Universidad de Granada

Departamento de Arquitectura y Tecnología de Computadores

Programa Oficial de Doctorado en
Tecnologías de la Información y la Comunicación

TESIS DOCTORAL

ACELERACIÓN Y OPTIMIZACION DEL CONSUMO ENERGETICO DE CLASIFICADORES EN CASCADA PARA LA DETECCION DE ROSTROS SOBRE ARQUITECTURAS ASIMETRICAS

Realizada por:

Jesús Alberto Corpas Novo

Dirigida por:

Manuel Rodríguez Álvarez
Guillermo Botella Juan

Granada, Junio 2019

Editor: Universidad de Granada. Tesis Doctorales
Autor: Jesús Alberto Corpas Novo
ISBN: 978-84-1306-289-1
URI: http://hdl.handle.net/10481/56822

A la memoria de mi hermano Jorge. Mi héroe.

AGRADECIMIENTOS

A mis directores de tesis doctoral, D. Manuel Rodríguez Álvarez y D. Guillermo Botella Juan, por su apoyo e inestimable ayuda durante el tiempo empleado. Quiero hacer hincapié en que la realización de este trabajo no hubiera sido posible sin el especial apoyo, guía y comprensión por parte de ellos.

También deseo expresar mi más profunda gratitud a todos los que han estado próximos a mí durante la realización de esta memoria, como mi familia, mis amigos y, a todos los que me han ayudado a ver la Luz en este trabajo.

Gracias a todos.

NOTACIÓN

Las referencias a secciones (Sec.c.n), figuras (Figura c.n), tablas (Tabla c.n), ecuaciones y demás expresiones matemáticas ((c.n)), se hacen incluyendo en primer lugar el número, c, de capítulo (1,2,..., 8) o apéndice (A) del que se trata, seguido de un número, n, de orden correlativo dentro del capítulo o apéndice.

Las referencias bibliográficas se codifican entre corchetes, con las cinco primeras letras del primer apellido del primer autor, seguidas del año de publicación. Caso de que se efectúen varias citas en el mismo año o el primer autor tenga varias referencias, se diferencia en el año de publicación. Las referencias bibliográficas completas se incluyen al final de la memoria ordenadas según el código indicado anteriormente: orden alfabético/año de publicación.

Índice general

Índice general I
Índice de figuras V
Índice de tablas IX
Glosario de Siglas y Términos XI
Resumen XIII
Abstract XV
1. INTRODUCCIÓN 1
1.1 Motivación 2
1.2 Objetivos 5
1.3 Estructura de la tesis doctoral 8
1.4 Conclusiones 11
2. EL PROBLEMA DE LA DETECCION FACIAL 13
2.1 Introducción 14
2.2 Métodos actuales de detección facial 17
2.2.1 Métodos basados en el conocimiento 18
2.2.2 Métodos basados en características invariantes 18
2.2.3 Métodos de coincidencias de plantillas 20
2.2.4 Métodos basados en apariencias 21
2.3 Evaluación de los métodos de detección facial. 21
2.3.1 Ejemplos de algoritmos para la detección facial 23
2.3.1.1 Algoritmo de Viola-Jones 23
2.3.1.2 Extracción de funciones de Gabor 24
2.3.1.3 Análisis del perfil de proyección (APP) 25
2.3.1.4 Comparativa entre algoritmos 25
2.4 Aplicaciones de la detección facial 26
2.5 Conclusiones 30
3. EL ALGORITMO DE VIOLA-JONES 31
3.1 Introducción 32
3.1.1 Características tipo Haar 33
3.1.2 Calculo de características con la Imagen Integral 37
3.1.3 Clasificador simple 40
3.1.4 El algoritmo Adaboost 41
3.1.5 Cascada de clasificadores fuertes 46
3.1.6 Diseño de Cascada de clasificadores 48
3.1.7 Complejidad temporal del algoritmo de Viola-Jones 51
3.1.8 Mejoras del algoritmo de Viola-Jones 53
3.2 Conclusiones 56
4. IMPLEMENTACIONES HARDWARE DEL ALGORITMO 59
4.1 Plataformas hardware para implementar el algoritmo de Viola-Jones 60
4.1.1 Plataformas hardware para la detección de objetos 61
4.1.1.1 CPU Multi-núcleo 61
4.1.1.2 Procesador digital de señal (DSP) 64
4.1.1.3 Unidad de Procesamiento de Gráficos (GPU) 66
4.1.1.4 Circuitos Integrados Específicos (ASIC) 69

4.1.1.5 Matriz de Puertas Programable (FPGA) 70
4.1.2 Hacia un sistema de visión integrado heterogéneo en CHIP 73
4.1.2.1 Problemas y desafíos del diseño de hardware 74
4.1.3 Resumen comparativo entre las diferentes plataformas hardware 76
4.2 Entornos tecnológicos para implementar sistemas de detección facial 79
4.2.1 Entornos hardware multi-núcleo 81
4.2.1.1 Placa Raspberry Pi 82
4.2.1.2 Placa Odroid XU4 85
4.3 Conclusiones 87
5. DESARROLLO Y ACELERACIÓN DEL ALGORITMO 89
5.1 Implementación algoritmo de Viola-Jones en C++ 90
5.2 Entorno de trabajo 96
5.3 Paquetes y estructura funcional del sistema de detección 97
5.3.1 Descripción funcional del sistema 99
5.3.2 Depuración y optimización secuencial del software 100
5.4 Análisis de rendimiento. Ejecución secuencial 102
5.5 Aceleración de la ejecución del programa 109
5.5.1 Sistemas homogéneos y heterogéneos 112
5.5.2 Técnicas de mapeo 113
5.5.3 Mapeo del algoritmo de detección facial 116
5.5.3.1 Análisis de dependencias y transformación del algoritmo 116
5.5.3.2 Particionado y asignación de tareas. 116
5.5.3.3 Asignación, planificación y equilibrio de carga 119
5.5.4 Modelos de programación en paralelo 121
5.5.5 Metodología para la paralelización del programa detección facial 123
5.5.6 Open multi-processing (OPENMP) 124
5.6 Resultados de la paralelización del algoritmo de detección facial 128
5.7 Conclusiones 138
6. OPTIMIZACIÓN DEL CONSUMO ENERGÉTICO 141
6.1 Introducción 142
6.2 Procesadores multi-núcleo asimétricos 143
6.2.1 Modelos de ejecución para arquitecturas Big.LITTLE 144
6.3 Entorno de medida del consumo energético 145
6.4 Mejora del rendimiento y optimización del consumo energético en arquitecturas asimétricas 146
6.4.1 Técnicas de mapeo aplicadas a sistemas asimétricos 148
6.4.2 Alternativas para la paralelización en plataformas asimétricas 149
6.4.3 Modelos de programación paralela en arquitecturas asimétricas 150
6.4.4 Políticas de reducción del consumo 152
6.4.4.1 Políticas basadas en el escalado de frecuencia 153
6.4.4.2 Políticas basadas en la asignación de tareas en OmpSs 157
6.4.4.3 Políticas basadas en los parámetros “step” y “scalaFactor” 159
6.4.5 Adaptación de OmpSs a arquitecturas asimétricas 162
6.4.6 Resultados optimización del consumo energético 168
6.5 Conclusiones 181
7. IDENTIFICACIÓN DE ROSTROS Y ACELERACIÓN DEL SISTEMA CON NEURAL STICK MOVIDIUS 183
7.1 Introducción 184
7.1.1 Redes Neuronales Convolucionales (CNN) 185

7.1.1.1 Capa convolucional ... 187
7.1.1.2 Capa de reducción o *pooling* ... 188
7.1.1.3 Capa clasificadora totalmente conectada ... 189
7.2 Arquitectura de un sistema de identificación facial ... 189
7.3 Detección facial con Facenet. ... 190
7.3.1 Comparativa entre métodos de detección facial ... 192
7.4 Identificación de rostros usando Facenet Tensorflow ... 194
7.4.1 Ejecución del programa de identificación de rostros ... 197
7.5 Aceleración del sistema de identificación facial ... 201
7.5.1 Movidius Neural Compute Stick ... 201
7.5.2 Funcionamiento del programa de identificación facial ... 205
7.5.3 Resultados de la aceleración de sistema de identificación facial ... 210
7.6 Conclusiones ... 218
8 CONCLUSIONES Y TRABAJOS FUTUROS ... 221
8.1 Conclusiones ... 222
8.2 Trabajos futuros ... 230
9 BIBLIOGRAFIA ... 233
APÉNDICE A ... 247
APÉNDICE B ... 253
APÉNDICE C ... 257
APÉNDICE D ... 259

Índice de figuras

Figura 2.1: Modelo de Saber y Tekalp [Sabem, 1998]. 20
Figura 2.2: Grafica que determina la tasa de equierror ERR [Fuente: elaboración propia]. 23
Figura 3.1: Características tipo Haar utilizadas por Viola-Jones [Fuente: elaboración propia]. 35
Figura 3.2: Ejemplo de característica tipo Haar [Chang, 2008]. 35
Figura 3.3: Filtros Haar en una ventana 4×4 pixel, [Kassn, 2017] 36
Figura 3.4: Representación de una imagen genérica con una característica [Fuente: elaboración propia]. 36
Figura 3.5: Cálculo de un rectángulo a partir de la imagen integral. Para calcular el valor del rectángulo A se debe realizar la operación: A−B−C+D [Viola 2004]. 38
Figura 3.6: Cascada de clasificadores [Fuente: elaboración propia]. 46
Figura 3.7: Curva Roc Comparando un clasificador de 200 características respecto a un clasificador en cascada 10 etapas con 20 características cada una. [Viola, 2004]. 48
Figura 3.8: Curvas ROC para el detector de Viola-Jones para diferentes escalas y desplazamiento de la ventana de detección. [Viola, 2004]. 51
Figura 3.9: Diagrama de bloques para el entrenamiento del algoritmo de Viola-Jones 52
Figura 3.10: Características extendidas, utilizadas por Lienhart y Mayt. [Raine, 2002]. 54
Figura 4.1: Arquitectura CPU multi-núcleo formada por 4 núcleos [Fuente: elaboración propia]. 62
Figura 4.2: Diagrama de bloques de una GPU AMD FireStream [Harre, 2009] 67
Figura 4.3: Elementos básicos que constituyen una FPGA [Capot, 2016]. 71
Figura 4.4: Layout Raspberry Pi [Rasb2, 2017] 83
Figura 4.5: Arquitectura CPU de la Raspberry Pi 2 B [Fuente: elaboración propia]. 84
Figura 4.6: Raspberry Pi 2 B [Rasb2, 2017]. 85
Figura 4.7: Placa Odroid XU4 [Odroi, 2016]. 86
Figura 4.8: Arquitectura CPU de la placa Odroid XU4 [Fuente: elaboración propia]. 86
Figura 5.1: Desplazamiento de la ventana de detección sobre la imagen [Zheny, 2017] 90
Figura 5.2: Diagrama de bloques del programa de detección facial. 91
Figura 5.3: Imagen de entrada y de salida del programa de detección facial [Natio, 2016]. 92
Figura 5.4: Pirámide de imágenes. Representación multi-escala de una imagen [Zheny, 2017]. 94
Figura 5.5: Pseudo-código para implementar el algoritmo de Viola-Jones 95
Figura 5.6: Conjuntos de ficheros que componen el sistema de detección. 97
Figura 5.7: Contenido del fichero: "info.txt". Cada línea representa el número de características por etapa del clasificador en cascada. 98
Figura 5.8: Estructura del contenido del fichero: "class.txt". 99
Figura 5.9: Muestra representativa de las 10 imágenes usadas en las pruebas experimentales. 103
Figura 5.10: Resultados obtenidos en placa Odroid XU4 (izquierda) y Raspberry Pi 2 B (derecha). 103
Figura 5.11: Valor de la imagen integral (N) para cada una de la imágenes de prueba con diferente número de rostros. 104

Figura 5.12: Muestra de las 10 imágenes de un solo rostro usadas en las pruebas experimentales. 105
Figura 5.13: Resultados obtenidos en la placa Odroid XU4. 106
Figura 5.14: Resultados obtenidos en la placa Raspberry Pi 2 B. 106
Figura 5.15: Resultados obtenidos en placa Odroid XU4. 107
Figura 5.16: Resultados obtenidos en placa Raspberry Pi 2 B. 108
Figura 5.17: Resultados profiling en Odroid XU4 (izquierda) y Raspberry Pi 2 B (derecha). 111
Figura 5.18: DAG del programa de detección facial [Fuente: elaboración propia] 118
Figura 5.19: Fragmento de código para el cálculo secuencial del valor de las características. 119
Figura 5.20: Fragmento de código para el cálculo en paralelo del valor de las características. 119
Figura 5.21: Diferencia entre arquitecturas de memoria distribuida y 123
Figura 5.22: Modelo FORK – JOIN [Fuente: elaboración propia] 126
Figura 5.23: Ventajas y desventajas de OpenMP. 128
Figura 5.24: Fragmento de código fuente donde se llama a la función: runCascadeClassifier. 129
Figura 5.25: Fragmento de código fuente con las Directivas #pragma incluidas. 130
Figura 5.26: Fragmento de código fuente de la función "runCascadeClassifier" donde se llama a la función "evalWeakClassifier". 131
Figura 5.27: Fragmento de código fuente de la función "runCascadeClassifier" con las Directivas #pragma incluidas. 132
Figura 5.28: Tiempo de ejecución al aplicar directivas OpenMP. 134
Figura 5.29: Comparación entre los tiempos de ejecución secuencial y paralela 134
Figura 5.30: Profiling algoritmo paralelizado con OpenMP en Raspberry Pi 2 B. 135
Figura 5.31: Comparación entre consumo energético en la ejecución secuencial y paralela en la plataforma Raspberry Pi 2 B+. 137
Figura 5.32: Comparación entre consumo energético en la ejecución secuencial y paralela en la plataforma Odroid XU4. 137
Figura 6.1: Diagrama de bloques del procesador Exynos 5422 de la placa Odroid XU4, constituido por el cluster Big de 4 núcleos y el cluster LITTLE de 4 núcleos [Fuente: elaboración propia]. 146
Figura 6.2: Información sobre núleo CPU 4 obtenida al ejecutar la instrucción "cpufreq-info" en la placa Odroid XU4. 155
Figura 6.3: Fragmento deCódigo fuente para el control de frecuencia de ejecución 156
Figura 6.4: Evolución error total en función de los parámetros "step" y "scalaFactor". .. 161
Figura 6.5: Gráfico Acíclico Dirigido (DAG) del programa de detección facial. 163
Figura 6.6: Fragmento de código fuente de la función "ScaleImage_Invoker" donde se llama a la función "runCascadeClassifier", con las directivas de OmpSs incluidas. 166
Figura 6.7: Fragmento de Código fuente de la función "runCascadeClassifier" donde se llama a la función "evalWeakClassifier", con las directivas de OmpSs incluidas. 166
Figura 6.8: Comparación entre tiempos de ejecución secuencial y paralela al aplicar la directivas de OmpSs al programa de detección facial, y ejecutarlo en la placa Odroid XU4. 167
Figura 6.9: Comparación entre consumos energético en la ejecución secuencial y paralela al aplicar la directivas de OmpSs al programa de detección

facial en la placa Odroid XU4, sin aplicar políticas de optimización del consumo energético. 168
Figura 6.10: Consumo de energía de ejecución vs tiempo de ejecución y parámetros "scalaFactor" y "step", para las frecuencias del cluster: Big = 2000 MHz, LITTLE = 1400MHz 170
Figura 6.11: Tiempo de ejecución, consumo de potencia y error de detección según los parámetros "scalaFactor" y "step", para las frecuencias del clúster: Big = 2000 MHz , LITTLE = 1400 MHz 171
Figura 6.12: Consumo de energía de ejecución vs tiempo de ejecución y parámetros "scalaFactor" y "step", para las frecuencias del cluster: Big = 1500 MHz, LITTLE = 1400MHz 172
Figura 6.13: Tiempo de ejecución, consumo de potencia y error de detección según los parámetros "scalaFactor" y "step", para las frecuencias del clúster: Big = 1500 MHz , LITTLE = 1400 MHz 173
Figura 6.14: Consumo de energía de ejecución vs tiempo de ejecución y parámetros "scalaFactor" y "step", para las frecuencias del cluster: Big = 1000 MHz, LITTLE = 1400MHz 174
Figura 6.15: Tiempo de ejecución, consumo de potencia y error de detección según los parámetros "scalaFactor" y "step", para las frecuencias del clúster: Big = 1000 MHz , LITTLE = 1400 MHz 175
Figura 6.16: Consumo de energía de ejecución vs tiempo de ejecución y parámetros "scalaFactor" y "step", para las frecuencias del cluster: Big = 800 MHz, LITTLE = 1400MHz 176
Figura 6.17: Tiempo de ejecución, consumo de potencia y error de detección según los parámetros "scalaFactor" y "step", para las frecuencias del clúster: Big = 800 MHz , LITTLE = 1400 MHz 177
Figura 7.1: Esquema red neuronal artificial [Fuente: elaboración propia]. 186
Figura 7.2: Convolución de matrices [Fuente elaboración propia]. 187
Figura 7.3: Estructura de las Redes Neuronales Convolucionales [Fuente: elaboración propia]. 187
Figura 7.4: Operación Max-Pooling [Fuente: elaboración propia]. 188
Figura 7.5: Arquitectura de un sistema convencional de identificación facial [Fuente: elaboración propia] 189
Figura 7.6: Imágenes de prueba con la que se han obtenido los resultados experimentales en el sistema de detección facial basado en el algoritmo de Viola-Jones optimizado y en el sistema Facenet. 193
Figura 7.7: Representación gráfica del tiempo de ejecución y consumo energético en función del número de rostros de la imagen de prueba. Obtenidos ejecutando sobre la placa Odroid XU4: 194
Figura 7.8: Resultado de la ejecución del programa de identificación que usa el modelo Facenet sobre la placa Odroid XU4 para la identificación del rostro del ex-presidente de los Estados Unidos Barak Obama. Izquierda identificación positiva, derecha identificación negativa. 197
Figura 7.9: Imágenes de muestra para la identificación facial. 198
Figura 7.10: Imagen con el rostro a identificar en las imágenes de muestra. 199
Figura 7.11: Representación gráfica del tiempo de ejecución y consumo de potencia del sistema de identificación facial Facenet en función de número de rostros de la imagen de prueba. 200

Figura 7.12: Resultado de la ejecución del programa de identificación facial, para un caso positivo (izquierda) y otro negativo (derecha). 200
Figure 7.13: Movidius Neural Compute Stick [Intel, 2018]. 202
Figura 7.14: Diagrama de bloques detallado de Intel / Movidius Myriad 2 [Molon, 2014]. 203
Figura 7.15: Diagrama de bloques del programa de identificación facial 205
Figura 7.16: Esquema de funcionamiento del programa de identificación facial desarrollado en lenguaje de programación Python [Fuente: elaboración propia]. 208
Figura 7.17: Entorno de trabajo: Odroid XU4, y cuatro Movidius NCS. 210
Figura 7.18: Representación gráfica de los Resultados de tiempo de ejecución medio y consumo de potncia medio durante la ejecución del programa de identificación facial sobre la placa Odroid XU4, usando los sistemas de detección facial Viola-Jones y Facenet para diferente número de NCS y para resoluciones de las imágenes de prueba de 640×480, 320×240 y 230×160. 212
Figura 7.19: Representación gráfica de los Resultados de pixeles procesados por segundo y rendimiento por vatio del programa de identificación facial sobre la placa Odroid XU4, usando los sistemas de detección facial Viola-Jones y Facenet para diferente número de NCS y para resoluciones de las imágenes de prueba de 640×480, 320×240 y 230×160. 212
Figura 7.20: Representación gráfica de rendimiento por vatio y tiempo de ejecución del programa de identificación facial sobre la placa Odroid XU4, usando los sistemas de detección facial Viola-Jones y Facenet para diferente número de NCS y una resolución de las 10 imágenes de prueba de 640×480 pixel. 214
Figura 7.21: Uso de CPU de la Odroid XU4 con Viola-Jones y Ningún NCS. 216
Figura 7.22: Uso de CPU de la Odroid XU4 con Facenet y Ningún NCS. 216
Figura 7.23: Uso de CPU de la Odroid XU4 con Viola-Jones y 4NCS. 216
Figura 7.24: Uso de CPU de la Odroid XU4 con Facenet y 4 NCS. 216
Figura 7.25: Representación del porcentaje medio de uso de CPU durante la ejecución del programa de identificación facial en la placa Odroid XU4, usando Viola-Jones y el sistema Facenet para la detección facial, sin NCS, un NCS, dos NCS, tres NCS y cuatro NCS. 217

Índice de tablas

Tabla 2.1: Comparación entre algoritmos y requisitos para el sistema de detección de rostros. 26
Tabla 3.1: Complejidad temporal del entrenamiento de los clasificadores del algoritmo de Viola-Jones. 53
Tabla 4.1: Implementaciones del Algoritmo de Viola-Jones en CPU multi-núcleo. 63
Tabla 4.2: Implementaciones del Algoritmo de Viola-Jones en DSP. 65
Tabla 4.3: Implementaciones del Algoritmo de Viola-Jones en GPU. 68
Tabla 4.4: Ejemplo de implementaciones del Algoritmo de Viola-Jones en ASIC. 70
Tabla 4.5: Implementaciones del Algoritmo de Viola-Jones en FPGA. 72
Tabla 4.6: Implementaciones del Algoritmo de Viola-Jones. 77
Tabla 4.7: Comparativa entre plataformas hardware, para los requerimientos del 78
Tabla 4.8: Modelos actuales de Raspberry Pi [Rasb2, 2017] 83
Tabla 5.1: Funciones principales del programa de detección facial. 100
Tabla 5.2: Estructura de OpenMP en C/C++ y Fortran. 130
Tabla 5.3: Resultados de tiempo de ejecución obtenidos para la ejecución secuencial y paralela del programa de detección facial en las placas Odroid XU4 y Raspberry Pi 2. 133
Tabla 6.1: Principales políticas de programación de OmpSs [Omps, 2018] 158
Tabla 6.2: Datos obtenidos para diferentes valores del parámetro "step", y "scalaFactor"=1,2. 160
Tabla 6.3: Datos obtenidos para diferentes valores del parámetro "step", y "scalaFactor"=1,2. 160
Tabla 6.4*: Datos obtenidos para diferentes valores del parámetro "scalaFactor", y "step"=1.* 160
Tabla 6.5: Datos obtenidos para diferentes valores del parámetro "scalaFactor", y "step"=1. 160
Tabla 6.6: Resultados de consumo de energía, tiempo de ejecución y parámetros "scalaFactor" y "step", para las frecuencias del cluster: Big = 2000 MHz, LITTLE = 1400MHz 170
Tabla 6.7: Resultados de consumo de energía, tiempo de ejecución y parámetros "scalaFactor" y "step", para las frecuencias del cluster: Big = 1500 MHz, LITTLE = 1400MHz 172
Tabla 6.8: Resultados de consumo de energía, tiempo de ejecución y parámetros "scalaFactor" y "step", para las frecuencias del cluster: Big = 1000 MHz, LITTLE = 1400MHz 174
Tabla 6.9: Resultados de consumo de energía, tiempo de ejecución y parámetros "scalaFactor" y "step", para las frecuencias del cluster: Big = 800 MHz, LITTLE = 1400MHz 176
Tabla 6.10: Valores óptimos para reducir el consumo de energía y acelerar el tiempo de ejecución para una tasa de detección del 90%. 178
Tabla 6.11: Comparación entre los resultados de aplicar la función de OpenCV: detectMultiScale, y los obtenidos para el sistema de detección facial con "step"=1 y "scalaFactor"=1,2, sobre las dos bases de rostros experimentales: Base-450 y Base-750. 179

Tabla 6.12: Resultados de Precision y Recall para la función OpenCV detectMultiScale y el sistema de detección en las bases experimentales: base-450 y base-750. ... 180
Tabla 7.1: Resultados de tiempo de ejecución y consumo energético sobre la placa Odroid XU4, para el método de Viola-Jones optimizado en capítulo 6, y el método de detección facial de Facenet, usando las imágenes de prueba del capítulo 5 que contienen un número diferente de rostros. ... 193
Tabla 7.2: Modelos de Facenet pre-entrenados proporcionados por David ... 196
Tabla 7.3: Resultados de tiempo de ejecución y consumo de potencia del programa de identificación facial usando Facenet como algoritmo de detección de rostros para ... 199
Tabla 7.4: Resultados de tiempo de ejecución, consumo de potencia y rendimiento del programa de identificación facial ejecutado sobre la placa Odroid XU4, comparando los sistemas de detección facial Facenet con el que usa el algoritmo de Viola-Jones, y haciendo uso de ningún NCS, un NCS, dos NCS, tres NCS y cuatro NCS. Y para resoluciones de las 10 imágenes de prueba de 640×480, 320×240 y 230×160. ... 211
Tabla 7.5: Porcentaje medio de uso de CPU de la placa Odroid XU4, durante la ejecución del programa de identificación facial, comparando los sistemas de detección facial de Facenet con el que usa el algoritmo de Viola-Jones, y haciendo uso de ningún NCS, un NCS, dos NCS, tres NCS y cuatro NCS para las 10 imágenes de prueba con una resolución de 640×480 pixel. ... 215

Glosario de Siglas y Términos

AdaBoost	Adaptive Boosting. Aumento adaptativo
AMP	Asymmetric Multicore Processor.
ANN	Artificial Neural Networks. Redes Neuronales Artificiales.
API	Application programming interface. Interfaz programación Aplicaciones.
APP	Análisis del perfil de proyección.
ARM	Advanced RISC Machine.
ASIC	Circuito integrado para aplicaciones específicas.
CART	Classification and Regression Trees. Árboles de Clasificación Regression.
CBIR	Content Based Image Retrieval.
CCTV	Circuito cerrado de televisión.
CLB	Bloques lógicos configurables.
CNN	Redes Neuronales Convolucionales.
CPD	Centro de Procesado de Datos.
CPU	Central Processing Unit. Unidad Central de Procesamiento.
CPUM	CPU Migration.
CSM	Cluster Switching Mode. Modo de conmutación de clúster.
DAG	Directed Acyclic Graph. Gráfico Acíclico Dirigido
DMA	Direct Memory Access. Memoria de acceso directo.
DNN	Red Neuronal Profunda.
DR	Detection Rate. Tasa de detección.
DRAM	Dynamic Random Access Memory. Memoria dinámica acceso aleatorio.
DSP	Digital Signal Processor. Procesador digital de señales.
DVFS	Dynamic voltage and frequency scaling.
ERR	Equal Error Rate.
FAR	False Acceptance Rate. Tasa de aceptación falsa.
FIFO	First In, First Out. Primero en Entrar, Primero en Salir.
FP	Falsos Positivos.
FPGA	Field Programmable Gate Array.
FRR	False Rejection Rate.
GCC	GNU Compiler Collection.
GPU	Graphics Processing Unit. Unidad de Procesamiento de Gráfico.

GTS	Global Task Scheduling. Programación Global de Tareas.
HDL	Hardware Description Language. Lenguaje de descripción hardware.
HPC	High performance Computing.
IKS	In-Kernel Switcher.
INCITS	Int. Nat. Committee for Information Technology Standards.
ISA	Industry Standard Architecture.
LIFO	Last In, First Out. Último en entrar, primero en salir.
MPI	Message Passing Interface. Interfaz de paso de Mensajes.
NCS	Neural Computer Stick.
PBM	Portable Bitmap Format.
PC	Personal Computer. Ordenador Personal.
PCA	Principal component analysis. Análisis de Componentes Principales.
PCB	Printed Circuit Board. Placa de circuito impreso.
PGM	Portable Graymap Format.
PPM	Formato portable pixmap.
RISC	Reduced Instruction Set Computer.
RIT	Relación entre el Tiempo de ejecución e Imagen integral.
ROC	Receiver Operating Characteristic.
RPF	Raspberry Pi Foundation.
RSAT	Rotated Summed Area Table.
SAT	Summed Area Table.
SBC	Single Board Computer. Computadora de placa única.
SIMD	Single Instruction Multiple Data.
SMP	Symmetric Multi-Processing. Multiprocesamiento simétrico.
SoC	System-On-Chip. Sistema en un solo Chip.
SVM	Máquina de vector de soporte.
TLP	Thread-Level Parallelis. Paralelismos a nivel de hilos.
TPU	Unidad de Procesamiento de Tensor.
USB	Universal Serial Bus. Bus Universal en Serie.
VC	Virtual Core, Núcleos Virtuales.
VLIW	Very Long Instruction Word.
VPU	Video Processing Unity. Unidad de Proceso de Video.

Resumen

Esta tesis doctoral propone un mecanismo para acelerar y optimizar el consumo de energía de un software de detección facial basado en clasificadores en cascada tipo Haar, aprovechando las características de los procesadores asimétricos multi-núcleo (AMP) de bajo coste con un consumo de energía limitado. Se propone un modelado de asignación de tareas para hacer un uso eficiente de las funciones existentes en los procesadores ARM Big.LITTLE, que incluyen:

I Adaptación de código fuente para la ejecución en paralelo, que permite la aceleración del código aplicando el modelo de programación OmpSs, que es un modelo de programación basado en tareas que maneja las dependencias de datos entre tareas de manera transparente.

II Aplicación de diferentes políticas de asignación de tareas que tienen en cuenta la asimetría del procesador y pueden establecer dinámicamente los recursos de procesamiento de una manera más eficiente en función de sus características particulares.

El mecanismo propuesto se puede aplicar de manera eficiente para aprovechar los elementos de procesamiento existentes en dispositivos integrados multi-núcleo de bajo coste y bajo consumo de energía que ejecutan algoritmos de detección de objetos basados en clasificadores en cascada. Aunque estos clasificadores ofrecen los mejores resultados para detección de objetos en el campo de la visión artificial, sus altos requerimientos de cómputo impiden que se ejecuten en estos dispositivos en tiempo real.

Por otro lado, se compara la eficiencia energética de una arquitectura heterogénea basada en procesadores multi-núcleo asimétricos con una programación de tareas adecuada, con la de una arquitectura simétrica homogénea. Para hacer esto, se realiza una comparación entre placas con procesadores integrados de bajo costo con características similares, como son la placa Odroid XU4 y la placa Raspberry Pi 2 B.

Finalmente, se hace una aplicación práctica del sistema de detección facial desarrollado en la tesis doctoral, para ello se implementa un programa de identificación facial en leguaje de programación Python a partir de un sistema de identificación pre-entrenado basado en Redes Neuronales Convolucionales (CNN), llamado Facenet. Sobre dicho programa se instala el sistema de detección facial y se comprueba el rendimiento del mismo como parte del sistema de identificación facial global. Por otro lado, se procede a acelerar el programa de identificación desarrollado con uno, dos, tres y cuatro dispositivos Intel Movidius Neural Compute Stick (NCS) en paralelo, para analizar el efecto de los mismos en el rendimiento global del sistema de identificación.

Abstract

This doctoral thesis proposes a mechanism to accelerate and optimize the energy consumption of a face detection software based on Haar-like cascading classifiers, taking advantage of the features of low-cost Asymmetric Multicore Processors (AMPs) with limited power budget. A modeling and task scheduling/allocation is proposed in order to efficiently make use of the existing features on big.LITTLE ARM processors, including:

I Source-code adaptation for parallel computing, which enables code acceleration by applying the OmpSs programming model, a task-based programming model that handles data-dependencies between tasks in a transparent fashion.

II Different task allocation policies which take into account the processor asymmetry and can dynamically set processing resources in a more efficient way based on their particular features.

The proposed mechanism can be efficiently applied to take advantage of the processing elements existing on low-cost and low-energy multi-core embedded devices executing object detection algorithms based on cascading classifiers. Although these classifiers yield the best results for detection algorithms in the field of computer vision, their high computational requirements prevent them from being used on these devices under real-time requirements.

On the other hand, we compare the energy efficiency of a heterogeneous architecture based on asymmetric multicore processors with a suitable task scheduling, with that of a homogeneous symmetric architecture. To do this, a comparison is made between boards with embedded low-cost processors with similar characteristics, such as the Odroid XU4 board and the Raspberry Pi 2 B board.

Finally, a practical application of the facial detection system is made, for which a program of facial identification in Python programming language is developed from a pre-trained identification system based on Convolutional Neural Networks (CNN), called Facenet. The developed facial detection system is installed on the program and its performance in the facial identification system is checked. Moreover, we proceed to accelerate the identification program developed with one, two, three and four in parallel, Intel Movidius devices, Neural Compute Stick (NCS), to analyze effect of these devices on overall performance of the identification system.

CAPÍTULO 1

1. INTRODUCCIÓN

En este capítulo se expone la motivación que ha llevado a realizar este trabajo de investigación. El trabajo se basa en las muchas aportaciones que se pueden realizar en el campo de la detección facial, incluidas las mejoras del rendimiento y aceleración de los algoritmos en dispositivos empotrados, y en la optimización del consumo energético en dispositivos de bajo coste con arquitecturas asimétricas actuales. Por otro lado, se indicarán los principales objetivos y retos que se van a considerar en su desarrollo. Finalmente, se mostrará la estructura en capítulos de la tesis doctoral con una breve descripción de cada uno de ellos.

1.1 Motivación

La evolución tecnológica de los componentes electrónicos ha seguido un crecimiento de carácter exponencial en las últimas décadas. Esto ha propiciado una transición desde los gigantescos superordenadores, a los potentes dispositivos móviles disponibles en la actualidad. Igualmente, se han llegado a desarrollar mini computadores y sistemas empotrados de muy bajo coste del tamaño de una tarjeta de crédito, y, por tanto, al alcance de cualquier persona. Este hecho pone a disposición del usuario una gran cantidad de herramientas para la implementación de algoritmos complejos que requieren una gran cantidad de recursos de computación y memoria, y que antes, por su baja eficiencia, no se podían ejecutar en este tipo de dispositivos en tiempo real.

Como muestra de algoritmos complejos se pueden tomar algunos de los que se utilizan para el reconocimiento facial en imágenes o videos, que requieren de una gran cantidad de recursos computacionales para poder procesar en tiempo real imágenes y localizar en ellas la existencia de caras de personas. La imagen del rostro humano puede tener grandes variaciones entre diferentes individuos, lo que hace difícil el desarrollo de un sistema estándar de detección. Las variaciones pueden ser, por ejemplo, la posición de la cabeza en el momento de tomar la imagen, la iluminación, la expresión del rostro (risa, enojo, etc.), ocultamiento de ciertas partes del rostro debido al uso de accesorios como gafas, sombreros, o rasgos faciales como bigote, barba, etc.

La detección facial está cada vez más presente en el mercado gracias al enorme abanico de aplicaciones prácticas (monitorización de individuos, identificación biométrica, etc.), implementadas en aplicaciones de redes sociales, de video vigilancia, o de defensa, entre otros. Debido al alto interés del mercado en estas aplicaciones han surgido diversos métodos y algoritmos [Karnr, 2012], [Peers, 1999] de los que cabe destacar el algoritmo desarrollado por Paul Viola y Michael Jones [Viola, 2001][Viola, 2004]. Éste ha obtenido unos muy buenos resultados con una alta tasa de detección facial y baja posibilidad de errores. Todos a un coste computacional muy reducido en comparación con otros algoritmos. Además se puede utilizar para detectar otro tipo de

objetos, no solo rostros, ya que puede ser implementado en cualquier aplicación donde se necesite detectar objetos con variabilidad estructural en tiempo real. Otra de las ventajas es que se trata de un método no intrusivo, es decir, los datos pueden ser adquiridos incluso sin que el sujeto se percate de ello. Además, el aspecto facial es el método más utilizado de manera natural por los seres humanos para reconocerse unos a otros.

Por otro lado, el paradigma de diseño de microprocesadores ha evolucionado mucho en los últimos tiempos. La rápida escalada del consumo energético y problemas de disipación térmica pusieron en evidencia que el incremento de frecuencia como técnica de mejora de rendimiento no podía mantenerse. A partir de ese momento, se ha optado por incluir un mayor número de elementos de procesamiento (núcleos) en un mismo chip, lo que ha sido posible gracias a la disminución del tamaño de los circuitos integrados. Estos nuevos diseños multi-núcleo complementados con otras mejoras arquitectónicas permitieron realizar un mayor uso del paralelismo a nivel de hilo (*Thread Level Parallelism* - TLP), mediante la inclusión de múltiples unidades funcionales en un mismo chip.

Los procesadores multi-núcleo convencionales, están formados por núcleos idénticos y están presentes en un gran número de dispositivos electrónicos. Actualmente los dispositivos móviles evolucionan para pasar de una CPU de un solo núcleo a una CPU multi-núcleo, de manera similar a la que se ha producido en el mundo del PC durante la última década. Mientras que las CPU de un solo núcleo eran la corriente principal en los teléfonos móviles inteligentes de hace todavía pocos años, hoy en día, incluso los teléfonos móviles de gama baja cuentan con dos o más núcleos, y los modelos de teléfonos inteligentes de las principales marcas (Samsung, Motorola, LG, Sony, etc.) ya cuentan con CPU multi-núcleo, y sin duda se verán aún mayor número de núcleos en los futuros productos. La misma tendencia se aplica también a las familias de microcontroladores empotrados.

En este sentido se pueden distinguir entre los procesadores de alto rendimiento, que integran núcleos de elevado consumo y alta frecuencia de trabajo, como el Intel

Core I9-7960X (16 núcleos), que basan su diseño en la optimización de la capacidad de cómputo mediante el uso de técnicas como la ejecución fuera de orden o el lanzamiento múltiple (con un claro aumento del consumo energético como contrapartida), y los procesadores constituidos por núcleos de baja frecuencia y consumo reducido, cuya filosofía se centra en el ahorro energético, como los Intel Atom o los ARM Cortex A7 o A53. Estos últimos procesadores están presentes en dispositivos móviles en los que el ahorro de energía es un aspecto fundamental. Cabe destacar que estos procesadores, pueden ofrecer buen rendimiento en aplicaciones con gran paralelismo a nivel de hilo (*Thread-Level Parallelism* - TLP), ya que las paradas de *pipeline* que pueden suceder mientras un hilo se ejecuta en un núcleo no impiden que otros hilos prosigan su ejecución en otros núcleos.

En los últimos años, el consumo energético ha emergido como una de las principales barreras que ha frenado la mejora de rendimiento en los sistemas de cómputo, desde dispositivos móviles hasta grandes centros de procesamiento de datos (CPDs). Con el fin de mejorar la eficiencia energética, resulta necesario explotar mayores grados de especialización en el hardware. Las arquitecturas actuales proporcionan diversos tipos de heterogeneidad, que van desde configuraciones heterogéneas en CPDs, hasta procesadores con aceleradores hardware integrados en el propio chip, plataformas híbridas formadas por procesadores de propósito general (CPUs) y aceleradores de propósito específico (como pueden ser GPUs, Intel Xeon Phi, etc.), multi-núcleos asimétricos (Asymmetric Multicore Processor - AMP) con repertorio común de instrucciones o multi-núcleo heterogéneos con múltiples repertorios de instrucciones [Mitta, 2016].

La correcta explotación del rendimiento y la eficiencia energética ofrecidos por estas arquitecturas conlleva, de forma necesaria, un incremento en la complejidad del software que permita, de forma lo más transparente posible para el usuario final, la adaptación de aplicaciones ya existentes o de un nuevo desarrollo para aumentar el rendimiento y reducir su consumo energético [Luisc, 2016]. Trabajos previos han demostrado que para explotar el potencial de los procesadores multi-núcleo asimétricos,

el sistema debe tener en cuenta el beneficio relativo que cada aplicación experimenta al ejecutar en un núcleo rápido frente a un núcleo lento [Chron, 2015].

En la actualidad los planificadores por defecto de los sistemas operativos de propósito general no explotan la diversidad de procesos para mejorar el rendimiento de un sistema en una carga de trabajo multiprogramada. Uno de los principales retos en este sentido es la correcta asignación de trabajo a los recursos computacionales existentes, de modo que, de forma simultánea, se optimice su grado de ocupación y la adaptación del recurso escogido a la tarea específica a desarrollar aumentando así la eficiencia del dispositivo. En respuesta a estas necesidades una de las soluciones propuestas por la comunidad científica es el desarrollo de modelos de programación que que exploten el paralelismo a nivel de tareas en tiempo de ejecución a través de planificadores de tareas (comúnmente conocidos como *runtimes* [Duran, 2011]). Esta capa de software es capaz de gestionar de forma transparente al usuario los procesos de gestión de dependencias de datos entre tareas.

Atendiendo a todo lo expuesto en los apartados anteriores, a continuación, se describirán brevemente los objetivos de la tesis doctoral.

1.2 Objetivos

Este trabajo nace de la necesidad de implementar y aumentar la velocidad de ejecución en tiempo real de un algoritmo de detección facial eficiente, como puede ser el algoritmo desarrollado por Paul Viola y Michael Jones [Viola, 2001], en dispositivos empotrados modernos de bajo coste. En este sentido, una premisa importante es que las técnicas propuestas para mejorar la velocidad no deben aumentar significativamente el consumo de energía para no reducir el tiempo de vida de las baterías que normalmente alimentan a este tipo de sistemas.

Por lo tanto, el objetivo principal de esta tesis doctoral es demostrar la posibilidad de construir un sistema de detección facial eficiente, que funcione en tiempo real en los minidispositivos de bajo coste multi-núcleo, con tecnología similar a los

actuales teléfonos móviles inteligentes y sistemas empotrados, aprovechando todos sus recursos y prestaciones. Para tal propósito se propone utilizar la placa Raspberry Pi 2 B y la placa Odroid XU4. La primera dispone de un SoC (*System-On-Chip*) con un procesador de 4 núcleos simétricos (ARM Cortex A7), y la segunda contiene un procesador de 8 núcleos asimétricos (4 ARM Cortex A15 y 4 ARM Cortex A7) con igual repertorio de instrucciones.

La utilización de una placa con una arquitectura multi-núcleo simétrica y otra asimétrica va a permitir hacer un estudio comparativo en cuanto a la eficiencia energética que supone el uso de un sistema simétrico frente a otro asimétrico con un planificador de tareas específico y consciente de la asimetría de la CPU.

Para conseguir el objetivo principal se plantea desglosarlo en los siguientes subobjetivos:

1. Seleccionar un algoritmo eficiente para la detección facial, con una tasa de detección de rostros alta, y una tasa de error mínima, que ofrezca una detección de caras robusta y en tiempo real, y permita procesar imágenes de manera muy rápida y consiga una razón de detección alta. Para la realización de este algoritmo se tomará como base el propuesto por Viola-Jones [Viola, 2001], en 2001.

2. Desarrollar y optimizar el algoritmo de Viola-Jones en lenguaje de programación C++, usando técnicas de optimización y depuración de código que permitan conseguir un software óptimo acorde al objetivo de la tesis doctoral. La aplicación de detección facial estará principalmente enfocada a optimizar la carga computacional y minimizar los tiempos de detección, adaptándose a las limitaciones de los recursos que se van a utilizar.

3. Preparar, configurar y adecuar los entornos tecnológicos experimentales elegidos: la placa Raspberry Pi 2 B y la placa Odroid XU4. Ejecutar el software desarrollado y evaluar los tiempos de ejecución secuencial en cada una de las placas. Analizar los resultados obtenidos, a través de una herramienta de

Profiling como es Gperftools [Gperf, 2018], que es una interesante y rápida herramienta proporcionada por Google, que opera mediante un muestreo basado en tiempo, que permite un correcto análisis de aplicaciones con múltiples subprocesos. Además se evaluarán aspectos no funcionales como el consumo de energía.

4. A partir de los resultados obtenidos del *profiling*, extraer el paralelismo a nivel de tareas usando la API de OpenMP [Openm, 2017] y de OmpSs [Ompss, 2018]. Esta última, además de utilizarse para la aceleración del sistema, se usará para optimizar el consumo energético sobre la arquitectura asimétrica de la placa Odroid XU4, ya que dispone de un planificador de tareas consciente de dicha asimetría. Para ello, se generará el DAG (*Directed Acyclic Graph*), que es una representación gráfica del paralelismo de tareas asociada al algoritmo y que permitirá optimizar los recursos computacionales y acelerar la ejecución del programa. Esta técnica podrá integrarse en el entorno de análisis de prestaciones, de forma que su implantación pueda ser evaluada conjuntamente con otros parámetros funcionales para ver la relación entre el consumo de energía y el tiempo de ejecución.

5. Desarrollar técnicas de reducción del consumo energético que exploten de manera eficiente todos los recursos computacionales que ofrece la arquitectura asimétrica utilizada, la placa Odroid XU4.

6. Establecer unas reglas óptimas para la ejecución del programa en función de los parámetros del algoritmo implementado que permitan una ejecución óptima en tiempo real, con el menor impacto posible en el consumo de energía. El objetivo primordial de esta tarea es proporcionar un método que permita fijar a nivel funcional y no funcional los mejores parámetros para la ejecución del algoritmo en los dispositivos experimentales utilizados para ejecutar el programa desarrollado.

7. En función de los parámetros óptimos, realizar una comparativa de rendimiento entre las arquitecturas multi-núcleo simétrica y asimétrica.

8. Finalmente, realizar una experiencia práctica de uso eficiente del sistema de detección facial desarrollado, usándolo para la identificación de personas dentro de un conjunto de imágenes almacenadas.

1.3 Estructura de la tesis doctoral

Esta tesis doctoral propone diversas técnicas para acelerar y optimizar el consumo energético de un algoritmo para la detección facial basado en clasificadores en cascada tipo Haar, a través del aprovechamiento de las características que nos proporciona un procesador multi-núcleo simétrico y otro asimétrico (*Asymmetric Multicore Processor* - AMP) de bajo coste. Este último está formado por varios procesadores con el mismo repertorio de instrucciones pero distintas características de rendimiento y consumo. La tesis doctoral se ha estructurado en 8 capítulos.

En el capítulo 1 se describen los objetivos propuestos así como la estructura de la tesis doctoral.

En el capítulo 2 se describe el problema de la detección facial y una clasificación de los diferentes modelos que se suelen considerar para la resolución del mismo, así como los indicadores que marcan la precisión en la detección facial de estos sistemas, lo cual va a permitir hacer una comparativa entre diferentes algoritmos utilizados en la actualidad. Finalmente se verán algunas de las aplicaciones más conocidas de los sistemas de detección facial.

En el capítulo 3 se describe detalladamente el algoritmo presentado por Viola-Jones en 2001. Se describen las partes más importantes del mismo como son la imagen integral, el algoritmo de aprendizaje llamado Adaboost [Viola, 2001] y el diseño de clasificadores en cascada en función de los parámetros de precisión que se quiere alcanzar con el sistema de detección. Finalmente se mencionarán los resultados obtenidos en diferentes trabajos que se han desarrollado y que han mejorado el rendimiento del algoritmo, a través de modificaciones en el algoritmo de aprendizaje o

en la elección de las características de detección. Una de estas modificaciones es la que se encuentra implementada en la biblioteca de funciones de visión artificial desarrollada por Intel, denominada OpenCV.

En el capítulo 4 se introduce una visión general de las diferentes plataformas hardware utilizadas en el diseño de sistemas de visión para la detección facial, con el fin de detectar qué plataforma se ajusta mejor a los requerimientos de la tesis doctoral. Posteriormente, se definen dos entornos hardware de bajo coste, con tecnología similar a la disponible en los actuales teléfonos inteligentes, que se pueden utilizar para la implementación de sistemas de detección facial, como son la placa Raspberry Pi 2 B que dispone de un procesador ARM (Advanced RISC Machine) de cuatro núcleos simétricos Cortex-A7, y la placa Odroid XU4 cuyo procesador ARM big.LITTLE de ocho núcleos, está formado por dos *clúster* de 4 núcleos cada uno, el *clúster* Big: formado por 4 núcleos Cortex A15 de alto rendimiento, y el clúster Little: formado por núcleos Cortex A7 de bajo consumo (se trata por tanto de una arquitectura asimétrica).

En el capítulo 5 se describirán la metodología y el proceso llevado a cabo para la implementación y aceleración del software del algoritmo para la detección facial, así como los entornos de desarrollo experimentales donde se va a ejecutar. Para ello, se parte del código fuente de una implementación simplificada del algoritmo de Viola-Jones, realizada en C++, que dispone de unos parámetros de entrenamiento fijados y que presenta una tasa de detección alta para una amplia gama de imágenes de entrada. Dicha implementación se adecuará y optimizará al entorno de trabajo y se procederá a la extracción de paralelismo a nivel de tareas usando la API de OPenMP [Garci, 2003], con el fin de acelerar el programa de detección y aprovechar al máximo los recursos de los dispositivos. En este sentido, se llegará a un punto en el que mejorar el rendimiento de ejecución de un programa en C++ escrito originalmente para la ejecución tradicional en una sola CPU, implica la modificación de los algoritmos de procesamiento para utilizar varios núcleos de CPU en paralelo. En este caso, la API OpenMP [Garci, 2003], es una de las mejores opciones dentro de las tecnologías de programación en paralelo debido a que es soportado por diferentes sistemas operativos, herramientas de

compilación y otros dispositivos hardware, por lo que funciona hoy en día, incluso en dispositivos móviles.

En el capítulo 6, se realizará un estudio de las diferentes técnicas de reducción del consumo energético que exploten de manera eficiente todos los recursos computacionales que ofrece la arquitectura asimétrica de la placa Odroid XU4. Para ello, se hará uso de las funcionalidades que ofrece el planificador de tareas de OmpSs, consciente de la asimetría de la arquitectura de la CPU, así como de las políticas de asignación de recursos y explotación de los modos de bajo consumo proporcionado por dicha arquitectura, para permitir un aprovechamiento eficiente de la energía [Luisc, 2016]. Además, se realizará una evaluación de los resultados obtenidos bajo ejecución secuencial y paralela en cada una de las placas: Raspberry Pi 2 B [Raspb, 2017] y Odroid XU4 [Odroi, 2016]. Finalmente, se llevará a cabo un análisis experimental comparando los resultados obtenidos en dichas placas, con el fin de contrastar la mejora en la eficiencia energética que se produce en la arquitectura asimétrica frente a la simétrica, además de proporcionar algunas soluciones para optimizar el consumo energético.

En el capítulo 7 se llevará a cabo una experiencia práctica de uso del sistema de detección facial en el marco de un sistema general de identificación facial. En este capítulo se comparará su rendimiento respecto a otro sistema de detección facial basado en el modelo Facenet [Schro, 2015], que hace uso de Redes Neuronales Convolucionales (CNN). Seguidamente, se procederá a la aceleración del sistema de identificación facial a través de uno, dos, tres y cuatro dispositivos Intel Movidius Neural Compute Stick (NCS) [intel, 2018] en paralelo, comprobando de este modo la viabilidad del uso de estos dispositivos para la aceleración del sistema de identificación facial.

Por último, en el capítulo 8, se recogen las principales conclusiones y aportaciones de la tesis doctoral, así como algunos ejemplos de líneas de trabajo futuro relacionadas con la investigación llevada a cabo.

1.4 Conclusiones

En este capítulo se ha presentado la motivación que mueve a realizar la tesis doctoral, indicando la interesante evolución tecnológica que se está produciendo en los procesadores que pasan de tener un solo núcleo de procesamiento a tener varios. Esto hace necesario disponer de planificadores de tareas que permitan una distribución óptima entre los diferentes núcleos. Y todo ello sin incrementar considerablemente el consumo de energía. Es en este punto donde juegan un papel importante las arquitecturas asimétricas que contienen núcleos de bajo consumo que se pueden orientar en las tareas no críticas de los programas con el consiguiente ahorro de energía.

También se han expuesto los principales objetivos y retos planteados.

Por último, se han descrito brevemente cada uno de los capítulos en los que se ha estructurado la tesis doctoral.

CAPÍTULO 2

2. EL PROBLEMA DE LA DETECCION FACIAL

En este capítulo se realiza una introducción al problema de la detección facial. En primer lugar se presenta un planteamiento inicial general del problema, desde sus orígenes, pasando a explicar después distintos modelos que se suelen considerar para la resolución del mismo. Posteriormente se realiza una comparativa entre los distintos algoritmos con el fin de justificar la elección del modelo elegido en la presente tesis doctoral.

2.1 Introducción

La detección facial constituye una parte muy importante en el marco de la visión por computador e inteligencia artificial, sobre todo para la comunicación y la interacción hombre-máquina. Sin embargo al ser el rostro humano un objeto dinámico que tiene un alto grado de variabilidad en su apariencia, su detección es un problema difícil de tratar en visión por computador.

En este sentido se puede decir que la detección facial a través de medios computacionales es el proceso por el cual la computadora puede localizar las caras presentes en una imagen o video. Por tanto, dada una imagen, el objetivo del sistema de detección facial es identificar todas las regiones que contienen una cara sin importar su pose, orientación o condiciones de luz.

Este problema presenta una gran dificultad ya que las caras son objetos diferentes según el sujeto y variables en tamaño, forma, color y textura. Además, hay otros componentes faciales que puede aparecer como gafas, bigotes y expresiones faciales que pueden influir en el rendimiento del sistema de detección.

Los métodos de detección se han abordado desde diferentes enfoques [Gueva, 2008]:

- **Enfoques basados en rasgos faciales**, en los que se buscan determinados elementos que componen el rostro, como los ojos, la nariz o la boca.
- **Enfoques holísticos**, en este caso los métodos trabajan con la imagen completa o zonas concretas de la misma de la cual se extraen características que puedan representar el objeto buscado.
- **Enfoques híbridos**, estos métodos usan tanto la información local como la global para la detección, basándose en el hecho de que el sistema de percepción humano distingue tanto las características locales como globales del rostro.

Históricamente los primeros intentos para establecer un sistema de reconocimiento facial se remontan a 1896 [Villeg, 2005], año en el cual, Francis Galton

fue el primero en proponer un método sobre clasificación de caras. Éste recogía perfiles faciales como curvas, encontrando su norma y después se clasificaban perfiles por sus desviaciones con respecto a la norma. El resultado era un vector que podía ser comparado con otros vectores de la base de datos.

En 1960 se creó el primer sistema de reconocimiento facial que fue desarrollado por Woody Bledsoe, Helen Chan Lobo y Charles Bisson [Villeg, 2005] que introdujeron un sistema semiautomático que hacían marcas en las fotografías para localizar los rasgos principales: ojos, orejas, nariz y boca. Las distancias y radios se calculaban a través de dichas marcas para construir un sistema de referencia y poder comparar los datos. En 1970 Goldstein, Harmon y Lesk [Golds, 1971] crearon un sistema con 21 marcadores que incluían color del pelo y grosor de labios. Sus pruebas eran también difíciles de automatizar porque muchas de estas medidas se tomaban a mano.

Pocos años después Fisher y Elschlagerb [Karnr, 2012] introdujeron un sistema más autónomo que utiliza plantillas para medir los rasgos de diferentes partes de la cara, con las que construían un mapa global. Tras una continuada investigación resultó que las medidas no contenían suficientes datos únicos representativos como para identificar una cara de adulto.

En 1987 se introdujo por primera vez el procedimiento Eigenfaces por Sirovich y Kirby [Sirov, 1987], posteriormente desarrollado por Alex Pentland y Matthew Turk [Pentl, 1991], en 1991. El término "eigen" se refiere a un conjunto de 14 vectores propios, también conocido en Álgebra Lineal como "vectores característicos". La ventaja principal de este método es que puede representar un conjunto de imágenes utilizando una base formada de imágenes "eigen" cuya dimensión es mucho más pequeña que el conjunto original. La identificación se puede lograr mediante la comparación de dos imágenes representadas en la base “eigen” del conjunto de entrenamiento. El enfoque de "eigen" caras comenzó con la necesidad de encontrar una representación de pocas dimensiones de imágenes de rostros. Kirby y Sirovich demostraron que el Análisis de Componentes Principales (PCA) se puede utilizar en un grupo de imágenes de la cara para formar un conjunto de características básicas. Este

conjunto es conocido como "imágenes eigen" y se puede utilizar para reconstruir la colección de imágenes original. Cada rostro original sería reconstruido como una combinación lineal del conjunto base para extraer sólo la información crítica de la imagen de prueba y codificarla con la mayor eficacia posible. Fotografías de caras se proyectan en un espacio de características que ilustra mejor la variación entre las imágenes de rostros conocidos. Esta característica del espacio está definida por los vectores propios o "Eigenfaces". El vector de pesos expresa la contribución de cada cara eigen a la imagen de entrada.

En la década de los noventa fue cuando el desarrollo de algoritmos de detección de rostros inició su mayor crecimiento, proponiéndose una gran variedad de técnicas, desde algoritmos básicos de detección de bordes hasta algoritmos compuestos de alto nivel que utilizan métodos avanzados de reconocimiento de patrones. Como por ejemplo, el de la Agencia de Proyectos de Investigación Avanzada de Defensa (DARPA) y el Instituto Nacional de Estándares y Tecnología que lanzaron el programa de Tecnología de Reconocimiento Facial (FERET) a partir de la década de 1990 con el fin de fomentar el mercado de detección y reconocimiento facial comercial. El proyecto involucró la creación de una base de datos de imágenes faciales. La idea era que una gran base de datos de imágenes de prueba para el reconocimiento facial podría inspirar la innovación, que podría dar como resultado una tecnología de reconocimiento facial más poderosa.

A principios de la década de 2000, el Instituto Nacional de Estándares y Tecnología (NIST, por sus siglas en inglés), partiendo de FERET, diseñó un sistema de detección para proporcionar evaluaciones gubernamentales independientes de los sistemas de reconocimiento facial que estaban disponibles comercialmente. Estas evaluaciones se diseñaron para proporcionar a las agencias encargadas de hacer cumplir la ley y al gobierno de EE.UU. La información necesaria para determinar las mejores formas de implementar la tecnología de detección y reconocimiento facial.

A partir de 2010, la red social Facebook comenzó a implementar la funcionalidad de detección y reconocimiento facial que ayudó a identificar a las personas cuyas caras pueden aparecer en las fotos que los usuarios de la red social

actualizan a diario. Si bien la función fue instantáneamente criticada por los medios de comunicación, lo que generó una gran cantidad de artículos relacionados con la privacidad, a los usuarios de Facebook en general no pareció importarles. Al no tener un impacto negativo aparente en el uso del sitio web, más de 350 millones de fotos se cargan y procesan cada día mediante este sistema de detección y reconocimiento.

En 2017 Apple lanzó el iPhone X, anunciando la incorporación en el dispositivo de un sistema de detección y reconocimiento facial como una de sus características principales. Dicho sistema se usa en el teléfono con fines de seguridad facilitando el acceso a las funciones del móvil a su propietario. El nuevo modelo de iPhone se vendió rápidamente, lo que demuestra que los consumidores aceptan esta tecnología como un nuevo estándar para la seguridad.

En la actualidad, la evolución tecnológica ha permitido el desarrollo de sistemas de identificación facial muy precisos y eficientes, a través de las llamadas redes neuronales profundas (DNN), que ha demostrado su capacidad para resolver problemas de clasificación de imágenes utilizando un modelo jerárquico que aprende a partir de grandes bases de datos. Las DNN se han puesto muy de moda tras los logros conseguidos. Por ejemplo, Google ha logrado mejorar a su propio reCAPTCHA (reconocedor de texto en imágenes) con este tipo de redes neuronales. Estos avances hacen que cada vez se acerquen más a la idea de reproducir el funcionamiento del cerebro humano en un ordenador.

2.2 Métodos actuales de detección facial

Actualmente se pueden distinguir cuatro grandes categorías de métodos de detección de caras: basados en el conocimiento, basados en caracteres invariantes, basados en plantillas y basados en apariencia.

2.2.1 Métodos basados en el conocimiento

Las reglas en estos métodos están basadas en el conocimiento humano sobre las características que definen una cara. La mayoría de las reglas utilizan la relación entre determinadas características de la cara. Por tanto se diseñan principalmente para localización de rasgos faciales.

El desarrollo de un sistema de detección facial basado en estos métodos implica que una serie de reglas son definidas antes de la implementación del sistema. Estas reglas se derivan del conocimiento del investigador sobre las caras humanas. Un ejemplo de regla se puede representar en una cara a través de las distancias relativas y posiciones de los ojos, nariz, boca, etc. En general estos métodos carecen de entrenamiento previo del sistema de detección.

Un ejemplo de este método es el ideado por Yang y Huang, que utilizaron un método basado en conocimiento jerárquico para detectar caras [Yangh, 1994]. Su sistema consiste en tres niveles de reglas. En el nivel más alto todas las posibles caras son encontradas mediante un escaneado con una subventana sobre la imagen y la aplicación de un conjunto de reglas en cada localización. Otro método de este tipo es el de Ohya, Tang, Kawato y Nakatsu [Ohyar, 2000]. La técnica presentada por estos investigadores basa la detección en dos factores clave: distribuciones Gaussianas del color de la piel y el pelo, y diferenciación entre datos de piel relevantes utilizando una curva de decisión‖ que localice la línea del pelo en una cara humana. Finalmente, otro método es el presentado por Peer y Solina [Peers, 1999], donde se plantea una segmentación de las áreas de piel de una imagen a color y se aplican filtros y algoritmos de detección de bordes para localizar la presencia de ojos humanos en las muestras de piel. El proceso de detección es simple en esta técnica y ha sido diseñado para maximizar la velocidad del sistema.

2.2.2 Métodos basados en características invariantes

Estos algoritmos tienen por objetivo encontrar características estructurales que existen incluso cuando varía la pose, el punto de vista, las condiciones de luz, etc. Localizar en una imagen características útiles para la clasificación es una tarea compleja. A diferencia de los métodos basados en conocimiento, en este caso se buscan características

invariantes para la detección facial. Se ha observado que los seres humanos pueden detectar caras y objetos sin esfuerzo en diferentes poses y condiciones de luz y por eso deben existir propiedades o características que sean invariantes sobre los conjuntos que se nos presentan. A partir de esta presunción numerosos métodos han sido propuestos para detectar características faciales y deducir si se da o no la presencia de una cara.

Un ejemplo de este tipo de métodos es el conocido como algoritmo de Viola-Jones [Viola, 2001]. El sistema de detección propuesto por estos autores utiliza grupos de características simples rectangulares para llevar a cabo la detección. Las características utilizadas son como las del tipo Haar propuestas por Papageorgiou et al. [Papag, 1998] en 1998. Las características de Haar son funciones rectangulares simples de 2 dimensiones en las que se varía el tamaño y la posición de recuadros blancos y negros. Todas ellas se basan en la suma y diferencia de los valores de los píxeles dentro de los rectángulos. El algoritmo de Viola-Jones utiliza cuatro tipos de características Haar en su sistema de detección, que está basado en árboles de decisión con entrenamiento supervisado. De ahí que también sean conocidos estos métodos como clasificadores en cascada o clasificadores Haar. En el Capítulo 3 se describirá este método más detalladamente ya que será parte del objeto de la tesis doctoral.

Otro método es el propuesto por García y Tziritas [Garct, 1999], que consideran que el color y la textura también pueden ser considerados como un conjunto de características invariantes en una cara. García y Tziritas presentaron una técnica que utiliza regiones de color de piel cuantificadas y un análisis de paquetes de Wavelet*s* [Garct, 1999]. Su método tiene dos etapas principales: la detección del color de piel y el análisis efectivo de la textura utilizando una aplicación de análisis mediante filtros de paquetes Wavelets.

Por último, mencionar el método de Menser y Müller [Mense, 1999], basado también en la distribución de color, ya que utiliza el análisis de color y la aplicación de un mapa de distancias así como el análisis de componentes principales (PCA, *Principal Component Analysis*) para reducir la gran dimensión del espacio de la imagen en una muestra. La deducción se lleva a cabo a través de una comparación entre la energía de una sub-ventana con un umbral predeterminado.

2.2.3 Métodos de coincidencias de plantillas

En estos métodos a partir de un conjunto de muestras se construye un patrón facial estándar (normalmente de una cara en pose frontal). La relación entre la imagen de muestra y el patrón definido es observada y utilizada para hacer una deducción. Dada una imagen de entrada, se calculan unos valores de correlación utilizando independientemente los patrones estándar para el contorno facial, los ojos, la nariz y la boca. La existencia de una cara se determina en base a los valores de correlación. Estos métodos, como por ejemplo el propuesto por [Brune, 1993], tienen como ventaja una implementación sencilla. Sin embargo, se ha probado que son ineficaces para la tarea de la detección facial ya que no pueden tratar efectivamente la variación en escala, pose y forma de la cara.

Un ejemplo de este tipo es el propuesto por Saber y Tekalp [Sabem, 1998], que utiliza una función de coste basada en la simetría y un módulo de clasificación de formas basado en la deducción sobre un conjunto de muestra. Mediante el uso de un proceso de segmentación de piel esta técnica es capaz de trabajar basándose en una plantilla predefinida, que es usada junto a un modelo elíptico y un algoritmo de búsqueda de características faciales. Esto se realiza principalmente mediante el uso de un modelo de clasificación de forma. El modelo es una plantilla de cara basada en la naturaleza elíptica de la cara humana. El centro de la elipse se determina mediante el cálculo de *Eigenvectores* que se derivan de las coordenadas espaciales de los pixeles de piel identificados en una región. La plantilla se representa como una elipse y una función de coste de simetrías para las regiones de los ojos, la nariz y la boca (ver Figura 2.1).

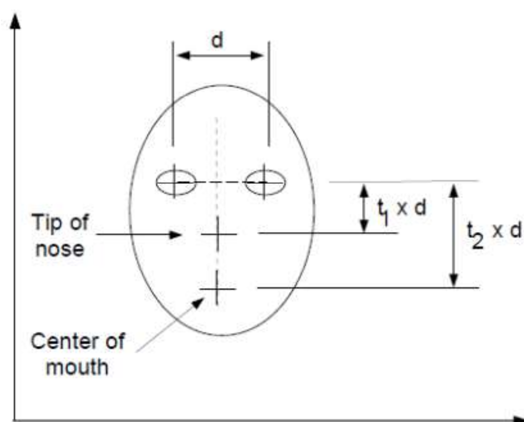


Figura 2.1: *Modelo de Saber y Tekalp [Sabem, 1998].*

2.2.4 Métodos basados en apariencias

Son similares a los anteriores, pero a diferencia de ellos las plantillas son obtenidas a partir de un conjunto de imágenes de entrenamiento que deberían capturar la variabilidad representativa de la apariencia de una cara. Estos métodos se usan principalmente para la detección facial. En general estos métodos se basan en técnicas de análisis estadístico y aprendizaje automático para encontrar características relevantes de imágenes faciales o de imágenes que no contengan caras. Las características aprendidas se expresan como modelos de distribución o funciones discriminantes que son utilizadas para la detección facial.

Un ejemplo de este tipo de métodos es el propuesto por Pham, Worring y Smeulders [Phamw, 2001], en cuyo método se presenta una red Bayesiana, con una estructura en forma de árbol para resolver un problema de clasificación de una sola clase [Phamw, 2001]. Se eligen los clasificadores Bayesianos debido a la gran velocidad que estos pueden alcanzar. La técnica también hace uso de *bagging* para generar un clasificador agregado ya que proporciona una manera natural de resolver los problemas de clasificación referidos a una sola clase. Otro de estos métodos es el de Rowley, Kanade y Balaujua [Rowle, 1996], que aplica un método basado en redes neuronales para proporcionar medios efectivos para la decisión. Se presenta como una red neuronal que se entrena para reconocer caras frontales. También combina algunas de estas redes para mejorar el rendimiento total del sistema. Los datos usados para entrenar cada red varían, específicamente con la frecuencia de imágenes de no caras. Cada red se entrena con imágenes con una resolución de 20 por 20 píxeles, donde cada ventana es pre-procesada para ecualizar la intensidad.

2.3 Evaluación de los métodos de detección facial.

La cara es la característica biométrica más comúnmente utilizada por los seres humanos. Debido a ello, las imágenes faciales han sido utilizadas para identificar a los individuos en los documentos personales. Los avances tecnológicos han permitido que las imágenes digitales de caras sean además la base de sistemas biométricos importantes como la verificación e identificación de personas, la video-vigilancia y el análisis de las

acciones humanas. Con el aumento de los sistemas biométricos basados en las imágenes de caras, ha surgido la necesidad de definir un formato de datos estándar, que establezca los requisitos indispensables para que las imágenes que se utilizan tengan valor identificativo y que permita la interoperabilidad entre diferentes sistemas.

La norma internacional ISO/IEC 19794-5 [Incit, 2004], creada por el Comité Internacional para los Estándares de Información y Tecnología (INCITS), y adoptada por la Organización Internacional de la Aviación Civil (ICAO), tiene como objetivo establecer los requisitos de las imágenes de rostros para aplicaciones de reconocimiento de personas y definir un formato para el almacenamiento e intercambio de las fotografías.

Con el fin de mejorar la precisión y efectividad del reconocimiento de rostros, ya sea de manera automática o llevada a cabo por humanos, la norma incluye más de 15 requisitos, que se dividen en tres grupos generales:

- Especificaciones de escena (pose facial, expresión, uso de accesorios).
- Características fotográficas (iluminación, nitidez, distancia focal, exposición, saturación).
- Atributos propios de las imágenes digitales (resolución, nivel de compresión, formato de los archivos, formas de almacenamiento).

En este contexto, referente a la calidad de las imágenes para la detección facial, para evaluar las prestaciones o rendimiento de este sistema, se suelen analizar y valorar los siguientes parámetros estándares [Biome, 2017]:

- Tasa de falsos positivos (FAR, *False Acceptance Rate*): Este índice mide la cantidad de objetos que no son caras y que son detectados.
- Tasa de falsos negativos (FRR, *False Rejection Rate*): que mide la cantidad de caras que no son detectadas como tal por el sistema.
- Tasa de detección (DR, *Detection Rate*): que mide el porcentaje de detecciones respecto al total de caras.
- Tasa de equierror (ERR, *Equal Error Rate*): corresponde al punto en que FAR y FRR coinciden, por lo que permite conocer el mejor funcionamiento mutuo de ambas tasas de error (ver figura 2.2).

Los parámetros FAR y FRR son parámetros inversamente proporcionales, y variarán en función de las condiciones prefijadas por el algoritmo de detección de rostros. Si el algoritmo se va a utilizar en un sistema de seguridad, se ha de intentar que el FAR sea lo más pequeño posible, aunque esto signifique el incremento del FRR. Se debe fijar un umbral que permita igualar estos dos factores, momento en el que se considera óptimo el funcionamiento del sistema.

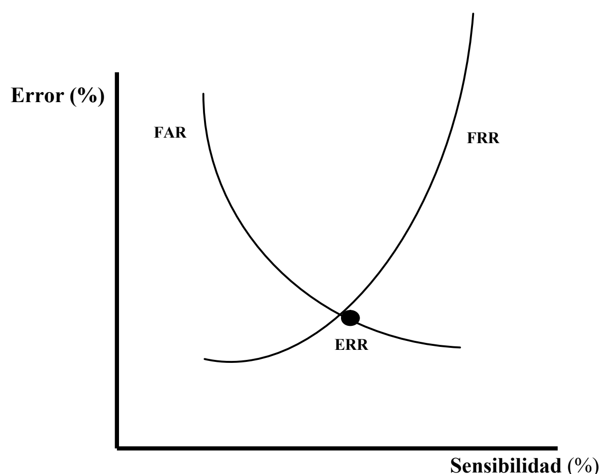


Figura 2.2*: Grafica que determina la tasa de equierror ERR [Fuente: elaboración propia].*

2.3.1 Ejemplos de algoritmos para la detección facial

Vistos la clasificación y los diferentes métodos que actualmente se utilizan para la detección facial, a continuación se van a estudiar, brevemente, tres algoritmos eficientes en tiempo real para la detección facial con el fin de seleccionar el más adecuado para el desarrollo del sistema de detección objetos de la tesis doctoral. En concreto se verá el algoritmo de Viola-Jones, el de extracción de funciones de Gabor, y el de análisis del perfil de proyección [Krist, 2012].

2.3.1.1 Algoritmo de Viola-Jones

El algoritmo de Viola-Jones se ha convertido en una herramienta muy común en los sistemas de detección de objetos, incluida la detección de rostros. Viola y Jones propusieron este algoritmo, que se encuentra dentro de los métodos de características invariantes, como un aprendizaje automático enfocado a la detección de objetos

haciendo énfasis en la obtención rápida de resultados y con altas tasas de detección. El algoritmo usa tres aspectos importantes. El primero es una representación de imagen estructurada llamada imagen integral [Viola, 2001]. En este algoritmo las características se calculan tomando la suma de los valores de los píxeles que están dentro de las áreas rectangulares que forman las características de Haar. Es decir, la suma de las regiones blanca y sombreada de cada rectángulo se calcula de forma independiente y se obtiene un valor que luego es comparado con un umbral que determina si cumple o no cumple una determinada condición.

El segundo aspecto introducido por Viola y Jones es al algoritmo de boosting Adaboost [Viola, 2001], que se utiliza para el aprendizaje del sistema y para la selección de las características que mejor identifican un rostro.

La última contribución del algoritmo de Viola-Jones es el uso de clasificadores en cascada por etapas que intentan descartar rápidamente aquellas áreas de la imagen que no contienen un rostro, un área que supera todas las etapas es probable que contenga un rostro.

2.3.1.2 Extracción de funciones de Gabor

Otro método de características invariantes de detección de rostros es usar filtros de Gabor para extraer las características de una imagen. Estos pueden ser aplicados a una red neuronal artificial (ANN) o a una máquina de vector de soporte (SVM) para la clasificación de imágenes. Sahooliza Deh et al. [Sahoo, 2008] y Gupta et al. [Gupta, 2010], usaron este método en la detección de rostros. En este algoritmo, se realiza una transformación de Wavelet de Gabor sobre la imagen permitiendo que el sistema encuentre puntos característicos en la misma. Estos se agregan al vector de características que es pasado a SVM o ANN, que ha sido previamente entrenado.

Una vez que se extraen estas características, el sistema usa uno de los métodos de aprendizaje (SVM o ANN) para verificar las caras alrededor los puntos característicos basados en un umbral de los píxeles circundantes. Más información sobre las matemáticas y el sistema se pueden ver en [Sahoo, 2008] y [Gupta, 2010].

2.3.1.3 Análisis del perfil de proyección (APP)

Este algoritmo se enmarca dentro de los métodos basados en apariencias y funciona primero fragmentando la imagen en dos segmentos: antecedentes y regiones de interés. El píxel que forma parte del segmento de interés, que suele ser el cuerpo humano, se marca como una máscara de un píxel. Una vez completado la segmentación del umbral del histograma, se realiza una reconstrucción morfológica de la imagen para llenar cualquier agujero en el segmento.

Las filas y columnas de esta máscara se usan para determinar cuántos píxeles de interés contiene cada fila o columna, de forma que se pueden usar para localizar la ubicación de la cabeza en función a los cambios esperados. Donde la cabeza se encuentra con los hombros, se produce un cambio repentino de intensidad en el píxel vertical [Krist, 2012]. De este modo se puede determinar donde se encuentra el rostro.

2.3.1.4 Comparativa entre algoritmos

Atendiendo a los resultados publicados por Reese et al. [Krist, 2012] y teniendo en cuenta los requerimientos de diseño para el sistema de detección, como son: fiabilidad, exactitud, rendimiento en tiempo real, Tasa de detección (DR) superior al 95%, tiempo de aprendizaje, flexibilidad y facilidad de implementación, en la Tabla 2.1 se muestra una comparativa entre los tres algoritmos descritos respecto a cada uno de dichos requerimientos.

Para ello, se han tenido cuenta los resultados experimentales para cada uno de los algoritmos, incluidos el tiempo promedio para completar una sola imagen en la base de datos experimental y la precisión, publicada en [Krist, 2012] [Viola, 2001] [Sahoo, 2008] y [Gupta, 2010]. Además se han usado los colores verde y rojo respectivamente para indicar la bondad o no del requerimiento en cada uno de los algoritmos.

Requisitos / Algoritmo	Fiabilidad	Precisión	Rendimiento en tiempo real	DR	Tiempo de Aprendizaje	Flexibilidad	Facilidad de implementación
Viola-Jones							
Gabor							
APP							

***Tabla 2.1**: Comparación entre algoritmos y requisitos para el sistema de detección de rostros.*

Como se puede apreciar en la tabla el algoritmo de Viola-Jones destaca positivamente por su precisión y negativamente porque requiere mucho tiempo de aprendizaje. En cuanto al de Gabor la extracción y expansión de características es muy lenta. El tiempo que toma el algoritmo para completar una detección de rostros en una imagen hace que este no se pueda implementar en un sistema de seguridad en tiempo real [Krist, 2012]. La precisión tampoco es mejor que las que presenta el algoritmo de Viola-Jones. Por último, indicar que el método de Análisis del Perfil de Proyección muestra los menos éxitos de todos los algoritmos estudiados. Aunque el mismo es rápido, es incapaz de encontrar rostros en una imagen con una precisión adecuada para los requerimientos indicados para el sistema de detección facial objeto de la tesis doctoral.

En general, se puede concluir que el algoritmo de Análisis del Perfil de Proyección es el menos confiable de los métodos analizados para detectar caras, y el de Gabor no presenta un rendimiento temporal adecuado para su ejecución en tiempo real. Por tanto, de los algoritmos analizados, el que mejor se ajusta a las prestaciones seleccionadas para el sistema de detección facial es el de Viola-Jones.

2.4 Aplicaciones de la detección facial

La detección facial junto con otras soluciones de identificación biométrica es aceptada ampliamente por la sociedad actual por razones sobre todo de seguridad. Por ello, actualmente existen gran cantidad de científicos e ingenieros trabajando en innumerables investigaciones al respecto. Sin embargo, hay muchos otros campos en los

que el análisis facial puede ofrecer gran cantidad de posibilidades. Desde un punto de vista genérico se pueden destacar las siguientes aplicaciones:

- Identificación facial.
- Autenticación digital.
- Interfaces para interacción hombre-maquina.
- Estimación de la edad.
- Detección de emociones.
- Video vigilancia.
- Diagnostico y detección de enfermedades.

Los sistemas de reconocimiento facial identifican a las personas por sus imágenes faciales. Estos pueden identificar y localizar la presencia de una persona autorizada en un lugar determinado. Estos sistemas suelen comparar directamente las imágenes de la cara de la persona detectada con una base de datos de rostros. Otro ejemplo de este tipo pueden ser los sistemas de control de acceso de una empresa. En este caso el tamaño del grupo de personas que necesitan ser reconocidas es relativamente pequeño. Otra aplicación de la tecnología de reconocimiento facial, puede ser, por ejemplo, en seguridad informática, para monitorizar continuamente quien está frente a una computadora. Esto permite al usuario salir del terminal sin cerrar archivos ni cerrar la sesión. Cuando el usuario se va por un tiempo predeterminado, un protector de pantalla cubre el trabajo y desactiva el ratón y el teclado. Cuando el usuario regresa y es reconocido, el protector de pantalla se borra y la sesión anterior aparece tal como quedó. Cualquier otro usuario que intenta acceder, no será reconocido y se le denegará el inicio de sesión.

Una de las aplicaciones más demandada actualmente, tiene que ver con la seguridad en lugares de tránsito de personas como pueden ser los aeropuertos. Los sistemas de protección que usan tecnología de reconocimiento de rostros están implementados en muchos aeropuertos de todo el mundo, como por ejemplo el aeropuerto internacional Fresno Yosemite en California que, en octubre de 2001, implementó una tecnología de reconocimiento facial para fines de seguridad aeroportuaria. Este sistema se diseñó para alertar al personal de seguridad del aeropuerto cada vez que un individuo coincide con la apariencia de un conocido

sospechoso de terrorismo. De igual forma, en 2011, el gobierno de Panamá autorizó un programa piloto con una plataforma de reconocimiento facial para reducir la actividad ilícita en el aeropuerto Tocumen de Panamá. Poco después de la implementación, el sistema resultó en la aprehensión de múltiples sospechosos de Interpol.

Se ha creado también aplicaciones para detección e identificación facial en bases de datos multimedia para la búsqueda, por ejemplo de personas desaparecidas. Por otro lado, el incremento de imágenes en Internet ha provocado que cada vez más, haya más aplicaciones CBIR (Content Based Image Retrieval) de recuperación de imágenes. Este es el caso de Informedia [Haupm, 1997], un proyecto de recuperación de programas de televisión y documentales. En sistemas con bibliotecas digitales de terabytes de video y audio la clasificación de las imágenes cumple un papel fundamental. El planteamiento de partida es tener un nivel de conocimiento previo de imágenes (etiquetadas) almacenadas en forma de base de datos, a partir de las cuales se quiere entrenar algún tipo de sistema de aprendizaje que permita resolver el problema concreto de identificación.

Otra posible utilidad es la vigilancia a gran escala de lugares públicos de grandes núcleos de población, como por ejemplo el sistema de vigilancia que se implanto en el distrito Newham Borough de Londres [Roger, 2002], que dispone de 300 cámaras conectadas a un circuito cerrado de TV (CCTV). El ayuntamiento afirma que esta tecnología ha ayudado a lograr una disminución de los delitos desde su instalación.

Otra aplicación de interés, que tiene que ver con crear interfaces para la interacción hombre-máquina, en concreto con la accesibilidad de personas con movilidad reducida a las Tecnologías de la Información y el Conocimiento (TIC). Para ello se han desarrollado sistemas que permiten a partir de la detección del rostro del usuario a través de la cámara del computador, una interacción directa con el mismo mediante movimientos leves de la cara, parpadeo de ojos, etc. Estos sistemas, por ejemplo, han permitido que persona con tetraplejia hayan podido manejar el ordenador de forma independiente sin ayuda externa.

Algunas de las aplicaciones de detección y reconocimiento facial más interesantes desarrolladas en el año 2018, se muestran a continuación:

- **Face2Gene** [Face2, 2018] es una aplicación de salud innovadora que utiliza reconocimiento facial para ayudar a los médicos y a la bioinformática a priorizar y determinar ciertos trastornos y variantes para sus pacientes. Funciona utilizando un algoritmo patentado que compara las caras de los individuos con aquellos con síndromes que presentan una morfología similar. Como resultado, los pacientes pueden ser diagnosticados más rápido y de manera más eficiente.
- **BioID** [BioID, 2018] es una aplicación de reconocimiento facial móvil que actúa como un autenticador biológico. A medida que el pirateo de contraseñas continúa aumentando, esta aplicación elimina la necesidad de una contraseña y en su lugar utiliza datos biométricos para demostrar su identidad.
- **FindFace** [Findf, 2018] es una aplicación de reconocimiento facial que el periódico The Guardian afirma que está "tomando a Rusia por sorpresa". Funciona comparando fotografías con imágenes de perfil en Vkontakte, una red social popular en Rusia, con más de 200 millones de cuentas. El objetivo de los diseñadores de la aplicación es un mundo donde la gente que pasa frente a ti en la calle pueda encontrar tu perfil de red social al filmar tu foto furtivamente. Los creadores de la aplicación afirman que es capaz de coincidir con 250 millones de caras en 0,3 segundos.
- **Luxand** [Luxan, 2018] ofrece un SDK de reconocimiento facial y API de detección de rostros que ofrecen todo tipo de características para aplicaciones, como la transformación de rostros en avatares 3D, la predicción de cómo se verán los niños y más.
- **Face Phi** [Facep, 2018] es una aplicación móvil que se utiliza en bancos de todo el mundo para verificar a sus clientes cuando se dedican a la banca móvil. Los clientes simplemente toman una foto de ellos mismos con su teléfono inteligente, que luego puede ser utilizado como un medio de identificación, lo que permite el acceso a la aplicación móvil del banco.
- **Face Detection Screen Lock** [Faced, 2018]. Esta aplicación de detección de rostros ayuda a mejorar la seguridad del dispositivo móvil al permitir a los

usuarios de Android bloquear y desbloquear su teléfono o archivos específicos mediante el reconocimiento facial.

2.5 Conclusiones

En este capítulo se ha presentado una introducción general al problema de la detección facial, presentando los modelos más comunes que se aplican para la resolución de dicho problema.

Se han definido los principales indicadores para medir y evaluar la bondad de un sistema de detección facial general como son: la Tasa de Falsos Positivos (FAR, *False Acceptance Rate*), la Tasa de falsos negativos (FRR, *False Rejection Rate*), la Tasa de Detección (DR, *Detection Rate*) y la Tasa de equierror (ERR, *Equal Error Rate*).

Una vez definido los indicadores para evaluar un sistema de detección se ha llevado a cabo una comparativa entre tres algoritmos diferentes utilizados en la detección de rostros en tiempo real como son: el algoritmo de Viola-Jones, el de Extracción de Funciones de Gabor y el de Análisis del Perfil de Proyección. De esto se ha concluido que la elección que mejor se adapta al sistema de detección objeto de la tesis doctoral es el algoritmo de Viola-Jones.

Por último, se ha ilustrado algunos ejemplos de aplicaciones de interés para la detección facial.

CAPÍTULO 3.

3. EL ALGORITMO DE VIOLA-JONES

En este capítulo se describe detalladamente el algoritmo presentado por Paul Viola y Michael Jones en el año 2001 [Viola, 2001]. Se describirán las partes más importantes del mismo como son la imagen integral, el algoritmo de aprendizaje llamado Adaboost, el diseño de clasificadores en cascada en función de los parámetros de precisión que se pretende alcanzar con el sistema de detección, y finalmente se mencionarán los resultados obtenidos en diferentes trabajos que se han desarrollado y que han mejorado el rendimiento del algoritmo, a través de modificaciones en el algoritmo de aprendizaje o en la elección de las características de detección.

3.1 Introducción

Como ya se ha comentado existen diversos métodos para la detección facial. Sin embargo, muchos de ellos no son aplicables cuando se busca hacerlo en tiempo real. El método propuesto por Paul Viola, de Mitsubishi Electric Research Labs., y Michael Jones, de Compaq CRL [Viola, 2001], es un algoritmo de detección facial con un coste computacional bajo que fue el primero en ofrecer una detección de rostros robusta y en tiempo real. Según los autores, operando sobre imágenes de 384×288 píxeles se detectarían caras a una velocidad de 15 imágenes por segundo utilizando un ordenador Pentium III de 700 MHz. Posee dos etapas principales: una de entrenamiento del detector, que es la que mayor tiempo de procesamiento demanda, y otra de detección donde se emplea el detector entrenado en la primera etapa sobre cada imagen a analizar. Esta segunda etapa es muy rápida y permite realizar la detección en tiempo real.

Para la detección de rostros, Viola y Jones analizan la imagen en busca de características relevantes que aporten información acerca de la presencia de un rostro en una imagen. Dicho algoritmo tiene una probabilidad de verdaderos positivos del 99,9 % y una probabilidad de falso positivos del 3,33%, y a diferencia de otros algoritmos utilizados en métodos de caracteres invariantes procesa sólo la información presente en una imagen en escala de grises. No utiliza directamente la imagen sino que utiliza una representación de la imagen llamada imagen integral, para determinar si en una imagen se encuentra una cara o no. El algoritmo divide la imagen integral en subregiones de tamaños diferentes y utiliza una serie de clasificadores (clasificadores en cascada), cada uno con un conjunto de características visuales. En cada clasificador se determina si la subregión es una cara o no. La utilización de este algoritmo supone un ahorro de tiempo considerable ya que no serán procesadas subregiones de la imagen que se sepa con certeza que no contienen una cara y sólo se invertirá tiempo en aquellas subregiones que posiblemente si contengan una cara. Este detector se ha hecho muy popular debido a su velocidad a la hora de detectar rostros en imágenes y por su implementación en la biblioteca de visión artificial OpenCV.

El sistema de detección de Viola-Jones utiliza grupos de características simples para llevar a cabo la detección. Las características se componen de rectángulos que

abarcan un conjunto de píxeles cuya suma de niveles de gris (luminancia) se utiliza para la evaluación. El uso de estas características posibilita una velocidad de detección mucho mayor a la hora de encontrar caras frente a un sistema basado en píxeles y beneficia la codificación en el dominio ad-hoc. Las características utilizadas son parecidas a aquellas del tipo Haar relacionadas con las propuestas por Papageorgiou et al. [Papag, 1998] en 1998. El algoritmo de Viola-Jones utiliza 4 tipos de características en su sistema de detección: dos tipos basados en dos rectángulos, un tipo basado en tres rectángulos y un tipo basado en cuatro rectángulos.

El trabajo de Viola y Jones realiza tres contribuciones fundamentales:

1. La creación de una nueva representación de la imagen, llamada imagen integral, que disminuye los tiempos de extracción de las características de una imagen al disminuir considerablemente el número de operaciones sobre los píxeles.
2. La construcción de un clasificador utilizando el algoritmo AdaBoost que se verá detenidamente más adelante, y que es utilizado tanto para seleccionar las características, como para entrenar los clasificadores.
3. Combinación de una sucesión de clasificadores cada vez más complejos cuando se avanza una etapa. La idea de este método es que los clasificadores más sencillos de bajo coste computacional descarten una gran cantidad de subventanas o subimágenes que no contienen rostros en las primeras etapas. dejando así concentrarse los clasificadores más complejos en las zonas donde es más posible encontrar una cara.

3.1.1 Características tipo Haar

La detección facial, se basa en tener en cuenta ciertas características de los rostros. En este sentido, una de las ideas principales del detector de caras propuesto por Viola-Jones son las denominadas características tipo Haar [Papag, 1998], que son funciones rectangulares simples de 2 dimensiones en las que se varía el tamaño y la posición de recuadros blancos y negros, y se usan para codificar la existencia de

contraste entre regiones en una imagen. La extracción de características es realizada aplicando a la imagen original filtros con bases Haar. Estos filtros pueden ser calculados eficientemente sobre la imagen integral (que es una representación de la imagen original que se explicará más adelante), son selectivos en la orientación espacial y frecuencia, y permiten ser modificados en escala y orientación. De esta manera, un conjunto de características pueden ser usadas para codificar los contrastes encontrados en los rostros. Las características tipo Haar se definen sobre regiones rectangulares de una imagen en escala de grises. Una característica está formada por un número finito de rectángulos y su valor escalar consistirá en la suma de los píxeles que componen cada rectángulo, que a su vez son sumados aplicando un cierto factor de peso. La fórmula para calcular el valor de una característica se representa matemáticamente según la ecuación 3.1:

$$característica_j = \sum_{1 \leq i \leq N} \omega_i \cdot Suma_rectangulo(r_i) \qquad (3.1)$$

Donde $\{r_1, \ldots, r_N\}$ son los rectángulos que forman la característica y ω_i el peso de cada uno que puede ser *1, -1, 2, -2* según el numero de cuadrados. La propuesta original de Viola-Jones, considera cuatro tipos de características, tres si se clasifican por el número de rectángulos que la forma [Viola, 2004]:

- Dos rectángulos: Su valor se calcula con la diferencia entre la suma de los píxeles dentro de las dos regiones rectangulares.
- Tres rectángulos: Su valor se calcula con la suma de los píxeles dentro de las dos regiones rectangulares exteriores y la substracción de los píxeles de la región interior.
- Cuatro rectángulos: Su valor se calcula con la diferencia entre las diagonales de los pares de rectángulos.

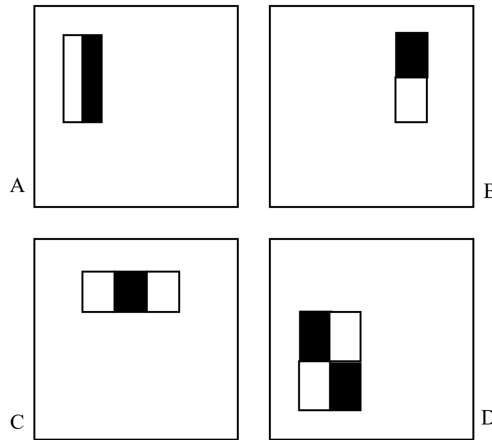


***Figura 3.1**: Características tipo Haar utilizadas por Viola-Jones [Fuente: elaboración propia].*

En la figura 3.1, se puede ver cada uno de los cuatro tipos de características. El valor de una característica se obtiene sumando todos los píxeles del rectángulo blanco y restándole todos los píxeles del rectángulo negro. Por ejemplo, la característica central de tres rectángulos trataría de representar que en general la región de los ojos es más oscura que las regiones de alrededor (Figura 3.2). También se puede ver que cada característica (excepto la de cuatro rectángulos) se puede rotar 90 grados para obtener nuevas características, también se puede invertir el peso de cada uno de los rectángulos.

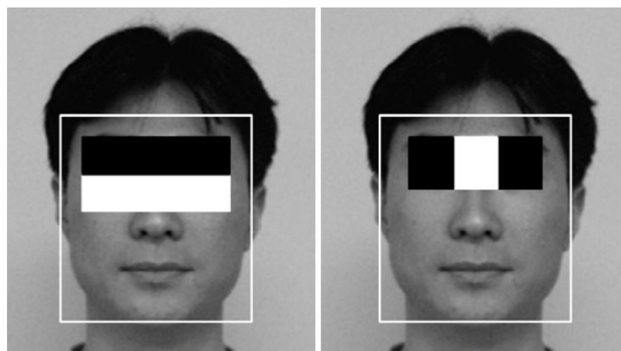


***Figura 3.2:** Ejemplo de característica tipo Haar [Chang, 2008].*

Las características no sólo se distinguen por su forma, sino también por su tamaño y posición dentro de una ventana, así como por la posible contribución de la misma a la detección de un rostro. Para ello es necesario calcular el valor de la característica, como se ha indicado anteriormente. Si este valor sobrepasa un

determinado umbral (umbral de clasificación), se considera que contribuye con un determinado valor α a la detección de una cara.

Algunas de las características tipo Haar que se pueden encontrar, por ejemplo, en una ventana de 4×4 píxeles se pueden ver en la Figura 3.3. El número de características diferentes que se pueden generar para una imagen de resolución 24×24 píxeles, si se considera la posición, escala, tamaño y tipo, por imagen es de más de 160.000 [Viola, 2004], cantidad mucho mayor que la cantidad de pixeles contenidos en la imagen de 24×24 pixeles. El número de características de cada tipo que se pueden obtener en una ventana determinada se puede obtener siguiendo el método de Lienhart y Maydit. Para hacer estos cálculos se hace uso de la figura 3.4, que nos sirve de modelo para identificar los parámetros utilizados en la ecuación 3.2 [Lienh, 2003].

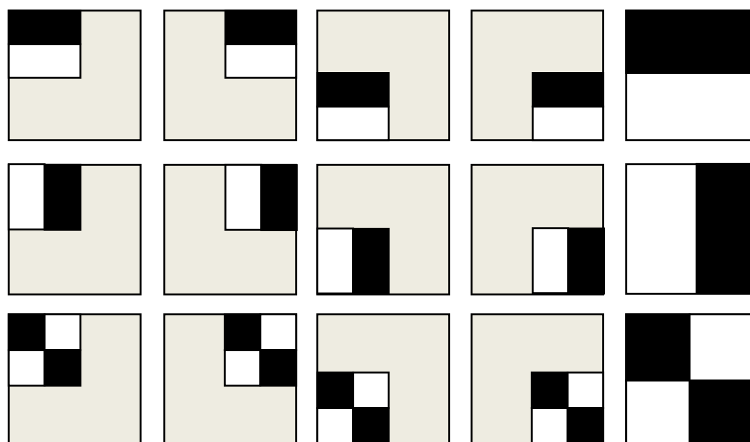


Figura 3.3: *Filtros Haar en una ventana 4×4 pixel, [Kassn, 2017].*

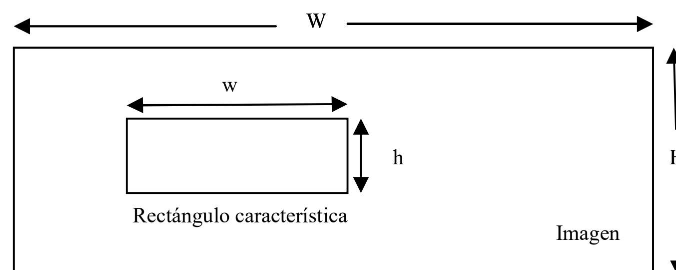


Figura 3.4: *Representación de una imagen genérica con una característica [Fuente: elaboración propia].*

$$F = X \cdot Y \left(W + 1 - w \frac{X+1}{2} \right) \left(H + 1 - h \frac{Y+1}{2} \right) \tag{3.2}$$

En la ecuación 3.2 se puede identificar: X=W/w, Y=$H/$h, siendo X e Y los valores máximos de escalado de la característica, W y H la anchura y altura máximas de la imagen respectivamente, w y h la anchura y altura máximas de la característica, y F el número de características que se pueden extraer de dicha imagen.

La experiencia realizada por Viola-Jones, demuestra que un número reducido de este tipo de características, escogido de entre todas la posibles en una imagen de resolución 24 por 24 pixeles, atendiendo a su máxima contribución en la detección de las caras, pueden determinar con cierto grado de exactitud si la imagen contiene o no un rostro. Así, por ejemplo, para un conjunto de 200 características se podría construir un clasificador que produce resultados razonables, alcanzando un 95% de tasa de detección [Viola, 2004].

3.1.2 Calculo de características con la Imagen Integral

Una vez definidas las características tipo Haar que se utilizan para determinar si una imagen contiene o no una cara, se necesita de un método que permita calcular de forma rápida el valor de cada una de ellas. Y a partir de éste ver cuál es su contribución en el sistema para la detección de rostros. Como se ha visto, se pueden definir más de 160.000 características diferentes en una ventana de 24×24 pixeles, modificando el tamaño de las características tipo Haar y su polaridad. Sumar uno a uno los pixeles de cada uno de los diferentes rectángulos tiene un coste computacional muy elevado, proporcional al área de dicho rectángulo. En este contexto, Viola-Jones hacen en su trabajo una de sus principales aportaciones introduciendo la llamada Imagen Integral, que es una representación de la imagen original que permite una rápida extracción del valor de las características al disminuir considerablemente el número de operaciones sobre los píxeles.

Se define la imagen integral de una Imagen I de tamaño n×m pixeles, como una función II definida por la siguiente ecuación 3.3.

$$II(x,y) = \sum_{\substack{1 \le x' \le x \\ 1 \le y' \le y}} I(x',y') \ para\ 1 \le x \le n, 1 \le n \le m \tag{3.3}$$

El valor de la imagen integral en el punto II(x,y) será igual al valor del pixel I(x,y) de la imagen original sumado a todos los pixeles de la imagen que estén a la izquierda y arriba de la posición (x,y). La imagen integral se puede calcular iterativamente comenzando por la posición (0,0). El pixel de abajo a la derecha contiene la suma de todas las intensidades de los pixeles de la imagen, como se puede ver en la ecuación 3.4.

$$\begin{aligned} S(x, y) &= S(x, y - 1) + I(x, y) \\ II(x, y) &= II(x - 1, y) + S(x, y) \end{aligned} \tag{3.4}$$

Donde S(x,y) es la suma acumulada de la fila, S(x,−1) =0, y II(−1, y) = 0. Gracias a esta representación se puede calcular la suma de los pixeles de un rectángulo de cualquier tamaño utilizando únicamente 4 accesos a memoria. Para calcular el valor de un rectángulo cuya esquina superior izquierda está en (x, y) y cuyo tamaño es (sx, sy) únicamente se necesita acceder al valor de la imagen integral de 4 esquinas. En la figura 3.5, se puede ver el esquema de la ecuación 3.5.

$$v(x, y, sx, sy) = II(x+sx, y+sy) - II(x-1, y+sy) - II(x+sx, y-1) + II(x-1, y-1) \tag{3.5}$$

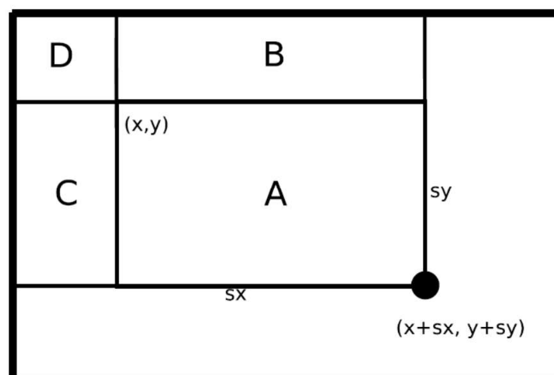


Figura 3.5: *Cálculo de un rectángulo a partir de la imagen integral. Para calcular el valor del rectángulo A se debe realizar la operación: A−B−C+D [Viola 2004].*

Además, como las características tipo Haar consisten en 2, 3 ó 4 rectángulos contiguos se pueden evitar accesos a memoria calculando el valor de todos los rectángulos conjuntamente.

La única desventaja de la imagen integral es que ocupa hasta 4 veces más en memoria que la imagen original. Al ser una suma de los pixeles de la imagen, no puede ser definida como matriz de 8 bytes tal y cómo se representan imágenes en escala de grises, así que se debe utilizar una matriz de enteros que en la mayor parte de sistemas ocupan 4 bytes [Morel, 2011].

La imagen integral se utiliza, además, para normalizar las imágenes de entrenamiento y la imagen de test en forma previa a la detección. De esta forma se reducen los efectos causados por diferencias de iluminación. Para normalizar la imagen, se calcula la desviación estándar utilizando la ecuación 3.6:

$$\sigma^2 = m^2 - \frac{1}{N}\sum x^2 \tag{3.6}$$

Donde σ^2 es la desviación estándar, m es la media de la imagen, N es igual a w (ancho) × h (altura) de la imagen y x es el valor de la imagen en un punto. El valor de la media m se puede obtener a partir de la imagen integral, mientras que el valor de $\sum x^2$ se obtiene de una segunda imagen integral que se construye según la ecuación 3.7.

$$II_{sqr}(x,y) = \sum_{x' \leq x,\, y' \leq y} (I(x',y'))^2 \;\; para \;\; 1 \leq x,\, x' \leq n, \;\; 1 \leq y, y' \tag{3.7}$$

Es decir, que por cada imagen se generan dos imágenes integrales *II* e II_{sqr}. Una vez obtenida la desviación estándar σ de una imagen I, se puede normalizar I multiplicando cada píxel por el factor σ^{-1}. En vez de ello, Viola-Jones obtienen el mismo efecto de normalización, multiplicando por este mismo factor los valores de las características después de evaluarlos.

3.1.3 Clasificador simple

La clasificación es un componente muy importante en los sistemas inteligentes. El problema que se plantea a la hora de realizar una clasificación consiste en decidir a qué “clase” pertenece un objeto. Sobre dicho objeto se realizan varias mediciones, que son las llamadas características. Por ello, lo que interesa es aprender y encontrar una relación entre dichas características y las diversas clases a las que se enfrentan el algoritmo.

El objetivo de un sistema de detección del tipo que se está considerando es encontrar aquellas características que mejor definan una cara y ayuden a localizarla en una imagen. La hipótesis planteada en el artículo de Viola-Jones establece que un pequeño número de esas características rectangulares pueden combinarse para formar un clasificador más preciso. Para ello establecen una variante del algoritmo de *boosting* AdaBoost de Freund y Shaphire [Freund, 1996]. Este algoritmo se utiliza para mejorar el rendimiento de un algoritmo de aprendizaje simple que define el entrenamiento de los llamados clasificadores débiles (*weak learner*, en inglés), cuya combinación forma un clasificador fuerte más preciso.

Formalmente un clasificador simple basado en una característica j se puede expresar según la ecuación 3.8.

$$h_j(x) = \begin{cases} 1 & si\ p_j f_j(x) < p_j \theta_j \\ 0 & en\ otro\ caso \end{cases} \quad (3.8)$$

Donde $f_j(x)$ es el valor que se obtiene al aplicar la característica j sobre la imagen x, θ_j es el umbral de la característica y p_j es la polaridad. Este último valor puede ser *1* o -1 y permite invertir una característica convirtiendo los rectángulos positivos en negativos y viceversa. Una de las utilidades de la polaridad es que en caso de que el clasificador obtuviera un error ϵ_j mayor que *0,5*, se podría utilizar la característica inversa y obtener un error de $(1 - \epsilon_j)$ que será menor que *0,5*.

Con un clasificador tan sencillo no se puede crear un detector de caras robusto. Viola y Jones propusieron combinar diversas características sencillas en un sólo clasificador entrenado con el ya mencionado algoritmo de *boosting* Adaboost, que se describirá en el apartado 3.1.4.

3.1.4 El algoritmo Adaboost

Para implementar los clasificadores simples, que forman parte de los clasificadores fuertes se tienen varias opciones utilizando diversos sistemas de aprendizaje, como pueden ser: SVM, perceptron, MLP, Naive Bayesian Classifier, etc.

Viola-Jones proponen en su trabajo un algoritmo de *Boosting* llamado Adaboost. El *Boosting* tiene su origen en la llamada teoría PAC (*Probably Approximately Correct*) propuesta por Valiant [Valia, 1984]. Donde se preguntan si se pueden combinar clasificadores débiles para dar lugar a un clasificador con prestaciones arbitrariamente elevadas, "fuerte". Fue Schapire quien introdujo el primer algoritmo de *Boosting* en forma de filtrado [Schap, 1990], y posteriormente Freund propuso una forma simple y elegante de combinar linealmente clasificadores débiles [Freund, 1996]. Años más tarde, Freund y Schapire introdujeron el primer algoritmo *Boosting* entrenable, denominado "AdaBoost" (*Adaptive Boosting*) [Freund, 1996]. Desde entonces son muchos los algoritmos *Boosting* que se han propuesto. Una descripción más detallada del origen de los algoritmos *Boosting* se puede encontrar en [Schap, 1990] y [Freund, 1996].

Por tanto, el *Boosting* es un sistema de aprendizaje cuya técnica consiste en unir varios clasificadores simples (en este caso basados en características tipos Haar), para crear un clasificador más complejo que tiene una menor tasa de error que los clasificadores simples individuales que lo forman. El algoritmo se encarga, a través de un sistema de aprendizaje previo de seleccionar las mejores características para la detección. A modo de resumen, la técnica que sigue el algoritmo de *Boosting* en cada iteración son:

- Evaluar cada característica, aplicando cada filtro rectangular, sobre cada imagen de ejemplo.
- Ordenar las imágenes según los valores obtenidos en el paso anterior.
- Seleccionar el mejor umbral para cada característica.
- Seleccionar el mejor filtro/umbral, es decir, la mejor característica.
- Actualizar los pesos de la característica para la detección del objeto en cuestión.

La ventaja principal del *Boosting* sobre otros métodos es la velocidad de aprendizaje y la capacidad de retroalimentación gracias a que posee, en cada iteración, información de las iteraciones anteriores representada en el peso de cada muestra de entrenamiento. Mediante los pesos se especifica la importancia de cada muestra en cada iteración.

El algoritmo de *Boosting* más conocido es el Adaboost. Es un algoritmo adaptativo porque va ajustando los clasificadores débiles que se van agregando a un clasificador fuerte, en base a las muestras mal clasificadas por los clasificadores débiles ya agregados.

En la primera fase del algoritmo de Viola-Jones se entrena el detector. Para el entrenamiento se utiliza una gran cantidad de imágenes de caras y de no caras de tamaño fijo. El entrenamiento consiste en seleccionar entre un gran conjunto de características, aquellas que mejor representan una cara, y utilizarlos para formar clasificadores débiles y clasificadores fuertes que una vez entrenados permiten hacer, en una segunda fase del algoritmo, una rápida detección de las cara contenidas en una imagen.

Adaboost se utiliza para seleccionar los mejores clasificadores simples. Teniendo en cuenta que existe un clasificador simple por cada característica se tiene que evaluar un total de $K \times N$ clasificadores, siendo K el número de características por imagen y N el número de imágenes que contenga nuestro conjunto de entrenamiento. Dicho de otro modo, Viola-Jones utiliza el algoritmo Adaboost para construir T clasificadores débiles para formar un clasificador fuerte que es una combinación lineal

de las T características seleccionadas, y cuyos pesos son inversamente proporcionales a los errores de entrenamiento. Formalmente se describe el algoritmo en los siguientes pasos:

1. Dado un conjunto de n imágenes $(x_1, y_1), \ldots, (x_n, y_n)$ donde x_i es una imagen e $y_i = \{0, 1\}$, el valor 1 representa que la imagen es una muestra positiva (es una cara) y 0 que no lo es.

2. Sean m y l el número imágenes con caras y no caras respectivamente. Se deben inicializar los pesos de cada imagen w_{1i} con los valores $1/(2 \times m)$ para las muestras positivas y $1/(2 \times l)$ para las muestras negativas.

3. Para cada una de las características $t = 1, \ldots, T$ (cada t se construye usando una única característica, es decir t viene a ser un clasificador simple). Se normalizan los pesos de las muestras para la característica t, según la ecuación 3.9:

$$w_{t,i} \leftarrow \frac{w_{t,i}}{\sum_{j=1}^{n} w_{t,j}} \quad (3.9)$$

3.1 Se selecciona el clasificador h_t con menor error ϵ_t. Para cada característica t se entrena un clasificador h_j. De modo que el error ϵ_t para ese clasificador sea mínimo (ecuación 3.10):

$$\epsilon_t = \min_{f,p,\theta} \sum_i w_i \, |h(x_i, f, p, \theta) - y_i| \quad (3.10)$$

El clasificador h_j que minimiza el error es guardado como: $h_j = h_t$

- Por tanto, se define el clasificado débil h de la característica t, h_t, como se expresa en la ecuación 3.11.

$$h_t(x) = h(x, f_t, p_t, \theta_t) \quad (3.11)$$

Donde f_t, p_t, θ_t son los minimizadores de ϵ_t.

- La característica se añade al clasificador final y se actualizan los pesos de modo que se reducen los pesos de las muestras clasificadas correctamente (ver ecuación 3.12), y se selecciona una nueva característica.

$$w_{t+1,i} = w_{t,i}\beta_t^{1-e_i} \qquad (3.12)$$

Donde $e_i = 0$ si se clasifica correctamente y 1 en otro caso. $\beta_t=\varepsilon_t/1-\varepsilon_t$.

El algoritmo continúa hasta que se han seleccionado T características. A cada clasificador débil se le asignará un valor α que será más grande cuanto mejor sea la característica.

4. Atendiendo a la combinación de todos los clasificadores débiles 1,…,T se puede obtener un clasificador fuerte según la ecuación 3.13.

$$C(x) = \begin{cases} 1 & \sum_{t=1}^{T} \alpha_t h_t(x) \geq \frac{1}{2} \sum_{t=1}^{T} \alpha_t \\ 0 & en\ otro\ caso \end{cases} \qquad (3.13)$$

Donde $\alpha_t = 1/\beta_t$

Se aplican las características del clasificador a la muestra y se suman los valores α de aquellas que consideren que la muestra es una cara. Si la suma es mayor que la mitad del total los valores α se etiqueta la muestra como cara.

Es posible utilizar la constante *1/2* como potenciador. Si se utiliza un valor menor se clasificarán más muestras como caras, mientras que si se aumenta el factor el clasificador será más exigente con las muestras y como resultado habría una menor detección del número de caras.

Uno de los problemas que tiene Adaboost es que no minimiza directamente los falsos negativos sino que minimiza el error total (falsos negativos + falsos positivos).

Esto hace que al entrenar un clasificador con Adaboost en el que se busca una tasa de falsos negativos mucho menor que la de falsos positivos, se obtenga un clasificador con más características de las necesarias [Plane, 2009].

Las mejores características se eligen basándose en el error ponderado que se produce. Este error ponderado es una función que utiliza los errores pertenecientes a los ejemplos de entrenamiento. El peso de un ejemplo clasificado correctamente se modifica mientras que el peso de un ejemplo mal clasificado se mantiene constante. Con esto se consigue que sea más difícil que la segunda característica clasifique erróneamente un ejemplo que haya sido clasificado erróneamente por la primera característica frente a un ejemplo clasificado correctamente por esa primera característica. Otra manera de ver esto sería que la segunda característica, y las sucesivas, se ven forzadas a tener más en cuenta los ejemplos clasificados erróneamente por características anteriores.

Para elegir la característica adecuada, las imágenes se ordenan según el valor de la característica que se evalúa en cada momento. El umbral de decisión óptimo para la característica en cuestión se calcula utilizando cuatro valores:

- T_+, suma total de los pesos de los ejemplos de entrenamiento positivos.
- T_-, suma total de los pesos de los ejemplos negativos.
- S_+, suma de los pesos de los ejemplos positivos que se encuentran por debajo del que se está evaluando.
- S_- suma de los pesos de los ejemplos negativos que se encuentran por debajo del ejemplo que se está evaluando.

Al elegir una característica y aplicar un umbral de decisión, la evaluación del error que conlleva es fácilmente calculable aplicando la ecuación 3.14:

$$e=min(S_+ + T_- - S_- \, , \, S_- + T_+ - S_+) \tag{3.14}$$

Con esta ecuación se evalúa el error de elegir entre clasificar todos los ejemplos por debajo del actual como negativos y los que se encuentran por encima positivos, y

viceversa. Se elige el error mínimo ya que al ser un decisor binario basta con cambiar el signo de la elección para conseguir el error mínimo de manera inmediata.

3.1.5 Cascada de clasificadores fuertes

Otra de las aportaciones del trabajo de Viola-Jones, es un método para combinar clasificadores sucesivamente más complejos en cascada que permiten incrementar rápidamente la velocidad del detector, ya que facilita el descarte rápido de las regiones que no tienen caras y se centran computacionalmente en aquellas regiones que tienen más probabilidad de tener caras. Es decir, la primera etapa la formará un clasificador fuerte (formado por un número pequeño de clasificadores débiles) optimizado para que con pocas operaciones pueda descartar el mayor número de no caras, de modo que el mayor consumo computacional se realice en las zonas donde hay mayor probabilidad de que exista un rostro. Este método recibe el nombre de Cascada de clasificadores (ver figura 3.6).

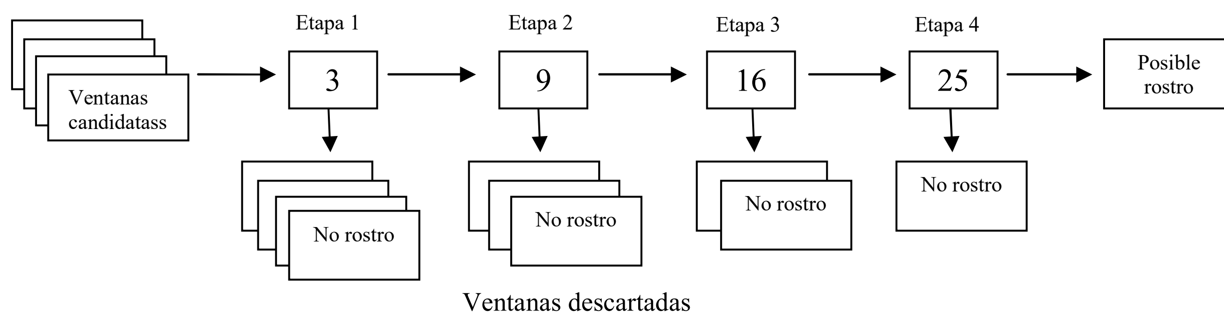


Figura 3.6: *Cascada de clasificadores [Fuente: elaboración propia].*

Así por ejemplo, la etapa 1 de la figura 3.4, está formada por un clasificador fuerte construido a partir de 3 clasificadores débiles. En otras palabras, en la primera etapa se evalúan 3 características en la ventana de detección. Si la evaluación de estas tres supera el umbral establecido en el proceso de entrenamiento del clasificador fuerte, se puede afirmar que hay probabilidad de que en esa región exista una cara, y por lo tanto pasa a ser evaluada por la siguiente etapa donde se evalúan más características. Si no supera el umbral, la región es descartada y se pasa a evaluar otra región de la imagen.

Este hecho es muy importante porque para localizar en una imagen cualquier rostro de diferente tamaño y ubicación, es necesario realizar un análisis de todas las zonas de la misma a través de una ventana de tamaño variable que se desplaza por toda la imagen, y sobre la que se aplica la cascada de clasificadores. En este sentido, mientras que ventana de caras puede haber unas pocas en una imagen, se pueden encontrar miles o millones de ventana de no caras. Este desequilibrio hace que sea importante que las primeras etapas descarten el mayor número de no caras al menor esfuerzo computacional posible. Si la ventana no es descartada en alguna de las etapas, se marca como un posible rostro. El proceso continúa desplazando la ventana (en sentido horizontal o vertical) según un factor de escaneo determinado. Una vez que se haya recorrido toda la imagen con un tamaño de ventana determinado, se incrementa el tamaño de la ventana (según un factor de escalado determinado) y se repite el procesamiento de las etapas y de barrido de la imagen. Los valores de los factores de escaneo y de escalado son obtenidos también durante el proceso de entrenamiento.

Por tanto, se puede concluir que el detector consistirá en una serie de clasificadores fuertes (etapas) puestos en cascada, como se muestra en la Figura 3.4. En este caso la cascada de clasificadores está formada por cuatro etapas formadas con 3, 9, 16 y 25 Clasificadores débiles respectivamente. Una ventana se clasificará con la primera etapa, si la etapa considera que es una cara continuará con la siguiente etapa. En cambio, si la etapa lo clasifica como no cara, la ventana se desecha. Las ventanas irán atravesando uno tras otro clasificadores cada vez más complicados y sólo las que lleguen al final de todos se considerarán caras.

Por otro lado, con el objetivo de demostrar que el clasificador en cascada es más veloz frente a un solo clasificador fuerte y, además, no pierde calidad en la detección, Viola y Jones entrenaron un clasificador con una etapa de 200 características por un lado, y otro con 10 etapas de 20 características cada una. En la figura 3.5, se puede ver la curva ROC (*Receiver Operating Characteristic*) combinando ambos clasificadores. A partir de la misma podemos ver una pequeña diferencia en términos de calidad, sin embargo la diferencia en términos de velocidad es mucho mayor.

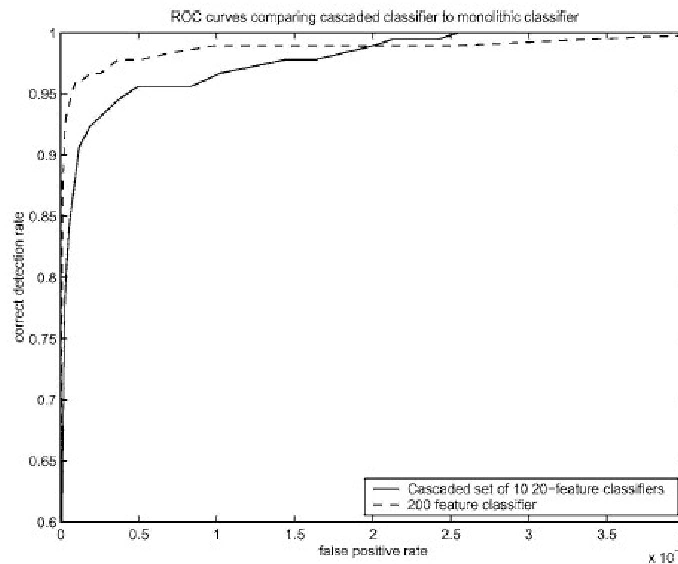


Figura 3.7: *Curva Roc Comparando un clasificador de 200 características respecto a un clasificador en cascada 10 etapas con 20 características cada una. [Viola, 2004].*

3.1.6 Diseño de Cascada de clasificadores

El proceso de diseño de la cascada de clasificadores debe considerar las limitaciones a las que se ve sometido el proceso de detección. En la mayoría de los casos, los clasificadores construidos utilizando más características tendrán una mayor tasa de detección y una menor tasa de falsos positivos. Al mismo tiempo, los clasificadores con más características necesitarán más tiempo para determinar si una ventana contiene o no una cara. A la hora de entrenar el clasificador se debe optimizar el número de etapas del clasificador, el número de características y el umbral de cada etapa. Sin embargo, realizar esta optimización es un trabajo tremendamente costoso que constituye lo que se denomina entrenamiento de la cascada de clasificadores.

Para simplificar el coste computacional de este proceso Viola y Jones idearon un algoritmo para poder entrenar de manera efectiva la cascada de clasificadores. En este caso el usuario elige las tasas de falsos positivos y de detección de cada etapa así como la tasa de falsos positivos referente a todo el detector. Cada etapa del detector se entrena utilizando AdaBoost del modo descrito en el apartado 3.1.3, incrementando el número de características de la etapa hasta que se obtengan las tasas de falsos positivos y de

detección deseadas. Dichas tasas se determinan confrontando el detector con un conjunto de validación. Si la tasa de falsos positivos global no es la que interesa se añade otra etapa al clasificador.

Los valores que determina el mejor diseño de una cascada son los siguientes índices:

- Tasa de falsos positivos (FAR - *False Acceptance Rate*), que mide la cantidad de objetos que no son caras y que son detectados como si lo fueran.
- Tasa de falsos negativos (FRR - *False Rejection Rate*), que mide la cantidad de caras que no son detectadas como tal.
- Tasa de detección: DR - *Detection Rate*), que mide el porcentaje de detecciones respecto al total de caras.

En una cascada de K etapas, el FAR total (F) de la cascada, y el DR total (D) de la cascada están dados por las ecuaciones 3.15 y 3.16.

$$F = \prod_{i=1}^{K} f_i \tag{3.15}$$

$$D = \prod_{i=1}^{K} d_i \tag{3.16}$$

Donde f_i y d_i es el FAR y el DR de cada etapa de la cascada respectivamente, es decir, de cada clasificador fuerte.

En función a estos índices, Viola y Jones plantean el entrenamiento de la cascada. El mismo recibe como parámetros: el máximo FAR aceptable por etapa (f), el mínimo DR aceptable por etapa (d), el FAR general de la cascada (F_{target}), el conjunto de muestras positivas (P), el conjunto de muestras negativas (N) y el conjunto de muestras de validación (V), compuesto por muestras positivas y negativas.

Cada una de las etapas de la cascada es entrenada utilizando Adaboost. Así, por ejemplo, en la primera etapa se tiene el primer clasificador fuerte compuesto por dos características (dos clasificadores débiles). Este clasificador se entrena para obtener los siguientes resultados: se ajusta el umbral para obtener 100% de detecciones y un índice de falsos positivos del 40%. Si una ventana se da por buena en una etapa pasa a la segunda y así sucesivamente hasta llegar a la última etapa en la que se determina si es o no un rostro. En este sentido, las primeras etapas del clasificador se diseñan de manera más simple para que puedan rechazar rápidamente el mayor número de ventanas en las que no existe probabilidad de que contengan rostros, dejando las etapas más complejas de la cascada, con mayor gasto computacional, para aquellas ventanas donde hay más probabilidad de que exista un rostro.

Por tanto, para el diseño de la cascada es necesario fijar las metas de detección y desempeño. Lo usual sería lograr niveles de detección entre el 85% y 95%, con tasa de falso positivos muy bajas, del orden de 10^{-5} o 10^{-6}. La manera de lograr esto es jugando con el umbral del perceptrón generado por Adaboost.

La estructura final del clasificador ideado por Viola-Jones consta de 32 etapas en cascada y un total de 4.297 características. La primera etapa del clasificador está construida con 2 características y rechaza el 60% de las no caras mientras alcanza prácticamente un 100% de detecciones. La segunda etapa está construida con 5 características y rechaza el 80% de las no caras mientras alcanza un 100% de detecciones. Después siguen 3 etapas de 20 características cada una, dos etapas de 50 características cada una, 5 etapas de 100 y 20 de 200 características. La cantidad de características en cada etapa fue conseguida mediante pruebas, ensayo y error, y después se fueron agregando etapas hasta conseguir el desempeño deseado. Para los escalados y desplazamientos de la ventana de barrido se han obtenido buenos resultados para S= 1,25 y A = 1 ó 2 pixeles, donde S es el factor de escala de la ventana de desplazamiento y A el paso de desplazamiento de dicha ventana [Viola, 2004].

En la publicación de Viola-Jones se muestran resultados utilizando tanto A = 1, como A = 1,5, utilizando escalas de 1 y 1,25 respectivamente (ver Figura 3.8).

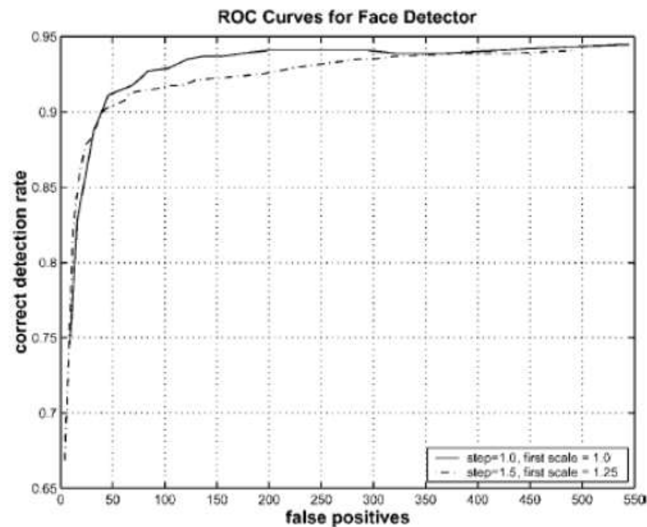


Figura 3.8: *Curvas ROC para el detector de Viola-Jones para diferentes escalas y desplazamiento de la ventana de detección. [Viola, 2004].*

3.1.7 Complejidad temporal del algoritmo de Viola-Jones

El entrenamiento de un detector facial basado en clasificadores en cascada como el que se define en el algoritmo de Viola-Jones es computacionalmente muy costoso, ya que requiere de un entrenamiento de los clasificadores que puede durar semanas en sistemas con una sola CPU. Así por ejemplo, el entrenamiento de un solo clasificador débil (una característica) puede suponer varios minutos de computo.

Se denomina complejidad temporal de un algoritmo a la función T(N) que mide el número de instrucciones realizadas por el mismo para procesar los N elementos de entrada. Cada una de estas instrucciones tiene asociado un costo temporal. Las funciones de N pueden ser de diferente tipo:

- Funciones constantes: f(N) = 5.
- Funciones logarítmicas: f(N) = log (N), o bien g(N) = n × log(N).
- Funciones polinomiales: $f(N) = 2\ N^2$, o bien $g(N) = 8\ N^2 + 5\ N$.
- Funciones exponenciales: $f(N) = 2^N$, o bien $g(N) = 2^{5N}$.
- Mezclas de las anteriores, o cualquier función de n.

En general, a medida que aumenta N, las exponenciales son mayores que las polinomiales; a su vez éstas son mayores que las logarítmicas.

Para clasificar funciones se define la función O. Se dice que T(N) es $O(N^i)$, si existen C y K tales que: $T(K) <= CK^i$.

En el caso del entrenamiento del algoritmo de Viola-Jones en términos de etapas de alto nivel de cómputo, en el figura 3.9, se puede ver el diagrama de bloques que sigue el algoritmo para su entrenamiento. El diagrama muestra que se trata de un flujo secuencial, ya que el conjunto de clasificadores que constituye una etapa se construye lentamente con cada clasificador débil.

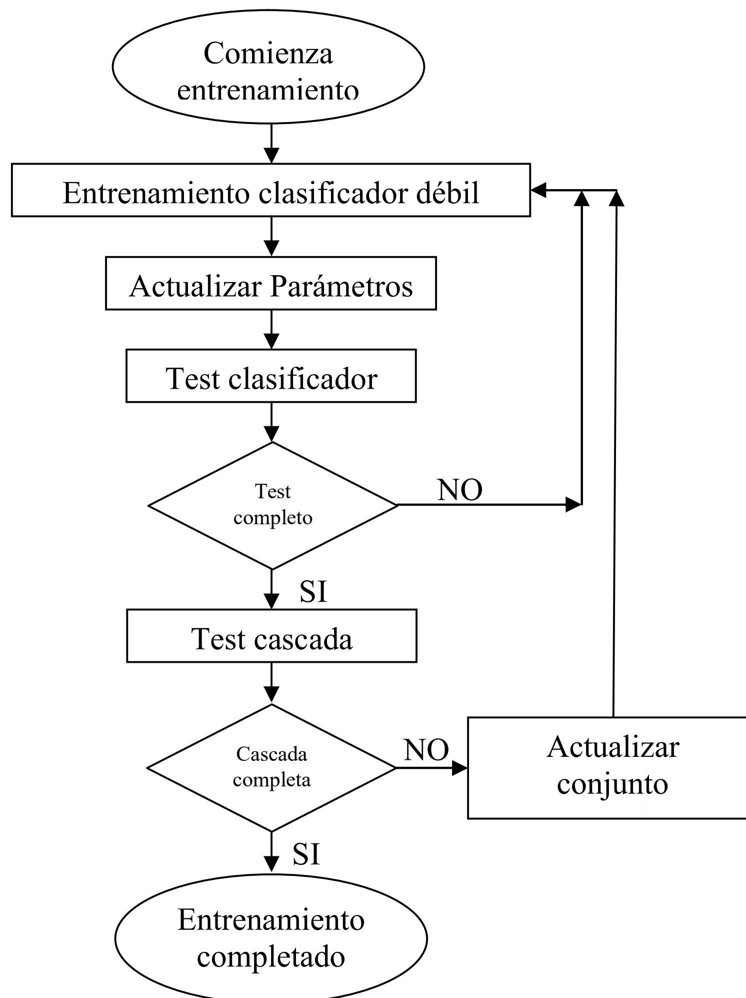


Figura 3.9: *Diagrama de bloques para el entrenamiento del algoritmo de Viola-Jones*

[Fuente: elaboración propia].

En la tabla 3.1, se muestra la complejidad temporal para el entrenamiento del método de Viola- Jones, en función de la descripción de Adaboost vista en el apartado 3.1.4.

ETAPA DE CÓMPUTO	COMPLEJIDAD TEMPORAL
Entrenamiento Clasificador Débil	$O(W \times N \times \text{Log } N)$
Actualizar Parámetros	$O(N)$
Test clasificador	$O(V^p \times \text{Log } V^p + V^n)$

Tabla 3.1*: Complejidad temporal del entrenamiento de los clasificadores del algoritmo de Viola-Jones.*

Donde N es el tamaño del conjunto de entrenamiento, V^p es el número de ejemplos de validación positivos, V^n es el número de ejemplos de validación negativos y W es el número de clasificadores débiles para entrenar. Las rutinas de actualización de parámetros involucran operaciones O(N) para actualizar los pesos del conjunto de entrenamiento. Los test implican clasificar los ejemplos positivos del conjunto de validación, así como la iteración a través del conjunto de validación negativa, y, por lo tanto, requiere $O(V^p \times \text{Log } V^p + V^n)$ operaciones en general. Sin embargo, entrenar a cada clasificador débil implica dos iteraciones sobre el conjunto de entrenamiento, así como la clasificación del conjunto de entrenamiento. Por lo tanto, este paso requiere $O(W \times N \times \text{Log } N)$ de tiempo. Como se indico anteriormente, hay cientos de miles de características en una ventana de imagen, por lo tanto, como en el entrenamiento el tamaño del conjunto crece, el entrenamiento débil del clasificador domina rápidamente el tiempo total de cálculo. Por tanto se puede concluir que la complejidad temporal del entrenamiento del algoritmo es O(W N Log N).

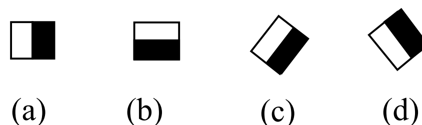
3.1.8 Mejoras del algoritmo de Viola-Jones

En este apartado se presentan diferentes avances realizados por diferentes autores, que mejoran el método de detección de rostros de Viola-Jones. Se explicará brevemente el trabajo publicado por Rainer Lienhart y Jochen Maydt [Raine, 2002],

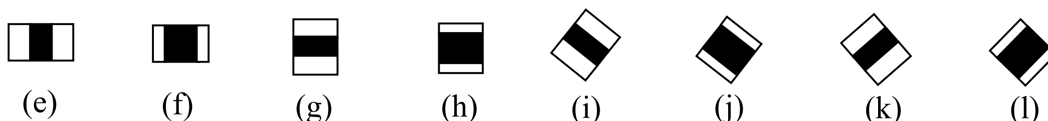
donde se extiende el algoritmo de Viola-Jones [Viola, 2001] a través de dos contribuciones principales: un conjunto de características extendido y un algoritmo de post-optimización para una cascada de clasificadores. Seguidamente se verá el algoritmo de Gentle Boost [Jerom, 2000], con el que Lienhart, Kuranov y Pisarevsky [Lienh, 2003] han realizado experimentos, obteniendo mejores resultados que con otros algoritmos de *Boosting*.

En primer lugar, Lienhart y Maydt, amplían el conjunto original de características, agregando nuevas características rotadas a 45º. Para que la evaluación de las características rotadas sea eficiente, proponen una representación análoga a la imagen integral. En la publicación se menciona que, al utilizar estos nuevas características, se obtiene para una tasa de detección (DR) dada, un 10% menos de falsos positivos (FP). El conjunto nuevo de características se puede observar en la figura 3.9.

Características de Detección de Bordes:



Características de Detección de Líneas:



Características de Detección de Centros:

Figura 3.10: *Características extendidas, utilizadas por Lienhart y Mayt. [Raine, 2002].*

Para calcular las características sin inclinación de la Figura 3.7 (a), (b), (e), (f), (g), (h) y (m), utilizan una función denominada SAT (*Summed Area Table*), que es similar a la imagen integral propuesta por Viola-Jones. Como se vio anteriormente, puede calcularse en una sola pasada por los píxeles de la imagen y está dada por la ecuación 3.17.

$$SAT(x,y) = \sum_{\substack{1\le x'\le x \\ 1\le y'\le y}} I(x',y') \tag{3.17}$$

Además agregan la función RSAT (*Rotated Summed Area Table*), que se utiliza para calcular las características a 45º de la Figura 3.7 (c), (d), (i), (j), (k), (l) y (n). Puede calcularse con dos pasadas por los píxeles de la imagen y está dada por la ecuación 3.18.

$$RSAT(x,y) = \sum_{\substack{1\le x'\le x \\ x'\le x-|y-y'|}} I(x',y') \tag{3.18}$$

Como segunda contribución, Lienhart y Maydt [Raine, 2002], proponen un procedimiento de post-optimización para una cascada, que permite mejorar el FAR de la misma un 12.5% en promedio. El mismo consiste en optimizar cada clasificador fuerte de la cascada, ajustando los umbrales de las características que forman el clasificador fuerte y buscando bajar el FAR pero sin perder el DR para el cual fue entrenado. De esta manera se logra reducir el error total del clasificador (FAR + FRR). Dado el clasificador fuerte que se expresa en la ecuación 3.19:

$$C(x) = signo\left(\sum_{t=1}^{T} \alpha_t \cdot h(x, f_t, p_t, \theta_t) + b\right) \tag{3.19}$$

Los parámetros libres son los umbrales θ_t y b, dado que los α_t deben ser elegidos de acuerdo al algoritmo Adaboost para preservar sus propiedades. Empezando por el θ_t original (el encontrado para los clasificadores débiles), se va incrementando y decrementando su valor en forma de gradiente descendente, verificando su desempeño.

En el trabajo publicado por Lienhart, Kuranov y Pisarevsky [Lienh, 2003], se exponen diversos experimentos realizados sobre tres variantes del algoritmo Adaboost, que son: Discrete Adaboost [Jerom, 2000], Real Adaboost [Jerom, 2000] y Gentle Adaboost [Jerom, 2000]. El algoritmo de *Boosting* propuesto por Viola-Jones en el apartado 3.1.3, es una variante de Discrete Adaboost. Lienhart, Kuranov y Pisarevsky

muestran empíricamente que Gentle Adaboost es más efectivo que Discrete y Real Adaboost, requiriendo una menor cantidad de características para obtener el mismo desempeño, y reduciendo el tiempo de cómputo.

Lienhart, Kuranov y Pisarevsky [Lienh, 2003], realizaron pruebas para comprobar la eficacia de los clasificadores débiles. En ellas muestran la superioridad de los clasificadores que emplean las características extendidas (ver figura 3.7) sobre los propuestos originalmente por Viola-Jones (ver figura 3.1). Con el conjunto de características extendidas, se obtienen en promedio, un 10% menos de falsos positivos. Por otro lado, se demuestra, en forma empírica, que la detección mejora considerablemente cuando se utilizan como clasificadores débiles pequeños árboles de decisión denominados CART (*Classification and Regression Trees*) [Johnm, 2009] en lugar de los clasificadores débiles o árboles de un solo nodo utilizados por Viola-Jones, también denominados *stumps*. Las velocidades de detección que se obtienen son similares, dado que con una mayor cantidad de nodos en los CARTS se requieren menos clasificadores débiles para alcanzar el performance requerido en cada etapa. Según los autores, los clasificadores débiles compuestos por *stumps*, no permiten aprender las dependencias existentes entre distintas características.

3.2 Conclusiones

En este capítulo se ha presentado el algoritmo para la detección de rostros en tiempo real propuesto por Viola y Jones en 2001, explicando cuáles son sus tres contribuciones principales:

1. Se introduce la imagen integral como una representación de la imagen original que permite calcular fácilmente las características tipo Haar de la imagen.
2. Diseño de un clasificador fuerte. Se elige un conjunto de características significativas utilizando el algoritmo Adaboost. Este es utilizado tanto para seleccionar las características, como para entrenar los clasificadores.
3. Se combina una sucesión de clasificadores en cascada formando etapas cada vez más complejas, dejando así que se concentren los clasificadores más complejos

y con mayor coste computacional en las zonas donde es más probable encontrar un rostro.

Por último se han expuesto algunas mejoras de dicho algoritmos a partir de la contribuciones realizadas por Rainer Lienhart y Jochen Maydt que proponen un nuevo conjunto de características rotadas y la aplicación del algoritmo Gentle Boost, que mejora los resultados obtenidos respecto a Adaboost como algoritmo de *Boosting* para el entrenamiento del detector.

CAPÍTULO 4.

4. IMPLEMENTACIONES HARDWARE DEL ALGORITMO

Este capítulo tiene como objetivo proporcionar una visión general de las diferentes plataformas hardware utilizadas en la implementación de sistemas de visión para la detección de objetos en tiempo real [Kjaer, 2010], de manera que puedan ser entendidas las capacidades y limitaciones de cada plataforma en términos de rendimiento de velocidad de detección, disipación de potencia y tamaño del sistema, y particularizándolo para el algoritmo de detección facial de Viola-Jones. Con ello se pretende justificar y enmarcar las plataformas hardware de detección que se van a usar en la tesis doctoral.

En este sentido, las plataformas que se van a analizar son la Unidad de procesamiento central (CPU) multi-núcleo, el procesador digital de señales (DSP), la unidad de procesamiento de gráficos (GPU), la matriz de puertas programables (FPGA) y los circuitos integrados de aplicación específica (ASIC). Sin embargo, debe tenerse en cuenta que la elección de la plataforma también depende del escenario de la aplicación, del algoritmo utilizado y de sus limitaciones, así como del coste y del tiempo de desarrollo.

4.1 Plataformas hardware para implementar el algoritmo de Viola-Jones

Por la forma en que se afronta el problema de detección de rostros y por la relativa alta velocidad de ejecución del algoritmo propuesto por Viola-Jones en comparación con otras propuestas, se ha podido encontrar en la bibliografía consultada muchos autores que han abordado la implementación y aceleración del mismo.

L. Hung-Chih et al. En [Hungc, 2007] utilizan un módulo hardware para implementar una pirámide de imágenes que le permita realizar el barrido en distintas resoluciones. Luego utilizan un buffer para calcular la imagen integral de la ventana que están procesando y a continuación realizar el proceso de cálculo de los clasificadores. Los autores utilizan una placa que contiene una FPGA Virtex-II Pro XC2VP30 e implementan 52 clasificadores en una sola etapa. Las imágenes donde se realiza la detección deben tener un tamaño fijo de 640×480 puntos y el conjunto de los clasificadores no puede ser modificado.

G. Changjian y L. Shih-Lien en [Chang, 2008] utilizaron una tarjeta que contaba con un conector PCIe y un FPGA Virtex 5 LX110T de Xilinx. En conjunto con un software que se ejecuta en un procesador Intel Core 2 Duo a 2.66 GHz con 8 GB de memoria RAM los autores reportan una velocidad de detección de 37 imágenes por segundo. En este trabajo las imágenes deben ser grabadas en la FPGA como paso previo a la detección y su tamaño de 256×192 puntos no puede variar.

J. Cho, B. Benson, S. Mirzaei y R. Kastner en [Chobe, 2009] utilizan una FPGA de Xilinx Virtex-5 LX330 y en ella implementan la interfaz para capturar las imágenes directamente de un sensor de visión. Como el cálculo de los clasificadores se realiza utilizando una ventana de tamaño fijo, los autores implementan el hardware necesario para disminuir progresivamente este tamaño de la imagen para poder realizar el barrido de la misma. Al realizar esta implementación, se fijan los factores de escala por los cuales será modificada la imagen. En imágenes de 320×240 puntos se realiza el barrido en 14 escalas pre-fijadas que no se pueden modificar. En el trabajo se reporta el procesamiento de 61,02 imágenes por segundo y en comparación con una realización

software equivalente los autores logran acelerar el proceso de detección. Una de las desventajas de esta implementación es que el tamaño de las imágenes a procesar es fijo y un cambio en el tamaño de las mismas representaría un cambio en todo el sistema.

4.1.1 Plataformas hardware para la detección de objetos

El sistema de visión para la detección de rostros que interesa implementar en este trabajo debe cumplir una serie de requisitos: fiabilidad, rendimiento en tiempo real, bajo coste, tamaño pequeño, bajo consumo de energía y flexibilidad. El coste, el consumo de energía y las restricciones espaciales dificultan aún más el cumplimiento de los demás requisitos, que también son importantes. La flexibilidad es un tema importante a tener en cuenta durante el diseño de la arquitectura. Una implementación flexible del sistema de detección debería poder actualizarse fácilmente para corregir errores detectados. De lo contrario, sería necesario reemplazar todo el hardware, lo que implica mayores costes de mantenimiento. No existe una fórmula ideal, y aunque el bajo coste es una prioridad en el mercado actual altamente competitivo, el resto de los requisitos de diseño también deben considerarse. En este apartado se revisa estos requisitos para comprender mejor la decisión de diseño tomada al elegir una plataforma de implementación de hardware y software [Chris, 2014].

4.1.1.1 CPU Multi-núcleo

Las CPU multi-núcleo han sido la tendencia de las arquitecturas principales de CPU en las últimas décadas en un intento de encontrar otras formas de mejorar el rendimiento principalmente debido a tres factores limitantes (consumo de energía, latencia de memoria y retardos de cableado, y límites de paralelismo de nivel de instrucción) que han estancado la progresión de las arquitecturas de CPU individuales. Como su nombre sugiere, en lugar de tener un solo núcleo con muy alta frecuencia, se reemplaza con dos o más núcleos más simples que procesan diferentes hilos (una pequeña secuencia de instrucciones del programa). Las arquitecturas de CPU multi-núcleo han sido la plataforma de procesamiento tradicional para la implementación de algoritmos de visión por computadora ya que ofrecen alta flexibilidad, facilidad de uso

y tiempos de desarrollo rápidos. Los sistemas multi-núcleo recientes consisten en núcleos de CPU físicos con frecuencias operativas del orden de GHz. También tienen soporte de coma flotante y soporte para el procesamiento de vectores a través de instrucciones SIMD (Single Instruction, Multiple Data), ver figura 4.1.

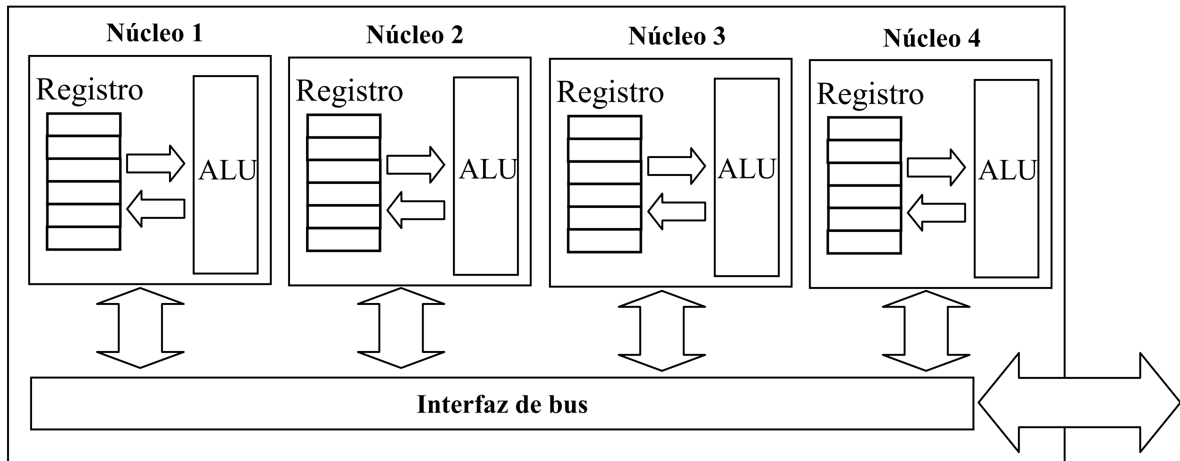


Figura 4.1: *Arquitectura CPU multi-núcleo formada por 4 núcleos [Fuente: elaboración propia].*

Sin embargo, el paralelismo y el rendimiento alcanzable no es proporcional al número de núcleos, que son menores en comparación con otras plataformas, estableciendo un límite para el rendimiento máximo alcanzable [Kjaer, 2010]. Además, los sistemas de visión existentes desarrollados usando CPUs de alta gama que pueden proporcionar velocidad de video en tiempo real requieren la utilización de todos los núcleos y, por lo tanto, exhiben un alto consumo de energía. La reducción del consumo de energía requiere la reducción de la frecuencia o la reducción de los recursos de procesamiento, que terminan por reducir el rendimiento. Por lo tanto, los sistemas de CPU multi-núcleo se han utilizado tradicionalmente para sistemas de seguridad y vigilancia en computadoras de escritorio donde el consumo de energía no es una restricción clave, en lugar de los entornos integrados. Pero en la actualidad, con la aparición de las computadoras de placa simple (SBC), de bajo coste y alto rendimiento, como las placas Raspberry Pi, Odroid XU4, etc. Se puede decir que este tipo de dispositivos cumplen con los requisitos propuestos para el sistema de detección. Por tanto, puede ser una alternativa viable como plataforma hardware para la implementación del algoritmo de Viola-Jones. En la tabla 4.1, se puede ver un resumen

comparativo de algunos trabajos relacionados con la implementación del algoritmo de Viola-Jones haciendo uso de CPU multi-núcleo.

Autor	CPU Multi-núcleo	Resolución imagen (pixel)	Tiempo de ejecución (s)	Implementación SW	Precisión de detección	Año
[Abyan, 2011]	Dual core ARM Cortex A8	320×240	0,95	OpenCV	95%	2011
[Acasa, 2011]	Dual-Core CPU T4300,	640×480	0,46	OpenCV	95%	2011
[Cheng, 2011]	8 Core ARM V5	720×576	0,7	OpenCV	95%	2011
[Hsuan, 2012]	8 Core ARM V5	512×512	0,376	OpenCV	93%	2012
[Young, 2016]	Quad-core Krait 400 CPU	720×576	0,253	OpenGL ES	-	2016
[Ranja, 2017]	CPU 4 core Intel Core i7-2655LE	640×480	0,2	OpenCV	95%	2017
[Shife, 2018]	E5-2660v3@2.60	512×512	0,05	CNN	96%	2018

Tabla 4.1*: Implementaciones del Algoritmo de Viola-Jones en CPU multi-núcleo.*

En [Abyan, 2011] se presenta un trabajo sobre la optimización del rendimiento de los algoritmos generales de visión de computadora, en sistemas integrados con recursos limitados. En artículo de [Acasa, 2011] se propone una implementación optimizada de la velocidad del algoritmo de detección de rostros Viola-Jones basado en el uso de OpenCV. [Cheng, 2011] analiza el comportamiento de la memoria de un algoritmo paralelo Viola-Jones y propone un esquema para mejorar localización de datos de la memoria caché. [Hsuan, 2012] realiza un análisis exhaustivo del paralelismo de un algoritmo de detección de rostros en diferentes niveles algorítmicos. Propone un esquema de paralelización de niveles mixtos de varias etapas para mantener la escalabilidad del rendimiento y evitar los factores limitantes. En [Young, 2016] el objetivo de la investigación es acelerar el algoritmo de detección de rostros Viola-Jones, para ello el método propuesto en este estudio adapta el paralelismo de tareas CPU-GPU, el paralelismo de ventanas deslizantes, el paralelismo de imágenes a escala, la asignación dinámica de subprocesos y la optimización de la memoria local para mejorar el tiempo de cálculo del sistema de detección. En [Ranja, 2017] se hacen tres contribuciones clave. La primera es la presentación de un algoritmo altamente optimizado de detección y seguimiento de caras en serie que utiliza la estimación de movimiento y las ventanas de búsqueda local para lograr tasas de procesamiento rápidas. La segunda es la redefinición del proceso de detección de caras basado en un conjunto de escalas de caras independientes que se pueden procesar en paralelo en núcleos de CPU separados y al mismo tiempo lograr una tasa de procesamiento objetivo. La tercera contribución es la demostración de cómo se pueden usar múltiples núcleos para acelerar el proceso de seguimiento de caras, lo que proporciona aumentos de velocidad significativos

cuando se realiza el seguimiento de un gran número de caras simultáneamente. Finalmente, en [Shife, 2018] se propone un nuevo detector de rostro, llamado FaceBoxes, con Rendimiento superior tanto en velocidad como en precisión que permite un procesamiento en tiempo real y está basado en CNN.

En la actualidad gracias a la enorme evolución de las tecnologías aplicadas a los teléfonos móviles, las arquitecturas ARM lideran claramente el mercado de CPU multi-núcleo. Algunos algoritmos simples pueden integrarse completamente en un microprocesador a través de APP disponibles para su descarga en teléfonos inteligentes. Sin embargo, los algoritmos más complejos generalmente necesitan aceleración de hardware y software adicional.

4.1.1.2 Procesador digital de señal (DSP)

Un procesador digital de señal (DSP) es similar a un procesador de propósito general en el sentido de que también tiene una lógica fija, puede ejecutar un número finito de tipos de instrucción y las instrucciones tienen un flujo secuencial. Sin embargo, la distinción principal de las CPU de propósito general es que su ISA (Industry Standard Architecture), arquitectura de bus para poder conectar tarjetas de expansión, está optimizado para operaciones matriciales, particularmente la multiplicación y acumulación que es la operación más común en aplicaciones de procesamiento de señales, y esperan un flujo de programa lineal con declaraciones condicionales infrecuentes donde un gran la cantidad de datos debe procesarse con el mismo programa / operaciones matemáticos. Por este motivo, proporcionan instrucciones SIMD (instrucción única, datos múltiples) para explotar el paralelismo ejecutando la misma instrucción en múltiples flujos de datos, así como instrucciones VLIW (Palabra de instrucción muy larga) para procesar diferentes instrucciones con diferentes datos [Brani, 2009].

Los DSP de alto rendimiento a menudo empaquetan múltiples núcleos de procesamiento con una estructura de CPU de propósito general. También incluyen unidades DMA (Direct Memory Access) y unidades de E / S dedicadas para un acceso rápido a la memoria fuera del chip. Los DSP son una plataforma atractiva para sistemas

de visión integrados, que ofrecen programabilidad, bajo costo y baja potencia, y paralelismo en forma de SIMD, VLIW o ambos. Sin embargo, la cantidad fija de recursos de procesamiento en un DSP acoplado con frecuencias más bajas que las CPU multi-núcleo, puede ser el cuello de botella en el rendimiento más alto alcanzable.

Tradicionalmente, los DSP han sido la primera opción en aplicaciones de procesamiento de imágenes. Los DSP ofrecen operaciones de multiplicación y acumulación de ciclo único, además de capacidades de procesamiento en paralelo y bloques de memoria integrados. Hay muchos ejemplos en la literatura de implementaciones de visión por computadora en DSP, como en [Kjaer, 2010].

La TI C6000 DSP [Ronen, 2001], que es una de las plataformas de visión integradas programables más utilizadas, ofrece un buen rendimiento general en el procesamiento de bajo, medio y alto nivel. Otra opción interesante es TDA2x [Jerem, 2012]. Esta familia de SoC incorporan una arquitectura heterogénea y escalable que incluye una combinación de núcleos DSP, aceleradores de visión, procesadores ARM Cortex-A15 MPCore y dual Cortex-M4.

Los DSP son muy atractivos para aplicaciones de visión integradas, ya que ofrecen una buena relación precio/rendimiento. Sin embargo, requieren un coste mayor en comparación con otras opciones como las FPGA, y no son tan fáciles y rápidos de programar como los microprocesadores. En la tabla 4.2, se puede ver algunos trabajos relacionados con la implementación del algoritmo de Viola-Jones haciendo uso de DSP.

Autor	**DSP**	**Resolución imagen (pixel)**	**Tiempo de ejecución (s)**	**Implementación SW**	**Precisión de detección**	**Año**
[Jianf, 2008]	C6400	40×40	0,94	OpenCV	99%	2008
[Binko, 2011]	DM642	720×576	0,22	VC6.0	80%	2011

Tabla 4.2: *Implementaciones del Algoritmo de Viola-Jones en DSP.*

En [Jianf, 2008] se introduce una implementación optimizada del algoritmo de detección de rostro de Viola-Jones, utilizando un DSP. Se aplica el algoritmo AdaBoost para detectar rostros humanos y se utilizan algunos métodos para optimizar los datos de la imagen original, como la escala de la imagen, el filtro de la mediana, la ecualización del histograma y la detección de bordes. [Binko, 2011] En este artículo se explica cómo lograr la implementación del algoritmo de Viola-Jones en tiempo real mediante software utilizando un DSP que tiene una capacidad de procesamiento y memoria relativamente limitados. Se discuten varias técnicas de optimización y se presenta un ejemplo de resultado de implementación en una plataforma móvil real.

4.1.1.3 Unidad de Procesamiento de Gráficos (GPU)

Las unidades de procesamiento de gráficos (GPU) se ocupan del procesamiento de información para producir una imagen de una escena (compuesta por objetos conocidos a diferentes escalas, orientaciones, tamaños, distancias, colores, formas, etc.). Esto es lo contrario a la visión por computadora, donde el objetivo es construir una descripción de una escena a partir de una imagen de entrada. Recientemente, las GPU (ver figura 4.2) han evolucionado de un co-procesador gráfico dedicado a una CPU, a un motor de cálculo programable masivamente paralelo. Las GPU emplean redes de núcleos de procesamiento de flujos programables masivamente paralelos [Owens, 2008] que son eficientes para computación de alto rendimiento, mientras que usan hardware de función fija adicional para el procesamiento de gráficos. Por lo tanto, desde muy pronto ha habido un interés masivo en la utilización de GPU para aplicaciones informáticas de uso general [Owens, 2008], usando el estilo de programación del lenguaje C y compiladores como el de la plataforma CUDA (Arquitectura Unificada de Dispositivos de Cómputo) de NVIDIA.

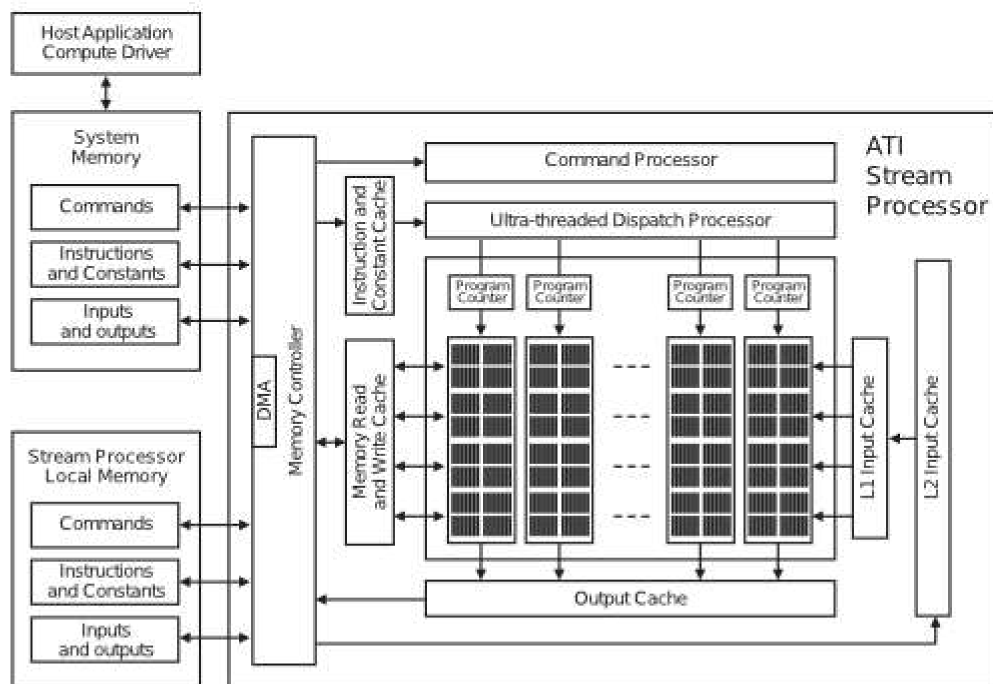


Figura 4.2: *Diagrama de bloques de una GPU AMD FireStream [Harre, 2009].*

La arquitectura de una GPU es drásticamente diferente de la de una CPU, ya que los transistores se usan para unidades de procesamiento computacional en lugar de cachés y predicción de bifurcación, y su arquitectura está optimizada para un alto rendimiento en lugar de una baja latencia. En consecuencia, la GPU ofrece mayor rendimiento y es ampliamente considerada como el motor computacional para el futuro. Las GPU también tienen una jerarquía de memoria diferente que tiene como objetivo proporcionar un gran ancho de banda a través de buses de memoria amplios y memoria gráfica especializada. Además, las GPU emplean pequeñas cachés de solo lectura que ayudan a reducir los requisitos de ancho de banda en la memoria principal y se benefician de pequeñas cachés que capturan la ubicación espacial.

A medida que aumenta el número de procesadores programables, se utilizan cantidades significativas de almacenamiento en chip para mantener el contexto de ejecución, los datos de transmisión y los datos temporales. Las GPU brindan una potencia bruta para tareas de cómputo intensivo que requieren un rendimiento en tiempo real, como en aplicaciones de visión. Sin embargo, hay que hacer concesiones entre la potencia y el rendimiento, ya que las GPU de gama alta tienden a tener un alto consumo

de energía (del orden de cientos de vatios [Jerem, 2012]). Otro inconveniente potencial para el uso de la tecnología GPU en entornos integrados es el hecho de que, a pesar de que las capacidades de programación de las GPU han mejorado dramáticamente en los últimos años, la depuración sigue siendo una tarea desafiante [Crist, 2010]. Finalmente, las transferencias de datos necesarias entre la GPU y la CPU del host aumentan la latencia de la aplicación [Crist, 2010], algo que a menudo no se considera en las implementaciones de aplicaciones de visión integradas en GPU. Por otro lado, las GPU integrada (eGPU) requieren menos demandas en términos de consumo de energía, pero no ofrecen las mismas características de programación y tienen menos recursos de procesamiento y memoria. En la tabla 4.3, se puede ver un resumen comparativo de algunos trabajos relacionados con la implementación del algoritmo de Viola-Jones haciendo uso de GPU.

Autor	GPU	Resolución imagen (pixel)	Tiempo de ejecución (s)	Implementación SW	Precisión de detección	Año
[Haipe, 2012]	NVIDIA Tesla C2050	500×500	0,25	OpenCL	95%	2012
[Masek, 2013]	NVIDIA GeF GTX 690	24×24	0,4	CUDA	95%	2013
[Wongy, 2015]	NVida C2075	320×240	0.31	CUDA	95%	2015
[Chouc, 2015]	NVIDIA GeForce 310 M	512×512	0,51	CUDA	95%	2015
[Bhati, 2016]	Nvidia GeForce 920m	320×240	0,15	CUDA	95%	2016
[Savat, 2018]	Nvidia Jetson	250×250	0,23	OPenCL	97%	2018

Tabla 4.3: *Implementaciones del Algoritmo de Viola-Jones en GPU.*

El trabajo de [Haipe, 2012] presenta una implementación OpenCL del algoritmo de detección facial Viola-Jones de alto rendimiento en una GPU NVIDIA, a través de varias técnicas: granularidad del sistema, hilos persistentes, Uberkernel y colas locales y globales. En [Masek, 2013] se propone una implementación CUDA multi-GPU de entrenamiento para la detección de objetos usando el algoritmo Viola-Jones, consiguiendo acelerar dos de las operaciones más lentas en el proceso de entrenamiento, a través de dos NVIDIA GeForce GTX 690 de doble núcleo. En [Wongy, 2015] se acelera con una GPU un sistema de detección de rostros que utiliza el algoritmo Viola-Jones utilizando OpenCV y CUDA. Los resultados de los experimentos muestran que la aceleración de la GPU propuesta para la detección de rostros es capaz de alcanzar una

velocidad de hasta 18 veces superior, en comparación con el algoritmo de la versión con una CPU convencional y manteniendo la precisión de detección. En el trabajo de [Chouc, 2015] se desarrolla una implementación de detección de rostros en tiempo real basada en aceleración de la GPU utilizando CUDA. La detección de rostros se realiza adaptando el algoritmo de Viola y Jones. En [Bhati, 2016] se desarrolla un trabajo cuyo objetivo principal es aumentar la velocidad computacional del algoritmo de Viola-Jones, que se puede lograr a través de la implementación paralela utilizando CUDA y OpenCV sobre la GPU, y luego proporciona un análisis comparativo de los mejores resultados computacionales entre la ejecución en serie y en paralelo. En el trabajo de [Savat, 2018] se propone un marco de reconocimiento facial múltiple que es implementado en el sistema de GPU integrado. Los resultados experimentales muestran que el sistema puede reconocer hasta 8 caras en tiempo real en 0,23 segundos de tiempo de procesamiento.

4.1.1.4 Circuitos Integrados Específicos (ASIC)

Los Circuitos Integrados para Aplicaciones Específicas (ASIC) son circuitos integrados (IC) personalizados para un uso particular, en lugar de para uso general. Los diseñadores de ASIC digitales suelen utilizar un lenguaje de descripción de hardware como Verilog o VHDL, para describir la funcionalidad de los ASIC.

Los ASIC tienen las ventajas de alto rendimiento y bajo consumo de energía. Se usan solo para la fabricación de gran cantidad y series largas debido a un coste de ingeniería inicial más elevado, por lo que no son adecuados para la creación rápida de prototipos. Además, tienen otro inconveniente importante: no son reconfigurables. Esto significa que una vez que se fabrican, no se pueden reprogramar. Esta falta de flexibilidad ha llevado al uso de otras alternativas como las matrices de puertas programables (FPGA). Sin embargo, aún se pueden encontrar en la literatura algunos ejemplos de implementaciones de sistemas de visión para la detección de objetos en ASIC [Mielke, 2011]. Esta tecnología también fue utilizada por Mobileye para construir sus productos EyeQ [Stein, 2005], que se componen de núcleos duales de CPU que se ejecutan en paralelo con múltiples núcleos adicionales dedicados y programables. En la

tabla 4.4, se puede ver un resumen de algunos trabajos relacionados con la implementación del algoritmo de Viola-Jones en ASIC.

Autor	ASIC	Resolución imagen (pixel)	Tiempo de ejecución (s)	Implementación SW	Precisión de detección	Año
[Zhouy, 2011]	ASIC	320×240	0,6	-	95%	2011
[Kimab, 2017]	ASIC	650×500	0,14	VHDL	81%	2017

Tabla 4.4: *Ejemplo de implementaciones del Algoritmo de Viola-Jones en ASIC.*

El artículo de [Zhouy, 2011] propone una arquitectura de detección de rostros, que los usuarios podrían ajustar según el entorno, la resolución del sensor, los diferentes requisitos de precisión y velocidad de detección. Este modo ajustable por el usuario hace que la reconfiguración sea muy simple y eficiente. En [Kimab, 2017] se presenta un acelerador de detección facial a partir de 2.000 clasificadores simples que se combinan de forma adaptativa para hacer una clasificación muy sólida.

4.1.1.5 Matriz de Puertas Programable (FPGA)

Las llamadas matrices de puertas programable (FPGA), como se muestra en la Figura 4.3, son un tipo de circuito integrado reconfigurable formado por bloques lógicos configurables (CLB) que consisten en tablas de búsqueda (LUT) que están programadas para almacenar salidas para todas las combinaciones de entrada, así como otras lógicas como registros y compuertas lógicas [Ronsa, 2010]. Los CLB están interconectados a través de una red de interconexión de conmutadores programables y los bloques de E / S están disponibles en el perímetro para permitir la comunicación con dispositivos externos. Además, las FPGA modernas también vienen equipadas con bloques de procesamiento integrados como procesadores integrados, unidades MAC y memoria integrada. Todos estos componentes hacen que las FPGA sean un circuito integrado totalmente programable que el diseñador puede usar para desarrollar cualquier circuito digital deseado y, al mismo tiempo, aprovechar los recursos dedicados disponibles.

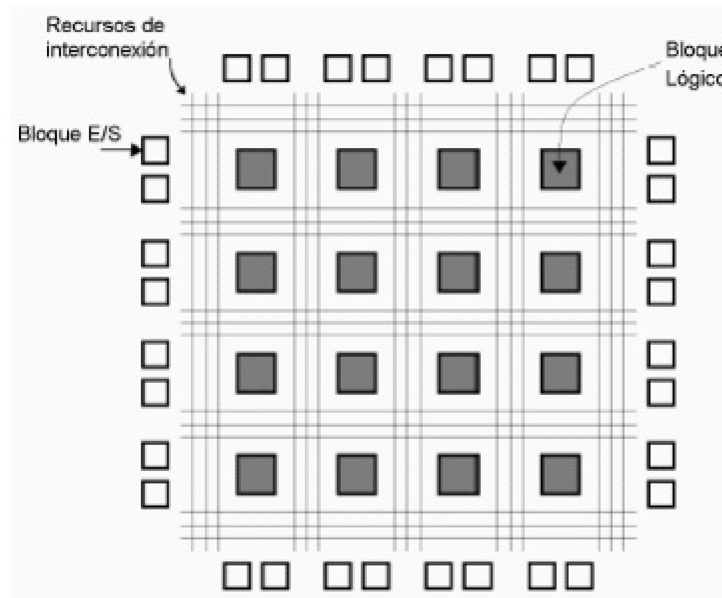


Figura 4.3: *Elementos básicos que constituyen una FPGA [Capot, 2016].*

Un circuito digital (arquitectura de hardware) se configura en la FPGA usando un lenguaje de descripción de hardware (HDL) que especifica su comportamiento y funcionalidad. El gran beneficio de las FPGA en comparación con las plataformas informáticas mencionadas anteriormente es que el circuito de aplicación se puede adaptar a las demandas de la aplicación con respecto a cómo se realizará el procesamiento, el flujo de datos, el control y la sincronización de varios componentes y la interfaz de entradas/salidas. Por lo tanto, las FPGA ofrecen una gran cantidad de recursos paralelos que pueden explotarse con la arquitectura hardware apropiada para proporcionar el rendimiento necesario, y dado que pueden reconfigurarse también proporcionan un alto grado de flexibilidad. Además, las FPGA ofrecen un rendimiento determinista y una solución completa, ya que un sistema completo junto con los periféricos se pueden integrar en la mismo FPGA con un número fijo de ciclos necesarios para llevar a cabo las evaluaciones. Sin embargo, la desventaja es que el diseño con FPGA requiere largos ciclos de desarrollo. Por lo tanto, las FPGA se utilizan principalmente como plataformas de prototipos para futuros aceleradores específicos de aplicaciones potenciales para aplicaciones de visión. Además, estos dispositivos a menudo tienen frecuencias más bajas que otras plataformas debido a la compleja red de interconexión fija que da como resultado trayectos de retardo más largos. Sin embargo, las bajas frecuencias pueden ser compensadas mediante el diseño

de una arquitectura altamente paralelizada. Como tal, existe un esfuerzo creciente de investigación para diseñar arquitecturas de hardware dedicadas para aplicaciones de visión con el fin de proporcionar un buen rendimiento en tiempo real junto con el menor consumo de potencia de las FPGA.

En este sentido las opciones más competitivas en FPGA son las series Intel Stratix [Strat, 2019] que combinan alta densidad y alto rendimiento con una rica función para habilitar más acciones y maximizar el ancho de banda del sistema, y la familia Xilinx Zynq-7000 [Xilin, 2018], que es capaz de arrancar independientemente de la lógica programable. Esta característica tiene una serie de ventajas, pero también significa que, desde el punto de vista de un ingeniero de software, el dispositivo Zynq-7000 se comporta como un procesador multi-núcleo de propósito general. Sin embargo, sólo vale la pena si realmente necesita la parte lógica programable. De lo contrario, es mucho más barato usar un microprocesador estándar. En la tabla 4.5, se puede ver un resumen de algunos trabajos relacionados con la implementación del algoritmo de Viola-Jones en FPGA.

Autor	FPGA	Resolución imagen (pixel)	Tiempo de ejecución (s)	Implementación SW	Precisión de detección	Año
[Chobr, 2009]	Xilinx Virtex-5	640×480	0,26	Verilog HDL	90%	2009
[Matai, 2011]	Xilinx Virtex-5	320×240	0,22	VHDL	90%	2011
[Dasat, 2012]	XilinxVirtex 5	640×480	0,17	VHDL	91,3%	2013
[Kimse, 2013]	Xilinx Virtex-5	320×240	0,24	VHDL	96%	2013
[Sused, 2015]	Xilinx Virtex-7	320×240	0,28	VHDL	95%	2015
[Archi, 2018]	Nexys 4 Artix-7	640×480	0,2	VHDL	96%	2018

***Tabla 4.5**: Implementaciones del Algoritmo de Viola-Jones en FPGA.*

El trabajo de [Chobr, 2009] presenta una arquitectura paralelizada para la aceleración por hardware de la detección de rostros diseñado con Verilog HDL e implementado sobre una FPGA Xilinx Virtex-5. Su desempeño ha sido medido y comparado con la implementación del software de detección de rostros de Viola y Jones. En [Matai, 2011] se propone un completo sistema de reconocimiento facial en tiempo real que consiste en un módulo de detección facial, un módulo de

reconocimiento y un muestreo descendente que utiliza un FPGA Xilinx Virtex-5. En el artículo [Dasat, 2012] se presenta una arquitectura hardware modificada con características clave para disminuir el uso de recursos de la FPGA y elevar la velocidad de las ventanas de detección de caras. El sistema se basa en el algoritmo de Viola Jones. El sistema esta implementado sobre una FPGA Xilinx Virtex-5, y produce una alta tasa de detección facial. El trabajo de [Kimse, 2013] desarrolla una arquitectura hardware eficiente para un sistema de detección de rostros en tiempo real que utiliza una FPGA Xilinx Virtex-5. Los resultados experimentales muestran que el tiempo de procesamiento para una imagen de 320×240 píxeles es de 42 cuadros por segundo con 100MHz. En [Sused, 2015] se presenta una nueva arquitectura hardware para la detección de patrones y la clasificación específica para la detección de rostro humano, que incluye la adquisición de imágenes en bruto, la creación de imágenes integrales, la extracción de ventanas, la generación de pirámides y los algoritmos de detección en pasos simultáneos, todo implementado sobre una FPGA Xilinx Virtex-7. El artículo de [Archi, 2018] presenta una síntesis del algoritmo de detección facial de Viola-Jones sobre una FPGA Nexys 4 Artix-7. Los resultados muestran una reducción en tiempo de ejecución y en consumo energético.

4.1.2 Hacia un sistema de visión integrado heterogéneo en CHIP

El análisis anterior de los principales conceptos y propiedades de cada plataforma informática demuestra que cada una de ellas tiene sus propias fortalezas y debilidades. Como tal, dada la naturaleza diversa de las aplicaciones de visión por computadora en términos de flujo de datos y operaciones, y la necesidad de especialización y flexibilidad, es razonable prever un sistema de visión integrado heterogéneo en un chip (SoC) [Gopal, 2013] donde las plataformas de procesamiento anteriores serán utilizadas para llevar a cabo diferentes tareas de aplicación. Por ejemplo, para una aplicación de procesamiento de visión los mecanismos de flujo de control de una CPU son adecuados para fines de supervisión y sincronización del sistema, las capacidades de procesamiento de señal de un DSP se pueden usar para manejar la transmisión y recepción de video, una GPU puede usarse para preprocesamiento de imágenes, manipulación y renderización, y una FPGA puede usarse para proporcionar un procesamiento en paralelo específico de la aplicación

optimizado y personalizado para el análisis de imágenes. Anticipando esta tendencia hacia una plataforma heterogénea [Dobai, 2013], la presente tesis doctoral aborda el diseño de una arquitectura hardware y software para la aceleración de un sistema para la detección de rostros basado en el algoritmo de Viola-Jones que puedan utilizarse para aplicaciones integradas en tiempo real.

También indicar que las FPGA se usan como un prototipo para evaluar las arquitecturas. Sin embargo, se pueden usar con diferentes plataformas de hardware programables. Las arquitecturas compactas de baja potencia y en tiempo real se pueden usar para diseñar procesadores especializados para incorporar capacidades de detección visual en cámaras, dispositivos móviles y robots humanoides. No obstante, hay varios desafíos de diseño asociados con el desarrollo de arquitecturas de hardware que se resumen en el siguiente apartado.

4.1.2.1 Problemas y desafíos del diseño de hardware

El diseño de una arquitectura de hardware para la detección de objetos que se puede usar con diferentes plataformas de hardware implica un compromiso entre diferentes factores que afectan tanto a la precisión de la detección como al rendimiento. Esto se menciona brevemente a continuación, pero será más evidente en las siguientes secciones y capítulos.

- **Representación de datos de entrada y capacitación**

La representación de datos desempeña un papel central en la implementación de una arquitectura de hardware para la detección de objetos. Por lo general, para aplicaciones de detección de objetos que se refieren a imágenes de píxeles, la entrada está representada por bits para imágenes en escala de grises (valores de intensidad de gris) y bits (bits por canal de color) para imágenes RGB (rojo-verde-azul). La gran mayoría de las aplicaciones de detección de objetos realizan clasificaciones en las imágenes en escala de grises. Sin embargo, no es necesario usar bits porque la entrada puede preprocesarse y normalizarse para ajustarse a los requisitos de la plataforma de implementación de hardware. Como tal, dependiendo del método de pre-procesamiento y los enfoques de extracción de características, los bits necesarios para representar el

valor pueden variar. Además, el valor puede no ser un número entero, como en las imágenes en escala de grises, sino un número real. Los datos de entrenamiento (peso y valores de umbral) también son típicamente números reales.

La representación de los números reales en el hardware se puede hacer ya sea mediante aritmética flotante o de punto fijo y complemento a dos. Por lo general, la representación de datos en punto fijo utiliza un hardware más simple que el que se necesita para las operaciones matemáticas en punto flotante. Sin embargo, es necesario que haya un diseño de la exploración del espacio para encontrar la cantidad óptima de bits para usar en la representación con el fin de minimizar los requisitos de memoria y conservar la tasa de precisión deseada.

- **Procesamiento paralelo**

Un algoritmo de reconocimiento de patrones requiere procesar cada dato de entrada con datos de entrenamiento para obtener el resultado de clasificación. La cantidad de datos de entrenamiento que necesita ser procesada por el algoritmo y la cantidad total de datos de entrada que deben procesarse por imagen de entrada afectan el rendimiento. Por lo tanto, para proporcionar un rendimiento en tiempo real, la arquitectura del hardware debe diseñarse de tal manera que facilite el procesamiento paralelo de datos. Hay dos formas posibles de explotar el paralelismo en los sistemas de detección de objetos. La primera es el paralelismo a nivel de píxeles que tiene como objetivo procesar los datos de una única ventana en paralelo con uno o más datos de entrenamiento. La segunda es el paralelismo a nivel de ventana que procesa varias ventanas con uno o más datos de entrenamiento, posiblemente sacrificando cierta velocidad para la clasificación de una ventana. También es posible una combinación de paralelismo a nivel de ventana y píxel, pero no es tan simple y requiere una exploración de espacio de diseño más rigurosa. Por supuesto, un factor importante a considerar son la cantidad de recursos disponibles y cómo el flujo de datos entre ellos puede facilitar cada nivel de paralelismo. Suponiendo que los recursos de hardware están disponibles, el factor limitante puede ser que el flujo de datos no sea tan rápido y pueda saturar la arquitectura. Por tanto, proporcionar un flujo de datos eficiente es igualmente importante para tener una arquitectura paralela. Además, la complejidad de controlar y administrar un sistema paralelo también es un aspecto importante a considerar.

- **Memoria y E / S**

Los problemas de memoria relacionados con un sistema de detección de objetos incluyen latencia, acceso, tamaño, estructura y ancho de banda. Las memorias externas como DRAM o flash compacta se utilizan para almacenar marcos de imagen / video. Un *buffer* de imagen se puede utilizar como memoria local para almacenar el marco de imagen activo en el chip para acelerar el procedimiento de clasificación y proporcionar más flexibilidad en la forma en que se accede a la memoria de imagen. El acceso a la memoria es importante para proporcionar un mayor grado de paralelismo. Además, las estructuras de memoria intermedia de imagen se pueden adaptar a los requisitos del sistema. En muchos casos donde la memoria en el chip prohíbe el almacenamiento de un marco de imagen completo, se usa un *buffer* de ventana para almacenar solo la ventana que necesita procesarse.

Además de los requisitos de almacenamiento de datos de imágenes, el algoritmo de reconocimiento de patrones también requiere el almacenamiento de los datos de entrenamiento utilizados para la clasificación. La frecuencia con la que se accede a los datos de entrenamiento afecta a la complejidad de la administración de la memoria. Si la solicitud de datos de entrenamiento es escasa, entonces es preferible almacenar los datos de entrenamiento en una memoria fuera del chip, y obtener solo los datos requeridos actualmente. En consecuencia, los recursos de memoria en el chip se pueden usar para los marcos de imagen. Sin embargo, si el acceso a los datos de entrenamiento es frecuente, entonces el uso de la memoria en el chip para proporcionar un acceso rápido y paralelo es la opción preferida.

4.1.3 Resumen comparativo entre las diferentes plataformas hardware

Atendiendo a todo lo indicado en los apartados anteriores, la tabla 4.6 recoge el análisis comparativo entre diferentes trabajos para la implementación del algoritmo en diferentes plataformas tecnológicas, y los requerimientos indicados de fiabilidad, rendimiento, coste, tamaño, consumo energético y flexibilidad. En la tabla 4.7 se puede ver un resumen comparativo entre las diferentes plataformas hardware estudiadas donde

se puede implementar el sistema de detección de rostros. En la misma se puede ver en color verde los requisitos eficientes y en rojo los requisitos poco eficientes para dicha plataforma en el caso de que se implementará el sistema de detección de rostros sobre la misma.

Autor	**Tecnología**	**Resolución imagen (píxel)**	**Tiempo de ejecución (s)**	**Año**
[Abyan, 2011]	CPU Multi-núcleo	320×240	0,95	2011
[Acasa, 2011]	CPU Multi-núcleo	640×480	0,46	2011
[Cheng, 2011]	CPU Multi-núcleo	720×576	0,7	2011
[Hsuan, 2012]	CPU Multi-núcleo	512×512	0,376	2012
[Young, 2016]	CPU Multi-núcleo	720×576	0,253	2016
[Ranja, 2017]	CPU Multi-núcleo	640×480	0,2	2017
[Shife, 2018]	CPU Multi-núcleo	512×512	0,05	2018
[Jianf, 2008]	DSP	40×40	0,94	2008
[Binko, 2011]	DSP	720×576	0,22	2011
[Zhouy, 2011]	ASIC	320×240	0,06	2011
[Kimab, 2017]	ASIC	650×500	0,14	2017
[Haipe, 2012]	GPU	500×500	0,25	2012
[Masek, 2013]	GPU	24×24	0,4	2013
[Wongy, 2015]	GPU	320×240	0.31	2015
[Chouc, 2015]	GPU	512×512	0,51	2015
[Bhati, 2016]	GPU	320×240	0,15	2016
[Savat, 2018]	GPU	250×250	0,23	2018
[Chobr, 2009]	FPGA	640×480	0,26	2009
[Matai, 2011]	FPGA	320×240	0,22	2011
[Dasat, 2012]	FPGA	640×480	0,17	2013
[Kimse, 2013]	FPGA	320×240	0,24	2013
[Sused, 2015]	FPGA	320×240	0,28	2015
[Archi, 2018]	FPGA	640×480	0,2	2018

Tabla 4.6: *Implementaciones del Algoritmo de Viola-Jones.*

Requisitos / P. Hardware	Fiabilidad	Rendimiento	Coste	Tamaño	Consumo energético	Flexibilidad	Facilidad de uso	Tiempo de desarrollo
CPU Multi-núcleo								
DSP								
GPU								
ASIC								
FPGA								

Tabla 4.7: *Comparativa entre plataformas hardware, para los requerimientos del sistema de detección de rostros.*

Como se puede ver en la comparativa, aunque no se puede hablar de un ganador claro entre los diferentes candidatos para ser la plataforma del sistema de detección facial, las CPU multi-núcleo son una alternativa a considerar para el procesamiento de la visión de alto nivel. Además, son fáciles de programar, ya que es posible utilizar las mismas herramientas y bibliotecas que se usan para las aplicaciones de PC estándar. Esto acorta significativamente la curva de aprendizaje necesaria para dominar una nueva arquitectura de hardware, que en el caso de los FPGA y las GPU debe tenerse especialmente en cuenta.

Además de los requisitos de diseño para plataformas integradas, la complejidad y las características del algoritmo implementado son dos parámetros que deben tenerse en cuenta al elegir una opción. Especialmente en qué medida el algoritmo se puede paralelizar e implementar fácilmente en una arquitectura adecuada para SIMD.

En cualquier caso, no es factible implementar una aplicación de visión completa en un FPGA o GPU. Estas arquitecturas no son adecuadas para la toma de decisiones de alto nivel, por lo que intentar implementar todo ese procesamiento en serie prolongaría demasiado el ciclo de desarrollo. Además, la portabilidad y la escalabilidad están mejor aseguradas usando soluciones de software. Por otro lado, usar una plataforma de software pura sería factible solo con aplicaciones simples. De lo contrario, la aceleración de hardware sería necesaria, especialmente para el procesamiento de bajo nivel.

Por todas estas razones, existe una tendencia creciente en adoptar arquitecturas SoC para visión integrada. Estos SoCs generalmente están compuestos por un microprocesador ARM y al menos un componente de hardware adicional, que puede ser una FPGA, una GPU o un DSP. La parte de bajo procesamiento del algoritmo se ejecuta en el FPGA, GPU o DSP, y el resto en el microprocesador, combinando las fortalezas de ambas arquitecturas. Además, como están ubicados físicamente en el mismo chip, el consumo total de energía es mucho menor que tenerlos en dos chips separados.

Actualmente, las soluciones basadas en GPU parecen la opción menos atractiva, debido a su mayor consumo de energía. Sin embargo, nuevos productos más eficientes energéticamente han comenzado a aparecer. Si bien los sistemas integrados han madurado significativamente en las últimas décadas, el relativamente nuevo campo de visión integrada para detección de objetos seguirá siendo un área activa de investigación en los próximos años y se espera que aparezcan nuevas soluciones innovadoras junto con herramientas más inteligentes para validarlas.

4.2 Entornos tecnológicos para implementar sistemas de detección facial

Debido a la complejidad de los algoritmos de detección facial se requiere una gran cantidad de recursos de computación y memoria. Por lo tanto, las implementaciones software de los algoritmos de detección resultan poco eficientes cuando deben ejecutarse sobre sistemas empotrados de bajo coste, bajas prestaciones, pocos recursos y bajo consumo de potencia. En estos casos el uso de técnicas de optimización usando técnicas de paralelización software puede aplicarse para acelerar las partes que requieren de mayor consumo computacional en los algoritmos de detección.

En este sentido, en la actualidad los algoritmos más exitosos para la detección de rostros en imágenes suelen ejecutarse adecuadamente en tiempo real en una CPU de un computador personal de última generación. En cambio, si se ejecuta el mismo algoritmo en una CPU de un dispositivo Android de gama baja, éste mismo algoritmo no

funcionará lo suficientemente rápido como para que su ejecución sea en tiempo real. Esta diferencia de rendimiento es debida a las diferentes capacidades de los procesadores utilizados en los dispositivos móviles que están optimizados para un bajo coste y bajo consumo de energía, y por tanto, sus resultados van por detrás de los procesadores utilizados en ordenadores personales.

Como se indico en el capítulo 1 de esta memoria, en la actualidad el aumento del rendimiento de las CPU se lleva a cabo incrementando el número de núcleos que la forman funcionando en paralelo. Por ello, los dispositivos móviles y sistemas empotrados evolucionan para pasar de una CPU de un sólo núcleo a una CPU multi-núcleo. En este sentido, se llega a un punto en el que mejorar el rendimiento de ejecución de un programa en C++ escrito originalmente para la ejecución tradicional en una sola CPU, implica la modificación de los algoritmos de procesamiento para utilizar varios núcleos de CPU en paralelo. En este caso, la API (Interfaz de Programación de Aplicaciones) OpenMP [Garci, 2003], es una de las mejores opciones dentro de las tecnologías de programación en paralelo debido a que es soportada por la mayoría de sistemas operativos, herramientas de compilación y otros dispositivos hardware, por lo que funciona hoy en día, incluso en dispositivos móviles.

Una extensión de OpenMP es OmpSs [Ompss, 2018], que es un modelo de programación portable y escalable que proporciona a los programadores una interfaz sencilla y flexible para el desarrollo de aplicaciones paralelas para plataformas que van desde PC y dispositivos móviles hasta superordenadores. La API de OmpSs utiliza un paradigma de programación basado en directivas que apoyan la paralelización incremental de un programa secuencial existente. Una vez que un código ha sido compilado con el compilador de OmpSs Mercurium [Mercu, 2018], éste puede ser ejecutado a través del *runtime* de OmpSs denominado Nanos++ [Nanos, 2018], consistente en una serie de bibliotecas encargadas de controlar la ejecución del programa e intentar finalizar la ejecución de la manera más eficiente posible.

4.2.1 Entornos hardware multi-núcleo

Desde la aparición de los procesadores multi-núcleo, y tras su auge durante la pasada década como respuesta a las limitaciones en el incremento de la frecuencia de reloj impuestas por la tecnología, este tipo de arquitecturas se han convertido en un pilar clave en el desarrollo de arquitecturas de altas prestaciones, reuniendo a la vez una gran capacidad de cálculo y una eficiencia energética notable.

Históricamente, los procesadores multi-núcleo han estado formados por varias Unidades de Procesamiento (CPUs), típicamente con idénticas características entre ellas. Es lo que se denomina arquitectura de multiprocesamiento simétrico (SMP-*Symmetric Multi-Processing*), que es un tipo de arquitectura de computadores en la que dos o más unidades de procesamiento comparten una única memoria central compitiendo en igualdad de condiciones en el acceso a memoria. Un ejemplo hardware de este tipo de arquitecturas es la placa Raspberry PI 2 [Raspb, 2017], que es un dispositivo empotrado de bajo coste muy versátil y dispone de una CPU de cuatro núcleos simétricos.

Sin embargo, a medida que la ley de escalabilidad de Dennard [Denna, 1972] comienza a ver su fin, los sistemas heterogéneos, que son aquellos que usan más de un tipo de procesador, son sistemas que ganan en rendimiento por añadir procesadores distintos. Incorporando capacidades de procesado especializadas para realizar tareas particulares, se han convertido en las arquitecturas más atractivas en el campo de la computación de altas prestaciones [Kumar, 2003]. Dentro de los sistemas heterogéneos es posible realizar una segunda clasificación en función de que los distintos núcleos dispongan de distintos repertorios de instrucciones (heterogeneous-ISA) o utilicen el mismo repertorio de instrucciones (single-ISA). Estos últimos se denominan habitualmente procesadores asimétricos (AMP, Asymmetric Multicore Processors) [Mitta, 2016] y han sido propuestos como alternativa de bajo consumo a los procesadores multi-núcleo simétricos convencionales. Los AMP combinan núcleos rápidos y complejos, de alto rendimiento, con núcleos lentos de consumo reducido.

En un procesador multi-núcleo asimétrico [Tulls, 2004], las diferencias entre los distintos núcleos se deben a distintas microarquitecturas (asimetría física) o se trata de procesadores con la misma microarquitectura pero distinta frecuencia o tamaño de

cache (asimetría virtual). En la actualidad, el uso de arquitecturas asimétricas es una tendencia en alza en el mercado de dispositivos móviles, en el que el tipo de tareas a ejecutar y su criticidad es altamente heterogéneo, y el tiempo de vida de las baterías es un bien escaso y muy a tener en cuenta. Es por ello que las arquitecturas asimétricas permiten optimizar la eficiencia energética asignando las tareas a núcleos de distintas características en función de los requisitos de rendimiento y consumo de cada tarea. Pero además, la arquitectura permite ajustar el rendimiento y consumo de cada núcleo utilizando las técnicas de escalado de frecuencia que se encuentran disponibles en los procesadores modernos. Dentro de las arquitecturas multi-núcleo asimétricas, tienen especial relevancia los procesadores de la familia ARM [Armco, 2017], que se denominan big.LITTLE [Brian, 2013], para reflejar que incluyen procesadores potentes (big) junto con procesadores de menor rendimiento y mucho menor consumo (LITTLE). Un ejemplo de este tipo puede ser la placa Odroid XU4 [Odroi, 2016] [Odro2, 2017] [Odro3, 2017].

A continuación, se describen detalladamente las dos placas experimentales que se utilizarán para el desarrollo de la investigación de la tesis doctoral. La placa Raspberry Pi 2 B+, que contiene una CPU de 4 núcleos simétricos y la placa Odroid XU4 que dispone de una CPU asimétrica de 8 núcleos distribuidos en dos *clúster* de 4 núcleos cada uno.

4.2.1.1 Placa Raspberry Pi

Raspberry Pi es un computador de placa única (SBC, Single Board Computer) de bajo coste (aproximadamente 35 euros) [Mattr, 2012] [Ebenu, 2012], del tamaño de una tarjeta de crédito (85.6 mm x 56.5 mm) y 45 gramos de peso que desarrollo la Raspberry Pi Foundation (RPF) con la intención de promover el aprendizaje de la informática básica en colegios. Esta fundación es una organización benéfica educacional sin ánimo de lucro creada en 2009 con sede en el Reino Unido. Un SBC es un ordenador completo construido en una sola PCB (*Printed Circuit Board*), que incluye microprocesadores (CPU, GPU, etc.), memoria de programa, elementos y puertos de entrada/salida, memoria masiva y otro hardware necesario. En ocasiones la memoria masiva funciona como periférico, como es el caso de la Raspberry Pi. Por

supuesto un SBC no incluye periféricos en sí mismo, pero estos pueden conectarse fácilmente a través de los distintos puertos de entrada/salida disponibles.

Las características y elementos de Hardware más importantes Raspberry Pi aparecen esquematizados en la figura 4.4.

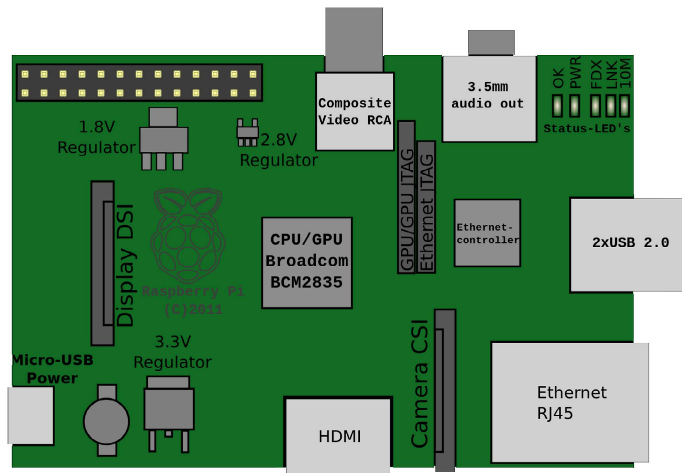


Figura 4.4: *Layout Raspberry Pi [Rasb2, 2017].*

Hasta el momento, aunque existen modelos intermedios, principalmente se comercializan tres modelos cuyas principales prestaciones se muestran en la tabla 4.8.

Características	Raspberry Pi 1	Raspberry Pi 2	Raspberry Pi 3
Chip	Broadcom BCM2835	Broadcom BCM2836	Broadcom SCO BCM2837
Procesador	ARM 1176JZF-S a 700 MHz	ARM Cortex A7, 900 MHz quad-core	ARM Cortex-A43, quad-core a 1.2 GHz
Procesador gráfico	VideoCore IV 520 MHZ OPENGLES 2.0	VideoCore IV 250 MHZ OPENGLES 2.0	VideoCore IV 400 MHZ OPENGLES 2.0
Memoria RAM	256 MB LPDDR SDRAM 400 MHZ	1 GB LPDDR2 SDRAM 450 MHZ	1 GB LPDDR2 SDRAM 450 MHZ
Vídeo	HDMI 1.4 1920x1200	Hdmi 1.4 1900x1200	Hdmi 1.4 1900x1200
Entradas y salidas de vídeo	Conector MIPI CSI, Conector RCA, Conector HDMI	Conector MIPI CSI, Conector RCA, Conector HDMI	Conector MIPI CSI, Conector RCA, Conector HDMI
Entradas y salidas de audio	HDMI, Minijack	HDMI, Minijack	HDMI, Minijack
Puertos USB	Uno (En el modelo B dos; en el modelo B+, cuatro)	Cuatro	Cuatro
Almacenamiento integrado	SD (En el modelo A+, microSD)	MicroSD	MicroSD
Conexión red	Ninguna	10/100 Ethernal via hub USB	WiFi 802.11n
Bluetooth	No	No	Bluetooth 4.1
Dimensiones	8.5 x 3.5 centímetros	8.5 x 3.5 centímetros	8.5 x 3.5 centímetros
Peso en gramos	45 (El modelo A+, 23)	45	45
Precio	29.95€	34.95€	35 dólares (se esperan unos 35€ al cambio)

Tabla 4.8: *Modelos actuales de Raspberry Pi [Rasb2, 2017].*

Como se ha comentado el modelo de placa que utilizaremos en la investigación es la Raspberry Pi 2 B. Se trata de un sistema compuesto por un SoC (System-on-Chip) con un Procesador Broadcom BCM2836 de 900 MHz ARM Cortex-A7 de cuatro núcleos con GPU VideoCore IV de doble núcleo que tiene implementado OpenGL ES 2.0 entre otras características. En la figura 4.5 se puede ver la arquitectura de la CPU de La Raspberry Pi 2 B [Raspb, 2017]. Dicha placa, ver figura 4.6, representa un buen ejemplo de dispositivo electrónico asequible comparable a las CPU de los teléfonos inteligentes actuales (Android, iPhone y Windows Móvil), que utilizan procesadores similares de varios núcleos ARM [Armco, 2017]. Por lo tanto, cualquier mejora de rendimiento que se lleve a cabo en esta plataforma es trasladable a los teléfonos inteligentes y tablets.

La Raspberry Pi 2 B usa mayoritariamente sistemas operativos basados en el núcleo Linux como Raspbian, una distribución derivada de Debian que está optimizada para hardware de Raspberry Pi. Se lanzó durante julio de 2012 y es la distribución recomendada por la fundación y es la que contiene todo el respaldo y funciones requeridas para la realización del objeto de esta tesis doctoral.

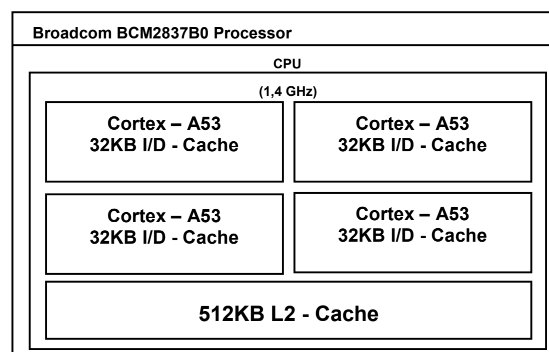


Figura 4.5: *Arquitectura CPU de la Raspberry Pi 2 B [Fuente: elaboración propia].*

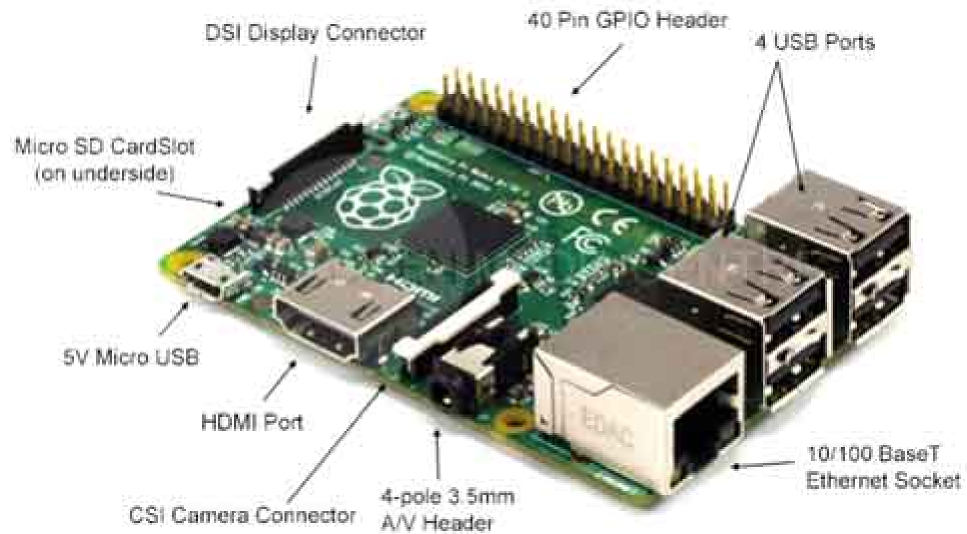


Figura 4.6: *Raspberry Pi 2 B [Rasb2, 2017].*

4.2.1.2 Placa Odroid XU4

La placa Odroid XU4, ver figura 4.7, como ya se ha comentado brevemente, es una placa de bajo coste (aproximadamente 70 euros), que tiene una arquitectura multi-núcleo asimétrica basada en ARM. Está provista de un SoC Exynos 5422 de Samsung fabricado en tecnología de 28 nm, que integra un procesador ARM big.LITTLE de ocho núcleos, formado por dos *clusters* de 4 núcleos cada uno, el Big, formado por 4 núcleos Cortex A15 de alto rendimiento, y el Little, formado por 4 núcleos Cortex A7 de bajo consumo, ver arquitectura del procesador en la figura 4.8. Fue creada por Hardkernel, una compañía de Hardware Open-Source con sede en Corea del Sur.

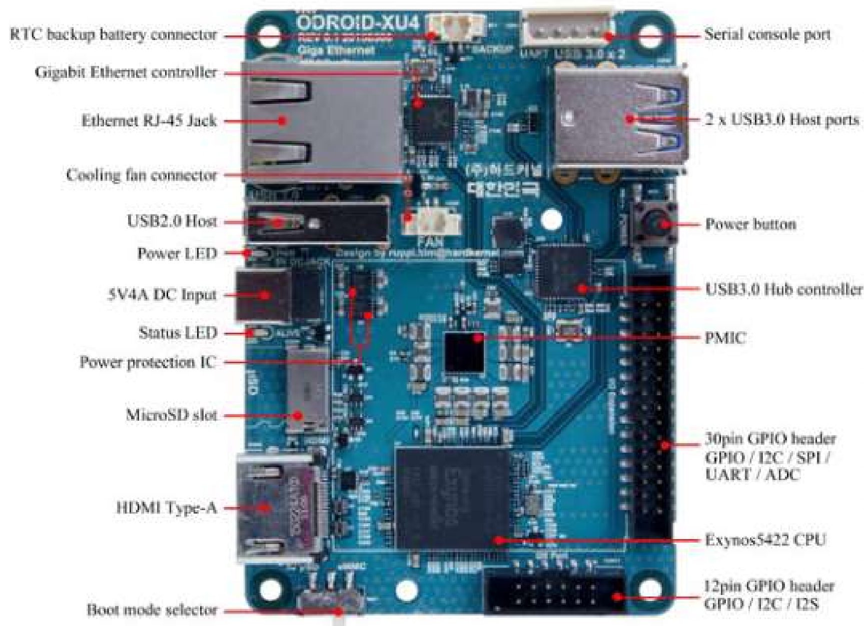


Figura 4.7*: Placa Odroid XU4 [Odroi, 2016].*

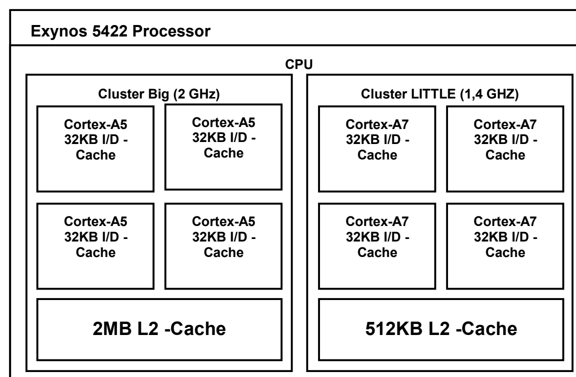


Figura 4.8*: Arquitectura CPU de la placa Odroid XU4 [Fuente: elaboración propia].*

Por otro lado, incorpora 2 Gb de memoria RAM LPDDR3 (*PoP Stacked*, integrada en el PCB) y un controlador de memoria flash eMMC 5.0 de 8 bits con conector para módulos externos de memoria eMMC, por lo que no incorpora almacenamiento integrado de serie. No obstante, admite tanto módulos eMMC como tarjetas microSD, así que, al igual que en la Raspberry Pi, se le puede instalar el sistema operativo en una tarjeta externa sin problema. Cabe mencionar que incorpora dos puertos USB 3.0 y un USB 2.0. También tiene una tarjeta de red Gigabit y salida de vídeo HDMI.

Los usos que se le pueden a esta placa de desarrollo son muy amplios, y se puede hacer con ella básicamente lo mismo que con Raspberry Pi pero con una mayor potencia y versatilidad al tener más conectores y más rápidos.

En cuanto a los sistemas operativos que se pueden ejecutar en esta placa, tienen que tener soporte para la arquitectura ARM y se pueden encontrar varios en la página oficial de Odroid [Odro2, 2017]. Los principales y más conocidos son Android 4.4, en sus diferentes versiones y Ubuntu 14.04 y 15.04, así como distribuciones derivadas de los mismos como son Lubuntu y Xubuntu. Por otra parte, existen sistemas operativos experimentales como Wheezy, que es una distribución basada en Debian. Para la selección del sistema operativo a instalar para este trabajo se ha optado por descartar los sistemas operativos no oficiales que no tienen respaldo ni las funciones requeridas y se ha elegido por tanto Ubuntu 14.04 como sistema operativo de la Odroid XU4, que contiene todas las funciones para la realización del objeto de la tesis doctoral.

4.3 Conclusiones

En este capítulo se ha presentado una introducción general a las diferentes plataformas hardware donde se puede implementar un sistema de detección facial como el del objeto de la presente tesis doctoral, y con ello elegir la plataforma tecnológica más adecuada. Para ello se han fijado una serie de requisitos de diseño y desarrollo como son la fiabilidad, rendimiento, coste, tamaño, consumo energético, flexibilidad, facilidad de uso y tiempo de desarrollo. Con todo ello se ha llevado a cabo una comparación entre las diferentes plataformas hardware. Como conclusión, se determina que la mejor opción hardware, en la actualidad, para la implementación de un sistema de detección facial basado en el algoritmo de Viola-Jones, es usar una CPU multi-núcleo.

Atendiendo a los resultados de la comparativa, por último, se han presentado las principales características de dos dispositivos hardware de bajo coste que cumplen con los requerimientos de desarrollo, y están constituidos por una CPU multi-núcleo, sobre los que se podría implementar el sistema de detección facial, y que disponen de una

tecnología similar a la que encontramos en los teléfonos móviles inteligentes actuales, como son la placa Raspberry Pi 2 B y la placa Odroid XU4.

CAPÍTULO 5.

5. DESARROLLO Y ACELERACIÓN DEL ALGORITMO

En este capítulo se describe la metodología y el proceso llevado a cabo para la implementación del algoritmo de detección facial, así como los entornos de desarrollo experimentales donde se ejecuta. Para ello se parte del código fuente de una implementación simplificada del algoritmo de Viola-Jones, desarrollada en C++, que dispone de unos parámetros de entrenamiento fijados y que presenta una tasa de detección alta para una amplia gama de imágenes de entrada. Se adecuará y optimizará el software al entorno de trabajo y se procederá a la extracción de paralelismo a nivel de tareas usando la API de OPenMP [Opeor, 2016], con el fin de acelerar el programa de detección y aprovechar al máximo los recursos de los dispositivos experimentales.

5.1 Implementación algoritmo de Viola-Jones en C++

El lenguaje de programación utilizado para desarrollar el detector facial basado en el algoritmo de Viola-Jones, es C++. Para su implementación se parte del código fuente de una implementación simplificada [Zheny, 2017] que dispone de unos parámetros de entrenamiento fijados. Es decir, la implementación solo considera la parte de detección facial y no incluye la parte de entrenamiento, ya que usa unos parámetros pre-entrenados para el clasificador en cascada.

El detector se aplica a lo largo de la imagen en distintas posiciones y a diferente escala, ya que se debe considerar la existencia de más de una cara y de diferentes tamaños según se puede ver en la figura 5.1. El escalado se consigue modificando o bien el tamaño de la ventana del detector o bien el tamaño de la imagen hasta que tenga el mismo tamaño que la ventana de detección. Este proceso tiene sentido ya que las características pueden ser evaluadas a cualquier escala con el mismo coste.

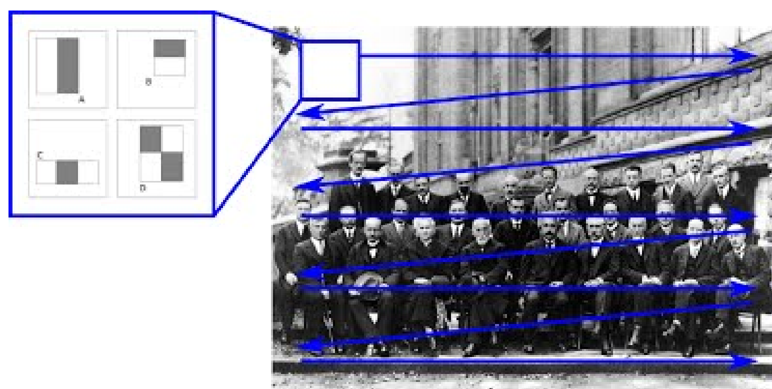


Figura 5.1: *Desplazamiento de la ventana de detección sobre la imagen [Zheny, 2017]*

El detector también se aplica en diferentes posiciones a lo largo de la imagen con lo que la ventana de detección se mueve a lo largo de la imagen un número determinado de píxeles, denominado desplazamiento (Δ). La elección de este número afectará tanto a la velocidad del detector como a su rendimiento. Tanto el tamaño de la ventana que se evalúa como el tamaño del desplazamiento de la misma sobre la imagen se incrementan utilizando el factor de escala y el factor de desplazamiento (Δ). En la publicación de

Viola-Jones se muestran resultados utilizando tanto Δ = 1 píxel, como Δ = 1.5 píxel, y con un factor de escala de 1 y 1.25 respectivamente [Viola, 2004].

En la figura 5.2, se puede ver una representación gráfica del diagrama de bloques secuencial que representa la organización de todo el proceso de detección facial con sus entradas y salidas, y el funcionamiento interno del sistema implementado, además de la relación entre los bloques.

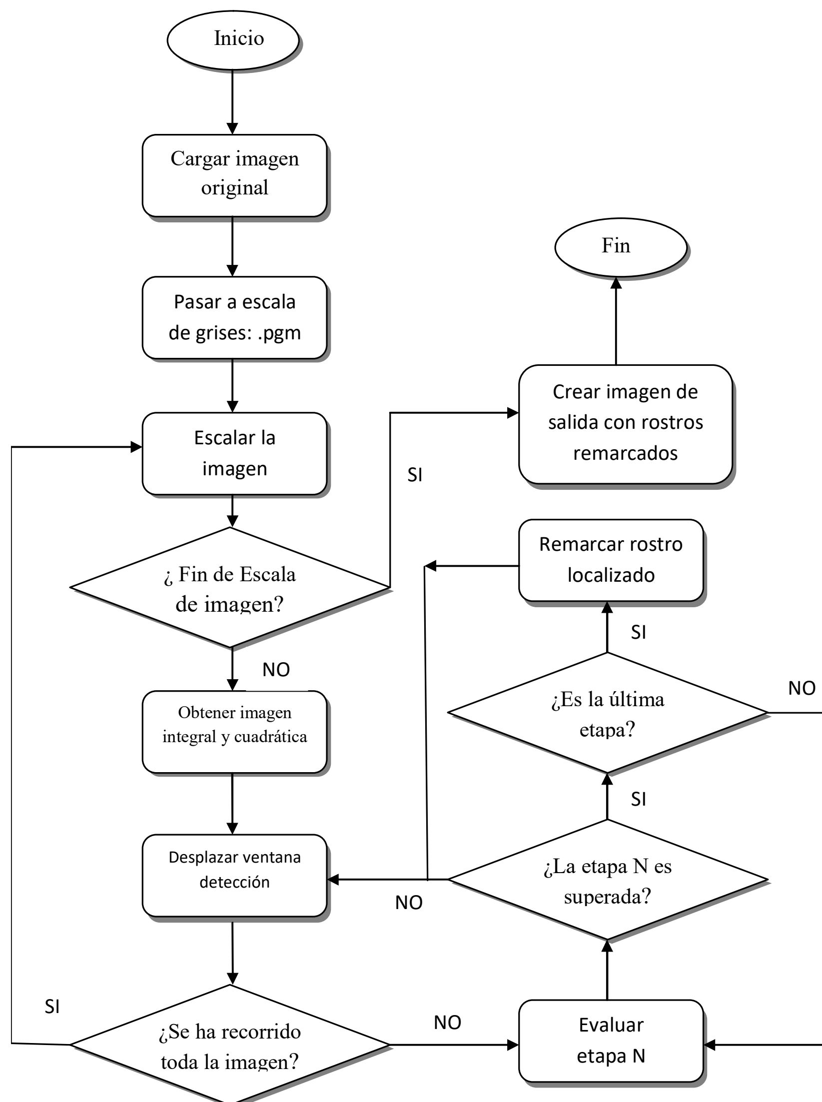


Figura 5.2: *Diagrama de bloques del programa de detección facial.*

La entrada del sistema es una imagen en escala de grises en formato PGM (Portable Graymap Format). Este formato de imagen simple en escala de grises sin compresión, está diseñado de forma sencilla y fácil de interpretar. Utiliza 8 bits por píxel y por tanto existen 256 niveles de grises. Un fichero PGM contiene texto plano y puede ser modificado con un simple procesador de texto; también existe la versión en binario, no legible por procesadores de texto normalmente.

A partir de la imagen de entrada se calcula la imagen integral normal (ecuación 3.2) y cuadrática ($\sum x^2$), que va a permitir calcular de forma sencilla el valor de las características en la imagen y la varianza a la que están normalizados los parámetros obtenidos en el entrenamiento del sistema. A continuación, se recorre la imagen con la ventana de detección analizando cada zona de la misma (a diferentes escalados). Si la región que ocupa dicha ventana supera todas las etapas del clasificador se selecciona la región como un rostro y su localización se remarca con un cuadrado. Seguidamente se desplaza la ventana hasta recorrer toda la imagen repitiendo todo el proceso de detección. Finalmente, se obtiene como salida la imagen de entrada con un cuadrado superpuesto en las zonas donde se ha localizado un rostro, tal como se puede ver en la figura 5.3.

Figura 5.3: *Imagen de entrada y de salida del programa de detección facial [Natio, 2016].*

El sistema de detección facial se puede implementar en dos modos de operación diferentes:

- **Modo 1:** Escalando la imagen. En este modo la imagen es escalada mediante interpolación hasta que se alcanza un tamaño mínimo predefinido. En cada momento del escalado se necesitan dos imágenes integrales normal ($\sum x$) y cuadrática ($\sum x^2$) para calcular la varianza, siendo x el valor (entre 0 y 255) de los píxeles de la imagen. La ventana de búsqueda tiene un tamaño fijo en todo el proceso de detección.

- **Modo 2:** Escalando los clasificadores. En este modo las imágenes integrales normal ($\sum x$) y cuadrática ($\sum x^2$) necesarias para calcular la normalización de la varianza se obtienen una sola vez de la imagen original (ver Ecuación 5.1). Sin embargo las características tipo Haar de los clasificadores son escaladas progresivamente hasta que sus dimensiones son similares a las de la imagen original. La ventana de búsqueda tiene, por lo tanto, dimensión variable durante el proceso de detección.

$$\left(\frac{\sigma}{WxH}\right)^2 = \frac{\sum x^2}{WxH} - \left(\frac{\sum x}{WxH}\right)^2 \quad (5.1)$$

En los dos modos los componentes de las características de tipo Haar (pesos y dimensiones) son escalados progresivamente con las dimensiones de la ventana de búsqueda. Esto significa que para una ventana de búsqueda de dimensión W×H (W = ancho, H = alto) el peso de cada rectángulo de la característica de tipo Haar es escalado por el valor W×H.

Para la implementación del sistema de detección facial objeto de esta investigación se elige el Modo 1, en el que se mantiene la ventana de detección al mismo tamaño y se escala la imagen en diferentes pasos del proceso construyendo lo que se conoce como pirámide de imágenes (ver figura 5.4). Por tanto, la pirámide de imágenes es una representación multi-escala de una imagen, de tal manera que la detección de rostros puede ser invariante en escala, es decir, la detección de caras grandes y pequeñas utiliza la misma ventana de detección. La implementación de la

pirámide de imágenes se realiza por reducción de la resolución de la imagen utilizando el algoritmo de los píxeles vecinos.

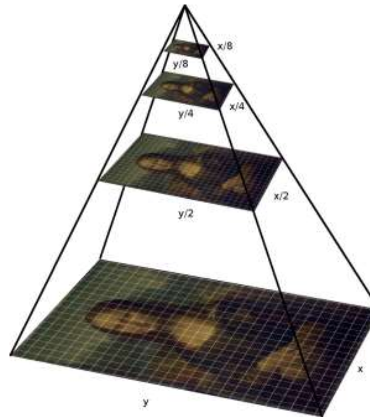


Figura 5.4: *Pirámide de imágenes. Representación multi-escala de una imagen [Zheny, 2017].*

Una vez escalada la imagen, se calcula la imagen integral normal (∑x) y la imagen integral cuadrática ($\sum x^2$), para poder calcular la varianza y desviación típica. Todos los ejemplos de sub-ventanas que se utilizan para el entrenamiento se hicieron con la varianza normalizada para reducir al mínimo el efecto de diferentes condiciones de iluminación. La normalización a través de la desviación típica es por lo tanto necesaria durante el proceso de detección. Dicha normalización con las dimensiones de la ventana de búsqueda viene dada por la ecuación 5.2, que reduce tanto el número de operaciones aritméticas (división, multiplicación) como los accesos a memoria para el cálculo. Esta ecuación es la que se implementa en el sistema y hace que el algoritmo sea más rápido [Acasa, 2011].

$$\left(\frac{\sigma}{N}\right)^2 = N\sum x^2 - \left(\sum x\right)^2 \qquad N = W \times H \tag{5.2}$$

Finalmente, indicar que en la implementación se ha considerado una ventana de detección de 24✕24 píxeles. El desarrollo final consta de un total de 25 etapas, cada una de ellas con un número determinado de clasificadores débiles, que forman el clasificador fuerte de la misma, siendo las primeras etapas las menos complejas computacionalmente para intentar descartar el mayor numero de ventanas en el menor tiempo posible. Entre todas las etapas hay un total de 2.913 clasificadores débiles, cada uno de los cuales requiere un total de 18 parámetros obtenidos previamente en el

proceso de entrenamiento y que se recogen en un fichero de texto. Es decir, la detección de un único rostro en una imagen, que pasa por todas las etapas del clasificador en cascada, requiere del cálculo de 2.913 características, cada una de las cuales queda definida en la ventana de detección por 18 parámetros. Una vez calculados los valores de las características que forman una etapa se hace una comparación con los umbrales obtenidos en el proceso de entrenamiento para la etapa de modo que si el valor obtenido supera dicho umbral la etapa es superada y se pasa a la siguiente, y así sucesivamente hasta la última etapa del clasificador que si es superada determinará que la ventana analizada contiene un rostro.

En la figura 5.5, se puede ver el pseudo-código con las principales etapas que sigue la implementación del algoritmo para cada uno de los escalados de la imagen de entrada. En la misma se puede ver que el número de etapas del clasificador es 25, y que el número de características Q asociado a cada una depende de la etapa que se esté considerando.

```
Mientras  R > Y hacer //R= tamaño imagen entrada, Y =tamaño ventana detección
  R= R-F  //Se reduce el tamaño de la imagen un factor de escala F.
  Obtener Σx y Σx² //imagen integral normal y cuadrática
  Para  d←0 hasta t hacer  //d=posición ventana detección, t posición final ventana detección en imagen
    d=d+Δ // Δ= desplazamiento de la ventana de detección por la imagen
    Para E←0 hasta  E<=25 hacer  //E= etapa del clasificador en cascada
      E=E+1
      Para  C←0 hasta Q(E) hacer  //C Característica etapa, Q número de características la etapa E
        Obtener V //V=valor de la característica
        A=A+V //A=Suma de los valores de las Características de la etapa
        C=C+1
        Si A<U hacer //U=umbral de la Etapa
          Rechazar esta ventana como una cara
          Romper bucle
      Fin para
    Fin para
    Si la ventana de detección no supera los umbrales en cada etapa hacer
      Aceptar la ventana como una cara
    Si no
      Rechazar la ventana como una cara
  Fin Para
Fin mientras
```

Figura 5.5: *Pseudo-código para implementar el algoritmo de Viola-Jones*

5.2 Entorno de trabajo

Los entornos hardware experimentales que se utilizan para ejecutar el programa de detección facial son la Placa Raspberry Pi 2 B y la placa Odroid XU4, tal y como se ha indicado en el Capítulo 4 de esta memoria. *GNU/Linux* es el nombre habitual del sistema operativo que se instala en ambas placas, concretamente una distribución de éste, que es una selección de paquetes compilados para una arquitectura concreta. En la placa Raspberry Pi 2 B se instala la distribución Raspbian versión Jessi [Raspb, 2017], que está optimizada para el hardware de dicha placa. Raspbian es una distribución libre y gratuita basada en la distribución Debian 8, creada por un pequeño grupo de desarrolladores con fines educativos. Actualmente dispone de una comunidad de programadores activa que actualiza periódicamente la distribución. Contiene unos 35.000 paquetes, pre-compilados, que se pueden instalar fácilmente según necesidad en la Raspberry Pi 2 B.

En el caso de la placa Odroid XU4 [Odroid, 2016], se instala la distribución Ubuntu Mate 16.04, también libre y gratuita que dispone de un *kernel* personalizado disponible en el sitio web de Hardkernel [Hardk, 2016], que permite que el sistema operativo se comunique con el hardware ODROID. Está página pública *kernels* que son específicos de la arquitectura ODROID, y mantiene un repositorio desde donde se pueden descargar los últimos que se van incorporando, así como una gran cantidad de bibliotecas y de aplicaciones que se pueden instalar, utilizar y modificar libremente.

Como entorno de desarrollo se ha utilizado la herramienta Codeblocks, herramienta de software libre multiplataforma que permite programar principalmente en C/C++ y Java. Lo más usual y sencillo para compilar un programa en GNU/Linux es el compilador **g++**, ya que es el que se incluye en todas las distribuciones, como Raspbian en la Raspberry Pi y Ubuntu Mate en la placa Odroid XU4.

G++ forma parte del proyecto *GNU Compiler Collection* (GCC), que engloba a un conjunto de compiladores de muy diversos lenguajes (C, C++, Objective-C, Java, D, Pascal, Ada, etc.) y de muy alta calidad. Se trata de una herramienta muy completa y compleja que se ha convertido en un estándar de facto en la industria, que se utiliza

incluso por los diseñadores de procesadores para probar los nuevos diseños antes incluso de su producción comercial.

5.3 Paquetes y estructura funcional del sistema de detección

El software desarrollado para la implementación del sistema de detección facial se estructura en un conjunto de archivos, donde cada uno de ellos contiene un conjunto de funciones específicas para una determinada tarea. En la figura 5.6 se puede ver, junto con una breve descripción, el conjunto de ficheros que compone el sistema.

- ***main.cpp****: función principal,*
- ***haar.cpp****: contiene los principales pasos del algoritmo Viola y Jones*
- ***haar.h****: archivo de cabecera del algoritmo Viola-Jones*
- ***rectángulos.cpp****: Dibuja rectángulos sobre rostros detectados*
- ***image.c****: funciones de asistencia para la manipulación de imágenes*
- ***image.h:*** *el archivo de cabecera para la manipulación de imágenes*
- ***Makefile****: Makefile para compilar el algoritmo*
- ***info.txt****: los parámetros del filtro de cascada pre-entrenado*
- ***class.txt****: los parámetros del filtro Haar pre-entrenados*
- ***Face.pgm****: la imagen de entrada de prueba*
- ***Output.pgm****: Imagen de salida del sistema de detección*

Figura 5.6: *Conjuntos de ficheros que componen el sistema de detección.*

El archivo más importante es "haar.cpp", que contiene todas las funciones para la implementación del algoritmo de detección facial de Viola-Jones. En cuanto a *Makefile,* es un fichero de texto que contiene todas las instrucciones necesarias para compilar los paquetes que forman el sistema de detección de manera sencilla a partir de la utilidad *make*.

El fichero. “info.txt”, es un fichero de texto que contiene el número de clasificadores simples o características que contiene cada una de las 25 etapas que forma el clasificador en cascada experimental pre-entrenado objeto de esta investigación. En la figura 5.7 se puede ver el contenido de dicho fichero. En el mismo,

se muestra que la primera etapa está formada por 9 características, la segunda etapa por 16, y así sucesivamente hasta alcanzar la última etapa que contiene un total de 211 características a evaluar en la región de la imagen que está siendo analizada. La zona de la imagen que supere todas estas etapas se considera que es un rostro y se remarca con un cuadrado en la imagen de salida del programa, como muestra la figura 5.3.

```
- 9
- 16
- 27
- 32
- 52
- 53
- 62
- 72
- 83
- 91
- 99
- 115
- 127
- 135
- 136
- 137
- 159
- 155
- 169
- 196
- 197
- 181
- 199
- 211
```

Figura 5.7: *Contenido del fichero: "info.txt". Cada línea representa el número de características por etapa del clasificador en cascada.*

Los parámetros de las características tipo Haar se almacenan en el archivo "class.txt", que contiene la información necesaria para evaluar adecuadamente una característica en una imagen (localización en la ventana de búsqueda, pesos y umbrales), y cuyos valores se han obtenido en el proceso de entrenamiento del detector para una ventana de detección de 24×24 píxeles. En concreto, cada una de las características requiere de un total de 18 parámetros para su correcta evaluación. En este sentido, el formato de archivo se muestra en la figura 5.8 [Zheny, 2017].

1.- Cada etapa del clasificador en cascada contiene:

18 parámetros por característica + 1 umbral por etapa.

2.- Para una etapa de 9 características, se tienen:

$(18 \times 9) + 1 = 163$ parámetros.

Que son los que aparecen entre la línea 1 a 163 de "class.txt".

3.- Cada uno de los 18 parámetros de cada característica representa lo siguiente:

- *1 a 4: coordenadas del rectángulo 1*
- *5: peso del rectángulo 1*
- *6 a 9: coordenadas del rectángulo 2*
- *10: peso del rectángulo 2*
- *11 a 14: coordenadas del rectángulo 3*
- *15: peso del rectángulo 3*
- *16: umbral de la característica*
- *17: alfa 1 de la característica*
- *18: alfa 2 de la característica*

***Figura 5.8:** Estructura del contenido del fichero: "class.txt".*

5.3.1 Descripción funcional del sistema

Diseñar e implementar un algoritmo como el de Viola-Jones en un lenguaje de programación como C++ puede ser una tarea difícil, ya que requiere de una buena dosis de experiencia y creatividad. Para el diseño se debe tener muy claro sin ambigüedad la definición del mismo, además del orden de cada uno de los pasos que se tiene que ir dando para realizarlo. En este contexto, el programa que se ha desarrollado para implementar el algoritmo de Viola-Jones sigue la programación estructurada propia del lenguaje C++, consta de 13 funciones principales definidas para la realización de la detección facial y que se listan en la tabla 5.1. También se definen dos parámetros configurables del software cuya variación puede producir una disminución del tiempo de ejecución del programa, en contraprestación a la exactitud de la detección de rostros. Dichos parámetros son el factor de escala: "*scalaFactor*", que recoge la disminución de la resolución de la imagen que se realiza en cada iteración, y el factor "*step*", que indica el desplazamiento por toda la imagen de la ventana de detección en cada iteración.

Función	Descripción
Main	Función principal que procesa una imagen en formato: PGM, y señala con un cuadrado los rostros que encuentra en dicha imagen
readTextClassifier	Lee los ficheros info.txt y class.txt que contiene la posición de cada una de las características (filtros) de cada una de las etapas en una ventana de 24×24 pixel
detectObjects	Detecta los rostros, recibe como parámetros: la imagen de entrada, el tamaño de la ventana de detección, factor de escala y desplazamiento y el objeto cascada
setImageForCascadeClassifier	Establece la ventana de detección en la imagen, y almacena en un vector los puntos de cada una de las características para luego ser evaluados y comparado con los que se marcan en class.txt
ScaleImage_Invoker	Desplaza la ventana de detección por toda la imagen
runCascadeClassifier	Controla el paso por las diferentes etapas que forman el clasificador en cascada
evalWeakClassifier	Evalúa cada una de las características en cada etapa y devuelve el resultado de la suma para que se compare con el umbral
integralImages	Obtiene la imagen integral e imagen integral cuadrática a partir de la imagen de entrada
nearestNeighbor	Escala la imagen usando el método del vecino más próximo. Esta función permite re-escalar la imagen (imagen piramidal)
int_sqrt	Calcula la varianza de la imagen
drawRectangle	Dibuja un cuadrado sobre los rostros detectados
readPgm	Lee la imagen en formato PGM
writePgm	Construye una nueva imagen insertando un cuadrado sobre los rostros detectados

Tabla 5.1: *Funciones principales del programa de detección facial.*

5.3.2 Depuración y optimización secuencial del software

Con la primera versión completa del programa se realizan las primeras pruebas de ejecución para verificar su correcto funcionamiento en la detección de rostros. En este sentido, el programa sigue un procedimiento secuencial en serie en el que las

instrucciones se ejecutan de una en una en el procesador, siguiendo el flujo de instrucciones del algoritmo implementado de principio a fin. Un programa secuencial tiene la ventaja de poder ejecutarse en un mayor número de plataformas tecnológicas, independientemente de su arquitectura. Pero tiene la desventaja de no aprovechar todos los recursos computacionales que puede proporcionar una arquitectura concreta para mejorar el rendimiento del mismo.

Comprobado su correcto funcionamiento para la detección de rostros con diferentes imágenes en formato PGM, se procede a depurar el programa con el fin de detectar posibles problemas de rendimiento o acciones redundantes que pueden ralentizar su ejecución secuencial, de forma que se pueda optimizar el tiempo de ejecución atendiendo a las características de las arquitecturas experimentales que se están usando, es decir la placa Raspberry Pi 2 B y la placa Odroid X4U.

Los métodos que se aplican se pueden resumir en optimizar los accesos a memoria y manejar correctamente el juego de instrucciones del procesador para optimizar el uso de las distintas unidades funcionales. En la medida de lo posible, los caminos recomendados pasan por utilizar bibliotecas estándar de alto rendimiento y usar adecuadamente las mejoras que el compilador realiza automáticamente. Las bibliotecas proveen además facilidad de uso y métodos robustos, evitando a la vez el tener que invertir una gran cantidad de tiempo en el desarrollo de algoritmos que pueden ser secundarios al problema que se está resolviendo.

En este contexto, se realizan diferentes pruebas del software sobre las dos placas experimentales utilizadas en esta investigación, con objeto de ajustar el programa de detección facial a las características del hardware utilizado. Para ello se realizan múltiples ejecuciones del software para que, en base a los mejores resultados obtenidos, optimizar la realización del programa sobre la arquitectura concreta que se está usando.

También se realiza un modelado del tiempo de ejecución a partir de la búsqueda de los parámetros que minimizan dicho tiempo como son el factor de escala "*scalaFactor*" y el factor de desplazamiento "*step*". Todo con la menor pérdida de precisión posible. En este sentido, los valores óptimos para estos parámetros se

establecen en 1.2 para el factor de escala y en 1 para el parámetro de desplazamiento (que representa 1 pixel de desplazamiento de la ventana de búsqueda en cada iteración). En el capítulo 6 se justificará adecuadamente la elección de estos valores para dichos parámetros.

5.4 Análisis de rendimiento. Ejecución secuencial

Claramente el tiempo de ejecución del software desarrollado para el reconocimiento de rostros depende:

a) Del Factor de Escala que se establece, como se ha indicado anteriormente, en 1.2 como óptimo para el reconocimiento de rostros, y que representa una reducción de escala de esa cantidad en cada iteración del programa, hasta que la escala de la imagen analizada sea inferior a la ventana de detección, que tiene una resolución de 24×24 pixeles.
b) Del valor del desplazamiento de la ventana de reconocimiento que se desplaza por toda la imagen,
c) De la resolución de la imagen, cuanta mayor resolución tenga, mayor será el tiempo invertido en su análisis.

En este contexto, para cada uno de los entornos de prueba creados (placa Odroid XU4 y Raspberry Pi 2 B), se ha procedido a la medida de tiempo de ejecución secuencial para una muestra de 10 imágenes en formato PGM, con resolución 640×480 píxeles, factor de escala 1,2 y factor de desplazamiento de la ventana de detección (1 píxel en cada iteración). Además, cada una de las imágenes consideradas contiene un número diferente de rostros y de diferente tamaño que va a permitir ver la evolución del tiempo de ejecución del software en función del número de rostros de la imagen. En la figura 5.9, se pueden ver las imágenes utilizadas en las pruebas experimentales. Cada una contiene 1, 2, 3, 4, 5, 6, 7, 11, 13 y 19 rostros respectivamente.

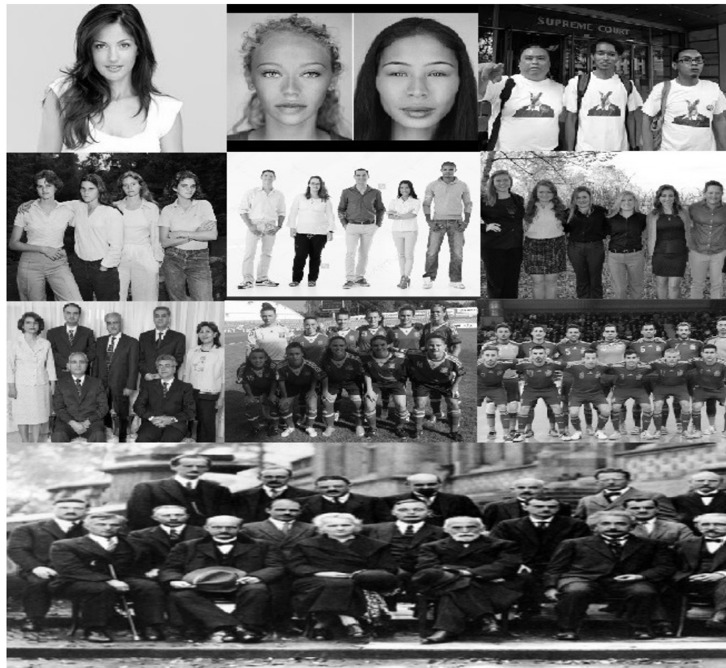


Figura 5.9: *Muestra representativa de las 10 imágenes usadas en las pruebas experimentales.*

La aplicación del programa de detección facial a cada una de las imágenes de la muestra produce los resultados que se muestran en la figura 5.10 para cada una de las placas experimentales usadas, Raspberry Pi 2 B y Odroid XU4.

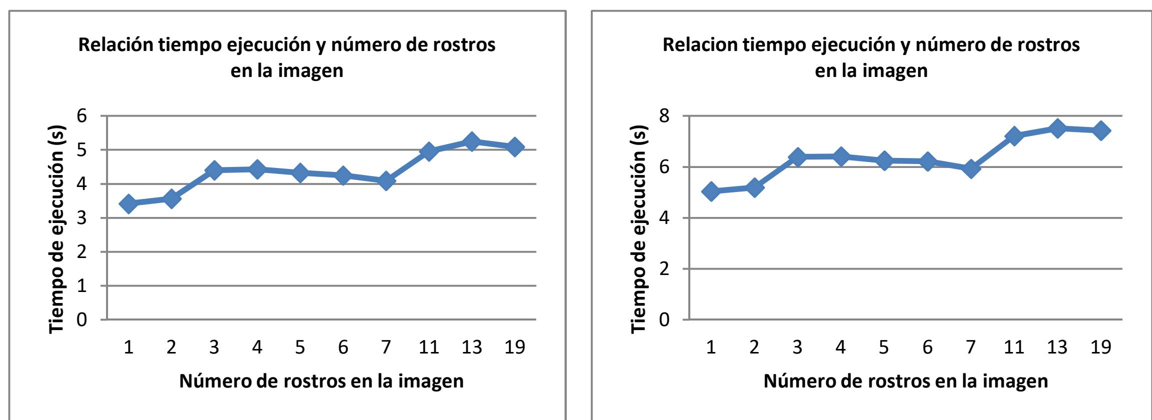


Figura 5.10*: Resultados obtenidos en placa Odroid XU4 (izquierda) y Raspberry Pi 2 B (derecha).*

El análisis de los resultados denota, como cabe esperar, que cuanto mayor número de rostros hay en la imagen analizada mayor tiempo de ejecución se produce, ya que cada rostro detectado requiere el paso por todas las etapas del clasificador en

cascada sin excepción. Pero como se puede apreciar en las gráficas, no siempre es así, no existe una relación directa. Se da el caso en el que una imagen con mayor número de rostros se ejecuta en menor tiempo que otra con menor número de rostros. Así por ejemplo en la figura 5.10, se puede ver que la imagen de muestra que contiene 7 rostros se procesa en menor tiempo que las imágenes con 5 y 6 rostros. Por tanto, tiene que haber otro factor o factores, además del número de rostros de la imagen que afecten al tiempo de ejecución y que permitan explicar la variación temporal en la ejecución de programa.

Tras la realización de diferentes experimentos sobre el conjunto de imágenes de prueba para ver qué otros factores pueden influir en el tiempo de ejecución del sistema de detección facial. Se observa (ver figura 5.11) que una diferencia considerable entre las imágenes que no cumplen la regla de a mayor número de rostros, mayor tiempo de ejecución, son aquellas que tienen un valor de imagen integral N mayor, Este valor es el resultado de sumar todos los pixeles que componen la imagen de prueba, es decir, es el valor del pixel del extremo inferior derecho de la imagen integral obtenida a partir de la imagen de prueba analizada en el sistema de detección facial.

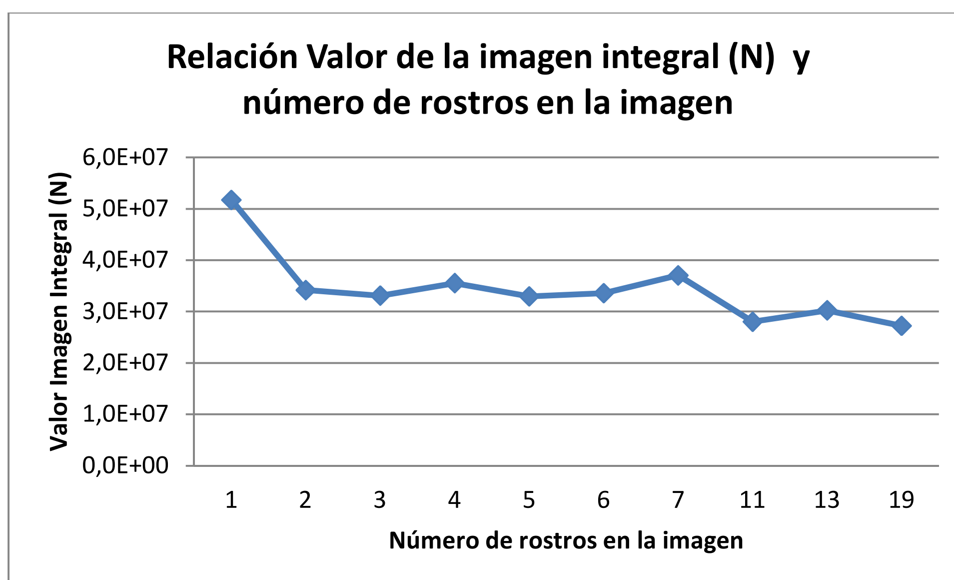


Figura 5.11: *Valor de la imagen integral (N) para cada una de la imágenes de prueba con diferente número de rostros.*

Para ver la influencia del valor de la imagen integral (N) sobre el tiempo de ejecución se ha repetido el experimento con una muestra de 10 imágenes (figura 5.12), con la misma resolución y condiciones que la muestra anterior (formato PGM y 640×480 píxeles de resolución), pero en este caso con un solo rostro, comparándose los valores de las imágenes integrales que tiene cada una. En las figuras 5.13 y 5.14, se puede ver la relación entre el tiempo de ejecución y el valor de la imagen integral de las imágenes de prueba en la placa Odroid XU4 y Raspberry Pi 2 B respectivamente. El número de cada imagen viene dado según el orden de aparición en la figura 5.12, desde la esquina superior izquierda y siguiendo cada una de las filas de imágenes.

Figura 5.12: *Muestra de las 10 imágenes de un solo rostro usadas en las pruebas experimentales.*

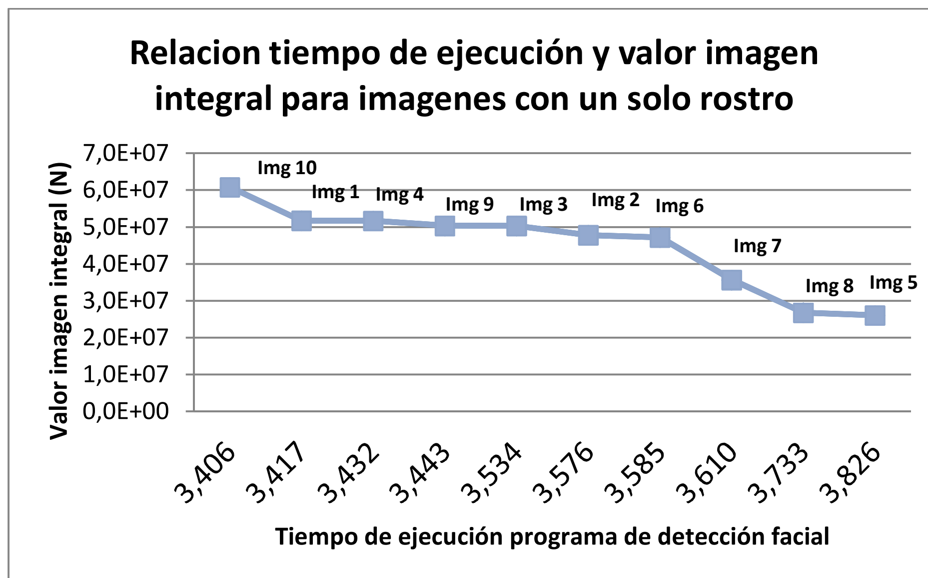


Figura 5.13: *Resultados obtenidos en la placa Odroid XU4.*

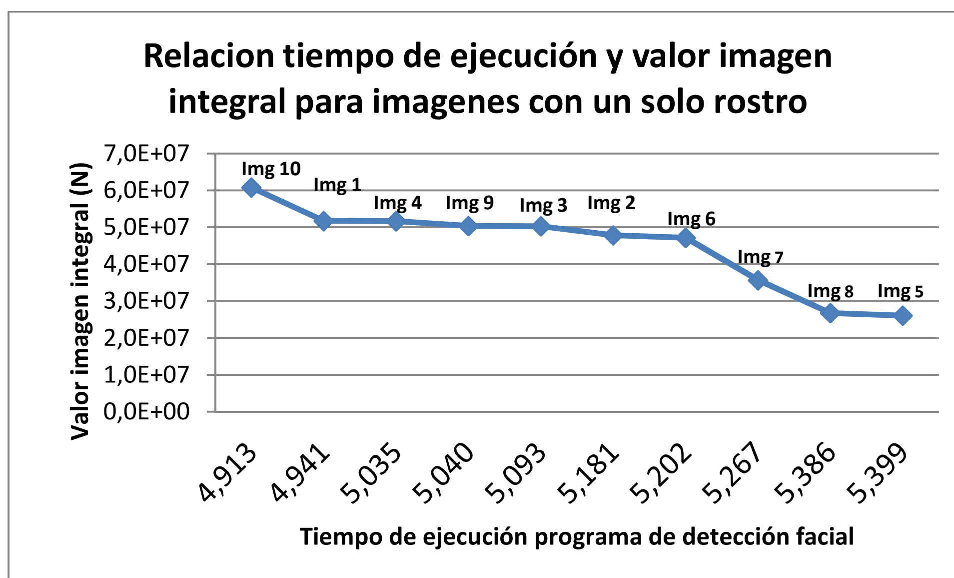


Figura 5.14: *Resultados obtenidos en la placa Raspberry Pi 2 B.*

Como se puede apreciar en las gráficas de resultados de las figuras 5.13 y 5.14, cuanto mayor es el valor de la imagen integral, menor tiempo de ejecución se produce en el sistema de detección facial. Por tanto, se puede ver que existe una relación entre el número de rostros que contiene la imagen, el valor de la imagen integral y el tiempo de ejecución. Esta relación que podemos denominar RIT (Relación entre el valor de la Imagen integral y el Tiempo de ejecución), se muestra en la ecuación 5.3. En las figuras 5.15 y 5.16 se muestra la representación grafica de los resultados de RIT obtenidos para las imágenes de prueba con diferente número de rostros vistas en la figura 5.9, para las placas Odroid XU4 y Raspberry PI 2 B respectivamente.

$$RIT = \frac{\boldsymbol{Tiempo\ ejecución\ *\ Valor\ imagen\ integral}}{\boldsymbol{Número\ de\ rostros\ en\ la\ imagen}} \tag{5.3}$$

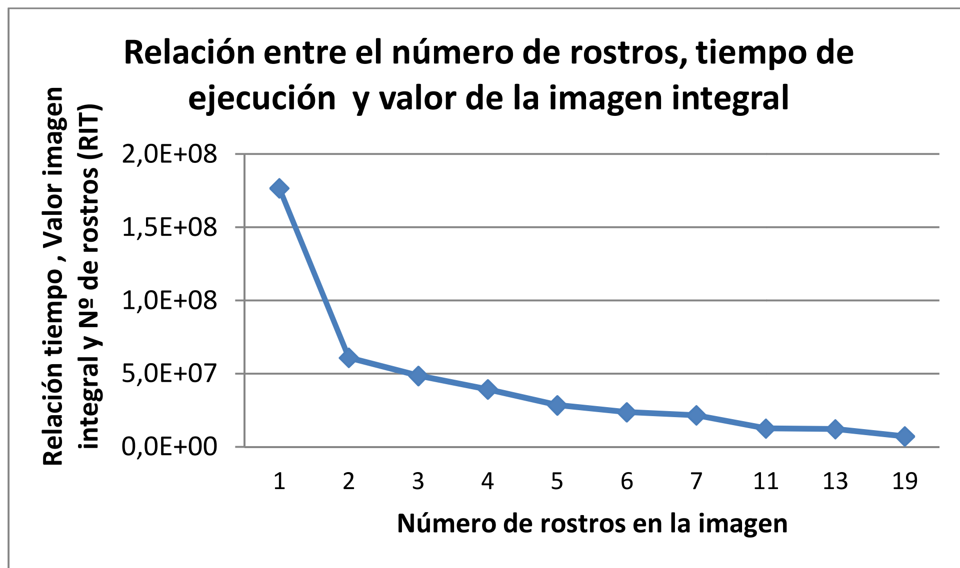


Figura 5.15: *Resultados obtenidos en placa Odroid XU4.*

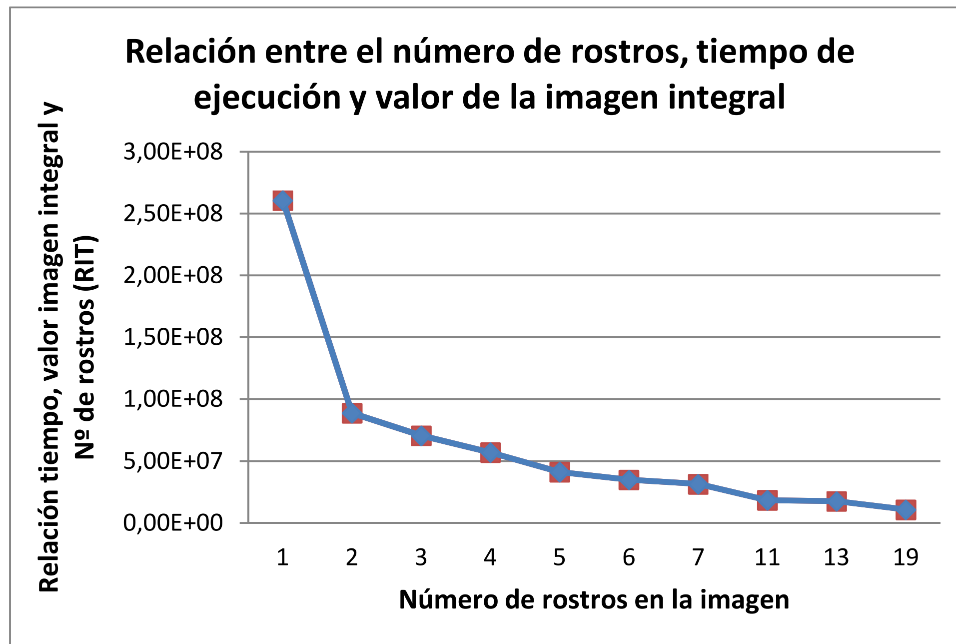


Figura 5.16: *Resultados obtenidos en placa Raspberry Pi 2 B.*

Atendiendo a los resultados, se puede concluir que el tiempo de ejecución suele estar afectado por el número de rostros y el valor de imagen integral que tiene la imagen procesada en el sistema de detección facial. En los casos estudiados las imágenes con un mayor valor de imagen integral suelen ser aquellas que tienen mayores tonalidades de grises, y por tanto, las que presentan un tiempo de ejecución menor. Esto puede ser debido a que en las imágenes con menos tonalidades de grises es más difícil catalogar una determinada zona de la misma como un rostro y por tanto el número de etapas del clasificador en cascada por las que tiene que pasar para llegar a esta conclusión es mayor.

Por tanto, del análisis experimental llevado a cabo en esta investigación se puede interpretar que partiendo de una imagen de entrada al sistema de detección facial en la que hay mayores tonalidades oscuras es más rápido determinar si una zona de la misma no contiene un rostro. Es decir, si se aplica un filtro a la imagen de entrada original que resalte las tonalidades de grises de la misma y con ello incremente el valor de la imagen integral, atendiendo a los resultados obtenidos, se puede conseguir una mejora en el tiempo de ejecución del programa de detección facial.

5.5 Aceleración de la ejecución del programa

Una vez que la aplicación está optimizada para procesamiento secuencial, el tiempo de ejecución puede reducirse aún más permitiendo que se ejecute en varios procesadores en paralelo a partir de un reparto adecuado de las tareas entre los núcleos de la CPU.

Durante el proceso de optimización, el software es modificado o adaptado a la arquitectura destino para conseguir una explotación eficiente de los recursos. En ocasiones se trata de optimizar un recurso de forma individual como puede ser la memoria, la distribución de los procesadores del sistema o el tiempo de ejecución del programa. Desde el punto de vista de la tecnología empleada, en la práctica es posible hablar de diferentes niveles de optimización: optimización a nivel de diseño, a nivel de código fuente, a nivel de compilación, a nivel de ensamblaje, a nivel de tiempo de ejecución, etc.

Este trabajo se centrará sobre todo en el marco de la optimización a nivel de tiempo de ejecución y, en menor medida, en el nivel de diseño y de código fuente. Se está particularmente interesado en desarrollar técnicas que permitan adaptar y ajustar el software a la arquitectura destino con el fin de reducir sus tiempos de ejecución y el consumo de energía. Las peculiaridades de los dos dispositivos experimentales usados en el estudio son muy comunes en la actualidad por su semejanza con las arquitecturas que se suelen utilizar en los teléfonos móviles de última generación.

Con el fin de continuar con el proceso de optimización, el siguiente paso en el análisis del rendimiento del programa secuencial es localizar los puntos donde se consume más tiempo y dónde resulta rentable enfocar los esfuerzos de optimización. Para ello, se utiliza una herramienta de trazado (*profiling*) del software para analizar dónde se produce el mayor consumo de tiempo de la CPU durante la ejecución secuencial del programa.

Las herramientas de *profiling* a menudo son específicas para ciertos grupos de herramientas de compilación, y algunas veces incluso se incluyen con el juego de

herramientas del compilador. En la actualidad, existen varias alternativas gratuitas disponibles para el *profiling* de software C++ en el entorno GNU:

1. Gperftools [Gperf, 2018]: es una herramienta fácil de usar que proporciona resultados correctos para algoritmos paralelos de múltiples subprocesos. Por lo tanto, es una herramienta adecuada para optimizar la paralelización de programas secuenciales.

2. Gprof [Fenla, 2017]: es un clásico del entorno GNU que está disponible con las herramientas del compilador *gcc*. Gprof es útil para el análisis de algoritmos que se ejecutan en CPU de un solo núcleo, pero no es adecuado para subprocesos en sistemas integrados multi-núcleo, por lo que se puede obtener resultados erróneos en algoritmos paralelos de subprocesos múltiples. Gprof podría usarse para la detección inicial de rutinas de puntos calientes antes de la optimización paralela.

3. Valgrind [Valgr, 2017]: es una herramienta de análisis de ejecución en profundidad que funciona simulando un procesador virtual. Este enfoque lo hace, sin embargo, muy lento en un entorno integrado. La simulación virtual también se realiza para un procesador de un solo núcleo, por lo que Valgrind no produce cifras de *profiling* realistas para algoritmos paralelos de múltiples hilos.

En este contexto, se usará Gperftools como herramienta de *profiling* para el análisis de sistema de detección facial objeto de esta tesis doctoral. Dicha herramienta proporcionada por Google opera mediante un muestreo basado en el tiempo. El resultado obtenido usando Gperftools durante la ejecución del programa lanzado sobre la placa Raspberry PI 2 B, se puede ver en la figura 5.17. En la misma se muestran las funciones ordenadas de mayor a menor tiempo de ejecución, y se aprecia que la función más costosa, temporalmente hablando, se encuentra en la parte superior de la lista, después viene la siguiente función más costosa, y así sucesivamente. Cada fila enumera detalles sobre la función que se detectó en el *profiling*, y cuánto tiempo de ejecución representa la misma en el marco del programa de detección facial.

Using local file ./vj.
Using local file vj.prof.
Total: 1018 samples

Time(s)	%Execution	%Total	Funtion
6.50	63.9%	63.9%	evalWeakClassifier
1.98	19.4%	83.3%	runCascadeClassifier
1.36	13.4%	96.7%	int_sqrt
0.18	1.8%	98.4%	integralImages
0.06	0.6%	99.0%	ScaleImage_Invoker
0.06	0.6%	99.6%	nearestNeighbor
0.01	0.1%	99.7%	munmap
0.01	0.1%	99.8%	partition
0.01	0.1%	99.9%	predicate
0.01	0.1%	100.0%	setImageForCascadeClassifier

Using local file ./vj.
Using local file vj.prof.
Total: 1940 samples

Time(s)	%Execution	%Total	Funtion
12.88	66.4%	66.4%	evalWeakClassifier
3.64	18.8%	85.2%	runCascadeClassifier
2.18	11.2%	96.4%	int_sqrt
0.36	1.9%	98.2%	integralImages
0.14	0.7%	99.0%	ScaleImage_Invoker
0.11	0.6%	99.5%	nearestNeighbor
0.06	0.3%	99.8%	setImageForCascadeClassifier
0.01	0.1%	99.9%	munmap
0.01	0.1%	99.9%	partition
0.01	0.1%	100.0%	predicate

Figura 5.17: *Resultados profiling en Odroid XU4 (izquierda) y Raspberry Pi 2 B (derecha).*

Se aprecia que las tres primeras funciones (evalWeakClassifier, runCascadeClassifier, int_sqrt), representan más del 96% del tiempo de ejecución del programa de detección. Por tanto, queda claro que el esfuerzo debe centrarse sobre todo en la optimización de estas tres funciones, ya que reducir el tiempo de ejecución en esas funciones proporcionará una buena reducción global. Para el resto de funciones (representan menos del 4%), se puede concluir que no es rentable optimizar el tiempo de ejecución porque el ahorro sobre el tiempo total será poco significativo.

Del resultado del *profiling* también se concluye, como cabía esperar, que el mayor gasto computacional se realiza en dos funciones. Por un lado, evalWeakClassifier, encargada de evaluar en la imagen de muestra cada una de las características que forman el clasificador fuerte de una etapa del clasificador. Y por otro, runCascadeClassifier, que es la función que controla el paso de la ventana de detección por las diferentes etapas que forman el clasificador en cascada.

Una vez detectadas las funciones más costosas desde el punto de vista temporal, y dado que las dos placas experimentales objeto de estudio son multi-núcleo, una forma de aceleración de la ejecución del programa es a través de la paralelización del sistema de detección facial, de forma que se haga un reparto en paralelo óptimo de tareas entre los núcleos que forman cada una de las CPU de las placas.

Para ello, se hará uso de OpenMP y OmpSs, que son unas APIs para la paralelización de programas en plataformas de memoria compartida. El primero,

además de ser uno de los modelos de programación paralela a nivel de tareas más extendido a día de hoy, permite paralelizar aplicaciones de forma sencilla [Garci 2003] [Ompss, 2018]. Ambos se basan en la inclusión de directivas (o pragmas) similares a las utilizadas en otros modelos de programación paralela. En el caso de OmpSs, está basado en OpenMP, pero tiene la particularidad, a diferencia del primero, de que su planificador de tareas es consciente de la asimetría de la CPU multi-núcleo, lo que va a permitir optimizar el consumo energético, como se verá más adelante, en este tipo de arquitecturas. Sin embargo, dichas directivas se limitan en su mayoría, a la anotación de ciertos bloques de código para informar de su carácter de tareas, es decir, unidades básicas de planificación para los recursos computacionales disponibles.

5.5.1 Sistemas homogéneos y heterogéneos

Antes de empezar a hablar sobre la metodología más adecuada a seguir para lograr la optimización del tiempo de ejecución, se definen los conceptos de sistema homogéneo y heterogéneo, que serán sobre los que se realizarán los experimentos, desarrollando primero la investigación sobre sistemas homogéneos y a continuación adaptándola a sistemas heterogéneos.

Un sistema homogéneo es aquel cuyos elementos de proceso tienen características físicas similares y la red de comunicación es idéntica vista desde los distintos nodos de cómputo. Esto quiere decir que los procesadores son idénticos, con la misma velocidad, memoria, etc., y que el coste de las comunicaciones entre dos procesadores cualesquiera es idéntico. En la categoría de sistema homogéneo entrarán normalmente los multi-computadores y los *clusters* de ordenadores del mismo tipo que utilizan una red común. En este caso el sistema experimental homogéneo que se usa en esta investigación es la Raspberry Pi 2 B.

Un sistema heterogéneo es aquel en que los elementos de proceso son distintos o en la red de comunicación se ven los procesadores de manera asimétrica (con distintas velocidades de transmisión entre pares de procesadores distintos). Algunas veces se considera heterogeneidad en la computación, pero no en la red de comunicación (aunque ésta sea heterogénea), y otras veces se considera heterogeneidad tanto en la

computación como en la red. En la categoría de los sistemas heterogéneos se encuentran los *clusters* de computadores, que pueden estar compuestos por procesadores de características distintas aunque compartan la red de comunicación, o los sistemas distribuidos en los que se conectan varios componentes que pueden ser a su vez *clusters*, supercomputadores, monoprocesadores, e incluso en un mismo nodo, con procesadores de distinto tipo en un mismo chip, como es el caso que se considera en esta tesis doctoral, a través de la placa Odroid XU4.

La mayor parte de los compiladores y las bibliotecas existentes han sido típicamente orientadas y optimizadas para sistemas homogéneos. Sin embargo, gracias a la constante reducción del precio del hardware y a su evolución, actualmente es habitual que los centros de procesamiento de datos e investigación dispongan de sistemas heterogéneos en que los elementos de proceso poseen características computacionales diferentes. Dado que los compiladores y bibliotecas han sido diseñados y optimizados para trabajar sobre entornos homogéneos, cuando se intenta su explotación en entornos heterogéneos se hace necesario un proceso de adaptación adicional. El elevado número de factores y su rango de variabilidad confieren al proceso de optimización en sistemas heterogéneos un nivel de dificultad adicional a la optimización en el caso homogéneo.

Por tanto, interesa desarrollar una metodología lo más general posible, que abarque la heterogeneidad tanto en la computación como en las comunicaciones, y que se pueda extender a los distintos tipos de sistemas mencionados. En este contexto, uno de los problemas más importantes para lograr reducir el tiempo de ejecución del programa es el adecuado reparto de los recursos hardware, lo que se denomina mapeo [Juanp, 2009].

5.5.2 Técnicas de mapeo

El concepto de mapeo es un importante problema en computación en paralelo que ha sido muy estudiado prácticamente desde el comienzo de esta disciplina. A lo largo de la literatura se encuentran ligeras variantes en la formulación del concepto de acuerdo con la aproximación seguida por cada autor. Por ejemplo en [Danmo, 1992] se

plantea que el problema del mapeo puede establecerse como el de ajustar un algoritmo o un problema a los recursos hardware con el fin de optimizar su resolución. El problema suele complicarse por el hecho de que normalmente el tamaño de un problema es mayor que el número de elementos de procesamiento disponibles en la máquina. En esos casos hay que contar con estrategias que permitan particionar la computación y asignarla a los recursos disponibles. Como resultado de la gestión del problema del mapeo podrían resolverse cuestiones como: ¿Qué plataforma es más adecuada para una aplicación dada? o ¿Con qué eficiencia puede una aplicación ser ejecutada sobre una plataforma determinada?

El mapeo de un programa ha sido abordado por diferentes autores a través de diversas técnicas. A continuación, se hace una breve descripción de algunas de las técnicas que se pueden encontrar en la literatura y que mejores resultados han obtenido.

- **Algoritmos Aleatorios**: Se realiza una asignación aleatoria de procesos a procesadores [Juanp, 2009].

- **Algoritmos Round Robin**: Se establecen estrategias de asignación de procesos a procesadores de manera circular. Cuando el último procesador recibe su asignación, se comienza con el primero. La granularidad de la asignación determina en cada iteración de asignación el tamaño de los bloques de procesos. En determinados problemas la búsqueda de la asignación óptima consiste en buscar el tamaño óptimo del bloque [Frana, 2004].

- **Bisección Recursiva**: Se realiza una división recursiva de los procesos en subgrupos de igual carga computacional y que minimicen el volumen de comunicaciones. Implica la existencia de algún mecanismo de evaluación de la carga a priori [Anton, 2003].

- **Algoritmos Heurísticos**: Se utiliza algún algoritmo heurístico que permita optimizar la asignación de procesos a procesadores de acuerdo a algún criterio. Cuando el criterio es el de minimizar el tiempo de ejecución la técnica suele

implicar el uso de alguna función del coste temporal de los procesos y de las comunicaciones asociadas [Moren, 2005].

- **Grafos de precedencia**: Esta aproximación consiste en formar un grafo de dependencias del programa en el que los nodos son procesos y los arcos son sus relaciones de dependencia. Una estimación del tiempo de ejecución puede proporcionarse también. Una vez que el grafo de dependencias ha sido construido, se puede realizar el mapeo sobre la arquitectura destino. Esta aproximación proporciona buenos rendimientos, pero el trabajo necesario para obtener el grafo de dependencias puede ser considerable si no se dispone de herramientas que automaticen su generación. La aproximación implica la obtención del camino crítico en el grafo de dependencias así como los tiempos de ejecución más temprano y más tardío de cada proceso [Grego, 2009]. En la presente tesis doctoral se han utilizado este tipo de grafos como herramientas de representación y evaluación de la asignación de procesos a procesadores.

- **Optimización Analítica**: Se dispone de una función que analiza el coste del algoritmo y prospectivamente se resuelve el problema de encontrar los parámetros que minimizan la ejecución. Presenta la ventaja de ser independiente de la arquitectura destino, pero el inconveniente es la dificultad asociada al proceso de minimización incluso para funciones relativamente simples [Juanp, 2009].

- **Recorridos en árbol:** Se formula el problema de asignación como un problema de tipo enumerativo en el que se consideran todas las posibles soluciones expresadas en forma de árbol de soluciones. Cada rama del árbol representa una posible asignación. El proceso enumerativo suele ir acompañado de funciones de coste que permiten evaluar que solución de las posibles es más óptima en cada nodo del árbol.

Un factor importante que afecta a la asignación es el carácter (homogéneo/ heterogéneo) de los procesadores que componen el sistema. Cuando el sistema es heterogéneo algunas de las estrategias de asignación previamente presentadas, deben ser revisadas para poder ser aplicadas con efectividad.

5.5.3 Mapeo del algoritmo de detección facial

Una metodología típica y muy eficiente para realizar el mapeo de un algoritmo dentro de la técnica de Grafo de Procedencia, descrita en el apartado anterior, es la que se presenta en [Juanp, 2009]. Esta es la técnica que se sigue para el mapeo del programa de detección facial objeto de este trabajo de investigación, que se resume en los puntos que se verán a continuación.

5.5.3.1 Análisis de dependencias y transformación del algoritmo

Este punto de la metodología hace referencia a que el algoritmo se puede transformar desde su forma secuencial hasta la versión paralela. Se trata de un análisis orientado a identificar el paralelismo existente en el programa desarrollado. Para ello, habrá que estudiar la existencia de tareas y el grado de dependencia entre ellas para determinar cuáles son independientes entre sí y pueden ejecutarse de forma paralela haciendo una correcta partición y asignación de tareas. En el caso del programa de detección facial desarrollado de forma secuencial, se observa que cada vez que se desplaza la ventana de detección por la imagen de entrada, se hace una llamada a la función RunCascadeClassifier. Ésta a su vez llama a EvalWeakClassifier que es la función que se encarga de evaluar cada una de las características, y por tanto, la que devuelve el resultado de la comparación de la suma del valor de las características con el umbral establecido en el entrenamiento del sistema. A partir de esta comparación se decide si se rechaza o pasa a la siguiente etapa del clasificador en cascada. En esta parte es donde se produce el mayor gasto computacional.

5.5.3.2 Particionado y asignación de tareas.

El problema del particionado puede formularse como el de dividir un problema en subproblemas más pequeños para su resolución [Juanp, 2009]. En general el problema no es sencillo ya que el particionado podría introducir efectos colaterales no deseados que degraden la estabilidad numérica de los algoritmos.

El particionado especifica también las unidades secuenciales de cómputo en el programa y de aquí la granularidad de la ejecución. Esto va a permitir identificar mejor aquellas partes del programa que se pueden ejecutar de forma independiente a la hora de paralelizarlo. El particionado puede llevar asociado tres posibles formas:

1. Un particionado de datos manteniendo el algoritmo a ejecutar.
2. Un particionado o transformación del código en el que, por ejemplo, se asocian secciones o iteraciones distintas del código con distintos procesos, manteniendo los datos de ejecución.
3. Una combinación de los dos anteriores.

El tipo de particionado viene generalmente impuesto por el tipo de dependencias en el cálculo que produce el algoritmo. Se trata de buscar un equilibrio entre el número de procesos que se generan, habitualmente superior al número de procesadores de que se dispone, y su posterior asignación a los procesadores físicos. En este sentido, hay que tener en cuenta que las comunicaciones debidas al intercambio y a la transferencia de datos y procesos, y las sincronizaciones entre procesos impuestas por las dependencias de datos juegan un papel importante en el tiempo final de ejecución del algoritmo.

Se puede comenzar el particionado del programa a partir de una representación gráfica del mismo llamada DAG (Directed Acyclic Graph, Gráfico Acíclico Dirigido), que viene a ser una representación del paralelismo de tareas asociado al programa de forma que no haya ciclos. Es decir, no hay ninguna manera de empezar en un vértice V y seguir una secuencia de aristas que eventualmente vuelva hacia V otra vez. Idealmente, un planificador de tareas en tiempo de ejecución podría explotar esta información para determinar distintas planificaciones (ordenación de tareas y asignación de las mismas a recursos computacionales disponibles) que satisfacen las dependencias fijadas en el DAG. Por tanto, se trata de una abstracción que proporciona características del rendimiento e ignora el resto de los aspectos, tales como los semánticos. En la figura 5.18, se puede ver el DAG obtenido a partir del análisis del algoritmo de detección facial objeto del trabajo. En el mismo un bloque básico representa a un conjunto de cómputo. Los nodos son los procesos y las aristas las dependencias entre procesos. Para cada iteración del algoritmo es necesario computar la función de coste y seleccionar la partición que minimiza la función de coste.

La función de coste del particionado es una combinación de dos términos: el camino crítico y el *overhead* (sobrecarga). El término asociado al *overhead* se decrementa con el número de iteraciones porque un movimiento hacia una granularidad más gruesa no puede incrementar el *overhead* total.

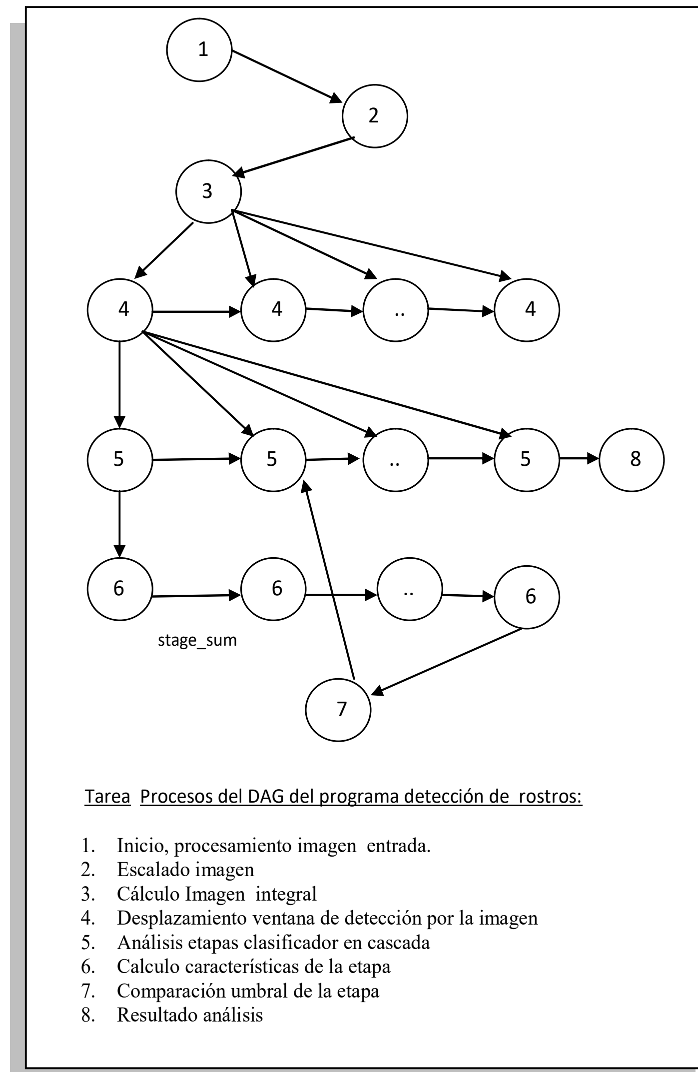


Figura 5.18: *DAG del programa de detección facial [Fuente: elaboración propia].*

Dadas las arquitecturas experimentales de estudio Raspberry PI 2 B, y Odroid XU4, de 4 núcleos simétricos, y 8 núcleos distribuidos en dos *cluster de 4* asimétricos respectivamente, se procede junto al DAG desarrollado a encontrar restricciones y dependencias con el fin de optimizar el código fuente para una adecuada paralelización.

En el sistema de detección facial objeto de esta tesis doctoral, cada etapa tiene dependencia entre sí porque una imagen puede ser rechazada en una etapa y no sería

necesario calcular las siguientes etapas pendientes del clasificador en cascada. Es decir, si se ejecuta cada etapa en tareas paralelas se podrían calcular etapas que no son necesarias. Sin embargo, fijándose en el cálculo de características de una etapa, a primera vista se puede pensar que son independientes, pero existe una variable compartida: “stage_sum”, que produce dependencia ya que se calcula de forma secuencial para obtener la suma acumulada del valor de cada una de las características de la etapa que se está evaluando. Para evitar esta dependencia que limita la paralelización del software, esta variable se puede dividir en partes haciendo uso de una variable tipo *Array*, que contenga tanto elementos como hilos del programa en paralelo se quieran crear. Esta variable sería por ejemplo, “stage_sum_array [numero de hilos]”, y los hilos solo tendrían que sincronizarse al final de la etapa. En las figuras, 5.19 y 5.20, se muestra la adaptación realizada en el programa para optimizar su ejecución en paralelo. En la primera se presenta el código fuente de la versión secuencial y en la segunda, el código adaptado a una paralelización de 4 hilos en la que se puede calcular el valor de 4 características de la etapa en paralelo.

```
for( j = 0; j < stages_array[i]; j++) {  // stages_array[i] es el número de características de la etapa i.
 stage_sum += evalWeakClassifier(variance_norm_factor, p_offset, haar_counter, w_index, r_index);}
```

Figura 5.19: *Fragmento de código para el cálculo secuencial del valor de las características.*

```
for( j = 0; j < stages_array[i]; j+=4) {  // stages_array[i] es el número de características de la etapa i.
stage_sum_array[0]= evalWeakClassifier(variance_norm_factor, p_offset, haar_counter, w_index, r_index);
stage_sum_array[1]= evalWeakClassifier(variance_norm_factor, p_offset, haar_counter, w_index, r_index);
stage_sum_array[2]= evalWeakClassifier(variance_norm_factor, p_offset, haar_counter, w_index, r_index);
stage_sum_array[3]= evalWeakClassifier(variance_norm_factor, p_offset, haar_counter, w_index, r_index);
suma+=stage_sum_array[0]+stage_sum_array[1]+stage_sum_array[2]+stage_sum_array[3];}
```

Figura 5.20*: Fragmento de código para el cálculo en paralelo del valor de las características.*

5.5.3.3 Asignación, planificación y equilibrio de carga

La planificación puede definirse como una función que asigna procesos a procesadores con el objetivo de distribuir la carga sobre todos ellos con la máxima eficiencia, minimizando la comunicación de los datos, lo que lleva a un tiempo de

ejecución total más corto. Las políticas de asignación pueden ser clasificadas como estáticas o dinámicas [Juanp, 2009]:

- **Asignación estática**. Bajo las políticas de asignación estática, los procesos se asignan a los procesadores por el programador o por el compilador antes de su ejecución. No hay sobrecarga en tiempo de ejecución y se incurre en la sobrecarga de asignación una sola vez cuando los programas van a ser ejecutados varias veces sobre diferentes secuencias de datos, aunque en muchas ocasiones, pequeños cambios en las secuencias de entrada obligan a una nueva asignación estática.

- **Asignación dinámica**. Bajo las políticas de asignación dinámica, las tareas se asignan a los procesadores en tiempo de ejecución. Este esquema ofrece mejor utilización de los procesadores, pero a cambio de asignaciones adicionales a los procesadores. La asignación dinámica puede ser distribuida o centralizada. En la asignación distribuida, hay una bolsa de tareas y cualquier procesador libre puede tomar procesos de esa bolsa. Con la asignación centralizada, pueden aparecer cuellos de botella cuando el número de procesadores crece.

Finalmente, en la fase de ejecución puede ocurrir, además, que algunos procesadores completen sus tareas antes que otros porque no se ha realizado un correcto balanceo de carga de trabajo entre los mismos. El reparto equilibrado de tareas entre los procesadores que forman la CPU, se conoce como equilibrio de carga y está vinculado con las diferentes fases que hay que considerar en el mapeo de un programa.

Para el caso de la presente investigación, después de realizar un particionado acorde al número de núcleos y a las arquitecturas experimentales utilizadas, se procede a realizar una asignación dinámica de tareas a través del planificador de tareas (*runtime*) que proporcionan los modelos de programación en paralelos que se van a emplear. Es decir, será el runtime el que hará la asignación dinámica de tareas para la paralización del sistema de detección facial dado los buenos resultados obtenidos, por ejemplo, por el planificador de tareas de OpenMP [Openm, 2017].

5.5.4 Modelos de programación en paralelo

Tras el análisis del mapeo y la adecuación del programa de detección facial, hay que llevar a cabo la paralelización del programa de detección facial en las arquitecturas experimentales estudiadas. Para ello se indicarán brevemente los modelos más importantes para la programación en paralelo, poniendo la atención en el modelo que ofrece OpenMP (Open Multi-Processing), que, como ya se ha mencionado,es uno de los que se utiliza en esta tesis doctoral.

El objetivo principal de la paralelización, no es otro que lograr completar un trabajo mucho más rápidamente, al desempeñar múltiples tareas simultáneamente en arquitecturas de computadoras que permiten procesamiento en paralelo.

Desde la perspectiva del sistema operativo, hay dos medios importantes de conseguir procesamiento paralelo: múltiples procesos y múltiples hilos. Un proceso es un programa en ejecución bajo control de un sistema operativo con un conjunto de recursos asociados. Estos recursos incluyen, entre otros, estructuras de datos con información del proceso, un espacio de direcciones virtuales conteniendo las instrucciones del programa y datos, y al menos un hilo de ejecución. Un hilo es un camino de ejecución o flujo de control independiente dentro de un proceso, compuesto de un contexto (que incluye un conjunto de registros) y una secuencia de instrucciones a ejecutar. Según esto, se puede considerar el procesamiento paralelo en tres categorías:

- **Paralelismo de procesos:** usar más de un proceso para desempeñar un conjunto de tareas.
- **Paralelismo de hilos:** usar múltiples hilos dentro de un único proceso para ejecutar un conjunto de tareas.
- **Paralelismo híbrido:** usar múltiples procesos, donde al menos uno de ellos es un proceso paralelo con hilos, para desempeñar un conjunto de tareas.

Aunque se puede obtener paralelismo con múltiples procesos, existen al menos dos razones potenciales para considerar paralelismo de hilos, que son la conservación de recursos del sistema y la ejecución más rápida. Los hilos comparten acceso a los datos del proceso, archivos abiertos y otros atributos del proceso. Compartir datos e

instrucciones puede reducir los requerimientos de recursos para un trabajo en particular. Por el contrario, una colección de procesos a menudo deberá duplicar las áreas de datos e instrucciones en memoria para un trabajo. La administración de hilos es más simple que la de los procesos, ya que los hilos no poseen todos los atributos de un proceso. Pueden ser creados y destruidos mucho más rápidamente que un proceso.

En este sentido, las técnicas más usadas comúnmente para paralelización de programas son:

1. El uso explícito de hilos, donde un hilo es una entidad en tiempo de ejecución que es capaz de ejecutar independientemente un flujo de instrucciones.
2. El uso de directivas al compilador, que para conseguir paralelismo tiene el fin de aliviar la complejidad y los problemas de la portabilidad a otros sistemas. En el paralelismo orientado a directivas, la mayoría de los mecanismos paralelos se ponen en marcha a través del compilador (generación de hilos, generación de construcciones de sincronización, etc.). El compilador traduce directivas del programa en llamadas al sistema necesarias para la administración de los hilos, y realiza cualquier reestructuración de código que sea necesaria.
3. El paso de mensaje, hace referencia a interfaces de E/S para la comunicación entre procesos que se ejecutan en paralelo.

Para la programación en paralelo se han creado diferentes lenguajes de programación, APIs y bibliotecas, entre las que cabe destacar la API denominada MPI (*Message Passing Interface*) que hace uso de la técnica de paso de mensajes y se suele usar mayoritariamente sobre arquitecturas con memoria distribuida (figura 5.21), de ahí que MPI está destinado a paralelismo de procesos, mientras que en las arquitecturas de memoria compartida dos de las APIs más utilizadas son POSIX Threads y OpenMP, orientadas al paralelismo de hilos.

En computación de alto rendimiento hay herramientas que ayudan a la programación con multi-hilos (*multi-threads*) en paralelo, procesamiento en memoria distribuida y plataformas de multiprocesadores de memoria compartida (figura 5.21). En

plataformas de multiprocesadores de memoria distribuida, cada procesador (P_i) tiene su propia memoria (M_i) y su contenido no está fácilmente disponible a otros procesadores. En estos casos, la información entre procesadores se comparte usualmente por el paso de mensajes usando los estándares de bibliotecas de paso de mensajes como MPI.

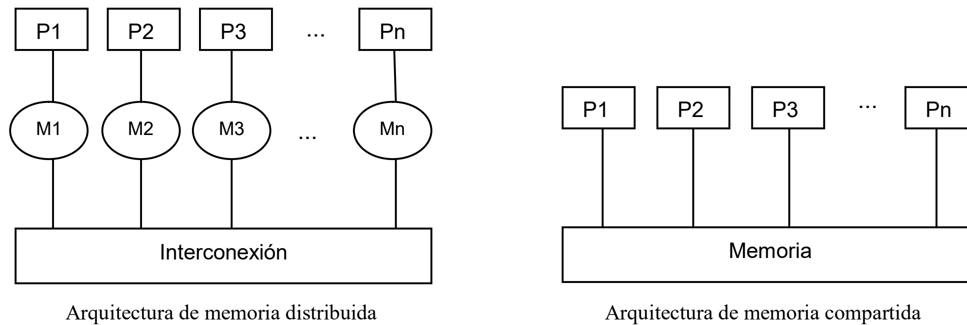


Figura 5.21: *Diferencia entre arquitecturas de memoria distribuida y compartida [Fuente: elaboración propia].*

En multiprocesadores como son el caso de la placa Raspberry Pi 2 B y Odroid XU4, la memoria entre procesadores se comparte. En estos casos, en la paralelización de procesos se suelen usar programas basados en directivas, como OpenMP que ha sido desarrollado específicamente para procesamiento de memoria compartida en paralelo. Por tanto, se usará OpenMP como paradigma de memoria compartida para llevar a cabo la paralelización del sistema de detección facial. Para este trabajo de tesis doctoral se emplea el término procesamiento paralelo para indicar que hay más de un hilo ejecutándose en un único programa. Esta definición admite la implementación de procesamiento paralelo con más de un proceso.

5.5.5 Metodología para la paralelización del programa detección facial

El primer paso para la paralelización, es elegir un modelo, identificar que partes del programa deben ser paralelizadas y determinar cómo dividir la carga de trabajo computacional entre los diferentes procesadores. Dividir la carga de trabajo computacional es crucial para el rendimiento, ya que determina el coste en

comunicación, sincronización y de desequilibrios de carga resultantes en un programa paralelizado.

En esta tesis doctoral se elige OpenMP como estándar de programación paralela, ya que esta específicamente desarrollado para sistemas que usan memoria compartida, como es el caso de las dos placas experimentales objeto de esta investigación. En el siguiente apartado se verá brevemente algunos detalles interesantes de OpenMP, indicando las principales ventajas que han permitido seleccionarlo como la mejor opción para llevar a cabo la paralelización del sistema de detección facial.

Después de seleccionar un modelo de paralelización, lo siguiente es optimizar su rendimiento, que se hace de forma similar a la optimización secuencial. Este proceso es iterativo e involucra mediciones repetidas seguidas de aplicar una o más técnicas de optimización para mejorar el rendimiento del programa. Las aplicaciones paralelas, sin importar el modelo utilizado, necesitan que exista una comunicación entre los procesos o hilos concurrentes. Se debe tener cuidado de minimizar los costes extras en comunicación y asegurar una sincronización eficiente en la implementación, y minimizar el desequilibrio de cargas entre las tareas paralelas, ya que esto degrada la escalabilidad del programa. También se deben considerar temas como la migración, la programación de procesos y la coherencia de caché.

Los cuellos de botella de un programa paralelo pueden ser muy diferentes de los presentes en una versión secuencial del mismo programa. Además de costes extras específicos de la paralelización, las partes secuenciales de un programa paralelo pueden limitar severamente la ganancia de velocidad de la paralelización si no están adecuadamente sincronizadas.

5.5.6 Open multi-processing (OPENMP)

OpenMP nace del trabajo conjunto de importantes fabricantes de hardware y software [Openm, 2017]. Se trata de una Interfaz de Programación de Aplicaciones (API), para paralelizar el funcionamiento de algoritmos que provee un modelo portable

y escalable para el desarrollo de aplicaciones paralelas de memoria compartida. Ofrece al usuario un modelo de programación de paralelismo explícito compuesto por un conjunto de directivas de compilador o “pragmas” que el usuario utiliza para especificar qué regiones de código van a ser paralelizadas, rutinas de bibliotecas de tiempo de ejecución, y variables de entorno para especificar paralelismo de memoria compartida. El usuario también cuenta con un limitado número de variables de entorno que permiten especificar, entre otros aspectos, las condiciones que definen la ejecución de la aplicación paralela.

OpenMP no es un lenguaje de programación, sino que es una notación que puede ser agregada a un programa secuencial en Fortran, C o C++ para describir cómo el trabajo debe ser compartido entre los hilos que se ejecutarán en diferentes procesadores o núcleos, y para organizar el acceso a los datos compartidos cuando sea necesario. La inserción apropiada de las características de OpenMP en un programa secuencial permitirá a muchas de las aplicaciones beneficiarse de una arquitectura de memoria compartida, a menudo con mínimas modificaciones al código. Uno de los factores del éxito de OpenMP es que es comparativamente sencillo de usar, ya que el trabajo más complicado para poner en marcha el programa paralelo se deja al compilador. Tiene además la gran ventaja de ser ampliamente adoptado, de manera que una aplicación OpenMP va a poder ejecutarse en muchas plataformas diferentes.

Las directivas de OpenMP permiten al usuario indicarle al compilador qué instrucciones ejecutar en paralelo y cómo distribuir las mismas entre los hilos que van a ejecutar el código. Las directivas son instrucciones en un formato especial que es entendido por compiladores OpenMP solamente [Opeor, 2016]. De hecho aparece como una directiva “pragma” para un compilador C/C++, de manera que el programa puede ejecutarse como lo hacía previamente si el compilador no conoce OpenMP. Pero no siempre es posible obtener alto rendimiento con una inserción sencilla e, incremental de directivas OpenMP en un código secuencial. Por esta razón OpenMP incluye varias características que habilitan al programador a especificar más detalle en el código paralelo.

La implementación de OpenMP utiliza, mediante un modelo Fork-Join (ver figura 5.22), múltiples hilos para obtener el paralelismo. De este modo, al encontrarse una directiva paralela que así lo indique, el hilo maestro se divide en un número determinado de hilos esclavos que se ejecutan concurrentemente, distribuyéndose las tareas entre ellos [David, 2013]. Dependiendo de los parámetros indicados, carga de la máquina y otros factores, el entorno de ejecución asignará uno o varios de los hilos esclavos a los distintos procesadores disponibles.

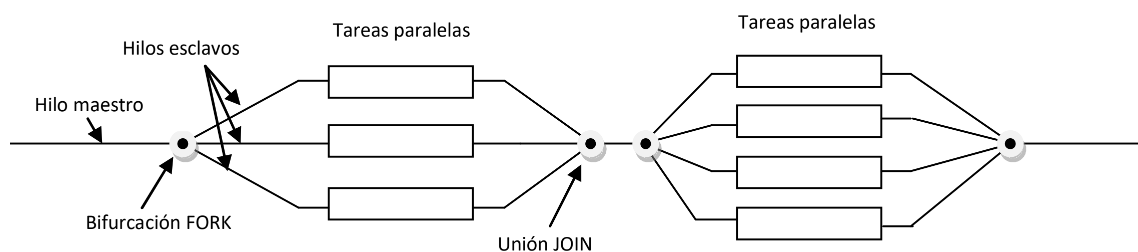


Figura 5.22: *Modelo FORK – JOIN [Fuente: elaboración propia].*

A modo de resumen se puede decir que OpenMP implementa el paradigma de paralelismo de datos. Se compone de tres componentes principales, que son:

- Un conjunto de directivas de compilación usado por el programador para comunicarse con el compilador para la paralelización del programa.
- Una librería de funciones en tiempo de ejecución que habilita la colocación de las directivas en el programa y requiere los parámetros paralelos que se van a usar, tal como número de los hilos que van a participar y el número de cada hilo.
- Un número limitado de variables de entorno que pueden ser usadas para definir en tiempo de ejecución parámetros del sistema en paralelo tales como el número de hilos.

Las principales formas de aprovechamiento para asignación de trabajo en hilos en OpenMP son:

- **Bucles Paralelos**. También llamado paralelismo de "grano fino", en el que el programa reparte la carga usando el índice del bucle.
- **Regiones Paralelas**. También llamado paralelismo de "grano grueso". En éste el programador reparte la carga en bloques de trabajo. Cualquier sección del código puede ser paralelizado. En este caso, el trabajo es distribuido explícitamente en cada uno de los hilos usando el identificador único que tiene cada hilo.

Según esto, OpenMP es adecuado principalmente en aquellos problemas donde se desea aplicar la misma función o conjunto de instrucciones a diferentes partes (disjuntas) de un universo de datos. Un caso típico de este tipo de problemas es el que se da cuando se trata de ejecutar instrucciones repetitivas o sobre listas numeradas del tipo:

```
for (i=inicio; i<fin; i++)
    x[i] = función(a[i])
```

Es decir, cuando el número de elementos sobre los que se opera es conocido se implementan normalmente con la instrucción *for*. Los diseñadores de OpenMP implementaron una operación especial para las construcciones *for* de los lenguajes de programación. De esta forma el programador no tiene que cambiar la semántica de sus programas, ya que el planificador de tareas de OpenMP se encargará de dividir los índices y asignarlos a los distintos hilos generados.

En resumen, se puede decir que la elección de la API para la paralelización del sistema de detección facial se debe fundamentalmente a que tiene un amplio apoyo de los grandes fabricantes de Hardware y Software. Por ello, de una forma similar al logro de MPI como el estándar para procesamiento de memoria distribuida en paralelo, OpenMP ha surgido como el estándar para la computación en paralelo en sistemas de memoria compartida. En la figura 5.23, se pueden ver las principales ventajas que determinan su elección en el desarrollo del presente trabajo de investigación, además de algunas desventajas, que también conviene conocer a la hora de hacer uso del mismo.

Ventajas de OpenMP:	**Desventajas de OpenMP:**
• Es altamente escalable, el rendimiento suele mejorar cuando se ejecuta sobre un número mayor de procesadores. • Por la forma como se diseñó, el código paralelo se mantiene igual al secuencial, lo que permite ejecuciones tanto en uno como en muchos procesadores sin necesidad de recompilar. • No requiere el control del programador sobre la sincronización (creación y eliminación de hilos).	• No tiene soporte para operaciones de bajo nivel para el manejo de memoria compartida. • El tiempo de arranque de los hilos es alto comparado con la ejecución de un hilo secuencial, por lo que en ocasiones (si el tiempo de ejecución del código es muy bajo) la implementación OpenMP resulta más ineficiente. • No tiene un manejo de errores eficiente, lo que dificulta el proceso de depuración.

Figura 5.23: *Ventajas y desventajas de OpenMP.*

En apéndice A, de esta tesis doctoral se describe como compilar programas en OpenMP. Además, de las principales directivas y cláusulas complementarias.

5.6 Resultados de la paralelización del algoritmo de detección facial

Como ya se vio en la figura 5.17 en el *profiling* del programa de detección facial, la función que más tiempo consume es "evalWeakClassifier". Esta función evalúa el valor de las características débiles que forman un clasificador fuerte, y es llamada desde la función: "runCascadeClassifier", que es la encargada de lanzar la cascada de clasificadores para cada una de las ventanas de detección (24×24) que se desplazan por toda la imagen de entrada. En la figura 5.24, se puede ver la parte del código fuente perteneciente a la función "scaleImage_Invoker", desde donde se llama a la función "runCascadeClassifier". Esta parte representa el desplazamiento de la ventana de detección o filtro, de 24×24 píxeles, por toda la imagen. En cada posición de dicha ventana se lanzan el proceso de detección facial a través de la función "runCascadeClassifier". El número de llamadas a esta función vendrá dado por el tamaño en píxeles de la imagen, el factor de escala y el factor de desplazamiento que se realiza en cada iteración.

```
//step indica el número de pixel de desplazamiento de la ventana de filtro por la imagen
//x2= margen en anchura hasta donde se puedes desplazar la ventana de filtro
//y2= margen en altura de hasta donde se puedes desplazar la ventana de filtro
step=1;
for( x = 0; x <= x2; x += step ) desplazamos la ventana de detección por columnas
   for( y = 0; y <= y2; y += step ) //desplazamos la ventana de detección por filas
    {
       p.x = x; //coordenada x donde comienza la ventana de filtro
       p.y = y; //coordenada y. La ventana de filtro es de 24x24 píxeles

       result = runCascadeClassifier( cascade, p, 0 );

       if( result > 0 )  //Sí se cumple se trata de un rostro y dibujo rectagulo en la zona
             {
              // factor= factor de escala de la imagen de entrada
              MyRect r = {myRound(x*factor), myRound(y*factor), winSize.width, winSize.height};
              vec->push_back(r);
             }
    }
```

Figura 5.24: *Fragmento de código fuente donde se llama a la función: runCascadeClassifier.*

Del análisis de código se puede deducir que, dado que se ejecuta la misma cascada de clasificadores cada vez que la ventana de detección se desplaza por la imagen, en este punto existe una oportunidad de optimización de software a través de la paralelización con OpenMP. A partir de las directivas de optimización especialmente diseñadas para los bucles *for*, donde en cada ciclo del bucle se ejecutan tareas similares pero independientes unas de otras (cada ciclo del bucle representa una ventana de detección diferente, sin dependencia entre ellas), se pueden establecer varios hilos de ejecución en paralelo, lo que permite evaluar varias ventanas de detección al mismo tiempo, con la consiguiente mejora temporal en la ejecución del software respecto a la ejecución secuencial del mismo.

La mayoría de las construcciones en OpenMP son directivas de compilación o pragmas, que atendiendo al lenguaje de programación utilizado siguen una estructura como la que se muestra en la tabla 5.2.

Lenguaje	Centinela	Directiva	Cláusula
C/C++	#pragma omp	Justo después del centinela y antes de las opciones	Opciones aplicables a la directiva
Fortran	!$OMP C$OMP *$OMP	Justo después del centinela y antes de las opciones	Opciones aplicables a la directiva

Tabla 5.2: *Estructura de OpenMP en C/C++ y Fortran.*

Las directivas son tomadas por comentarios por aquellos compiladores que no están preparados para interpretarlas. En el caso del programa de detección facial se utiliza el constructor de OpenMP: "Parallel", que define una región del programa que puede ser ejecutada por múltiples hilos en paralelo.

Por tanto, para optimizar el código, solo es necesario usar las directivas #pragma de OpenMP: "**#pragma omp parallel for**". Esta directiva le indica al compilador que divida la ejecución del bucle en varios hilos en paralelo, lo que permitirá ejecutar al mismo tiempo cada hilo en un núcleo de CPU. Después de esto el programa seguirá ejecutándose en un solo hilo. En la figura 5.25, se puede ver como quedaría el código mostrado en la figura 5.24, al añadir las directivas OpenMP.

```
#pragma omp parallel for
for( x = 0; x <= x2; x += step )
#pragma omp parallel for
  for( y = y1; y <= y2; y += step ) //desplazamos la ventana de detección por columnas
   {
        p.x = x; //coordenadas donde comienza la ventana de detección
        p.y = y; //la ventana de detección es de 24x24 pixels

        result = runCascadeClassifier( cascade, p, 0 );

        if( result > 0 )  //Sí se cumple se trata de un rostro y dibujo rectagulo en la zona
         {
          // factor= factor de escala de la imagen de entrada
          MyRect r = {myRound(x*factor), myRound(y*factor), winSize.width, winSize.height};
          vec->push_back(r);
         }
```

Figura 5.25: *Fragmento de código fuente con las Directivas #pragma incluidas.*

Se puede encontrar otra oportunidad de paralelización en la función "runCascadeClassifier", en la parte de código desde donde se produce la llamada a la función "evalWeakClassifier". En la figura 5.26, se puede ver una parte de código fuente, adaptado para la paralización realizada en el apartado 5.5.3.2, donde se realizan las llamadas a dicha función.

```
For ( j = 0; j < stages_array[i]; j+=2)   // stages_array[i] = número de características de la etapa i
 {
     stage_sum_array[0]= evalWeakClassifier(variance_norm_factor, p_offset, haar_counter, w_index, r_index);
     stage_sum_array[1]= evalWeakClassifier(variance_norm_factor, p_offset, haar_counter, w_index, r_index);
     stage_sum_array[2]= evalWeakClassifier(variance_norm_factor, p_offset, haar_counter, w_index, r_index);
     stage_sum_array[3]= evalWeakClassifier(variance_norm_factor, p_offset, haar_counter, w_index, r_index);
     stage_sum+=stage_sum_array[0]+stage_sum_array[1]+stage_sum_array[2]+stage_sum_array[3];

    // stage_sum = suma de los valores de los clasificadores debiles de la etapa
    // stages_thresh_array[i] = umbral de la etapa eveluada
   if( stage_sum  > stages_thresh_array[i] )  //si stage_sum supera el  umbral  la etapa del clasificador es satisfecha
   {
           return  1;
   }
}
```

Figura 5.26: *Fragmento de código fuente de la función "runCascadeClassifier" donde se llama a la función "evalWeakClassifier".*

En este caso se puede mejorar el rendimiento del programa de detección facial, agregando nuevamente la directiva "**#pragma omp parallel for**" antes de bucle *for***.** Sin embargo, una vez que el contenido del bucle *for* se ejecute en varios subprocesos y núcleos paralelos simultáneamente, es posible que varios subprocesos intenten modificar la variable compartida "stage_sum", que determina el valor del clasificador fuerte de la etapa i, que se está evaluando en ese momento. Esto se denomina conflicto de acceso a múltiples subprocesos, y se produce cuando se intenta modificar valores de variables compartidas al mismo tiempo por dos subprocesos, lo cual puede causar resultados impredecibles.

Lo que normalmente se produce cuando varios subprocesos pueden modificar los valores de las variables compartidas simultáneamente es que los resultados de los cálculos producidos no son correctos, y los mismos pueden ser impredecibles, por lo que el mismo algoritmo puede producir resultados aleatoriamente diferentes en tiempos de ejecución, incluso con los mismos parámetros de entrada.

Para evitar posibles conflictos de acceso a variables compartidas entre múltiples subprocesos y garantizar un resultado coherente, se necesita usar una directiva de OpenMP, para controlar el acceso a dichas variables compartidas. Esta directiva es "#pragma omp critical", y garantiza que la "sección crítica" identificada, la que sigue inmediatamente después de la directiva, se ejecute por un solo hilo evitando de este modo los posibles errores mencionados. Es decir, si varios subprocesos alcanzaran la sección crítica simultáneamente, solo un subproceso ejecutará el código en la sección crítica a la vez, de modo que todos los demás subprocesos que deseen acceder a ella se pausarán y esperarán su turno hasta que el subproceso anterior haya salido del bloque de la sección crítica.

La aplicación de la directiva de sección crítica hay que hacerla con precaución para evitar la degradación del rendimiento, ya que ésta puede hacer que los hilos paralelos se detengan y esperen. Si los subprocesos paralelos estuvieran demasiado tiempo esperando antes de una sección crítica, el código resultante posiblemente no se ejecutaría más rápido que empleando una ejecución secuencial, o, en el peor de los casos, incluso podría ser más lento. Por esta razón, se agrega en el código fuente que se muestra en la figura 5.27, una verificación adicional "if" antes de la sección crítica, con ello se consigue que solo se pueda entrar a la sección crítica solo cuando sea necesario modificar el valor de la variable compartida.

```
#pragma omp parallel for
For ( j = 0; j < stages_array[i]; j+=2)   // stages_array[i] = número de características de la etapa i
 {
     stage_sum_array[0]= evalWeakClassifier(variance_norm_factor, p_offset, haar_counter, w_index, r_index);
     stage_sum_array[1]= evalWeakClassifier(variance_norm_factor, p_offset, haar_counter, w_index, r_index);
     stage_sum_array[2]= evalWeakClassifier(variance_norm_factor, p_offset, haar_counter, w_index, r_index);
     stage_sum_array[3]= evalWeakClassifier(variance_norm_factor, p_offset, haar_counter, w_index, r_index);
     stage_sum+=stage_sum_array[0]+stage_sum_array[1]+stage_sum_array[2]+stage_sum_array[3];

    // stage_sum = suma de los valores de los clasificadores debiles de la etapa
    // stages_thresh_array[i] = umbral de la etapa eveluada
    if( stage_sum  > stages_thresh_array[i] )  //si stage_sum supera el  umbral  la etapa del clasificador es satisfecha
    {
          #pragma omp critical
          if ( stage_sum  > stages_thresh_array[i] )  return  1;
    }
}
```

Figura 5.27: *Fragmento de código fuente de la función "runCascadeClassifier" con las Directivas #pragma incluidas.*

Si con estos cambios se compila y se ejecuta el software en el entorno de referencia y en las mismas condiciones de procesamiento, se obtienen los resultados de tiempo de ejecución que se muestran en la tabla 5.3.

		Placa Odroid XU4		Placa Raspberry Pi 2 B	
		Ejecución secuencial	Ejecución con directivas OpenMP	Ejecución secuencial	Ejecución con directivas OpenMP
Imagen 1 1 rostro	**Real time**	3,476s	0,845s	5,040s	1,818s
	User time	3,450s	5,845s	5,010s	5,230s
	System time	0,020s	0,035s	0,020s	0,040s
Imagen 2 2 rostros	**Real time**	3,566s	0,845s	5,195s	1,868s
	User time	3,530s	5,850s	5,130s	5,290s
	System time	0,035s	0,030s	0,000s	0,070s
Imagen 3 3 rostros	**Real time**	4,405s	0,838s	6,390s	2,323s
	User time	4,380s	6,095s	6,350s	6,590s
	System time	0,020s	0,055s	0,030s	0,030s
Imagen 4 4 rostros	**Real time**	4,434s	0,907s	6,414s	2,372s
	User time	4,385s	6,295s	6,370s	6,650s
	System time	0,030s	0,050s	0,020s	0,060s
Imagen 5 5 rostros	**Real time**	4,327s	0,893s	6,248s	2,025s
	User time	4,320s	6,155s	6,200s	5,570s
	System time	0,015s	0,075s	0,040s	0,040s
Imagen 6 6 rostros	**Real time**	4,250s	0,989s	6,220s	2,251s
	User time	4,230s	6,940s	6,120s	6,380s
	System time	0,015s	0,030s	0,060s	0,060s
Imagen 7 7 rostros	**Real time**	4,092s	0,921s	5,928s	2,266s
	User time	4,070s	6,350s	5,870s	6,100s
	System time	0,015s	0,050s	0,000s	0,050s
Imagen 8 11 rostros	**Real time**	4,957s	0,981s	7,214s	2,548s
	User time	4,950s	6,805s	7,120s	7,310s
	System time	0,010s	0,070s	0,030s	0,140s
Imagen 9 13 rostros	**Real time**	5.251s	0,946s	7,520s	2,565s
	User time	5.220s	6,640s	7,460s	7,710s
	System time	0.025s	0,060s	0,030s	0,030s
Imagen 10 19 rostros	**Real time**	5.093s	1,069s	7,432s	2,572s
	User time	5,065s	7,120s	7,390s	7,550s
	System time	0,025s	0,045s	0,030s	0,080s

Tabla 5.3: *Resultados de tiempo de ejecución obtenidos para la ejecución secuencial y paralela del programa de detección facial en las placas Odroid XU4 y Raspberry Pi 2.*

En la figura 5.28 se puede ver gráficamente el tiempo de ejecución en relación a número de rostros de la imagen de entrada aplicando las directivas OpenMP y en la figura 5.29, la comparación entre los tiempos de ejecución antes y después de aplicar las directivas OpenMP. En la misma se puede apreciar cómo se ha reducido el tiempo de ejecución de forma directa al número de núcleos que tiene cada una de las placas de estudio.

Examinando los resultados de la tabla 5.2 y de las figuras 5.28 y 5.29, se observa que, en el caso de la Raspberry Pi 2 B que tiene cuatro núcleos de CPU, la mejora del tiempo de ejecución es algo más del doble, es decir, no se ha producido una reducción proporcional al número de núcleos disponibles en el dispositivo como era de esperar.

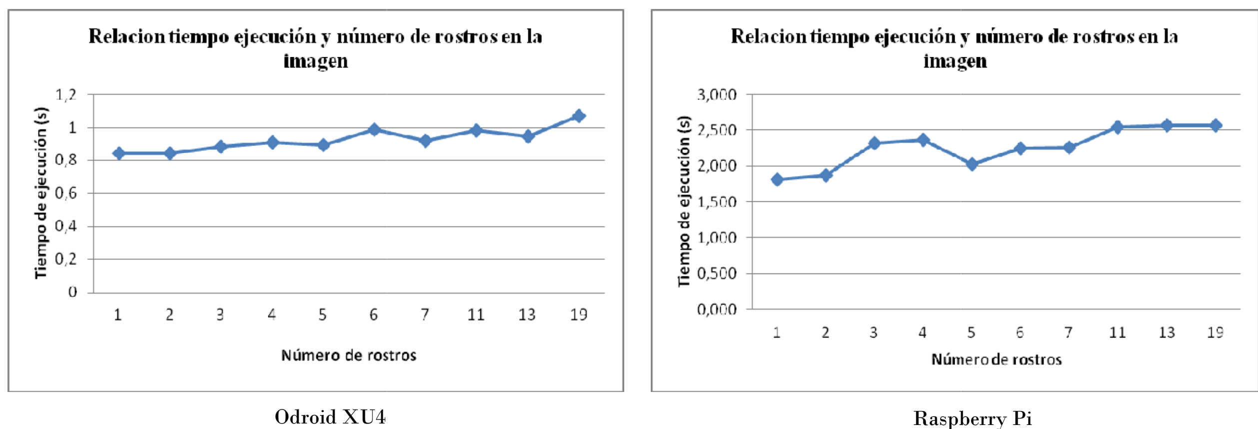


__Figura 5.28:__ Tiempo de ejecución al aplicar directivas OpenMP.

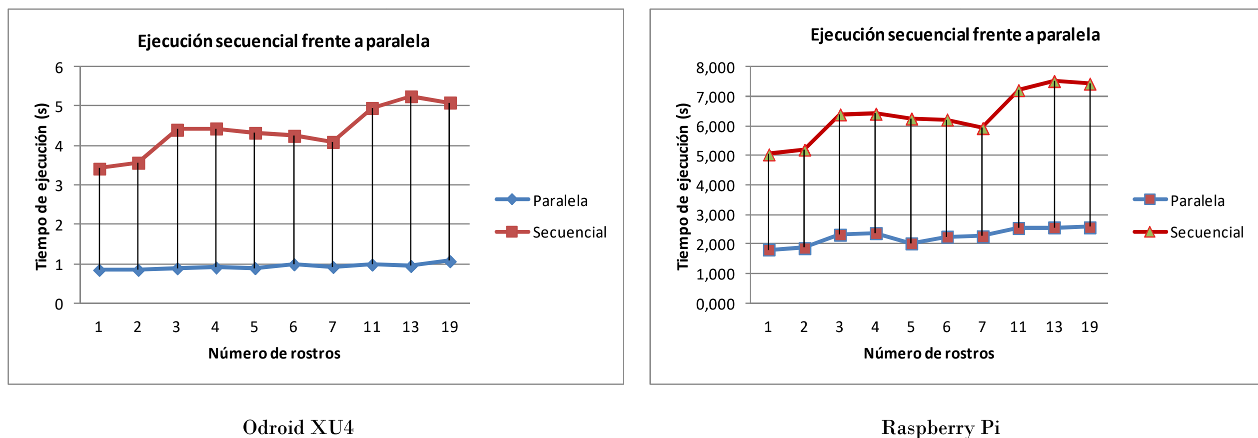


__Figura 5.29:__ Comparación entre los tiempos de ejecución secuencial y paralela

La razón de esta desviación es que la ejecución de un algoritmo de división en cuatro hilos en paralelo introduce inevitablemente una sobrecarga de tareas para sincronizar los resultados entre hilos, lo que impide que se puedan dedicar todos los núcleos a la ejecución en paralelo encomendada. Se puede ver esta sobrecarga en la figura 5.30, donde los resultados del *profiling* del algoritmo optimizado tienen ahora dos nuevas funciones que no existían cuando se ejecutaba de forma secuencial (ver figura 5.17). Estas funciones son: "ScaleImage_Invoker [clone._omp_fn.1]" y "ScaleImage_Invoker [clone ._omp_fn.0]".

```
Using local file ./vj.
Using local file vj.prof.
Total: 1975 samples
    930  47.1%  47.1%      930  47.1% ScaleImage_Invoker [clone ._omp_fn.1]
    354  17.9%  65.0%      354  17.9% runCascadeClassifier
    247  12.5%  77.5%      247  12.5% ScaleImage_Invoker [clone ._omp_fn.0]
    121   6.1%  83.6%      121   6.1% detectObjects
     88   4.5%  88.1%       88   4.5% int_sqrt
     77   3.9%  92.0%       77   3.9% myRound
     44   2.2%  94.2%       44   2.2% operator new
     39   2.0%  96.2%       39   2.0% integralImages
     15   0.8%  97.0%       15   0.8% setImageForCascadeClassifier
     12   0.6%  97.6%       12   0.6% _IO_acquire_lock_fct
     10   0.5%  98.1%       10   0.5% ScaleImage_Invoker
     10   0.5%  98.6%       10   0.5% nearestNeighbor
      7   0.4%  98.9%        7   0.4% _IO_getc
      6   0.3%  99.2%        6   0.3% fputc
      4   0.2%  99.4%        4   0.2% 0xffff0fe0
      3   0.2%  99.6%        3   0.2% myatoi
      1   0.1%  99.6%        1   0.1% _IO_fgets
      1   0.1%  99.7%        1   0.1% __memchr
      1   0.1%  99.7%        1   0.1% _init
      1   0.1%  99.8%        1   0.1% cpyPgm
      1   0.1%  99.8%        1   0.1% munmap
      1   0.1%  99.9%        1   0.1% readPgm
      1   0.1%  99.9%        1   0.1% write
      1   0.1% 100.0%        1   0.1% writePgm
```

Figura 5.30: *Profiling algoritmo paralelizado con OpenMP en Raspberry Pi 2 B.*

Estas nuevas llamadas son funciones de sincronización de hilos OpenMP que administran la sincronización entre los mismos, y también los tiempos de espera en el final de las secciones paralelas, Es decir, esperan hasta que todos los hilos hayan completado su trabajo antes de proporcionar los resultados de cada una de las tareas en paralelo.

También se puede ver la sobrecarga que produce OpenMP, si se compara el tiempo de usuario (*User time*) en la ejecución secuencial y en la paralela, como se mostraba en la tabla 5.2. Por ejemplo, para la Raspberry Pi y la imagen de entrada que contiene 19 rostros, el tiempo de usuario aumenta de 7.390s a 7.550s, lo que supone un tiempo de trabajo de la CPU extra para gestionar la sobrecarga por la gestión de los hilos de ejecución en paralelo.

A pesar de lo anterior, se puede decir que se ha mejorado el rendimiento del software C++ que había sido originalmente diseñado para una ejecución secuencial en un solo hilo. Se ha visto cómo mediante la introducción de forma adecuada de algunas directivas simples de la API de OpenMP al código fuente, se produce un decremento del tiempo de ejecución en todos los casos, como se puede ver en los resultados mostrados en la tabla 5.2.

Sobre la base de las pruebas realizadas, las mejoras con OpenMP producen una reducción del tiempo de ejecución del detector de rostros en, aproximadamente, un 65% respecto a la ejecución secuencial en una Raspberry Pi 2 B, con un procesador de cuatro núcleos ARM, y de aproximadamente una mejora del 75% para la placa Odroid XU4, que contiene un procesador de 8 núcleos asimétricos en dos *clusters* de 4 procesadores cada uno.

Por tanto, existe una reducción importante del tiempo de ejecución, lo que se traduce en un incremento de la velocidad de detección de rostros en dispositivos de bajo coste como son las placas Raspberry Pi 2 B y Odroid XU4. Lo que, a su vez, supone una mejora en el rendimiento del algoritmo para la detección en tiempo real de rostros.

Los buenos resultados obtenidos en la aceleración del código tienen una desventaja, ya que si se mide en la Raspberry Pi 2 B u Odroid XU4 el consumo energético de la ejecución paralela respecto a la secuencial, la ejecución paralela supone un incremento en el consumo superior al 50% respecto a la ejecución secuencial (ver figuras 5.31 y 5.32) [Corpa, 2018]. Esto es debido sobre todo al incremento de ciclos de procesamiento que requiere la paralelización del programa.

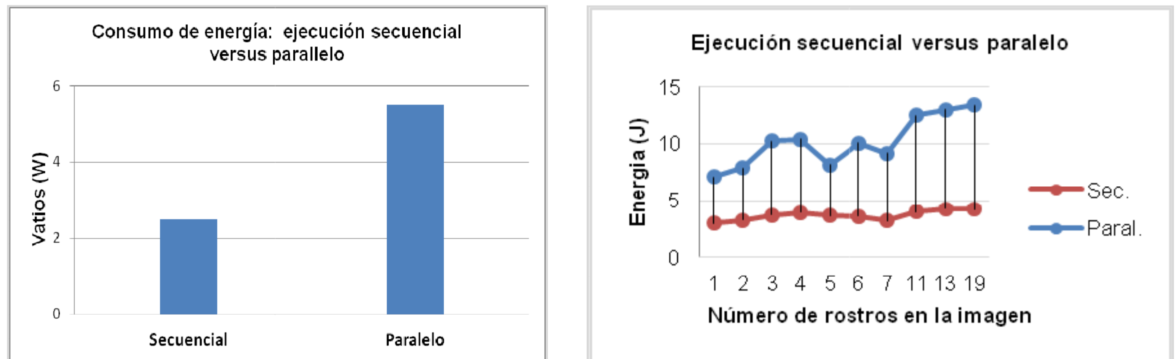


Figura 5.31: Comparación entre consumo energético en la ejecución secuencial y paralela en la plataforma Raspberry Pi 2 B+.

Ambas figuras 5.31 y 5.32, muestran el consumo medio de energía durante la ejecución de cinco pruebas experimentales del programa de detección facial para las imágenes de prueba.

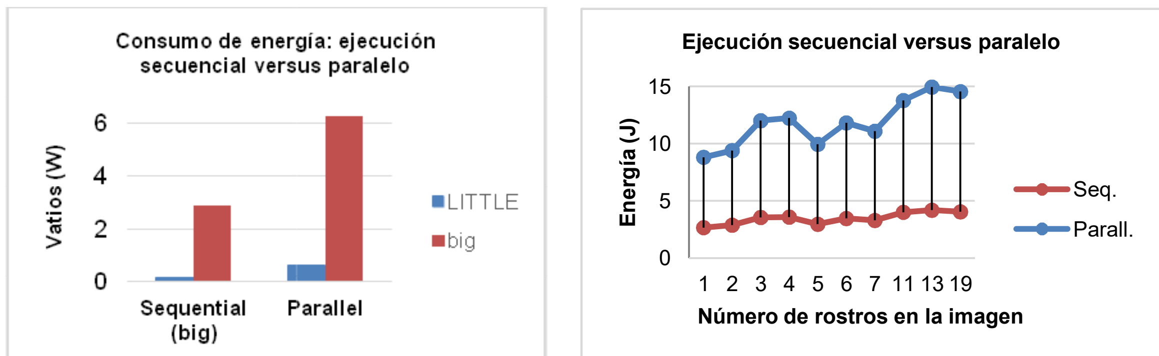


Figura 5.32: Comparación entre consumo energético en la ejecución secuencial y paralela en la plataforma Odroid XU4.

El consumo de energía durante la ejecución aumenta como se muestra en las figuras 5.31 y 5.32. Sin embargo, esta desventaja se puede superar mediante el uso de diferentes técnicas de optimización que se describirán en el Capítulo 6 de esta memoria, tales como son las técnicas de tiempo de ejecución del planificador de tareas de OmpSs (Nanos++) [Nanos, 2018], el escalado de frecuencia y voltaje dinámico (DVFS) o técnicas específicas para arquitecturas asimétricas.

5.7 Conclusiones

En este capítulo se ha presentado la implementación del algoritmo de detección facial y los entornos experimentales de pruebas que se han utilizado, que son las placas Raspberry Pi 2 B y Odroid XU4.

Seguidamente se ha llevado a cabo un análisis de la ejecución secuencial del programa de detección de rostros, para el cual se ha utilizado una muestra de 10 imágenes con la misma resolución, formato y con diferente número de rostros, y una herramienta de *profiling* llamada Gperftools. A partir de ésta se han obtenido las funciones que mayor esfuerzo computacional requieren y, por tanto, más tiempo de ejecución consumen.

Como resultado de las pruebas de la ejecución secuencial del programa de detección facial, cabe destacar el descubrimiento de una posible relación directa entre el tiempo de ejecución, el número de rostros y el valor de lo que se ha denominado imagen integral en el algoritmo de Viola-Jones.

Con el fin de mejorar el tiempo de ejecución del programa de detección se ha procedido a realizar un estudio detallado para la paralelización del mismo. Para ello, se ha adecuado el código fuente siguiendo las técnicas de mapeo estudiadas con la idea de optimizar dicha paralelización en las plataformas experimentales de prueba. Como entorno de programación paralela se ha hecho uso de la API que ofrece OpenMP, que, como se ha visto, es una de las mejores opciones dentro de las tecnologías de programación en paralelo en arquitecturas de memoria compartida.

El resultado de la paralelización, sobre la misma muestra de imágenes, ha sido sumamente satisfactorio, ya que se ha conseguido reducir el tiempo de ejecución considerablemente en ambas plataformas de bajo coste, aproximadamente, el 65% en el caso de la placa Raspberry Pi 2 B y aproximadamente el 75% en la placa Odroid XU4. Esta reducción guarda una relación directa con el número de procesadores o núcleos que tiene cada una.

En contraposición a estos buenos resultados, se ha producido un incremento superior al 50% en el consumo energético de ambos dispositivos respecto a la ejecución secuencial. Este problema es en general uno de los principales inconvenientes que se producen en la paralelización de un programa informático. Este punto es un tema de mucho interés en el ámbito de los dispositivos experimentales de prueba que se están usando, ya que este tipo de dispositivos suelen depender de baterías cuya autonomía se ve muy afectada por el tipo de procedimientos que se ejecuta en los mismos.

Para intentar salvar este importante problema, en el Capítulo 6, se estudiarán diferentes técnicas para optimizar, y, por tanto, reducir el consumo energético aprovechando los recursos disponibles en los dispositivos experimentales utilizados en esta investigación.

CAPÍTULO 6.

6. OPTIMIZACIÓN DEL CONSUMO ENERGÉTICO

En este capítulo se realiza un estudio de las diferentes técnicas de reducción del consumo energético para aprovechar de manera eficiente todos los recursos computacionales que ofrecen las arquitecturas experimentales estudiadas, sobre todo la asimétrica. Para ello, se hará uso de las funcionalidades que ofrece el planificador de tareas de OmpSs [Ompss, 2018], consciente de la asimetría de la arquitectura de la CPU, así como de las políticas de asignación de recursos y explotación de los modos de bajo consumo proporcionados por las arquitecturas, que permitan un aprovechamiento eficiente de la energía. El objetivo será conseguir un rendimiento energéticamente eficiente de todos los recursos computacionales existentes en los dispositivos de bajo coste estudiados en esta tesis doctoral, la placa Raspberry Pi 2 B, y la placa Odroid XU4.

6.1 Introducción

El paradigma de diseño de microprocesadores ha evolucionado mucho en los últimos tiempos. La rápida escalada del consumo energético y los problemas de disipación térmica pusieron en evidencia que el incremento de frecuencia como técnica de mejora de rendimiento no podía mantenerse. A partir de ese momento, se ha optado por incluir un mayor número de elementos de procesamiento (núcleos) en un mismo chip, lo cual ha sido posible gracias a la disminución del tamaño de los transistores.

Inicialmente, los núcleos dentro de un procesador eran idénticos entre sí, pudiendo distinguirse dos tipos [Adria, 2016]:

- Procesadores de alto rendimiento: integran núcleos de elevado consumo y alta frecuencia de trabajo, como el Intel Core i7-6700K. Estos núcleos basan su diseño en la optimización de la capacidad de cómputo mediante el uso de técnicas como la ejecución fuera de orden o el lanzamiento múltiple, con un claro aumento del consumo energético como contrapartida.

- Procesadores constituidos por núcleos de baja frecuencia y consumo reducido, su filosofía se centra en el ahorro energético, como los Intel Atom o los ARM Cortex A7 ó A53. Estos procesadores están presentes en dispositivos móviles en los que el ahorro de batería es un aspecto fundamental.

Sin embargo, la diversificación de las necesidades de computación que presentan las aplicaciones ha propiciado el inicio de una nueva etapa en el diseño de hardware, en la que se incluyen en un mismo chip componentes especializados en resolver cierto tipo de problemas [Johnl, 2019]. En este contexto se propusieron como alternativa los procesadores multi-núcleos asimétricos [Kumar, 2003] que proporcionan un mejor soporte que los multi-núcleo convencionales para cargas de trabajo heterogéneas.

En este sentido, los procesadores multi-núcleo asimétricos que poseen un repertorio común de instrucciones han sido propuestos recientemente como alternativa de bajo consumo a los procesadores multi-núcleo simétricos convencionales, ya que

combinan, en un mismo chip, núcleos rápidos de alto rendimiento con núcleos más lentos y sencillos de consumo reducido con los que se puede optimizar el consumo de energía del procesador. En el caso que nos ocupa, como se indicó anteriormente, para realizar las pruebas de optimización de consumo energético se utilizará la placa Odroid XU4, ya que presenta una arquitectura asimétrica.

Las arquitecturas asimétricas permiten reducir el consumo de energía asignando las tareas a núcleos de distintas características en función de los requisitos de rendimiento y consumo de cada tarea. Además, la arquitectura de la placa Odroid XU4 permite ajustar el rendimiento y consumo de cada núcleo utilizando las técnicas de escalado de frecuencia que se encuentran disponibles en los procesadores modernos.

Por tanto, este capítulo se centra en la optimización del consumo energético del programa de detección facial, aprovechando los recursos que ofrece la arquitectura asimétrica de la placa Odroid XU4. Los resultados experimentales obtenidos permitirán hacer una comparativa en cuanto a eficiencia energética entre esta placa y la Raspberry Pi 2 B, objeto también de estudio de esta tesis doctoral, y cuya arquitectura es simétrica.

6.2 Procesadores multi-núcleo asimétricos

Un procesador multi-núcleo asimétrico (*Asymmetric Multicore Processor*, AMP) integra núcleos con distintas características en un mismo chip. Por un lado, los AMPs integran un grupo de núcleos rápidos, de alto rendimiento y consumo, que trabajan a alta frecuencia e implementan complejas técnicas como la ejecución fuera de orden o el lanzamiento múltiple de instrucciones. Por otra parte, estos procesadores incluyen un grupo de núcleos más lentos y sencillos. Estos núcleos operan a una frecuencia de trabajo más baja e implementan un *pipeline* más sencillo y de bajo consumo [Hillm, 2008].

Los procesadores multi-núcleo asimétricos aportan una mayor flexibilidad que los procesadores convencionales, ya que mantienen la capacidad de un procesador de alto rendimiento para aplicaciones intensivas en CPU, sin sacrificar la eficiencia energética. Nótese que las aplicaciones intensivas en memoria, que provocan muchas

paradas en el *pipeline*, se podrían ejecutar en un núcleo lento sin penalizar su rendimiento de forma significativa, pero proporcionando una considerable mejora en la eficiencia energética del sistema.

El ejemplo más destacado a nivel comercial de multi-núcleo asimétrico es el procesador Big.LITTLE con tecnología ARM [Armbe, 2016] Exynos 5422 de Samsung, que se utiliza en la placa experimental objeto de estudio Odroid XU4, y que está presente en múltiples dispositivos móviles. En los procesadores multi-núcleo asimétricos actuales, los distintos núcleos exponen un repertorio de instrucciones común, lo que facilita enormemente el desarrollo de software. Esta aproximación contrasta con la que se sigue en otros sistemas heterogéneos como el IBM Cell BE [Gschw, 2007], donde los distintos núcleos exponen un repertorio de instrucciones diferente.

En definitiva, los AMPs permiten a los programadores controlar la ejecución de determinadas tareas de aplicaciones en determinados núcleos, pudiendo explotar el paralelismo de forma más eficiente. Por ejemplo, usar los núcleos rápidos para ejecutar las fases secuenciales de los procesos y los lentos y eficientes para ejecutar fases multi-hilo, puede proporcionar mayor eficiencia energética [Annav, 2005]. Por otra parte, es posible mejorar el rendimiento global si los núcleos rápidos se destinan a la ejecución de fases intensivas en CPU de un programa, y los lentos se destinan a la ejecución de las fases intensivas en memoria [Kumar, 2003].

6.2.1 Modelos de ejecución para arquitecturas Big.LITTLE

Las arquitecturas Big.LITTLE modernas ofrecen un conjunto de distintos modelos de ejecución soportados por el sistema operativo:

- **Clúster Switching Mode (CSM)**: El procesador se divide de forma lógica en dos *clusters*, uno conteniendo los núcleos rápidos, y otro conteniendo los núcleos lentos, pero sólo uno de ellos es utilizable simultáneamente en tiempo de ejecución. El sistema operativo activa/desactiva los *clusters* de forma transparente, en función de la carga de trabajo, para equilibrar el rendimiento y la eficiencia energética.

- **CPU migration (CPUM)**: Los núcleos físicos se agrupan en pares, cada uno formado por un núcleo rápido y otro lento, constituyendo Núcleos Virtuales (VC), a los que el sistema operativo mapea hebras. En un instante determinado, sólo un núcleo físico está activo en cada VC, dependiendo de los requisitos exigidos por la carga computacional activa. En aquellas implementaciones big.LITTLE en las que el número de núcleos lentos y rápidos no es el mismo, cada VC puede estar formado por diferente número de núcleos de cada tipo [Angel, 2008].

- **Global Task Scheduling (GTS)**: Se trata del modelo más flexible. Todos los núcleos (lentos y rápidos) están disponibles para la planificación de hebras, y el sistema operativo las mapea en función de la naturaleza de la carga computacional asociada a cada uno de ellos y la disponibilidad de núcleos.

GTS es la solución más flexible, ya que permite al planificador del sistema operativo asignar hebras a cualquiera de los núcleos disponibles, es decir, todos los núcleos disponibles se ofrecen al sistema operativo como candidatos a ejecutar el código de cualquier hebra, independientemente de su naturaleza o características.

6.3 Entorno de medida del consumo energético

Existen placas de desarrollo con procesadores ARM big.LITTLE que integran registros o sensores para obtener medidas de consumo de potencia instantánea o energía consumida a lo largo del tiempo para distintos componentes del SoC, como los *clusters* de núcleos, la memoria o la GPU integrada. Este es el caso de la placa de desarrollo Odroid XU4.

Para el desarrollo de este capítulo se ha utilizado un entorno experimental de medición de consumo energético sobre la plataforma Odroid XU4. El entorno se basa en la adaptación del software Pmlib [Alonso, 2012], desarrollado por la Universidad Jaume I de Castellón. Dicha herramienta, desarrollada en lenguaje Python, implementa un paradigma cliente/servidor, en el que la parte servidora se encarga de recoger continuamente muestras de potencia instantánea disipada por cierto dispositivo,

mientras que la utilización de una API propia, que actúa como cliente, permite instrumentar los códigos a estudiar desde el punto de vista energético, realizando peticiones al servidor Pmlib [Luisc, 2016].

El servidor de Pmlib permite interactuar con los sensores de consumo energético integrados en la placa y basados en resistencias de *shunt*. Dichos sensores ofrecen lecturas independientes para el consumo de energía del *cluster* formado por los Cortex-A7, Cortex-A15, GPU y RAM (ver figura 6.1), con una frecuencia de refresco de cuatro muestras por segundo [Luisc, 2016].

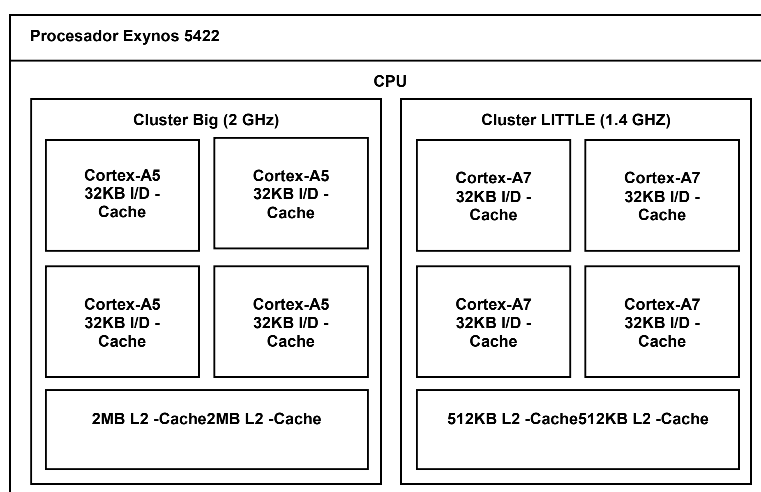


Figura 6.1*: Diagrama de bloques del procesador Exynos 5422 de la placa Odroid XU4, constituido por el cluster Big de 4 núcleos y el cluster LITTLE de 4 núcleos [Fuente: elaboración propia].*

6.4 Mejora del rendimiento y optimización del consumo energético en arquitecturas asimétricas

Las arquitecturas asimétricas, formadas por varios procesadores con el mismo repertorio de instrucciones pero distintas características de rendimiento y consumo, ofrecen muchas posibilidades de optimización del rendimiento y/o del consumo energético en la ejecución de aplicaciones paralelas. La planificación de tareas sobre dichas arquitecturas de forma que se aprovechen de manera eficiente los distintos recursos, es compleja y se suele abordar utilizando modelos de programación paralelos,

que permiten al programador especificar el paralelismo de las tareas, y entornos de ejecución que explotan dinámicamente dicho paralelismo.

Una técnica utilizada para obtener un mejor rendimiento en programas paralelos basados en tareas es intentar que las tareas críticas finalicen su ejecución en el menor tiempo posible. Se denominan tareas críticas a aquellas tareas que en caso de retrasar su ejecución, producen un retraso en la ejecución del programa. Las tareas críticas asociadas a un determinado programa, forman el llamado camino crítico del DAG (Directed Acyclic Graph). Por tanto, en el proceso de optimización se debe localizar el camino crítico para intentar ejecutar dichas tareas en los núcleos rápidos y evitar retardos innecesarios en la ejecución del programa.

En un sistema asimétrico, se pueden identificar tres variables que afectan al rendimiento y al consumo energético de una aplicación:

- El tipo de núcleo usado para ejecutar las distintas tareas que componen el problema. Ejecutar una tarea sobre un núcleo del *cluster* Big va a generar un mayor rendimiento que si se ejecuta sobre un núcleo del clúster LITTLE, que como contrapartida dará lugar a un mayor consumo de energía [Luisc, 2016].

- El número de núcleos activos durante la ejecución del problema. Un número bajo de núcleos activos puede suponer una pérdida de rendimiento global considerable, sin embargo también supone un ahorro en el consumo energético del procesador.

- La frecuencia a la que trabajan los distintos *cluster*. Frecuencias bajas pueden afectar al rendimiento de la aplicación al aumentar el tiempo de ejecución, pero pueden suponer un descenso de la potencia instantánea.

En esta tesis doctoral se intenta aprovechar todos los recursos al máximo para conseguir la mejor eficiencia energética posible. Para mejorar dicha eficiencia energética se van a incluir en el planificador de tareas (o *runtime*) políticas de explotación de los modos de bajo consumo de la arquitectura y también de apagado o no asignación de carga de trabajo a algunos de los núcleos, que se activan en tiempo de

ejecución cuando se detecta que la aplicación no necesita todos los recursos disponibles en la arquitectura.

6.4.1 Técnicas de mapeo aplicadas a sistemas asimétricos

Cuando el sistema consta de elementos con diferentes características computacionales es posible emplear diferentes enfoques para realizar el mapeo. En particular todas las técnicas descritas en el capítulo anterior para sistemas homogéneos pueden ser aplicadas teniendo en cuenta el carácter heterogéneo del sistema donde sea necesario. Se hará énfasis en esta sección, en dos estrategias habitualmente aplicadas al caso heterogéneo asimétrico [Alexe, 2001]:

- La estrategia HoHe, consiste en diseñar tareas que trabajan sobre volúmenes de datos distintos y asignar una tarea a cada procesador que se utilice. El volumen de trabajo dependería de las características del procesador sobre el que va a ejecutarse la tarea. Se tiene una distribución homogénea de procesos y una distribución heterogénea de datos.

- La estrategia HeHo, en la que cada tarea trabaja con un volumen idéntico de datos. Se crea un número de tareas independiente y habitualmente superior al número de procesadores, y se asigna un número variable de tareas a cada procesador en función de las capacidades de computación y comunicación de cada uno. Se tiene, en este caso, una distribución heterogénea de las tareas en los procesadores, y una distribución homogénea de los datos en los procesos.

La estrategia HoHe conlleva la reprogramación del código, lo que supone un importante trabajo de rediseño de los algoritmos que han sido desarrollados para los sistemas homogéneos. La estrategia HeHo, sin embargo, permite que se pueda utilizar el mismo código que en sistemas homogéneos, y sólo será necesario disponer de un método eficiente de asignación de tareas a procesadores. En este caso, se plantea un problema de asignación de tareas a procesadores con el fin de obtener un tiempo total de ejecución reducido. Por este motivo, en este trabajo de investigación se hace uso de la estrategia HeHo, en la que el planificador de tareas de OmpSs será el encargado de realizar una asignación eficiente de tareas, conforme a las políticas para la reducción del consumo energéticos que se apliquen y que se verán más adelante.

6.4.2 Alternativas para la paralelización en plataformas asimétricas

Existen principalmente dos alternativas principales a la hora de aprovechar los recursos disponibles en una plataforma asimétrica en la que se quiere implementar una determinada solución algorítmica:

- Utilizar un planificador de tareas consciente de la asimetría, en el que, por ejemplo, las tareas pertenecientes al camino crítico sean asignadas a núcleos de procesamiento rápidos. De este modo, la unidad básica de planificación de tareas es el núcleo individual, las tareas son invocaciones a versiones secuenciales de una determinada biblioteca, y es el planificador de tareas el encargado de explotar en tiempo de ejecución, sin la intervención del programador, los núcleos asimétricos subyacentes [Luisc, 2016].

- Utilizar una implementación de rutinas de biblioteca conscientes de la asimetría, sin extraer paralelismo a nivel de tareas. Una determinada invocación a una de estas rutinas se ejecutará eficientemente de forma paralela teniendo en cuenta las características de la arquitectura asimétrica subyacente, repartiendo su espacio de iteraciones, por ejemplo, según las capacidades de cómputo de cada tipo de núcleo disponible.

En este trabajo de investigación se hace uso del planificador de tareas, como ya se mencionó en el punto anterior, dado que las ventajas de su uso son claras: menor intervención por parte del programador o usuario, adaptación a pequeños cambios en el comportamiento dinámico de la arquitectura ante la ejecución de un código, o modificación de las políticas de planificación dinámicamente, entre otras. Sin embargo, en muchas ocasiones, el sobrecoste de este tipo de mecanismos no es desdeñable; además, su adaptación a arquitecturas asimétricas, como es el caso, requiere un desarrollo específico de nuevas políticas de planificación que exploten los recursos disponibles de manera eficiente.

6.4.3 Modelos de programación paralela en arquitecturas asimétricas

En general, los modelos de programación proporcionan al programador en primer lugar un mecanismo para expresar el paralelismo potencial a nivel de tareas existente en la aplicación, normalmente en forma de grafo de tareas, y en segundo lugar un planificador de tareas (o *runtime*) que realice una planificación dinámica de las mismas, respetando las dependencias de datos entre ellas y a la vez permitiendo un correcto aprovechamiento de las unidades de procesamiento disponibles. Entre las propuestas e implementaciones más extendidas para este tipo de modelo de programación cabe destacar, entre otros: Cilk Plus, StarPU, Superglue, QUARK, Kaapi y OmpSs [Duran, 2011].

OmpSs es un modelo de programación paralela muy extendido basado en la extracción de tareas. A grandes rasgos, el diseño del modelo de programación de OmpSs se basa en la inclusión de directivas (o pragmas) similares a las utilizadas en el modelo de programación paralela OpenMP. Como este, OmpSs proporciona mecanismos sencillos y no invasivos a través del uso de dichas directivas (pragmas) para anotar rutinas existentes en el código como tareas, e indicando a la vez la direccionalidad de sus operandos (entrada, salida o entrada/salida) a través de cláusulas adicionales [Ompss, 2018].

OmpSs dispone de un planificador de tareas llamado Nanos++ [Nanos, 2018], que se encarga de mapear, de forma eficiente, su ejecución en el mejor recurso computacional disponible en un momento dado, aunque no es consciente de la asimetría de la CPU. El planificador de OmpSs descompone el código en un conjunto de tareas, identificando las dependencias entre ellas, para su ejecución en los distintos núcleos computacionales del sistema. Al contrario que en el modelo de ejecución usado por OpenMP, Nanos++ no crea un nuevo hilo por cada tarea a ejecutar, sino que crea un conjunto de hilos al inicio de la aplicación [Ompss, 2018]. Y descompone el código en un conjunto de tareas, identificando las dependencias entre ellas, y lanzando a ejecución únicamente tareas listas (es decir, aquellas cuyas dependencias han sido satisfechas) para su ejecución en los distintos núcleos computacionales del sistema.

Una vez que un código fuente ha sido compilado con el compilador de OmpSs: Mercurium [Mercu, 2018], éste puede ser ejecutado a través del *runtime* Nanos++, que se compone de una serie de bibliotecas encargadas de controlar la ejecución del programa e intentar finalizar la ejecución de la manera más eficiente posible. Para ello, Nanos++ está compuesto por un núcleo principal encargado de la gestión de los distintos hilos y estructuras internas, y un conjunto de módulos encargados de distintas tareas como puede ser el módulo encargado de la planificación de las tareas sobre los hilos, el módulo encargado de comprobar las dependencias entre tareas, o el módulo encargado de obtener distintas trazas durante la ejecución [Luisc, 2016].

Una vez que un programa es lanzado sobre Nanos++, éste realiza una serie de pasos antes de comenzar la ejecución del programa original. Uno de estos pasos es la creación de un número fijo de hilos denominados *worker threads*, que van a ser los encargados de ejecutar las distintas tareas que estén preparadas para ser ejecutadas (el número de hilos creados es escogido por el usuario en el momento de iniciar la ejecución). El primero de estos *worker threads* creados, aparte de ejecutar tareas igual que el resto de *worker threads*, también es el encargado de ejecutar el código secuencial del programa, por lo que mientras que el resto de hilos pueden estar ociosos en el caso de no existir ninguna tarea lista para ejecución, este hilo siempre estará ocupado.

Cuando un *worker thread* finaliza la ejecución de una tarea, éste realiza dos acciones sobre el planificador de tareas [Ompss, 2018]:

1. El *worker thread* informa al *runtime* de la finalización de la tarea. Al finalizar la tarea, el módulo encargado de comprobar las dependencias comprueba qué nuevas tareas están listas para ser ejecutadas ya que se han satisfecho todas sus dependencias. Cuando se determina que una tarea está lista para ejecución, se informa al módulo encargado del planificador para que la almacene y trate de acuerdo a la política que implemente el módulo cargado.

2. El *worker thread* solicita una nueva tarea para ejecutar al módulo planificador. El módulo planificador, en función de la política que se haya decidido usar, y del *worker thread* que solicite la tarea, decidirá qué tarea debe ejecutar. En caso de no existir. ninguna tarea lista para ejecutar, o de que el *worker thread* no sea el idóneo para ejecutar ninguna de las tareas listas existentes, éste pasa a estado inactivo hasta la creación de nuevas tareas, evitando así consumir recursos de manera innecesaria.

La creación de nuevas tareas tiene lugar en el momento en el que un *worker thread* en ejecución se encuentra una anotación para la creación de una nueva tarea. En este momento, el *worker thread* informa al *runtime* para que cree la tarea correspondiente, esperando a que todas sus dependencias sean satisfechas y la nueva tarea esté lista para su ejecución.

En el apéndice A, de esta tesis doctoral se describe como compilar programas en OmpSs, a través del compilador Mercurium C/C++. Además, de las principales directivas y cláusulas complementarias de OmpSs.

6.4.4 Políticas de reducción del consumo

En cualquier arquitectura asimétrica, como es la placa Odroid XU4, las principales variables que afectan al rendimiento energético son:

1. El tipo de núcleo o *cluster* usado.
2. El número de núcleos activos durante la ejecución del programa.
3. Las frecuencias a la que trabajan los distintos núcleos o *cluster* en una arquitectura asimétrica.

A partir de estos tres parámetros, existen muchas formas de combinar los distintos factores para conseguir adaptar el rendimiento y el consumo energético de una determinada aplicación.

Por tanto, las políticas para la reducción del consumo se pueden dividir en tres grandes grupos: un primer grupo formado por políticas basadas en escalado de

frecuencia, un segundo grupo formado por políticas encargadas de modificar la asignación de las tareas a los diferentes *worker threads* generados por el planificador de tareas de OmpSs, y un tercer grupo basado en políticas de selección eficiente de los parámetros configurables del programa que pueden producir un ahorro del consumo energético.

En este sentido, en el marco del programa para la detección de rostros, se encuentran un par de parámetros que pueden afectar considerablemente al rendimiento y al consumo energético del dispositivo. El primer parámetro es ”scalaFactor”, que determina la reducción de la escala de la imagen en cada iteración. Cuanto mayor es su valor, el rendimiento y el consumo energético del programa mejoran considerablemente, en detrimento de la precisión, ya que el número de falsos negativos puede crecer rápidamente. El segundo parámetro que afecta es "step". Este parámetro determina el desplazamiento de la ventana de detección por la imagen analizada en cada iteración. Igualmente el incremento de su valor produce una mejora del rendimiento y del consumo energético en detrimento de la precisión en el reconocimiento facial. Lo más preciso es que se desplace 1 pixel cada vez (por eso su valor habitual es 1) pero cabe considerar otros valores sin que el número de falsos negativos se incremente considerablemente, sobre todo en imágenes de alta resolución.

A partir de todas estas variables se puede encontrar las políticas y los parámetros óptimos que permitan un rendimiento, un consumo energético y una precisión óptima en función de las necesidades marcadas para en programa de detección facial.

A continuación, se analizan más detenidamente cada una de las políticas mencionadas y la repercusión que las mismas pueden tener en el rendimiento y en el consumo energético del programa de detección facial.

6.4.4.1 Políticas basadas en el escalado de frecuencia

Las siglas DVFS (Dynamic Voltage Frequency Scaling) representan a un conjunto de técnicas caracterizadas por variar la frecuencia o el voltaje del procesador de manera dinámica en tiempo de ejecución con el objetivo de obtener una mejora

energética o disminuir la potencia instantánea consumida. En la actualidad, esta técnica ha cobrado especial relevancia en los dispositivos móviles, donde la duración de la batería es un factor muy importante a tener en cuenta. Sin embargo, esta técnica no se limita solamente a este tipo de dispositivos.

Los procesadores de la familia Big.LITTLE de ARM poseen soporte para realizar escalado de frecuencia, aunque únicamente se permite realizar el escalado a nivel de *cluster* y no de núcleo, lo que implica que todos los núcleos del *cluster* siempre van a estar ejecutando tareas a la misma frecuencia.

El acceso a esta característica de los procesadores en el *kernel* Linux se realiza a través del subsistema "cpufreq", que proporciona la biblioteca "libcpufreq" y los ejecutables "cpufreq-set" y "cpufreq-info" que permiten obtener y modificar la frecuencia de cada *cluster* en tiempo de ejecución. Esta gestión de frecuencias va a permitir contener el gasto energético y por lo tanto ahorra energía durante la ejecución del programa.

Una posible idea para ahorrar energía puede consistir en aprovechar los períodos de tiempo en los que la carga de trabajo sobre los núcleos lentos es menor para disminuir el consumo energético de los mismo forzando una reducción de frecuencia sobre el cluster LITTLE. Reducir la frecuencia en el cluster LITTLE implica que la potencia instantánea disipada disminuye. Aunque es cierto que al reducir la frecuencia de los núcleos el tiempo empleado en ejecutar una tarea aumenta, esta técnica se puede aplicar en aquellas fases de la ejecución paralela en las que la ejecución está limitada por el gran número de tareas críticas y el bajo número de tareas no críticas; por tanto, es esperable que el impacto final en el rendimiento no sea elevado, y sí lo sea la reducción de consumo energético [Luisc, 2016].

Para controlar la frecuencia de ejecución, lo primero que se hace es cargar el modulo que permite controlar la frecuencia, en el caso del *kernel* instalado en la placa Odroid XU4, el modulo es "cpufreq-dt". En este contexto, si se ejecuta en la ventana de comandos la instrucción "cpufreq-info", se obtiene toda la información sobre los diferentes núcleos del procesador necesaria para poder realizar una adecuada gestión de frecuencias sobre el dispositivo experimental. En este caso, para la placa Odroid XU4,

se obtiene la información que se muestra en la figura 6.2 para el núcleo: CPU 4, que forma parte del *cluster* Big (núcleos: CPU 0, CPU 1, CPU 2, CPU 3 forman el *cluster* LITTLE, y núcleos: CPU 4, CPU 5, CPU 6, CPU 7 forma el *cluster* Big).

```
cpufrequtils 008: cpufreq-info (C) Dominik Brodowski 2004-2009
Report errors and bugs to cpufreq@vger.kernel.org, please.
analyzing CPU 4:
  driver: cpufreq-dt
  CPUs which run at the same hardware frequency: 4 5 6 7
  CPUs which need to have their frequency coordinated by software: 4 5 6 7
  maximum transition latency: 157 us.
  hardware limits: 200 MHz - 2.00 GHz
  available frequency steps: 200 MHz, 300 MHz, 400 MHz, 500 MHz, 600 MHz,
700 MHz, 800 MHz, 900 MHz, 1000 MHz, 1.10 GHz, 1.20 GHz, 1.30 GHz, 1.40
GHz, 1.50 GHz, 1.60 GHz, 1.70 GHz, 1.80 GHz, 1.90 GHz, 2.00 GHz
  available cpufreq governors: conservative, userspace, powersave, ondemand,
performance, schedutil
  current policy: frequency should be within 200 MHz and 2.00 GHz.
                  The governor "performance" may decide which speed to use
                  within this range.
  current CPU frequency is 2.00 GHz.
  cpufreq stats: 200 MHz:0.04%, 300 MHz:0.00%, 400 MHz:0.00%, 500
MHz:0.00%, 600 MHz:0.00%, 700 MHz:0.00%, 800 MHz:0.00%, 900
MHz:0.00%, 1000 MHz:0.00%, 1.10 GHz:0.00%, 1.20 GHz:0.00%, 1.30
GHz:0.00%, 1.40 GHz:0.00%, 1.50 GHz:0.00%, 1.60 GHz:0.00%, 1.70
GHz:0.00%, 1.80 GHz:0.00%, 1.90 GHz:0.01%, 2.00 GHz:99.92%  (195)
```

Figura 6.2*: Información sobre núleo CPU 4 obtenida al ejecutar la instrucción "cpufreq-info" en la placa Odroid XU4.*

Como se puede apreciar la salida del comando "cpufreq-info" devuelve:

- La lista de las frecuencias que soporta la CPU 4 y su uso (200 MHz, 300 MHz, 400 MHz, 500 MHz, 600 MHz, 700 MHz, 800 MHz, 900 MHz, 1000 MHz, 1.10 GHz, 1.20 GHz, 1.30 GHz, 1.40 GHz, 1.50 GHz, 1.60 GHz, 1.70 GHz, 1.80 GHz, 1.90 GHz, 2.00 GHz).
- La frecuencia de funcionamiento actual (current CPU frequency is 2.00 GHz).
- El controlador actual (*current policy*).
- Los controladores disponibles (*available cpufreq governors*). Por defecto suele estar seleccionado "performance" (rendimiento máximo y máximo consumo de energía), pero también están disponibles "ondemand" (aumenta la frecuencia del procesador al llegar al 90% de su carga), "conservative" (aumenta la frecuencia del procesador al llegar al 70% de su carga), "powersave" (mínima frecuencia de funcionamiento para

ahorrar energía) y "userspace". Con este último el usuario puede cambiar la frecuencia de funcionamiento a placer, ajustando la frecuencia al valor especificado por el programa en el espacio de usuario.

De todos los controladores, "userspace" es el más adaptable y dependiendo de cómo se configure, puede ofrecer el mejor equilibrio entre rendimiento y consumo para un sistema determinado. En el caso de esta tesis doctoral, se va a controlar la frecuencia del procesador, para lo que se ejecuta en modo superusuario la instrucción:

#sudo cpufreq-set –g userspace

Una vez ejecutada la instrucción, ya se puede cambiar la frecuencia de funcionamiento del procesador a través del programa. Para ello, haciendo uso de la función "cpufreq_set_frequency", se modifica el código fuente para que tome como argumento la frecuencia (dentro de las permitidas por el dispositivo) a la que se quiere que funcione en cada ejecución cualquiera de los dos *cluster* (Big-Little) que incluye la placa Odroid XU4. En la figura 6.3, se muestra el código fuente incluido para la gestión de la frecuencia de ejecución de la placa.

```
if (argc==2)
{
frecuencia= atoi(argv[1]);
   switch(frecuencia)
   {
      case 200: cpufreq_set_frequency(4,200000); break;
      case 300: cpufreq_set_frequency(4,300000); break;
      case 400: cpufreq_set_frequency(4,400000); break;
      case 500: cpufreq_set_frequency(4,500000); break;
      case 600: cpufreq_set_frequency(4,600000); break
      case 700: cpufreq_set_frequency(4,700000); break
      case 800: cpufreq_set_frequency(4,800000); break
      case 900: cpufreq_set_frequency(4,900000); break
      case 1000: cpufreq_set_frequency(4,1000000); break
      case 1200: cpufreq_set_frequency(4,1200000); break
      case 1300: cpufreq_set_frequency(4,1300000); break
      case 1400: cpufreq_set_frequency(4,1400000); break
      case 1500: cpufreq_set_frequency(4,1500000); break
      case 1600: cpufreq_set_frequency(4,1600000); break
      case 1700: cpufreq_set_frequency(4,1700000); break
      case 1800: cpufreq_set_frequency(4,1800000); break
      case 1900: cpufreq_set_frequency(4,1900000); break
      case 2000: cpufreq_set_frequency(4,2000000); break
      default:
              printf("Opción de frecuencia no valida\n");
              break;
   }

   } else   {
              printf("Frecuencia de funcionamiento no especificada.\n");
            }
```

Figura 6.3: *Fragmento deCódigo fuente para el control de frecuencia de ejecución de la placa Odroid XU4.*

6.4.4.2 Políticas basadas en la asignación de tareas en OmpSs

La política de programación del OmpSs define cómo se ejecutan las tareas listas para ser ejecutadas. Estas tareas son aquellas cuyas dependencias han sido satisfechas, por lo tanto su ejecución puede comenzar inmediatamente. La política de programación tiene que decidir el orden de ejecución de las tareas y el recurso donde se ejecutará cada una.

Las políticas de programación se especifican en el planificador de OmpSs, Nanos++, mediante la variable de entorno: NX_SCHEDULE, o con la opción:”--schedule”, que se incluye en la variable de entono NX_ARGS. En la tabla 6.1, se pueden ver las principales políticas que se pueden aplicar para la ejecución de un programa. Para ello solo se tiene que exportar el valor de la variable de entorno como se indica en la columna “Declaración” de dicha tabla.

Política	Descripción	Declaración
Breadth first	Es la política por defecto. Esta política solo implementa una única cola lista global. Al crear una tarea sin dependencias (o cuando una tarea está lista después de que se hayan cumplido todas sus dependencias) se coloca en esta lista de espera	$export NX_SCHEDULE=bf o $export NX_ARGS="--schedule=bf"
Distributed breadth first	Cada hilo inserta sus tareas creadas en su propia lista de espera siguiendo una política FIFO	$export NX_ARGS="--schedule=dbf" $export NX_ARGS="—-Schedule-priority"
Work first	Esta directiva de planificador implementa una cola de preparación local por subproceso. Una vez que se crea una tarea, elige continuar con la nueva tarea creada, dejando la tarea actual	$export NX_ARGS="--schedule=wf" $export NX_ARGS="--wf-steal-parent"
Socket –aware scheduler	Este planificador asignará tareas de nivel superior (profundidad 1) a un nodo NUMA establecido por el usuario antes de la creación de la tarea, mientras que las tareas anidadas se ejecutarán en el mismo nodo que su padre. Para ello, el usuario debe llamar a la función nanos_current_socket antes de ejecutar tareas para establecer el nodo NUMA al que se asignará la tarea	$export NX_ARGS="--schedule=socket" $export NX_ARGS="--cores-per-socket"
Affinity	Tiene en cuenta dónde se encuentran los datos utilizados por una tarea.	$export NX_ARGS="--schedule=affinity"

Versioning	Este programador puede manejar múltiples implementaciones para la misma tarea y da soporte a la cláusula implements. El planificador perfila automáticamente cada implementación de tarea y elige la implementación más adecuada cada vez que se debe ejecutar la tarea. Cada subproceso tiene su propia cola de tareas, donde se agregan implementaciones de tareas dependiendo de las decisiones del planificador	$export NX_ARGS="--schedule=versioning"
Bottom level-aware scheduler (Botlev)	Este planificador está diseñado para máquinas heterogéneas de un solo ISA que mantienen dos tipos de núcleos (rápido y lento, como ARM big.LITTLE), por tanto es consciente de la asimetría de la arquitectura de la CPU. El planificador detecta dinámicamente la ruta más larga del gráfico de dependencia de tareas y asigna las tareas que pertenecen a esta ruta (tareas críticas) a los núcleos rápidos del sistema.	$export NX_ARGS="--schedule="--schedule=botlev"

Tabla 6.1: *Principales políticas de programación de OmpSs [Omps, 2018]*

Antes de tener en cuenta cualquier criterio de programación, se deben considerar cuatro modificadores del planificador de tareas, como son [Ompss, 2018]:

- **Política de regulación en la creación de tareas**. Justo cuando se va a crear una nueva tarea, el sistema de tiempo de ejecución determina (de acuerdo con una política de aceleración) si debe crearlo e insertarlo en el sistema de programación o simplemente crear una descripción mínima de la tarea y ejecutarla de inmediato en el entorno actual
- **Prioridad de Tarea**. Al insertar una nueva tarea en una cola de prioridad, si la nueva tarea tiene la misma prioridad que otra en la cola, la nueva se insertará antes / después de la existente (de acuerdo con el comportamiento FIFO / LIFO entre tareas con la misma prioridad).
- **Sucesor inmediato**. Liberando la última dependencia al salir de una tarea. Cuando una tarea se encuentra en el gráfico de dependencia y su último predecesor inmediato finaliza la ejecución, podemos ejecutar las tareas bloqueadas de forma inmediata en lugar de agregarlo a la lista de espera. A veces, el sucesor inmediato no es una opción configurable. Verifica la descripción y/o los parámetros específicos del programador para más detalles.

- **Continuación de los padres cuando se ejecutó el último hijo**. Al ejecutar aplicaciones OpenMP / OmpSs anidadas puede suceder que una tarea determinada se bloquee debido a la ejecución de una directiva “taskwait” o “taskwait on”. Cuando el último hijo haya finalizado su ejecución, el padre será reasignado de inmediato en lugar de ser devuelto a la lista de espera (lo que podría retrasar su ejecución).

En este trabajo de investigación en general se hará uso de la política de programación Bottom level-aware scheduler (Botlev), que hace que el planificador Nanos++ de OmpSs sea consciente de la asimetría de la CPU de la placa Odroid XU4, permitiendo hacer una optimización energética más eficaz durante la ejecución del programa de detección facial, como se verá más adelante.

6.4.4.3 Políticas basadas en los parámetros “step” y “scalaFactor”

Como se ha comentado en el Capítulo 5, en el programa de detección facial existen dos parámetros cuyo valor puede influir considerablemente en el rendimiento, precisión y consumo energético que se produce al ejecutar el programa de detección. Estos parámetros son “step”, que establece el número de *pixeles* que se desplaza la ventana de detección en cada iteración, y “scalaFactor”, que marca el re-escalado de la imagen de entrada en cada iteración.

Para visualizar el efecto que tiene estos dos parámetros en el programa de detección, en las tablas 6.2, 6.3, 6,4 y 6,5 se puede ver el número de falsos negativos, falsos positivos, el tiempo de ejecución y Error total (numero de falsos positivos más el número de falsos negativos), en función del valor de “scalaFactor” y ”step”. Para la obtención de los resultados se han usado dos bases de datos de rostros públicas sobre las que se han realizado las pruebas experimentales: **Base-750** [Steph, 2017], y **Base-450** [Calte, 2017]. La primera está compuesta por 750 imágenes con un solo rostro cada una y una resolución de: 480x640 pixeles. La segunda dispone de 450 imágenes con un solo rostro cada una y una resolución de: 896x592 pixeles.

BASE 450				
STEP (Scalafactor 1.2)	FALSO POSITIVOS	FALSOS NEGATIVOS	TIEMPO DE EJECUCION (MIN)	ERROR TOTAL
1	6	44	15:06	50
2	1	113	6:01	114
3	0	264	5:01	264
4	0	321	4:23	321
5	0	424	4:11	424

***Tabla 6.2**: Datos obtenidos para diferentes valores del parámetro "step", y "scalaFactor"=1,2.*

BASE 750				
STEP (Scalafactor 1.2)	FALSO POSITIVOS	FALSOS NEGATIVOS	TIEMPO DE EJECUCION (MIN)	ERROR TOTAL
1	18	17	13:46	35
2	0	58	6:19	58
3	0	225	5:09	225
4	0	475	4:53	475
5	0	576	4:21	576

***Tabla 6.3**: Datos obtenidos para diferentes valores del parámetro "step", y "scalaFactor"=1,2.*

BASE 450				
ScalaFactor (Step 1)	FALSO POSITIVOS	FALSOS NEGATIVOS	TIEMPO DE EJECUCION (MIN)	ERROR TOTAL
1,1	16	22	23:50	38
1,15	9	29	18:29	38
1,2	6	44	15:06	50
1,25	5	50	13:18	55
1,3	5	63	11:54	68
1,35	7	66	11:04	73
1,4	4	66	10:23	70
1,5	7	117	9:37	124

Tabla 6.4: *Datos obtenidos para diferentes valores del parámetro "scalaFactor", y "step"=1.*

BASE 750				
ScalaFactor (Step 1)	FALSO POSITIVOS	FALSOS NEGATIVOS	TIEMPO DE EJECUCION (MIN)	ERROR TOTAL
1,1	48	13	22:07	61
1,15	24	15	16:34	39
1,2	13	17	13:48	30
1,25	13	24	12:12	37
1,3	6	28	11:03	34
1,35	9	34	10:17	43
1,4	2	34	9:45	36
1,5	2	58	8:52	60

***Tabla 6.5**: Datos obtenidos para diferentes valores del parámetro "scalaFactor", y "step"=1.*

Las tablas anteriores se resumen en la figura 6.4, donde se puede ver la evolución gráfica del Error Total en función de distintos valores del parámetro “step” y “scalaFactor”, respectivamente.

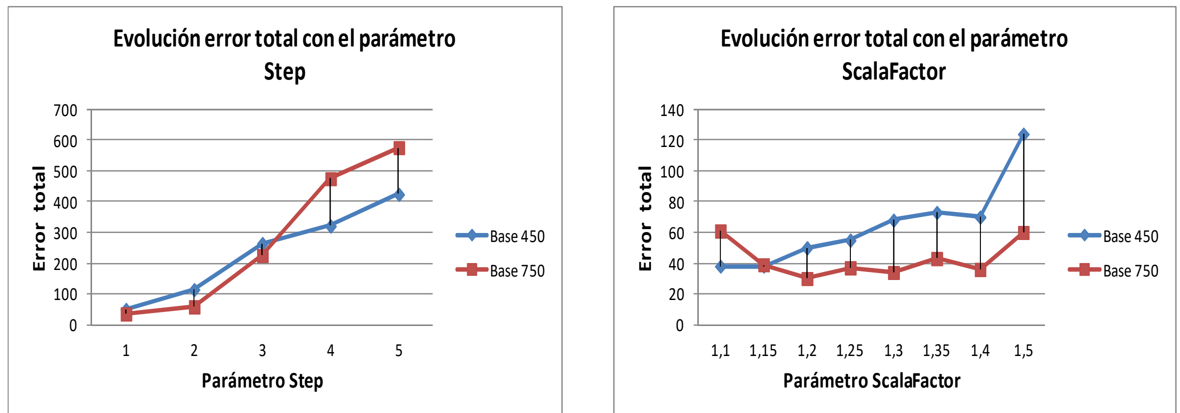


***Figura 6.4**: Evolución error total en función de los parámetros “step” y “scalaFactor”.*

De los resultados obtenidos se pueden destacar las siguientes conclusiones:

1. El parámetro “step”, es más sensible que “scalaFactor”, respecto a la precisión del programa. Cualquier valor mayor que 2 produce un gran incremento de los errores de detección, siendo el valor óptimo para la detección el 1. Sin embargo, se produce una mejora considerable del rendimiento del programa ya que disminuye considerablemente el tiempo de ejecución. Por tanto, para un “step” igual a 2, aunque se pierde algo de precisión por el incremento del error total, puede ser de interés su consideración cuando el objetivo sea incrementar el rendimiento y la disminución del consumo energético.

2. El parámetro “scalaFactor”, en contraposición con el parámetro “step”, es menos sensible en cuanto a la precisión, el error total aumenta lentamente a medida que se incrementa el valor de este parámetro. También aumenta el rendimiento del programa pero a menor escala que con el parámetro “step”. En este contexto, se podría evaluar como valor óptimo para este parámetro el valor que proporcione el mejor rendimiento y provoque un menor consumo energético.

6.4.5 Adaptación de OmpSs a arquitecturas asimétricas

Con el reciente auge de las arquitecturas asimétricas en el mundo de la computación de alto rendimiento (HPC, High performance Computing), Los desarrolladores de OmpSs han introducido recientemente un nuevo planificador dentro de las políticas de programación del planificador de tareas de OpmSs Nanos++, denominado *Bottom level-aware scheduler* (Botlev) [Chron, 2015] específico para este nuevo tipo de arquitecturas. Botlev recoge las ideas de los planificadores tradicionales basados en arquitecturas heterogéneas, distinguiendo únicamente dos tipos de nodos de cómputo (un nodo rápido formado por los núcleos de tipo Big, y un nodo de cómputo lento formado por los núcleos de tipo LITTLE) y eliminando el cálculo de los costes asociados a la transferencia de datos.

Una técnica utilizada para obtener un mejor rendimiento en programas paralelos basados en tareas es intentar que las tareas críticas finalicen su ejecución en el menor tiempo posible. El planificador Botlev persigue este objetivo intentando calcular de manera dinámica qué tareas pertenecen al camino crítico del DAG asociado al problema, y ejecutar estas tareas sobre núcleos rápidos con el objetivo de finalizar su ejecución cuanto antes. El objetivo de garantizar que las tareas del camino crítico se ejecutan en los núcleos rápidos cuanto antes es asegurarse de que al ejecutar las tareas críticas, éstas liberarán dependencias con nuevas tareas que pasarán a estar listas para ejecución, y así intentar conseguir que nunca haya "*worker threads*" ociosos a causa de que no existan tareas listas para ejecutar, lo que disminuiría el rendimiento global de la aplicación. La principal diferencia entre Botlev y los planificadores tradicionales para sistemas heterogéneos es que Botlev toma las decisiones de forma dinámica sin necesidad de conocer de antemano información sobre las tareas, o la forma del árbol de dependencias del programa [Luisc, 2016].

Por tanto, la principal aportación del planificador Botlev respecto a una ejecución del programa con el planificador Nano++ sin aplicar ninguna política de programación, es que Botlev es consciente de la asimetría de la arquitectura, lo cual permite tomar decisiones dinámicas basada en el camino crítico que hacen que la ejecución del programa sea más eficiente energéticamente que con el planificador

convencional Nanos++, que no es consciente de dicha asimetría. Por esta razón Botlev solo tiene sentido aplicarlo a la placa experimental Odroid 4XU.

En la figura 6.5, se puede ver, el DAG (Gráfico Acíclico Dirigido) desarrollado para encontrar restricciones y dependencias con el fin de optimizar el código fuente para una paralelización y consumo energético adecuada.

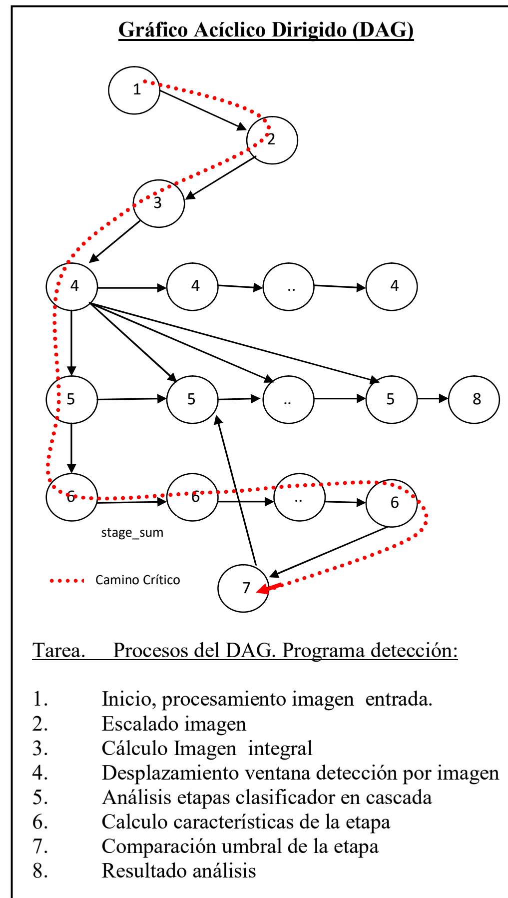


***Figura 6.5**: Gráfico Acíclico Dirigido (DAG) del programa de detección facial.*

La prioridad de una tarea viene dada por un número entero positivo, que representa la longitud del camino más largo desde la tarea actual hasta una tarea hija del grafo de dependencias. Hay que destacar que esta forma de calcular el camino crítico detecta las tareas que pertenecen al camino más largo, pero eso no implica que sean aquellas que más van a retrasar la ejecución en caso de que no se ejecuten de manera prioritaria. Para calcular esto, sería necesario conocer de antemano las necesidades de

cada tarea, o ir almacenando resultados parciales de ejecución de manera dinámica para tomar decisiones en el futuro.

Una tarea puede ser ejecutada cuando todas sus tareas predecesoras han finalizado la ejecución. Cuando esto ocurre, Botlev inserta la tarea en una cola de tareas pendientes para ir ejecutándolas en función de la prioridad asignada, o en orden de inserción en caso de empate. Estas tareas son asignadas a los diferentes núcleos según vayan finalizando la ejecución de las tareas previas. Según la prioridad de la tarea, Botlev distingue dos tipos de colas en las que insertar una tarea: la cola con las tareas pertenecientes al camino crítico, y la cola con el resto de tareas.

Cuando un núcleo Big finaliza la ejecución de una tarea, éste ejecutará la siguiente tarea de la cola de tareas críticas, mientras que un núcleo LITTLE ejecutará la primera tarea de la cola de tareas no críticas. De esta forma se consigue asignar las tareas críticas a núcleos Big, mientras que las tareas no críticas serán ejecutadas por núcleos lentos. En el caso de que un núcleo Big no disponga de tareas críticas que ejecutar, ejecutará la siguiente tarea de la lista de tareas no críticas. El robo de tareas en sentido contrario (es decir, que los núcleos LITTLE ejecuten tareas críticas) es un parámetro adicional que se puede seleccionar en tiempo de ejecución, pero que se encuentra desactivado por defecto.

El modelo de programación de OmpSs proporciona mecanismos sencillos y no invasivos para la paralelización de programas, a grosso modo, el planificador convencional de OmpSs descompone el código en un conjunto de tareas, identificando las dependencias entre ellas, y lanzando a ejecución únicamente tareas listas, es decir, aquellas cuyas dependencias han sido satisfechas para su ejecución en los distintos núcleos computacionales del sistema [Luisc, 2016].

Una vez que un código anotado ha sido compilado con el compilador de OmpSs denominado Mercurium [Mercu, 2018], éste puede ser ejecutado a través del *runtime* de OmpSs, consistente en una serie de bibliotecas encargadas de controlar la ejecución del programa e intentar finalizar la ejecución de la manera más eficiente posible.

Según esto, OmpSs, al igual que OpenMP, es muy adecuado en aquellos problemas donde se desea aplicar la misma función o conjunto de instrucciones a diferentes partes (disjuntas) de un universo de datos. Este tipo de problema es el más común cuando se trata de ejecutar instrucciones repetitivas sobre un número de elementos a operar conocido. Normalmente estas operaciones se implementan con la instrucción *for*. De esta forma el programador no tiene que cambiar la semántica de sus programas, ya que OmpSs se encargará de dividir los índices y asignarlos a los distintos hilos generados como ya se realizó cuando se aplicó OpenMP.

Se ha seguido la misma técnica para la paralelización del algoritmo de detección facial que se vio en el capítulo 5 apartado 5.6. Donde a partir del análisis con la herramienta de *profiling* Gperftools, y teniendo en cuenta las dependencias entre cada etapa del clasificador en cascada para optimizar el rendimiento del sistema, se ha paralelizado el software con OmpSs como se hizo con OpenMP, aplicando las directivas a las funciones a optimizar "evalWeakClassifier" y "runCascadeClassifier. También, se ha mantenido la división de la variable "stage_sum", de la que se obtiene la suma acumulada de cada una de las características de la etapa, para favorecer la creación de hilos del programa en paralelo.

En este caso, para optimizar el código, solo es necesario usar las directivas #pragma de OmpSs: "#pragma omp for Schedule(static)" y "#pragma omp task". La primera le indica al compilador que divida la ejecución del bucle en varios hilos en paralelo, lo que permitirá ejecutar al mismo tiempo cada hilo en un núcleo de CPU. La segunda marca la siguiente declaración (bucle *for*) como una tarea. Después de esto el programa seguirá ejecutándose en un solo hilo. En la figura 6.6, se puede ver el mismo fragmento de código fuente que se paralelizó en el capítulo 5 con OpenMP, pero en este caso con las directivas de OmpSs.

```
#pragma omp for schedule(static)
for( x = 0; x <= x2; x += step )
#pragma omp task
   for( y = y1; y <= y2; y += step ) //desplazamos la ventana de detección por columnas
     {
            p.x = x; //coordenadas donde comienza la ventana de detección
            p.y = y; //la ventana de detección es de 24x24 pixels

            result = runCascadeClassifier( cascade, p, 0 );

            if( result > 0 )  //Sí se cumple se trata de un rostro y dibujo rectagulo en la
zona
             {
               // factor= factor de escala de la imagen de entrada
               MyRect r = {myRound(x*factor), myRound(y*factor), winSize.width,
winSize.height};
               vec->push_back(r);
             }
     }
```

***Figura 6.6**: Fragmento de código fuente de la función "ScaleImage_Invoker" donde se llama a la función "runCascadeClassifier", con las directivas de OmpSs incluidas.*

Al igual que se vio en el apartado 5.6, para evitar posibles conflictos de acceso a variables compartidas, se usa la directiva también de OmpSs: "#pragma omp critical". Garantizando, de este modo, que la sección crítica que viene inmediatamente después se ejecute en un solo hilo, ver figura 6.7.

```
#pragma omp for schedule(static)
For ( j = 0; j < stages_array[i]; j+=2)   // stages_array[i] = número de características de la etapa i
 {
    stage_sum_array[0]= evalWeakClassifier(variance_norm_factor, p_offset, haar_counter, w_index, r_index);
    stage_sum_array[1]= evalWeakClassifier(variance_norm_factor, p_offset, haar_counter, w_index, r_index);
    stage_sum_array[2]= evalWeakClassifier(variance_norm_factor, p_offset, haar_counter, w_index, r_index);
    stage_sum_array[3]= evalWeakClassifier(variance_norm_factor, p_offset, haar_counter, w_index, r_index);
    stage_sum+=stage_sum_array[0]+stage_sum_array[1]+stage_sum_array[2]+stage_sum_array[3];

   // stage_sum = suma de los valores de los clasificadores debiles de la etapa
   // stages_thresh_array[i] = umbral de la etapa eveluada
  if( stage_sum  > stages_thresh_array[i] )  //si stage_sum supera el  umbral  la etapa del clasificador es satisfecha
  {
        #pragma omp critical
        if ( stage_sum  > stages_thresh_array[i] )  return  1;
  }
}
```

***Figura 6.7**: Fragmento de Código fuente de la función "runCascadeClassifier" donde se llama a la función "evalWeakClassifier", con las directivas de OmpSs incluidas.*

Si con estos cambios se vuelve a ejecutar el software sobre la placa Odroid XU4, se obtienen los resultados de tiempo de ejecución que se muestran en la figura 6.8, donde se puede ver una comparación entre los tiempos de ejecución antes de aplicar las directivas OmpSs (ejecución secuencial) y después de aplicar las directivas convencionales de OmpSs sin aplicar ninguna política de programación del planificador de tareas Nanos++. El resultado, tanto en tiempo de ejecución como en consumo energético, es muy similar al obtenido al aplicar las directivas de OpenMP (ver apartado 5.5.7).

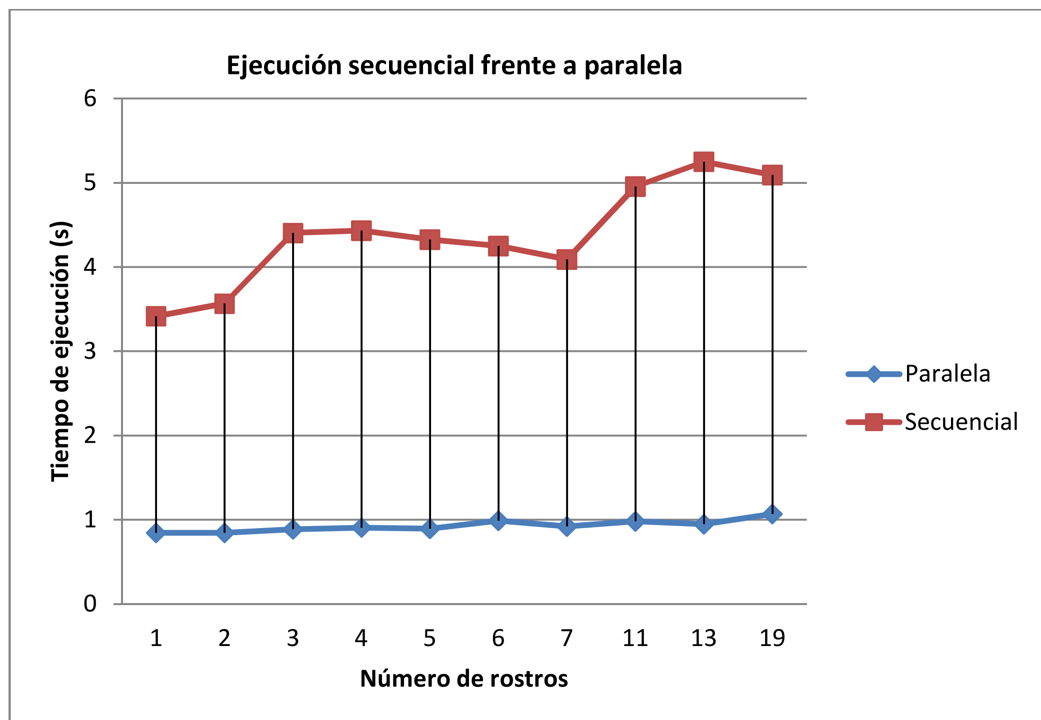


Figura 6.8: *Comparación entre tiempos de ejecución secuencial y paralela al aplicar la directivas de OmpSs al programa de detección facial, y ejecutarlo en la placa Odroid XU4.*

En la figura 6.9, se puede ver la comparación entre la energía consumida antes de aplicar las directivas OmpSs (ejecución secuencial) y después de aplicar las directivas OmpSs, pero sin asignar aún ninguna política de optimización del consumo energético.

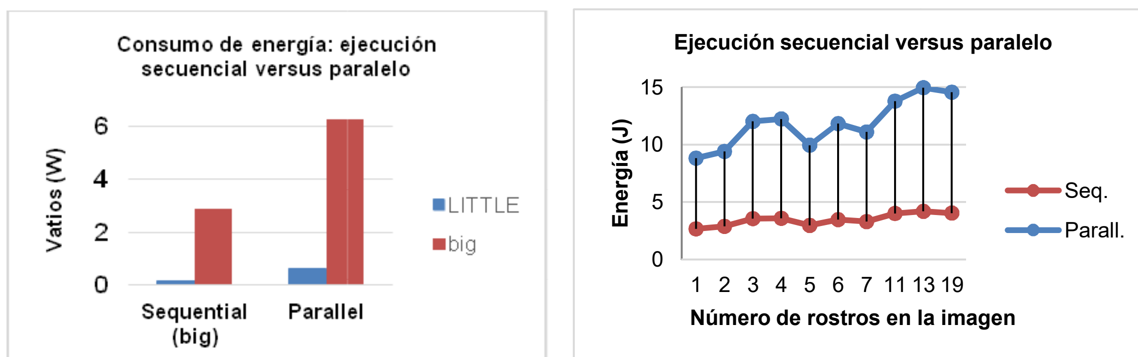


Figura 6.9: Comparación entre consumos energético en la ejecución secuencial y paralela al aplicar la directivas de OmpSs al programa de detección facial en la placa Odroid XU4, sin aplicar políticas de optimización del consumo energético.

6.4.6 Resultados optimización del consumo energético

Atendiendo a todos los procedimientos y técnicas que se han visto en los apartados anteriores, aplicables a arquitecturas asimétricas como la Big-LITTLE del procesador ARM de la placa de estudio Odroid XU4, que están relacionadas con el planificador de tareas de OmpSs Botlev, al escalado de frecuencias que se puede configurar en cada uno de los dos cluster Big y LITTLE de la CPU de dicha placa, y a la adecuada configuración de los parámetros del programa de detección facial: ”step” y “scalaFactor”, se ha realizado varias pruebas experimentales con las bases de rostros de referencia utilizadas: Base-750 [Steph, 2017], y Base-450 [Calte, 2017], con el objetivo de obtener los parámetros más adecuados para alcanzar una buena optimización del consumo energético sobre la placa Odroid XU4, aprovechando todos sus recursos y técnicas estudiadas.

Con ello se podrá comprobar experimentalmente si las arquitecturas asimétricas permiten optimizar la eficiencia energética asignando las tareas a núcleos de distintas características en función de los requisitos de rendimiento y consumo de cada tarea, además de ajustar el rendimiento y consumo de cada núcleo utilizando las técnicas de escalado de frecuencia que se encuentran disponibles en los procesadores modernos.

En cuanto a los dos parámetros configurables del software, el factor "scalaFactor", que determina la disminución de la resolución de la imagen que se realiza en cada iteración, y el factor "step", que determina el desplazamiento por toda la imagen de la ventana de detección en cada iteración, su variación puede producir una disminución del tiempo de ejecución y del consumo energético, en contraprestación a la precisión de la detección de rostros, como se ha mostrado en el apartado 6.4.4.3. Por tanto, es necesario hacer una correcta selección de dichos parámetros en el proceso de optimización para evitar perder precisión en la detección de rostros.

Para los experimentos, solo se modificó la frecuencia del *clúster* Big, ya que modificar la frecuencia del *clúster* LITTLE no tiene un impacto significativo en el consumo de energía, pero tiene un gran impacto en el tiempo de ejecución, como se muestra en [Luisc, 2016]. En este contexto, a continuación se muestran los resultados de consumo energético obtenidos para las diferentes frecuencias del *cluster* Big, en función del tiempo de ejecución, y los parámetros "step" y "scalaFactor", después de procesar 1.200 imágenes incluidas en las dos bases de datos experimentales (Base-450 y Base-750). Todas las pruebas se hacen con objeto de encontrar un equilibrio óptimo entre tiempo de ejecución y consumo de energía.

Para una frecuencia de 2000 Mhz la tabla 6.6 muestra los valores obtenidos sobre la placa Odroid XU4 y las figuras 6.10 y 6.11, una representación gráfica de los mismos para una mejor interpretación de los resultados.

Placa Odroid XU4						
Frecuencia Clúster Big (MHz)	Frecuencia Clúster LITTLE (MHz)	Parámetro "step"	Parámetro "scaleFactor"	Tiempo Ejecución (s)	Consumo Energético (Ws)	Error de detección
2000	1400	1	1,1	2414	5,638	72
2000	1400	1	1,2	1342	5,588	70
2000	1400	1	1,3	1003	5,816	88
2000	1400	1	1,4	854	5,815	105
2000	1400	2	1,1	698	5,621	90
2000	1400	2	1,2	387	5,364	177
2000	1400	2	1,3	303	5,070	270
2000	1400	2	1,4	258	4,894	341
2000	1400	3	1,1	467	4,556	222
2000	1400	3	1,2	269	4,180	476
2000	1400	3	1,3	200	3,959	728
2000	1400	3	1,4	173	3,791	887
2000	1400	4	1,1	391	3,716	428
2000	1400	4	1,2	217	3,415	782
2000	1400	4	1,3	167	3,257	1000
2000	1400	4	1,4	146	3,071	1088

Tabla 6.6: *Resultados de consumo de energía, tiempo de ejecución y parámetros "scalaFactor" y "step", para las frecuencias del cluster: Big = 2000 MHz, LITTLE = 1400MHz*

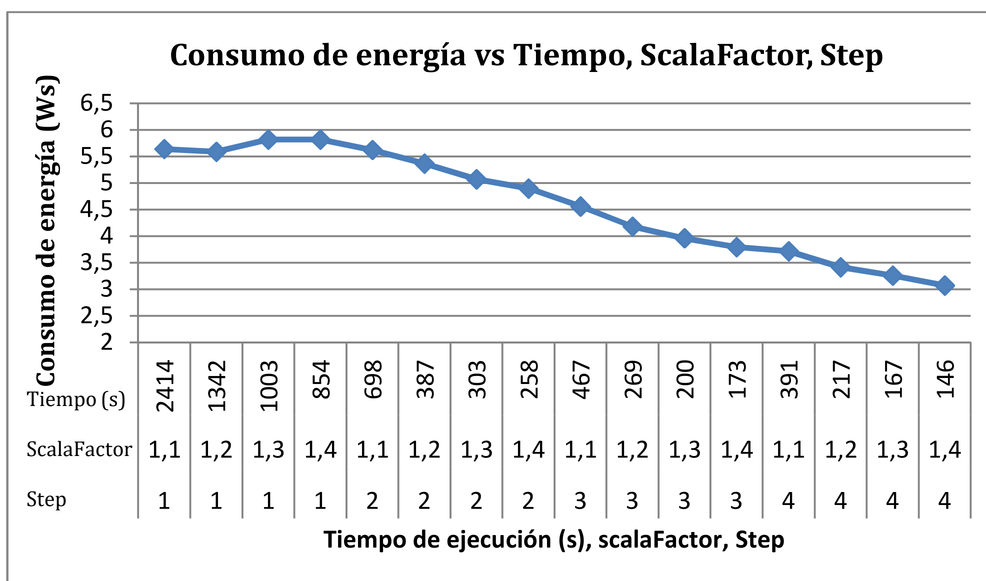


Figura 6.10*: Consumo de energía de ejecución vs tiempo de ejecución y parámetros "scalaFactor" y "step", para las frecuencias del cluster: Big = 2000 MHz, LITTLE = 1400MHz*

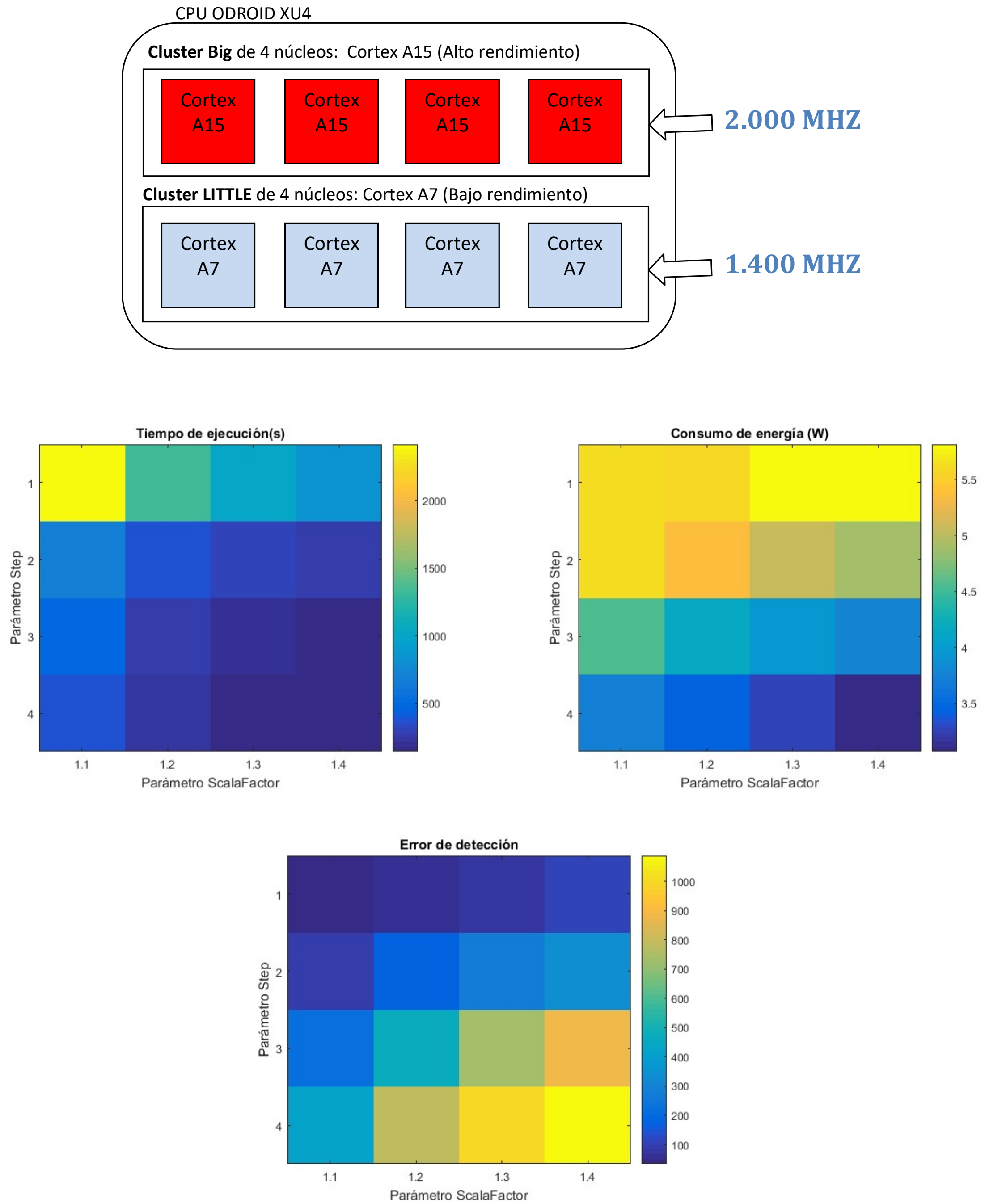


Figura 6.11: *Tiempo de ejecución, consumo de potencia y error de detección según los parámetros "scalaFactor" y "step", para las frecuencias del clúster: Big = 2000 MHz , LITTLE = 1400 MHz*

La tabla 6.7, muestra los valores exactos obtenidos sobre la placa Odroid XU4 para una frecuencia del *cluster* Big de 1500MHz, y las figuras 6.12 y 6.13, una representación gráfica de los mismos.

Placa Odroid XU4						
Frecuencia Clúster Big (MHz)	Frecuencia Clúster LITTLE (MHz)	Parámetro "step"	Parámetro "scaleFactor"	Tiempo Ejecución (s)	Consumo Energético (Ws)	Error de detección
1500	1400	1	1,1	2646	3,019	71
1500	1400	1	1,2	1490	2,955	73
1500	1400	1	1,3	1135	2,888	98
1500	1400	1	1,4	972	2,871	101
1500	1400	2	1,1	851	2,596	87
1500	1400	2	1,2	478	2,442	175
1500	1400	2	1,3	369	2,367	272
1500	1400	2	1,4	315	2,328	344
1500	1400	3	1,1	567	2,214	221
1500	1400	3	1,2	321	2,058	473
1500	1400	3	1,3	240	1,997	726
1500	1400	3	1,4	205	1,938	886
1500	1400	4	1,1	455	1,990	430
1500	1400	4	1,2	253	1,827	783
1500	1400	4	1,3	195	1,757	999
1500	1400	4	1,4	169	1,653	1087

Tabla 6.7: *Resultados de consumo de energía, tiempo de ejecución y parámetros "scalaFactor" y "step", para las frecuencias del cluster: Big = 1500 MHz, LITTLE = 1400MHz*

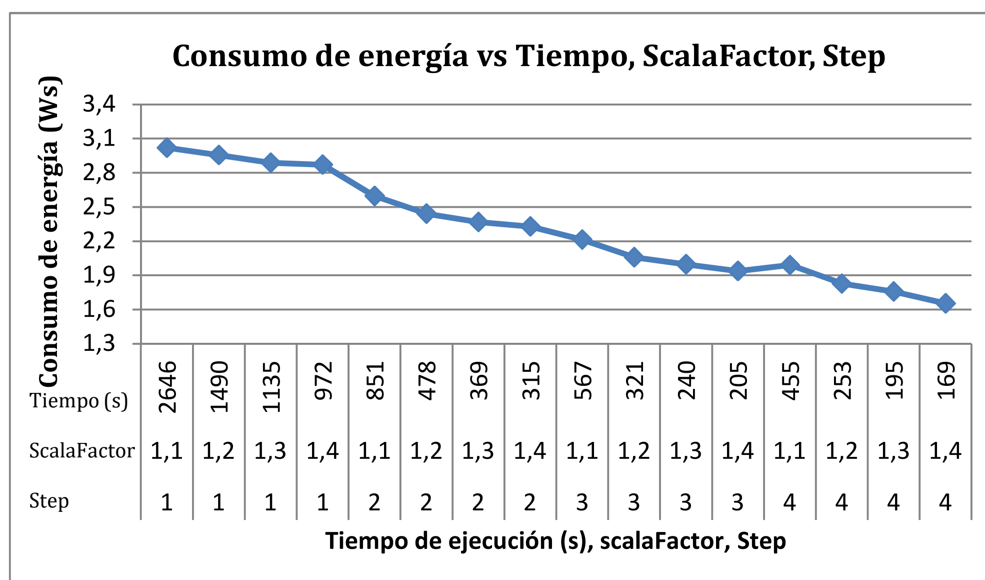


Figura 6.12*: Consumo de energía de ejecución vs tiempo de ejecución y parámetros "scalaFactor" y "step", para las frecuencias del cluster: Big = 1500 MHz, LITTLE = 1400MHz*

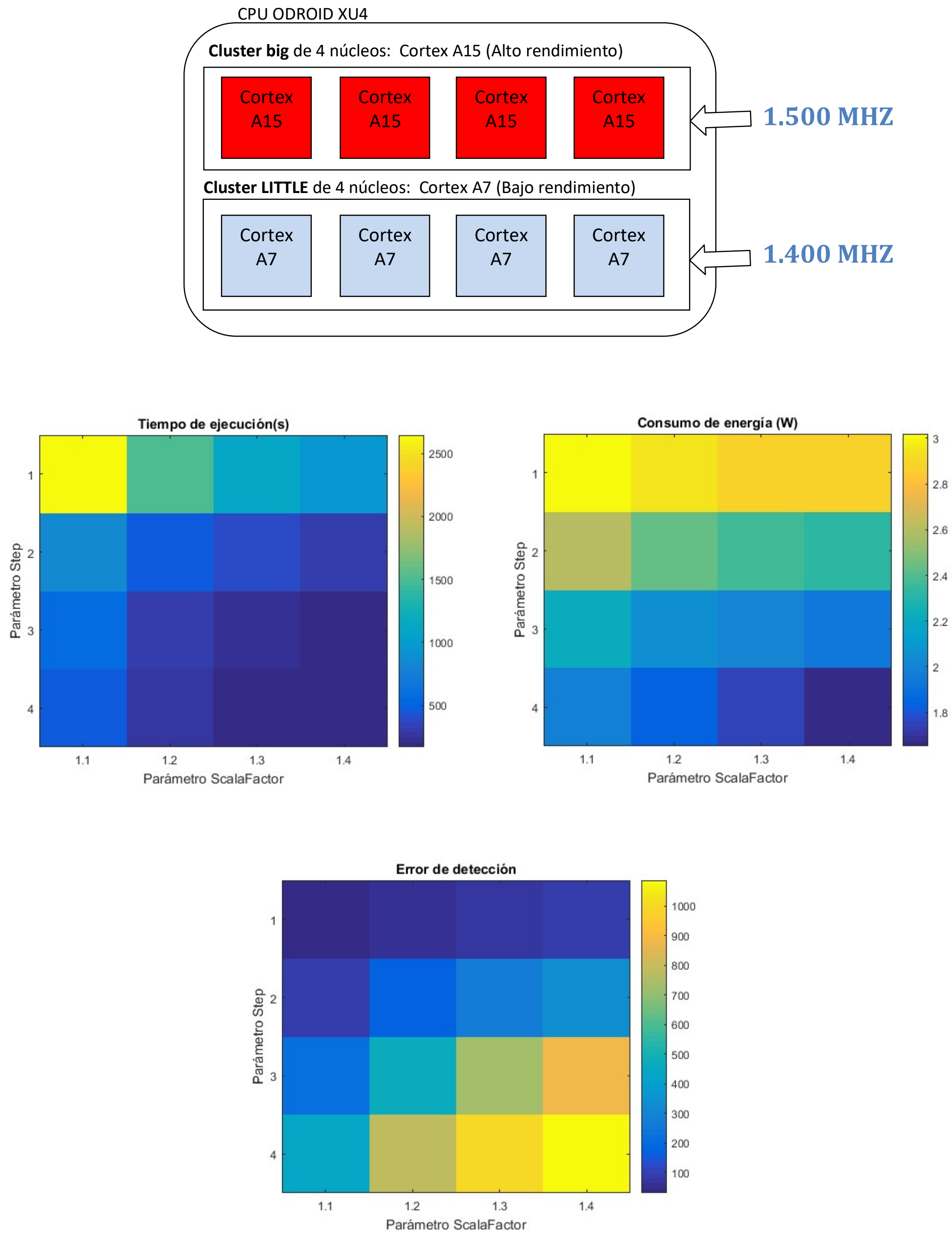


***Figura 6.13**: Tiempo de ejecución, consumo de potencia y error de detección según los parámetros "scalaFactor" y "step", para las frecuencias del clúster: Big = 1500 MHz , LITTLE = 1400 MHz*

La tabla 6.8, muestra los valores exactos obtenidos sobre la placa Odroid XU4 para una frecuencia del *cluster* Big de 1000MHz, y las figuras 6.14 y 6.15, una representación gráfica de los mismos.

Placa Odroid XU4						
Frecuencia Clúster Big (MHz)	Frecuencia Clúster LITTLE (MHz)	Parámetro "step"	Parámetro "scaleFactor"	Tiempo Ejecución (s)	Consumo Energético (Ws)	Error de detección
1000	1400	1	1,1	3534	1,859	65
1000	1400	1	1,2	2027	1,813	64
1000	1400	1	1,3	1530	1,784	86
1000	1400	1	1,4	1307	1,762	112
1000	1400	2	1,1	1165	1,593	92
1000	1400	2	1,2	664	1,519	176
1000	1400	2	1,3	506	1,470	276
1000	1400	2	1,4	428	1,448	343
1000	1400	3	1,1	749	1,417	223
1000	1400	3	1,2	420	1,324	476
1000	1400	3	1,3	321	1,300	727
1000	1400	3	1,4	273	1,273	882
1000	1400	4	1,1	568	1,337	431
1000	1400	4	1,2	327	1,244	783
1000	1400	4	1,3	247	1,207	1000
1000	1400	4	1,4	210	1,169	1087

Tabla 6.8: *Resultados de consumo de energía, tiempo de ejecución y parámetros "scalaFactor" y "step", para las frecuencias del cluster: Big = 1000 MHz, LITTLE = 1400MHz*

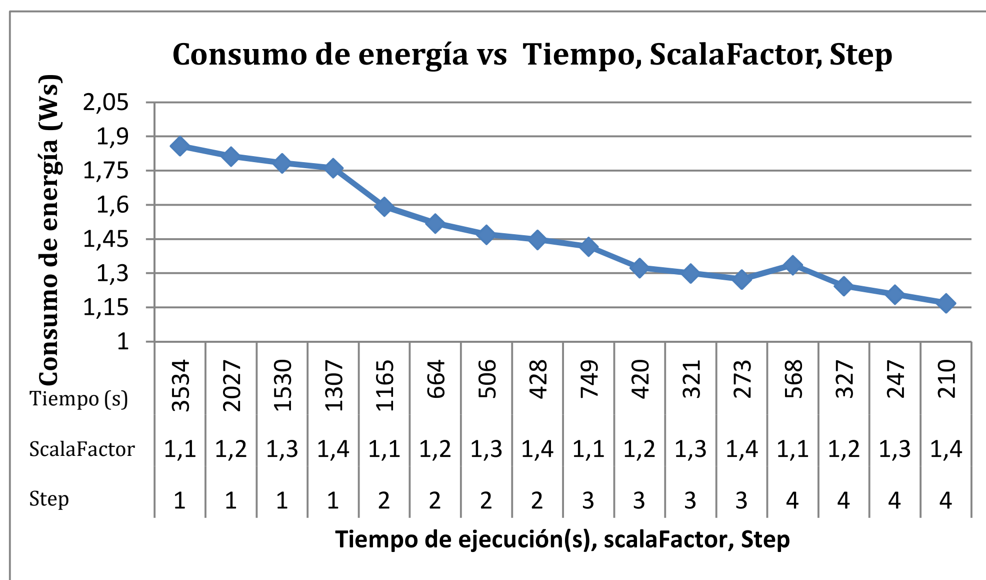


Figura 6.14*: Consumo de energía de ejecución vs tiempo de ejecución y parámetros "scalaFactor" y "step", para las frecuencias del cluster: Big = 1000 MHz, LITTLE = 1400MHz*

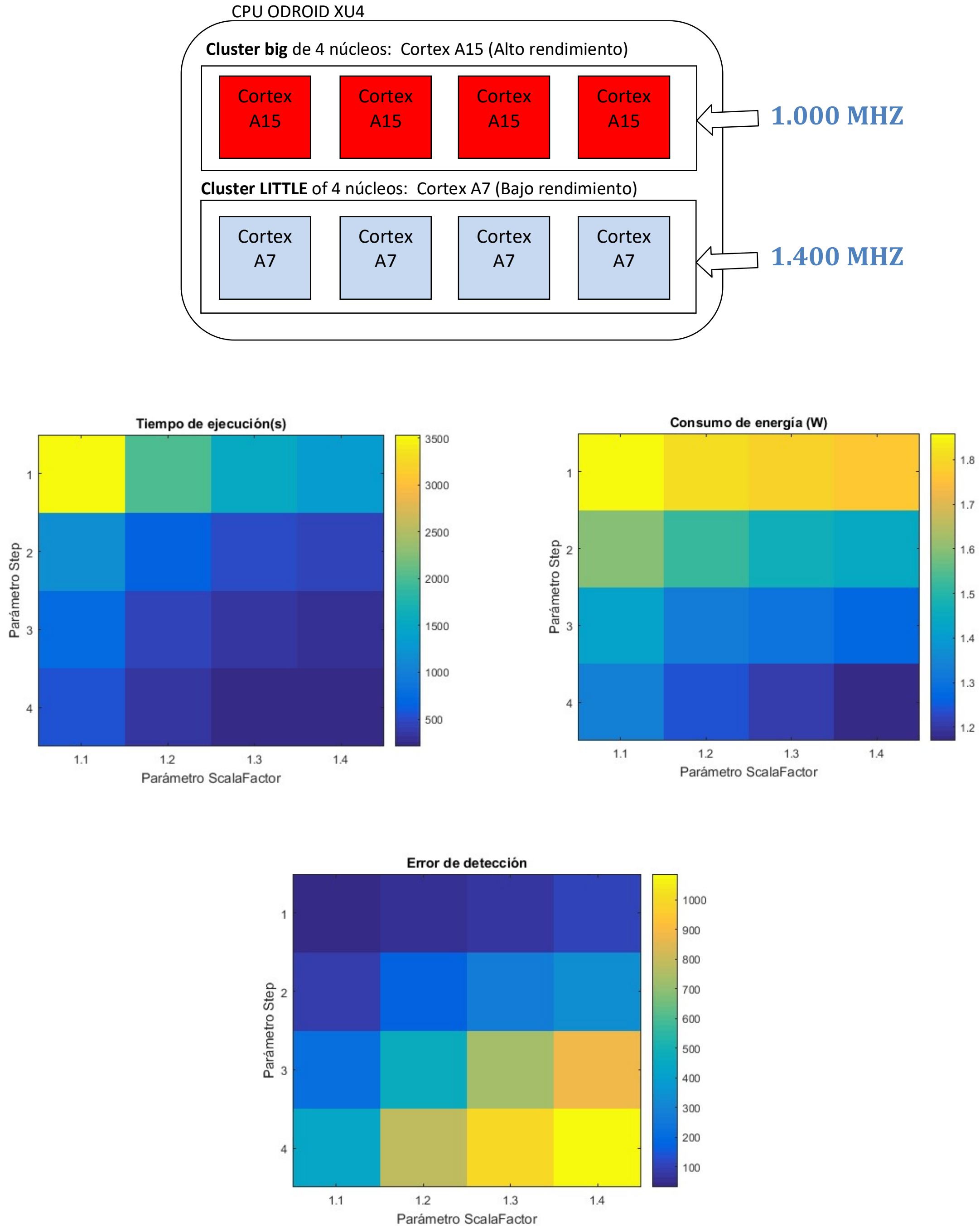


***Figura 6.15**: Tiempo de ejecución, consumo de potencia y error de detección según los parámetros "scalaFactor" y "step", para las frecuencias del clúster: Big = 1000 MHz , LITTLE = 1400 MHz*

La tabla 6.10, muestra los valores exactos obtenidos sobre la placa Odroid XU4 para una frecuencia del *cluster* Big de 800MHz, y las figuras 6.16 y 6.17, una representación gráficas de los mismos.

Placa Odroid XU4						
Frecuencia Clúster Big (MHz)	Frecuencia Clúster LITTLE (MHz)	Parámetro "step"	Parámetro "scaleFactor"	Tiempo Ejecución (s)	Consumo Energético (Ws)	Error de detección
800	1400	1	1,1	4244	1,525	60
800	1400	1	1,2	2405	1,490	67
800	1400	1	1,3	1817	1,467	88
800	1400	1	1,4	1543	1,449	105
800	1400	2	1,1	1409	1,305	86
800	1400	2	1,2	801	1,235	178
800	1400	2	1,3	611	1,207	274
800	1400	2	1,4	516	1,182	340
800	1400	3	1,1	879	1,170	225
800	1400	3	1,2	502	1,107	475
800	1400	3	1,3	382	1,071	731
800	1400	3	1,4	323	1,048	882
800	1400	4	1,1	689	1,106	432
800	1400	4	1,2	384	1,038	784
800	1400	4	1,3	293	1,003	1001
800	1400	4	1,4	248	0,975	1088

Tabla 6.9: *Resultados de consumo de energía, tiempo de ejecución y parámetros "scalaFactor" y "step", para las frecuencias del cluster: Big = 800 MHz, LITTLE = 1400MHz*

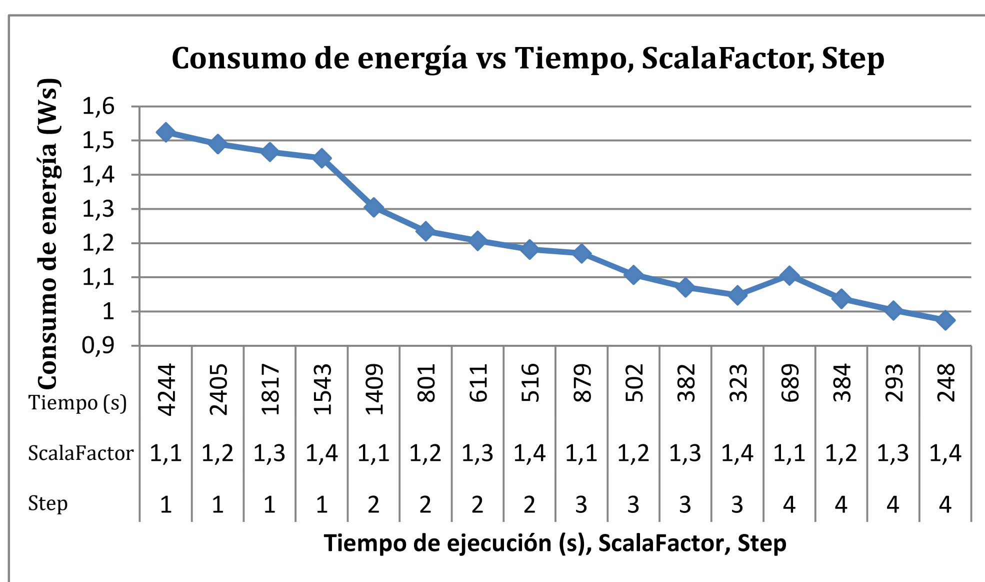


Figura 6.16*: Consumo de energía de ejecución vs tiempo de ejecución y parámetros "scalaFactor" y "step", para las frecuencias del cluster: Big = 800 MHz, LITTLE = 1400MHz*

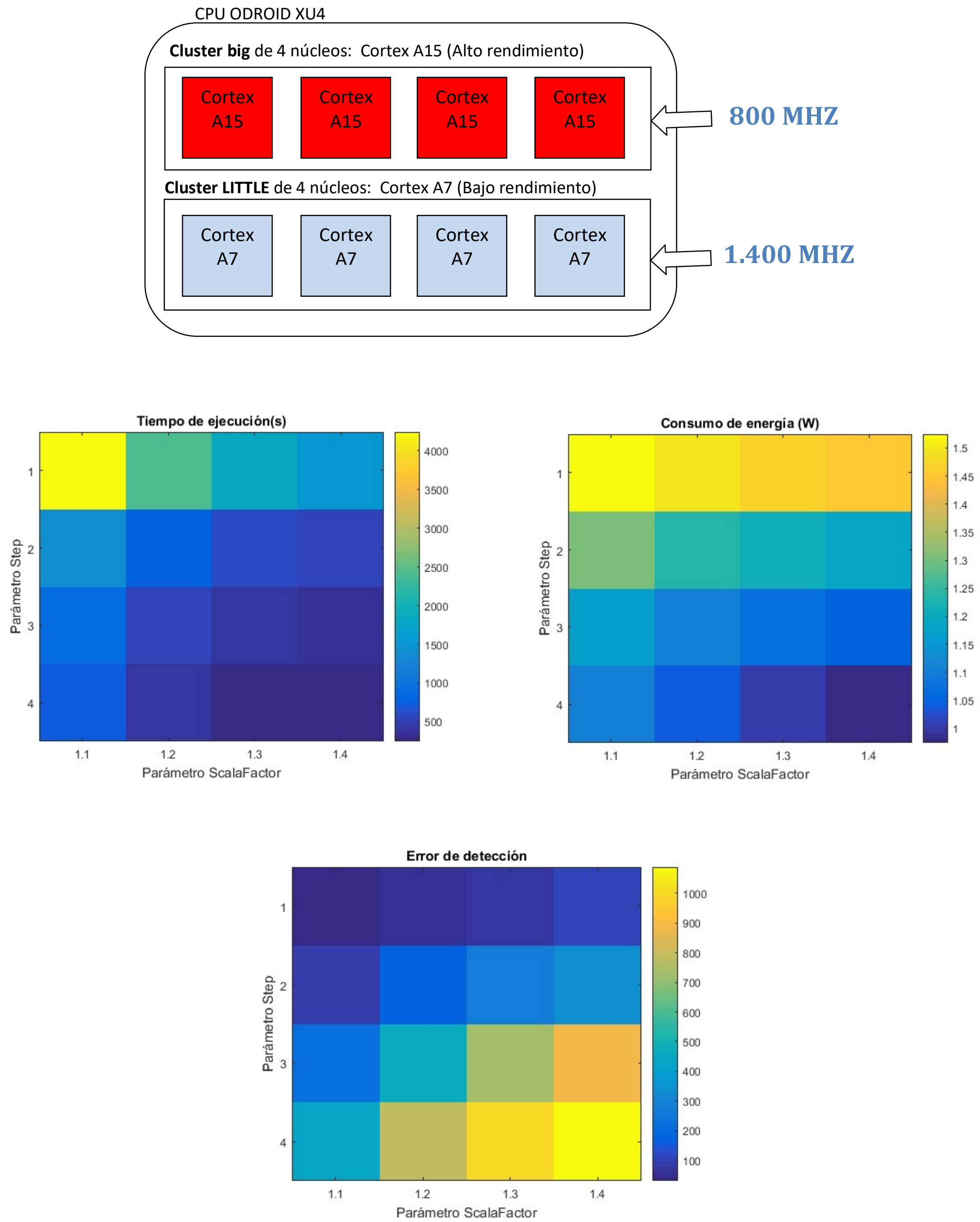


Figura 6.17: *Tiempo de ejecución, consumo de potencia y error de detección según los parámetros "scalaFactor" y "step", para las frecuencias del clúster: Big = 800 MHz , LITTLE = 1400 MHz*

Si se analizan los resultados obtenidos, reflejados en las tablas y figuras anteriores, atendiendo a los objetivos marcados en el presente trabajo para acelerar la ejecución del programa de detección facial y optimizar el consumo energético en la placa Odroid XU4, se puede observar que los valores óptimos para un error de detección inferior al 10% del total de caras de la muestra experimental se pueden conseguir fijando sobre dicha placa experimental los parámetros que se muestran en la tabla 6.10.

Frecuencia Cluster Big (Solo Odroid XU4)	Frecuencia Cluster LITTLE (Solo Odroid XU4)	Parámetro “step”	Parámetro “scalaFactor”
1500 MHz	1400 Mhz	1	1,2

***Tabla 6.10**: Valores óptimos para reducir el consumo de energía y acelerar el tiempo de ejecución para una tasa de detección del 90%.*

La aplicación de estos parámetros al programa de detección de rostros, junto con el ajuste de la frecuencia a 1.500 MHz en el clúster Big y la aplicación de las políticas de reducción de consumo relacionadas con el planificador de tareas de OmpSs Botlev, consciente de la asimetría de la CPU, produce una mejora en el tiempo de ejecución de, aproximadamente, un 68% para la placa Odroid XU4, respecto del tiempo de ejecución secuencial.

En el caso de la optimización del consumo energético se consigue una reducción en torno al 55% del consumo energético respecto de la ejecución anterior sin la aplicación de las políticas de optimización del consumo energético en la placa Odroid XU4. Todo gracias al aprovechamiento de los recursos que nos proporciona la arquitectura asimétrica de dicha placa, a través del planificador de tareas de OmpSs y el escalado de frecuencias. En este caso fijando la frecuencia del cluster Big a 1.500 Mhz y la del cluster Little a 1.400 Hz [Corpa, 2018].

Por último, se han seleccionado los valores óptimos de los parámetros: “step” y “scalaFactor”, con objeto de conseguir una adecuada precisión en la detección de

rostros (es decir, un error de detección inferior al 10%), el mejor tiempo de detección y el menor consumo energético posible.

A modo de comparación de los resultados obtenidos, se ha realizado una prueba con las mismas bases de datos: **Base-450** y **Base-750**, pero en este caso usando la función: “detectMultiScale” de OpenCV. Esta función implementa también el método de detección facial de Viola y Jones, incluyendo la mejora propuesta por Rainer Lienhart [Lienh, 2003]. Siendo una de las funciones más utilizadas por los programadores para implementar sistemas de detección facial. En este contexto, los resultados obtenidos aplicando dicha función de detección y los obtenidos por el sistema de detección facial objeto de la tesis doctoral, se muestra en la tabla 6.11.

Función detectMultiScale de OpenCV				Sistema de detección con parámetros seleccionados			
BASE 450				BASE 450			
FALSO POSITIVOS	FALSOS NEGATIVOS	TIEMPO DE EJECUCION (MIN)	TOTAL ERRORES	FALSO POSITIVOS	FALSOS NEGATIVOS	TIEMPO DE EJECUCION (MIN)	TOTAL ERRORES
151	3	29,18	154	9	29	18,29	38
BASE 750				BASE 750			
33	3	24,23	36	13	17	13,48	30

***Tabla 6.11**: Comparación entre los resultados de aplicar la función de OpenCV: detectMultiScale, y los obtenidos para el sistema de detección facial con “step”=1 y “scalaFactor”=1,2, sobre las dos bases de rostros experimentales: Base-450 y Base-750.*

Si se comparan los resultados obtenidos con los resultados experimentales del apartado 6.4.4.3, se puede ver que el rendimiento de la función “detecMultiScale” de OpenCV es menor, para valores de los parámetros “step” y “scalaFactor” del sistema de detección facial de 1 y 1.2 respectivamente. También se puede observar que el Error Total que se produce es considerablemente mayor, sobre todo sobre Base-450, respecto al programa de detección facial desarrollado en esta tesis doctoral. En este sentido, se puede concluir que el sistema de detección facial experimental presenta unos resultados viables en el contexto actual que, con lo parámetros adecuados, pueden mejorar fácilmente el rendimiento y la precisión del proceso de detección facial propuesto por Viola-Jones [Viola, 2001].

De los resultados obtenidos se puede ver cómo el sistema de detección objeto del presente trabajo mejora tanto el tiempo de ejecución como el número total de errores que se producen en la detección de rostros respecto a los obtenidos por la función de OpenCV “detectMultiScale” [Corpa, 2018].

La precisión y la sensibilidad son dos parámetros importantes que definen un sistema de detección o de reconocimiento de patrones. Se define la precisión (también llamada valor predictivo positivo) como la fracción de resultados relevantes entre los resultados recuperados. Mientras que “*Recall*” (también conocida como sensibilidad) se define como la fracción de resultados relevantes que se han recuperado sobre el total de resultados relevantes. Tanto la precisión como el “*Recall*” se basan en una comprensión y medida de la relevancia de los resultados. En este contexto, la precisión y *Recall* del sistema de detección facial se define como se indica en las ecuaciones 6.1 y 6.2.

$$\text{Precision} = \frac{\text{Verdaderos positivos}}{\text{Verdaderos positivos} + \text{Falsos Positivos}} \tag{6.1}$$

$$\text{Recall} = \frac{\text{Verdaderos positivos}}{\text{Verdaderos positivos} + \text{Falsos negativos}} \tag{6.2}$$

Los resultados obtenidos de *Precision* y *Recall* para la función OpenCV “detectMultiScale” y el sistema de detección objeto de la tesis doctoral, sobre las bases de rostros experimentales: Bases-450 y Base-750, se muestra en la tabla 6.12.

	OPENCV detectMultiScale		Sistema detección facial	
	Base-450	Base-750	Base-450	Base-750
Precisión	74,71%	95,76%	97,91%	98,26%
Recall	99,33%	99,60%	93,56%	97,73%

Tabla 6.12: *Resultados de Precision y Recall para la función OpenCV detectMultiScale y el sistema de detección en las bases experimentales: base-450 y base-750.*

De los resultados también se puede concluir que el sistema de detección objeto de esta tesis doctoral es más preciso que el obtenido por la función de OpenCV "detectMultiScale" [Corpa, 2018].

6.5 Conclusiones

En este capítulo, con objeto de paliar el incremento del consumo de energía que ha supuesto la paralelización del sistema de detección facial, visto en el capítulo 5, se ha optimizado el consumo energético del mismo sobre uno de los entornos de prueba utilizados como es la placa Odroid XU4. Para ello, se han aprovechado todos los recursos disponibles en un dispositivo de sus características.

En primer lugar, se han estudiado y aplicado diferentes técnicas y políticas para optimizar el consumo energético en el ámbito que brindan las arquitectura asimétricas como la que contiene la CPU de la placa Odroid XU4, que permiten optimizar la eficiencia energética asignando las tareas a núcleos de distintas características en función de los requisitos de rendimiento y consumo de cada tarea. Estas técnicas son:

- El uso del planificador de tareas de OmpSs Botlev, consciente de la asimetría de la arquitectura, permite tomar decisiones dinámicas basadas en el camino crítico, haciendo que la ejecución del programa sea más eficiente energéticamente.
- El uso de las técnicas de escalado de frecuencias aplicadas a dicha placa permiten ajustar el rendimiento y consumo de cada núcleo.
- La selección óptima de los parámetros del sistema "step" y "scalaFactor".

En este contexto, se han realizado diferentes pruebas experimentales sobre la placa Odroid XU4, utilizando el planificador Botlev de OmpSs, diferentes frecuencias para el cluster Big de la placa, y diferentes valores de los parámetros "step" y "scalaFactor", obteniendo resultados que permiten hacer una selección optima de parámetros y hacer comparativas de rendimiento y eficiencia energética respecto a otros sistemas de detección facial.

Seguidamente, teniendo en cuenta los requerimientos de diseño marcados para el sistema de detección facial en cuanto a precisión (error de detección inferior al 10%), consumo energético, tiempo de ejecución y fiabilidad, se han seleccionado los valores óptimos de frecuencia para los cluster Big y Little, y los parámetros “step” y “scalaFactor” del sistema de detección de rostros. Estos valores son para las frecuencias del Cluster Big y LITTLE de la placa Odroid XU4, 1500 MHz y 1400 MHz respectivamente, 1 y 1.2 para los parámetros “step” y “scalaFactor” respectivamente.

Con los valores óptimos seleccionados se comprueba el aumento en la aceleración del sistema de detección facial de, aproximadamente, el 65% y un ahorro del consumo energético de, aproximadamente, el 55% tras aplicar las políticas de optimización del consumo energético, lo que demuestra la utilidad de las arquitecturas asimétricas respecto a las simétricas.

Finalmente, se ha realizado una comparativa entre el sistema de detección facial y una función de OpenCV llamada “detectMultiScale”, muy usada por los programadores para la creación de sistemas de detección facial. De los resultados obtenidos se comprueba que el sistema de detección facial objeto de esta tesis presenta una mejor precisión y tiempo de ejecución que la función “detectMultiScale” de OpenCV.

CAPÍTULO 7.

7. IDENTIFICACIÓN DE ROSTROS Y ACELERACIÓN DEL SISTEMA CON NEURAL STICK MOVIDIUS

En este capítulo se realiza en primer lugar una comparativa entre el sistema de detección facial optimizado en el capítulo anterior con un sistema de detección facial denominado Facenet [Schro, 2015], basado en Redes Neuronales Convolucionales (CNN). En segundo lugar, una vez detectados los rostros existentes en una imagen, y con objeto de hacer una investigación completa sobre todos los aspectos del problema de la detección facial, se procede a identificar los rostros detectados utilizando las 128 características proporcionadas por Facenet. Que usa el *framework* de Tensorflow para hacer inferencias. Finalmente, se procede a acelerar el sistema con hasta cuatro dispositivos Neural Stick Movidus [Intel, 2018] para comprobar el efecto de dichos dispositivos en la aceleración del sistema de identificación facial final.

7.1 Introducción

La identificación facial es una parte del campo de la biometría, que estudia la capacidad de una computadora para reconocer a un humano a través de un rasgo físico único. El reconocimiento facial proporciona la facultad para que la computadora reconozca a un humano por sus características faciales, ya que se sobreentiende que cada rostro tiene alguna singularidad que permite identificarlo. Hoy en día, la biometría es uno de los campos de más rápido crecimiento en tecnología avanzada.

Un dispositivo de identificación facial toma una imagen o un fotograma de video que contiene un rostro humano y lo compara con otros rostros en una base de datos de imágenes previamente procesadas a través de un algoritmo de reconocimiento. La estructura, la forma y las proporciones de las caras se comparan durante los pasos de identificación facial. Además, también se comparan la distancia entre los ojos, la nariz, la boca y la mandíbula, los contornos superiores de las cavidades oculares, los lados de la boca, la ubicación de la nariz y los ojos [Findb, 2018].

En este contexto, conviene en este punto de la presente tesis doctoral recalcar la diferencia de enfoque entre la detección de rostros existentes en una imagen, como el que se ha llevado a cabo en los capítulos anteriores, con la identificación de dichos rostros y asociarlo a una persona concreta. En este sentido, durante la última década se han implementado multitud de programas de identificación facial. Cada uno utiliza diferentes métodos y algoritmos que implican en primer lugar, extraer las características de la cara de la imagen de entrada para su identificación posterior comparándola con los existentes en una base de datos.

La identificación facial ha logrado un gran progreso en los últimos años debido a la mejora en el diseño y aprendizaje de características y modelos de reconocimiento facial. Todos ellos intentan imitar la capacidad que tienen los humanos para reconocer a las personas independientemente de su edad, condiciones de iluminación y expresiones variables. El objetivo de la investigación de estos sistemas es poder igualar o incluso superar la tasa de reconocimiento humano. En este sentido, alguno de los algoritmos más actuales propuestos en la bibliografía [Findb, 2018] utilizan, por ejemplo, un sensor 3D para capturar información sobre la forma de la cara, de modo que sólo se usan

características distintivas del rostro, como el contorno de las cuencas de los ojos, la nariz y la barbilla, para el reconocimiento facial. Otros algoritmos analizan la textura a partir de los detalles visuales de la piel, tal como se capturan en imágenes digitales o escaneadas y luego convierte las líneas, los patrones y los puntos únicos que aparecen en la piel de la persona en un espacio matemático, a través del cual se pueden clasificar e identificar a las personas que aparecen en las imágenes. Otros enfoques usan características geométricas, caras *Eigen* (ver capítulo 5), coincidencia de patrones y coincidencia de gráficos para poder realizar dicha identificación facial.

Pero, de entre todas estas técnicas, actualmente se pueden destacar las llamadas Redes Neuronales Convolucionales (CNN) [Guosh, 2015] que han demostrado ser un método muy eficiente para la identificación facial. Las Redes Neuronales Convolucionales son simplemente redes neuronales artificiales que utilizan la convolución en lugar de la multiplicación general de matrices en al menos una de sus capas. Debido a que su aplicación se realiza en matrices bidimensionales, son muy efectivas para tareas de visión artificial, y en la clasificación y segmentación de imágenes, entre otras aplicaciones.

7.1.1 Redes Neuronales Convolucionales (CNN)

Una red neuronal artificial es un algoritmo inspirado en el cerebro humano diseñado para, entre otras funciones, reconocer patrones en conjuntos de datos numéricos. Los datos del mundo real, por ejemplo, imagen, texto, audio, video, etc. necesitan ser transformados en vectores numéricos para ser usados por las redes neuronales. Una red neuronal está compuesta de diferentes capas y una capa está formada por varios nodos. En función del tipo de patrón, la red neuronal trata de aprender de los datos de entrada al nodo y a cada uno se le asigna cierto peso. Estas ponderaciones determinan la importancia de los datos de entrada para producir el resultado final. La suma ponderada de los datos de entrada se calcula y, dependiendo de algunos sesgos de umbral, se determina la salida del nodo. El mapeo de la entrada a la salida se realiza mediante alguna función de activación, como se muestra en la figura 7.1.

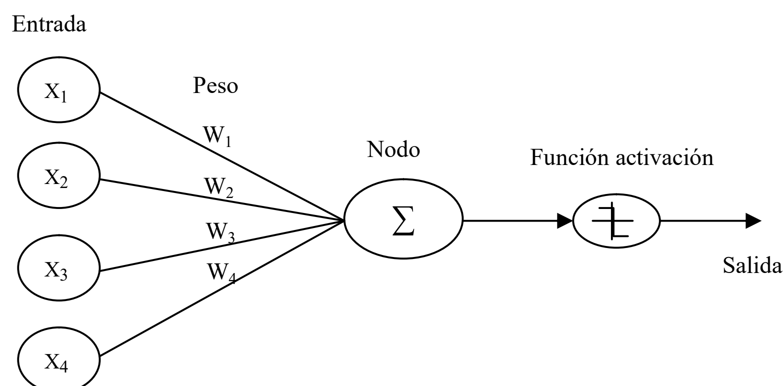


Figura 7.1: *Esquema red neuronal artificial [Fuente: elaboración propia].*

Como se comentó anteriormente, lo que distingue a las Redes Neuronales Convolucionales de cualquier otra red neuronal es que utilizan la operación llamada convolución en alguna de sus capas, en lugar de utilizar la multiplicación de matrices. En la convolución se realizan operaciones de productos y sumas entre la capa de partida y los *n* filtros (o kernel) con los que se genera un mapa de características. Las características extraídas corresponden a cada posible ubicación del filtro en la imagen original, como se muestra en la figura 7.2. La ventaja es que el mismo filtro (= neurona) sirve para extraer la misma característica en cualquier parte de la entrada. Con esto se consigue reducir el número de conexiones y el número de parámetros a entrenar en comparación con una red multicapa de conexión total. En este sentido, las CNN son muy potentes para todo lo que tiene que ver con el análisis de imágenes, debido a que son capaces de localizar características simples como por ejemplo detección de bordes, líneas, etc. Y componerlas en características más complejas hasta detectar lo que se busca.

Después de aplicar la convolución se aplica a los mapas de características una función de activación como en el caso de las redes neuronales artificiales definidas. En general, las CNN se construyen sobre una estructura que contendrá 3 tipos distintos de capas (ver figura 7.3):

1. Una **capa convolucional**, que es la que le da nombre a la red.
2. Una **capa de reducción o de *pooling***, la cual va a reducir la cantidad de parámetros al quedarse con las características más comunes.
3. Una **capa clasificadora** totalmente conectada, que va a producir el resultado final de la red que suele ser un perceptrón multicapa.

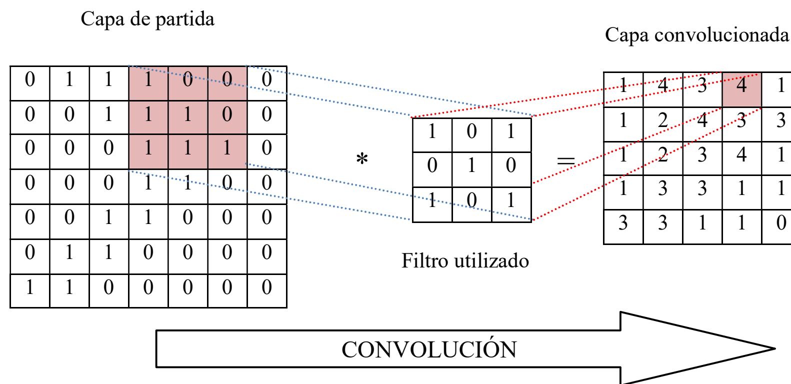


Figura 7.2: *Convolución de matrices [Fuente elaboración propia].*

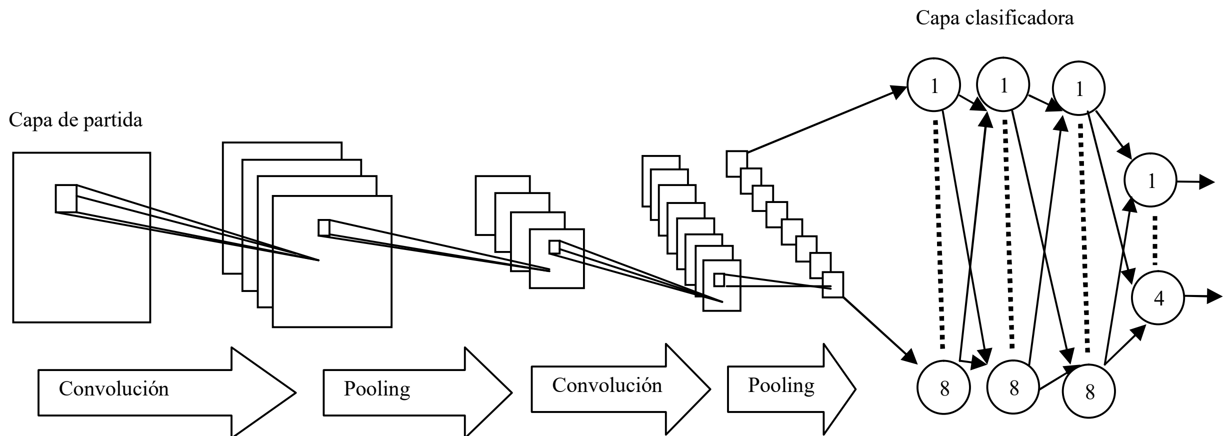


***Figura 7.3**: Estructura de las Redes Neuronales Convolucionales [Fuente: elaboración propia].*

7.1.1.1 Capa convolucional

La operación de convolución recibe como *entrada o input* la imagen y luego aplica sobre ella un *filtro o kernel* que devuelve un *mapa de las características* de la imagen original. De esta forma se consigue reducir el tamaño de los parámetros. La

convolución aprovecha tres ideas importantes que pueden ayudar a mejorar cualquier sistema de aprendizaje automático [Calvo, 2018], que son:

- **Interacciones dispersas**, ya que al aplicar un filtro de menor tamaño sobre la entrada original se puede reducir drásticamente la cantidad de parámetros y cálculos.
- **Parámetros compartidos**, que hacen referencia a compartir los parámetros entre los distintos tipos de filtros, ayudando también a mejorar la eficiencia del sistema.
- **Representaciones equivariantes**, que indican que si las entradas cambian, las salidas van a cambiar también en forma similar.

7.1.1.2 Capa de reducción o *pooling*

La capa de reducción o *pooling* se coloca generalmente después de la capa convolucional. Su utilidad principal radica en la reducción de las dimensiones espaciales (ancho×alto) del volumen de entrada para la siguiente capa convolucional. La operación realizada por esta capa también se llama *reducción de muestreo*, ya que la reducción de tamaño conduce también a la pérdida de información [Calvo, 2018].

La operación que se suele utilizar en esta capa se denomina *max-pooling*, que divide la imagen de entrada en un conjunto de rectángulos y, respecto de cada uno, se va quedando con el máximo valor, ver figura 7.4.

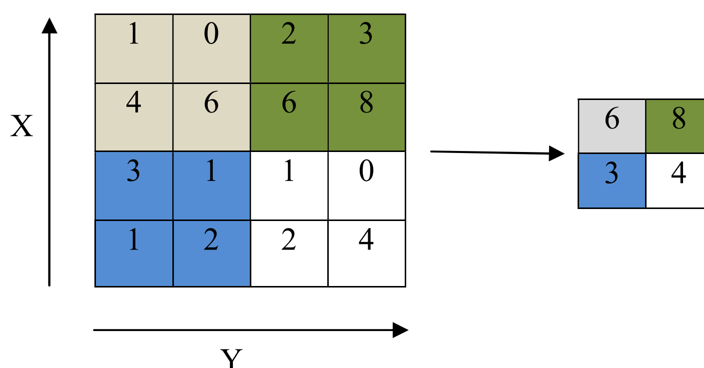


***Figura 7.4**: Operación Max-Pooling [Fuente: elaboración propia].*

7.1.1.3 Capa clasificadora totalmente conectada

Al final de las capas *convolucional* y de *pooling*, las Redes Neuronales Convolucionales utilizan generalmente capas completamente conectadas en las que cada pixel se considera como una neurona separada al igual que en una red neuronal regular. Esta última capa clasificadora tendrá tantas neuronas como el número de clases que se deben predecir.

7.2 Arquitectura de un sistema de identificación facial

La estructura típica de un sistema de identificación facial convencional consta de cuatro etapas: detección facial, alineación facial, extracción de características y clasificación, como se puede ver en la figura 7.5.

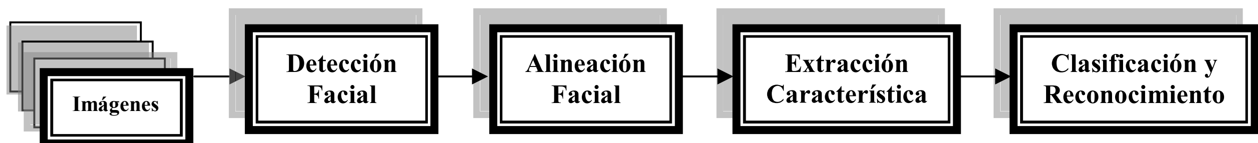


Figura 7.5: *Arquitectura de un sistema convencional de identificación facial [Fuente: elaboración propia].*

En el proceso de detección facial se recogen las imágenes de entrada de diferentes fuentes y se detectan todos los rostros que aparecen en ellas. A continuación, la etapa de alineación localiza los componentes de la cara y, mediante transformaciones geométricas, los normaliza respecto a propiedades geométricas, como el tamaño y la pose, y fotométricas, como la iluminación.

La etapa de extracción de características se puede definir como el proceso de extracción de información relevante de una imagen facial. En la extracción de características, se genera una representación matemática de la imagen original llamada plantilla biométrica o referencia biométrica, que se almacena en la base de datos y

formará la base (vector) de cualquier tarea de reconocimiento. Posteriormente estas características extraídas se utilizan en el reconocimiento. Un píxel en escala de grises puede ser considerado como característica inicial.

Una vez que se extraen y seleccionan las características, el siguiente paso es clasificar la imagen. Los algoritmos de identificación proporcionan información para distinguir entre las caras de diferentes personas según variaciones geométricas y/o fotométricas. En cuanto al reconocimiento, el vector de características extraído se compara con los vectores de características extraídos de las caras de la base de datos. Si encuentra uno con un porcentaje elevado de similitud, devuelve la identidad del rostro; si no, indica que es una cara desconocida. A veces se realiza el proceso de extracción y reconocimiento de características simultáneamente.

Un ejemplo de sistema que sigue una estructura como la descrita, es Facenet [Schro, 2015], que es un sistema unificado para la detección, alineación e identificación facial que utiliza una red neuronal de convolución profunda (CNN).

En este sentido, en este capítulo se va a llevar a cabo una comparativa entre la parte de detección facial de Facenet y el sistema de detección facial optimizado en el capítulo 6, con objeto de comprobar el rendimiento de ambos sistemas en los dispositivos experimentales usados en esta tesis doctoral y previo a la identificación de rostros. Para ello, a continuación, se va a describir un poco más en detalle el sistema Facenet.

7.3 Detección facial con Facenet.

Facenet es un sistema que aprende directamente de un mapeo de imágenes de rostros en un espacio euclidiano compacto donde las distancias corresponden directamente a una medida de similitud de caras. Una vez que se ha creado este espacio euclidiano, se pueden implementar fácilmente tareas como la detección, identificación y el agrupamiento mediante el uso de técnicas estándar con incrustaciones de Facenet como vectores de características, siendo la incrustación de

rostros una representación vectorial numérica multidimensional de un rostro que representa la identidad única del mismo.

El sistema Facenet utiliza una red CNN entrenada para optimizar directamente la incrustación en sí misma, en lugar de una capa intermedia que puede provocar un cuello de botella. Facenet mapea directamente las características faciales en 128 dimensiones y crea un modelo que mapea cualquier rostro humano en genérico. Es decir, cuando se proporciona una imagen de entrada al modelo, devuelve 128 bytes de datos vectoriales numéricos que se comparan con la representación de rostro genérica mapeada en el modelo determinando si es o no un rostro. Estos puntos de inserción son fácilmente comparables midiendo la distancia euclidiana y de este modo se puede llevar a cabo la identificación del rostro si así fuera.

Para entrenar, se usan muestras triples de rostros coincidentes / no coincidentes aproximadamente alineados y generados usando el método de minería de tres en línea, que consiste en la detección de patrones coincidentes en cada una de las muestras. El beneficio de este enfoque es una mayor eficiencia de representación, logrando un rendimiento óptimo de reconocimiento facial con solo 128 bytes por cara. Los autores [Schro, 2015], indican que el sistema alcanza un nivel de detección del 95,12% y una precisión del 99,65%.

Visto el sistema de mapeo de imágenes de Facenet, la etapa de detección de rostros en una imagen usando Facenet requiere de un método eficiente que permita escanear una imagen y encontrar los rostros existentes en el menor tiempo posible. En este sentido, el método que se usará está basado en el trabajo de [Zhang, 2016]. Éste utiliza CNN en una arquitectura de tres capas en cascada. En la primera etapa, se localizan rápidamente posibles ventanas de candidatos a través de una CNN poco profunda. La segunda etapa, refina las ventanas rechazando una gran cantidad de ventanas sin caras a través de una CNN más compleja. Finalmente, la tercera etapa utiliza una CNN más potente para refinar el resultado y verificar si se trata de un rostro.

7.3.1 Comparativa entre métodos de detección facial

En este apartado se hace una comparativa entre el rendimiento del sistema de detección optimizado en el capítulo 6, objeto de esta tesis doctoral, y el descrito en el apartado anterior, que usa Facenet para la detección de rostros en una imagen. Este último está implementado en Python y se puede descargar de [Manda, 2017].

Como se indicaba en el apartado anterior, el método de detección que usa Facenet también utiliza una arquitectura en cascada como el sistema de detección de Viola-Jones, pero en este caso formado por tres etapas de menor a mayor complejidad donde se van rechazando las ventanas que no contienen rostros. Aquellas ventanas de entrada que contengan un rostro superarán cada una de las tres etapas y se marcarán como posible rostro.

A pesar de que el sistema de aprendizaje de ambos algoritmos se puede considerar parecido, en general la diferencia entre los algoritmos de detección que usan redes neuronales y el de Viola-Jones radica en la elección de las características. En el caso de Viola-Jones utiliza las funciones de Haar (capítulo 5) y los clasificadores de umbrales. Pero las características son más artesanales en este caso. Sus pesos y formas están predeterminados y solo optimizados en el sentido de una selección de características que forman un clasificador fuerte de una etapa de la cascada. Mientras que en redes neuronales, las características son fijas en su tamaño y sus pesos están optimizados.

Instalado el sistema Facenet en la placa experimental Odroid XU4 se han realizado pruebas en ambos sistemas usando las imágenes utilizadas en las pruebas experimentales del capítulo 5, con resolución 640×480 pixel, que se muestra en la figura 7.6. En la tabla 7.1 se pueden ver los resultados obtenidos para cada sistema en tiempo de ejecución y consumo energético en función del número de rostros que contiene la imagen, mientras que en la figura 7.7, se muestra la representación gráfica de los mismos.

***Figura 7.6**: Imágenes de prueba con la que se han obtenido los resultados experimentales en el sistema de detección facial basado en el algoritmo de Viola-Jones optimizado y en el sistema Facenet.*

			Placa experimental Odroid XU4			
			Algoritmos de detección facial optimizado capítulo 6		**Modelo de detección facial Facenet**	
Imagen	**Resolución (pixel)**	**Número de rostros**	**Tiempo de ejecución (s)**	**Consumo de energía (Ws)**	**Tiempo de ejecución (s)**	**Consumo de energía (Ws)**
1	640×480	1	0,845	2,80	1,282	3,70
2	640×480	2	0,845	2,81	1,301	3,70
3	640×480	3	0,886	2,91	1,324	3,76
4	640×480	4	0,907	3,41	1,481	4,09
5	640×480	5	0,893	2,92	1,223	3,76
6	640×480	6	0,989	3,34	1,343	3,86
7	640×480	7	0,921	3,01	1,314	3,91
8	640×480	11	0,981	3,54	1,650	4,15
9	640×480	13	0,946	2,99	2,012	3,81
10	640×480	19	1,069	3,62	2,381	4,55

***Tabla 7.1**: Resultados de tiempo de ejecución y consumo energético sobre la placa Odroid XU4, para el método de Viola-Jones optimizado en capítulo 6, y el método de detección facial de Facenet, usando las imágenes de prueba del capítulo 5 que contienen un número diferente de rostros.*

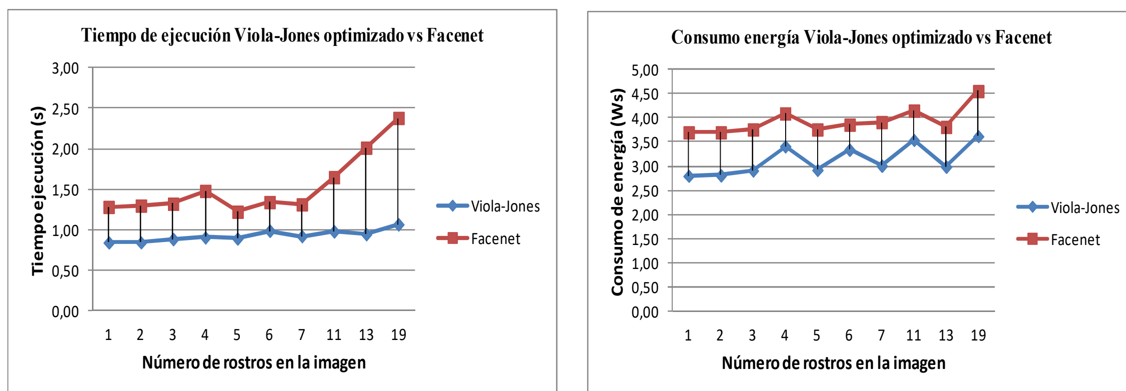


***Figura** 7.7: Representación gráfica del tiempo de ejecución y consumo energético en función del número de rostros de la imagen de prueba. Obtenidos ejecutando sobre la placa Odroid XU4:*
El programa de detección facial de Viola-Jones optimizado en el capítulo 6 y el sistema Facenet.

Los resultados demuestran que el rendimiento del sistema de Viola-Jones optimizado descrito en los capítulos 5 y 6 es mejor que el algoritmo que usa el modelo Facenet para la detección facial. Se mejora en torno al 50% el tiempo de ejecución del programa de identificación y aproximadamente un 24% el consumo energético. Esto permite pensar, dado que la identificación de rostros con Facenet es muy eficiente, en integrar los dos sistemas para obtener un mejor rendimiento global. Es decir, usar como sistema de detección de rostros en una imagen el sistema de detección basado en el algoritmo de Viola-Jones, optimizado en los capítulos 5 y 6, y utilizar el sistema Facenet para la identificación de dichos rostros.

7.4 Identificación de rostros usando Facenet Tensorflow

El sistema de identificación de rostros basado en Facenet que se va a implementar es la fusión de dos trabajos. El primero proporcionado por Intel como ejemplo de utilización de Facenet, que ha sido desarrollado por David Sandberg [Sandb, 2018], y que proporciona una red ya entrenada. Para realizar la inferencia de las imágenes se utiliza el *framework* Tensorflow Inception Resnet v1, con objeto de detectar la misma cara en dos fotografías diferentes. El segundo trabajo utilizado es el elaborado por Arun Mandal, que se ha descrito en el apartado anterior para la detección de rostros, y que está basado en el articulo [Manda, 2018]. Como resultado final de estos trabajos, el programa final consta de varios módulos, uno para la detección de

caras en una imagen, visto en el apartado 7.3, y otro para comparar dos caras de dos imágenes distintas.

Tensorflow [Tensor, 2018] es una biblioteca de software de código abierto para el cálculo numérico mediante gráficos de flujo de datos y aplicaciones de *Deep Learning*. Originalmente fue desarrollado por Google Brain Team dentro de la organización de investigación Machine Intelligence de Google para el aprendizaje automático y la investigación en redes neuronales profundas (DNN), pero el sistema es lo suficientemente general como para ser aplicable en una amplia variedad de dominios. Alcanzó la versión 1.0 en febrero de 2017, y ha continuado con un rápido desarrollo, con más de 21.000 aportaciones hasta el momento. Tensorflow es multiplataforma, se ejecuta en: GPU y CPU, plataformas móviles e integradas, incluso en unidades de procesamiento de tensor (TPU), que es hardware especializado (ASIC) para realizar cálculos tensoriales. Por otro lado, dado que los modelos de aprendizaje pueden complicarse mucho, Tensorflow permite fácilmente la distribución para poder iterar dentro de un marco de tiempo razonable.

Cada modelo de Tensorflow se define por tres ficheros:

- **Fichero meta:** Describe la estructura del grafo (conjunto de operaciones anidadas en base a nodos y aristas, donde un nodo representa una operación matemática o una entrada/salida de datos, y las aristas las relaciones entre nodos definidos para el modelo, pero sin incluir los valores de las variables.
- **Fichero index**: Contiene una tabla de cadenas de texto (*strings*) con nombres de tensores (estructura de datos básica en Tensorflow) y su metadata asociada.
- **Fichero data**: Este fichero contiene los valores de todas las variables del modelo.

Estos tres ficheros son necesarios cuando se trabaja con el modelo y se está entrenando el mismo, ya que se van modificando los pesos y las variables cambian cada cierto tiempo. Para trabajar con un modelo, éste se suele "congelar" previamente, tomando una foto fija del mismo en la que se elimina las variables (y se definen como constantes) y se compila toda la información en un fichero de tipo *Protocol Buffer* (o *protobuf*), que almacena la información del grafo y de las variables de forma conjunta.

Una vez que se dispone del modelo para Tensorflow congelado, se puede compilar para crear el fichero ".graph" que se utiliza para realizar las inferencias. En este caso, el modelo de Tensorflow que se utiliza, como ya se ha comentado, es el de Facenet, en concreto el ya entrenado por David Sandberg [Sandb, 2018], que presenta varios modelos pre-entrenados (ver tabla 7.2).

Modelo	Precisión LFW	Base de entrenamiento	Arquitectura
20180408-102900	0,9905	CASIA-WebFace	Inception ResNet V1
20180402-114759	0,9965	VGGFace2	Inception ResNet V1

Tabla 7.2: *Modelos de Facenet pre-entrenados proporcionados por David Sanderg [Sandb, 2018].*

El primero de ellos, el "20180408-102900", se ha entrenado usando la base de datos CASIA-WebFace [CBSRD, 2018], consistente en un total de 453.453 imágenes sobre 10.575 identidades.

El segundo, "20180402-114759" se ha entrenado con la base de datos de VGGFace2 [Qiong, 2017], que contiene más de 3,3 millones de caras sobre más de 9.000 identidades. Este es el que se usa en esta investigación.

La instalación de Tensorflow en la placa experimental Odroid XU4, ha sido un proceso complicado, debido a que esta biblioteca está preparada fundamentalmente para trabajar sobre sistemas de 64 bits y no para arquitecturas de 32 bits como la de la placa experimental objeto de esta tesis doctoral. En este contexto, existen algunas (muy pocas) referencias en internet de instalaciones sobre la placa Odroid XU4, como [Jacob, 2017]. El proceso que se ha seguido para la instalación de Tensorflow sobre la placa experimental Odroid XU4, se describe en el apéndice B, de la presente tesis doctoral, y se ha basado fundamentalmente en la publicación de [Jians, 2018].

Los módulos que forman el programa Facenet Tensorflow para la identificación facial descritos en este apartado se pueden ver en detalle en el apéndice C de esta memoria.

7.4.1 Ejecución del programa de identificación de rostros

El programa de identificación facial, desarrollado en lenguaje de programación Python, recoge las imágenes de muestra de un directorio denominado "*imagenes*", que contiene imágenes de personas (individuales o grupos). Dentro de ese mismo directorio, existe otro denominado "*imagen_validada*" donde se incluye una imagen que contiene el rostro de la persona que se desea reconocer respecto a las imágenes contenidas en el directorio "imágenes". El programa procesa las imágenes de este directorio detectando los rostros y los compara con el rostro incluido en la imagen incluida en el directorio "imagen_validada". En el proceso, el programa muestra de forma sucesiva las imágenes contenidas en el directorio "imágenes" e indica si la cara de la imagen almacenada en el directorio "*imagen_validada*" (denominada "*valid.jpg*") se encuentra en cada una de las imágenes mostradas. Para indicarlo muestra la foto con un recuadro verde si la cara se encuentra en la imagen, y por tanto ha sido identificada, y un recuadro rojo en caso contrario, como se muestra en la figura 7.8.

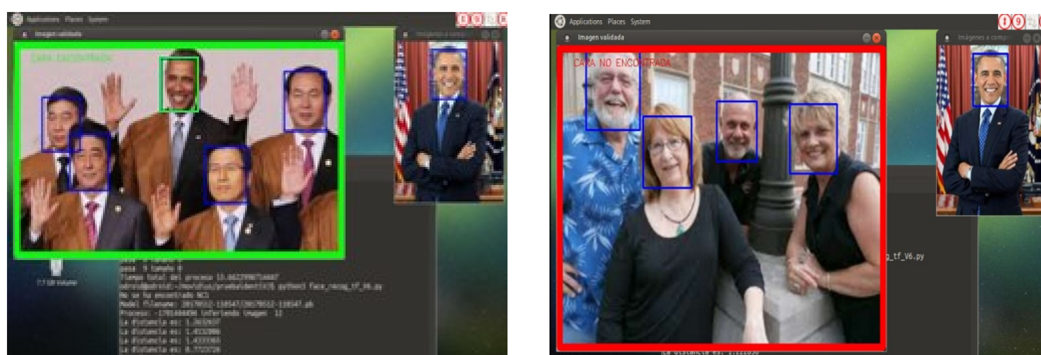


***Figura 7.8**: Resultado de la ejecución del programa de identificación que usa el modelo Facenet sobre la placa Odroid XU4 para la identificación del rostro del ex-presidente de los Estados Unidos Barak Obama. Izquierda identificación positiva, derecha identificación negativa.*

El rendimiento del programa depende de la resolución y del formato de las imágenes utilizadas, tanto de la imagen con el rostro a identificar, como de las imágenes a comparar del directorio “imágenes”. En este sentido, para realizar las pruebas se han usado 10 imágenes con una resolución similar de 640×480 pixel, en formato JPG. También el rendimiento depende del número de rostros que existen en la imagen. La muestra de imágenes que se han utilizado en la ejecución del programa se presenta en la figura 7.9. En ella se puede ver el recuadro verde o rojo sobre la imagen que indica si la identificación ha sido positiva o negativa respectivamente. En cuanto a la imagen “*imagen_validada*” que se utiliza para ser identificada en las imágenes de muestra, ésta se encuentra en formato JPG y a una resolución de 183×275 pixel. La imagen utilizada se expone en la figura 7.10.

Figura 7.9: *Imágenes de muestra para la identificación facial.*

***Figura 7.10**: Imagen con el rostro a identificar en las imágenes de muestra.*

Los resultados de ejecución y consumo energético del programa de detección facial se muestran en la tabla 7.3.

			Identificación con Facenet	
Imagen	**Número de rostros**	**Resolución (pixel)**	**Tiempo de ejecución (s)**	**Consumo de potencia (w)**
1	1	640×480	2,337	4,54
3	2	640×480	2,798	5,63
5	2	640×480	2,805	5,66
7	2	640×480	2,667	5,87
6	3	640×480	3,349	5,83
4	4	640×480	4,144	6,09
2	5	640×480	4,887	6,59
9	5	640×480	4,514	6,68
8	9	640×480	5,955	6,84
10	10	640×480	6,771	6,88
Tiempo ejecución de las 10 imágenes			**40,227**	

***Tabla 7.3**: Resultados de tiempo de ejecución y consumo de potencia del programa de identificación facial usando Facenet como algoritmo de detección de rostros para las 10 imágenes de prueba.*

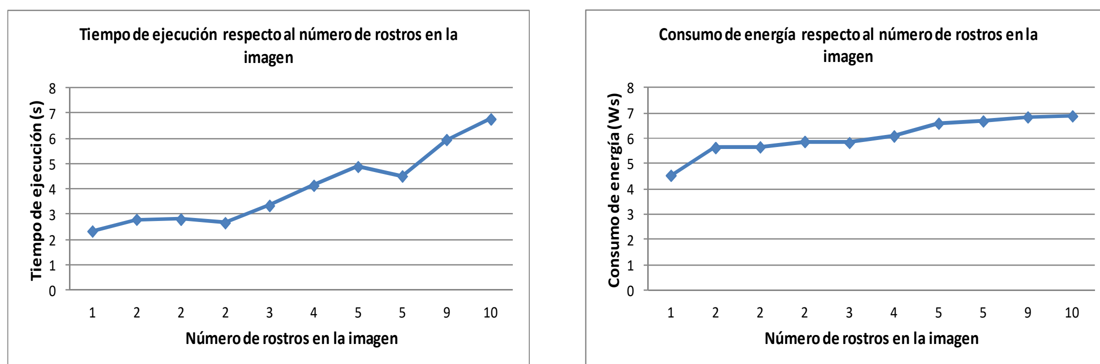


***Figura 7.11**: Representación gráfica del tiempo de ejecución y consumo de potencia del sistema de identificación facial Facenet en función de número de rostros de la imagen de prueba.*

Los resultados de la ejecución del programa son satisfactorios, se consigue identificar el rostro elegido en todas las imágenes de muestra que lo contienen, igualmente se rechazan todas aquellas imágenes que no contienen el rostro a identificar, como se puede ver en la figura 7.11. Tal como se indico en el apartado anterior, el número de rostros presentes en la imagen de prueba influye directamente en el rendimiento y en el consumo energético del sistema de identificación Facenet utilizado. Es decir, a más rostros mayor consumo. En la figura 7.12, se puede ver de forma gráfica dos ejemplos del resultado de la ejecución del programa de identificación facial.

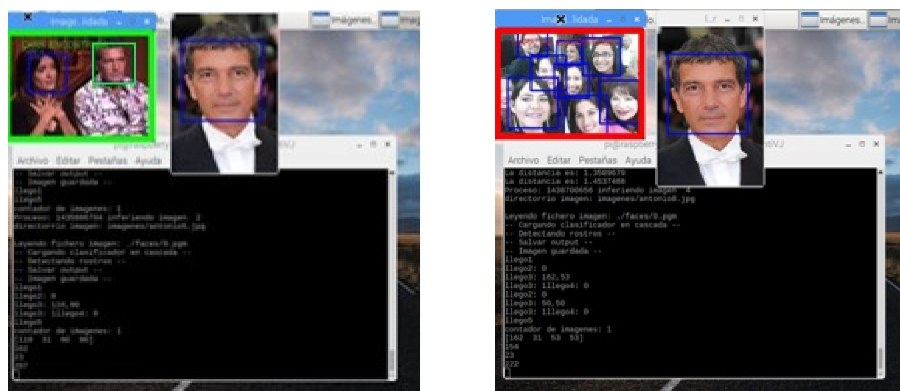


***Figura 7.12:** Resultado de la ejecución del programa de identificación facial, para un caso positivo (izquierda) y otro negativo (derecha).*

7.5 Aceleración del sistema de identificación facial

En la tabla 7.3, se puede ver que el tiempo de ejecución en el procesado de las 10 imágenes es bastante dilatado. Sí este tiempo se extrapola, por ejemplo, a 2000 imágenes con el mismo formato y resolución, el tiempo de ejecución podría ser aproximadamente una hora y media, lo cual se sobreentiende que es demasiado alto para poder llevar a cabo la identificación de un rostro con un rendimiento y tiempo de ejecución razonable. En este contexto, en este apartado se va a proceder a acelerar el sistema a partir de dos métodos diferentes. El primero será sustituir el modelo de detección de Facenet por el sistema de detección facial basado en el algoritmo de Viola-Jones objeto de la tesis doctoral, y que como se vio en el apartado 7.3.1, tiene mejor rendimiento. El segundo método, será usar varios aceleradores Intel Movidius, Neural Compute Stick (NCS). Con todo esto, se podrá ver la evolución del rendimiento del sistema de identificación facial estudiado.

7.5.1 Movidius Neural Compute Stick

Movidius Neural Compute Stick (figura 7.13) [Intel, 2018], es una unidad USB que está diseñada para trabajar con sistemas de inteligencia artificial y aprendizaje profundo que destaca por ofrecer una alta potencia en un tamaño compacto y un consumo energético y precio reducido. Para su desarrollo se ha utilizado una VPU (Vision Processing Unity) Myriad 2, que es un SoC multi-núcleo de baja potencia, diseñado para proveer un alto rendimiento en soluciones de visión por computador en dispositivos móviles, portátiles, y aplicaciones integradas tales como robótica, realidad aumentada y virtual.

La VPU Myriad 2 se basa en el procesador de aceleración hardware y el procesador de vectores SHAVE de 128 bits respaldado por un subsistema de memoria multi-núcleo compartido, y ocupa 27 mm^2 en HPM-CMOS de 28 nm. El dispositivo ha sido diseñado para funcionar a 0,9 V para un funcionamiento nominal de 600 MHz, y contiene 17 islas de alimentación diferentes, junto con una amplia sincronización de reloj bajo una API de software para minimizar la disipación de energía. Los 12 procesadores SHAVE integrados, combinados con aceleradores de hardware de video,

alcanzan 1000 GFLOPS (tipo fp16) con un consumo de 600 mW, incluidos periféricos y DRAM DDR2 LP de 32 MB de ancho de 32 bits apilados que funciona a 533 MHz. La VPU incorpora características de paralelismo, ISA y microarquitectura, como archivos de registro de múltiples puertos y soporte hardware para estructuras de datos dispersos, aceleradores de hardware de video, y bancos de memoria de múltiples puertos. Por lo tanto, proporciona una eficiencia de rendimiento excepcional y altamente sostenible en una gama de aplicaciones de imagen computacional y visión computacional, incluidas aquellas con requisitos de baja latencia (del orden de milisegundos) [Molon, 2014]. La figura 7.14, muestra el diagrama de bloques de Myriad 2.

Figure 7.13: *Movidius Neural Compute Stick [Intel, 2018].*

El NCS se conecta directamente a un puerto USB 3.0 aunque también es compatible con 2.0. Con esto, se puede añadir, por ejemplo, funciones de reconocimiento de imágenes a dispositivos que no disponen de suficiente potencia de cálculo como son la Raspberry Pi 3 B+ o la placa Odroid XU4 para aumentar su potencia de cálculo. Otra función interesante es que soporta Multi-Stick, lo que permite conectar varias unidades en paralelo para aumentar la potencia. Es capaz de ofrecer una potencia de 100 GFLOPS, manteniendo un consumo de apenas un vatio.

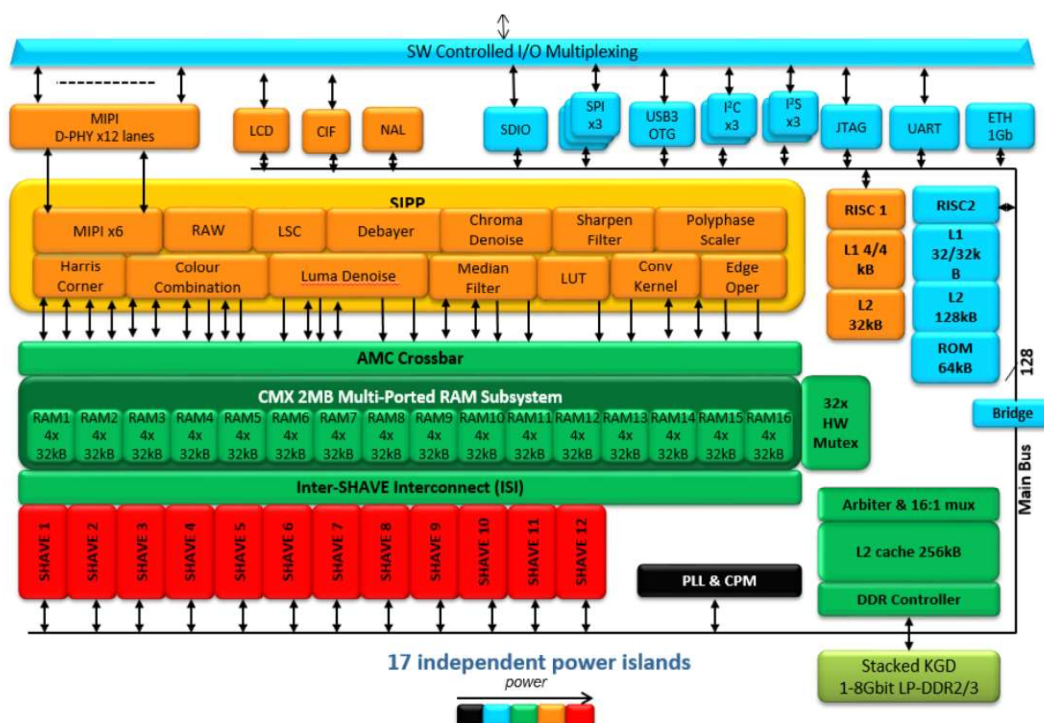


Figura 7.14*: Diagrama de bloques detallado de Intel / Movidius Myriad 2 [Molon, 2014].*

El NCS permite el desarrollo y prototipado de aplicaciones de inteligencia artificial (IA) en una amplia gama de dispositivos de última generación con un cómodo formato USB. El dispositivo está dirigido a desarrolladores que trabajan en aplicaciones de aprendizaje automático y ciencia de datos, y ofrece la máxima eficiencia energética de su clase siendo capaz de ejecutar CNN con operaciones de coma flotante y alto rendimiento.

También es compatible con la popular estructura de DNN Caffe [Yangq, 2018]. El dispositivo de computación neuronal es ideal como herramienta de desarrollo para prototipado y aceleración de redes neuronales. El motor de inferencia con factor de forma USB permite que los desarrolladores e investigadores liberen sus proyectos de la nube y conozcan rápidamente el rendimiento y precisión de sus aplicaciones para redes neuronales ejecutándolas en el mundo real. Los proyectos de redes neuronales se transfieren inmediatamente a través del compilador de computación neuronal Movidius para ejecutar inferencias de aprendizaje profundo en tiempo real en el dispositivo USB.

Con el NCS se pueden acelerar las actuales plataformas con restricciones de computación, como pueden ser los dispositivos experimentales objetos de esta tesis doctoral, habilitar la I+D de aprendizaje profundo y el prototipado en un portátil Linux o cualquier dispositivo anfitrión basado en x86. Además, la API de la plataforma de computación neuronal permite ejecutar aplicaciones en un anfitrión integrado, pudiendo inicializar la plataforma de destino, cargar un archivo gráfico y evitar interferencias [Intel, 2018].

En general, el Movidius NCS, se puede utilizar de dos formas principalmente:

1. Perfilado, ajuste y compilación de redes neuronales profundas. Esta operación debe ser realizada en un *host* con ayuda de las herramientas proporcionadas por Intel en su NCSDK (Neural Compute Software Development Kit). Dichas herramientas son las siguientes:

    - **mvNCCCompile**. Esta aplicación permite, usando un modelo de red Caffe o Tensorflow y sus pesos asociados, generar un fichero ".graph" que luego será utilizado por el NCS para realizar las inferencias.
    - **mvNCProfile**. Esta aplicación permite obtener estadísticas capa a capa, del rendimiento de la red que se está evaluando en el NCS.
    - **mvNCCCheck**. Para realizar validaciones de la red que se está implementando en el NCS, comparando las inferencias con él obtenidas, con las que se obtiene directamente con Caffe/Tensorflow.

2. Realizar el prototipado de aplicaciones que utilizan el NCS para acelerar la inferencia en redes neuronales profundas. Para esta función se utiliza el NCAPI (Neural Compute Application Programming Interface).

En este contexto, el NCS es el dispositivo ideal para acelerar el sistema de identificación facial descrito en los apartados anteriores.

7.5.2 Funcionamiento del programa de identificación facial

El método de identificación Facenet que se usará, requiere desarrollar un programa en Python adaptado al entorno de trabajo objeto de esta tesis doctoral y a los requerimientos fijados en las pruebas, así como de a las necesidades especificas que haya que definir para poder ejecutar el programa adecuadamente en los entornos experimentales. Los mecanismos de control para el correcto funcionamiento del programa son la lectura y procesado de los ficheros de prueba, los puntos de medición del tiempo de ejecución de programa para el análisis posterior, los módulos de transformación del modelo Facenet al *framework* de Tensorflow, etc. En la figura 7.15 se puede ver el diagrama de bloques del programa de identificación facial que se ha desarrollado.

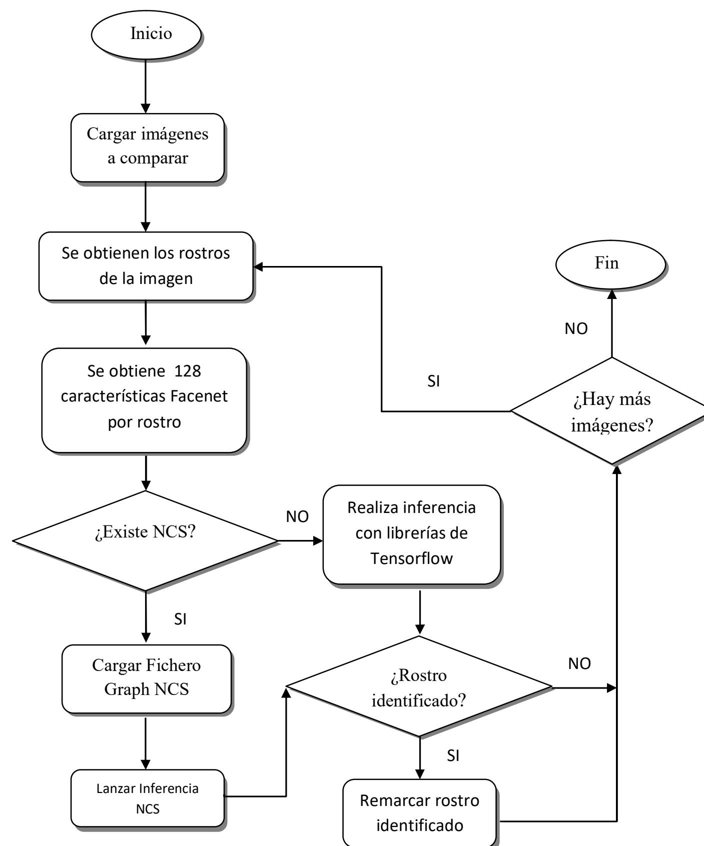


***Figura 7.15**: Diagrama de bloques del programa de identificación facial [Fuente: elaboración propia].*

Como se indica en el diagrama de la figura 7.16, previo a la inferencia existe un bloque del programa donde se obtienen los rostros que existen tanto en la imagen donde se encuentra el rostro a validar, como en la imagen de muestra con la que hay que comparar los rostros. Este bloque, como se ha comentado en el apartado 7.3, se puede llevar a cabo de dos formas diferentes. La primera con el método de detección facial basado en la cascada de tres etapas con CNN [Manda, 2018], y la segunda, a través del sistema de detección facial optimizado en los capítulos 5 y 6 de la tesis doctoral basado en el algoritmo de Viola-Jones. El primer método está desarrollado en Python, igual que el programa de identificación facial que se ha implementado, con lo cual no hay ningún problema en incluirlo en el programa. El segundo método, como se ha visto a lo largo de la tesis doctoral, está desarrollado en C++.

En este sentido, para poder hacer uso del programa de detección facial optimizado de Viola- Jones, es necesario incrustar el programa de detección escrito en C++ como un modulo de Phyton a través de alguna interfaz. Es bastante común que se integren módulos de extensión escritos en C/C++ en bibliotecas Python, sobre todo cuando se necesita mucha eficiencia o usar recursos o bibliotecas ya existentes. Algunas bibliotecas para cálculo científico, como NumPy y SciPy usan estas extensiones muy a menudo. A esto se le conoce como *"wrapping"*, y existen múltiples formas de hacerlo, desde simplemente llamar a funciones C/C++ desde un modulo o biblioteca estática hasta usar código compilado en bibliotecas dinámicas (*.so* y *.dll*). En este contexto, existen diferentes métodos para conseguir integrar el código C++ en Python, como son: Cython, Swig, Boost, Ctypes, etc. En esta tesis doctoral se hará uso de la API Python/C, ya que este método no requiere el uso de un lenguaje adicional intermedio o una modificación sustancial del programa escrito en C++, y está especialmente recomendado para extender Python con C. Además, se puede hacer uso del compilador habitual de C++, y los resultados presentan un rendimiento optimizado. El procedimiento que se ha seguido para incrustar el programa de detección facial, optimizado en el capítulo 6, en el sistema de identificación facial escrito en Python, se puede ver en el apéndice D de esta memoria.

Por otra parte, para hacer uso de la capacidad de disponer de varias unidades NCS funcionando de forma paralela realizando inferencias, se ha decidido utilizar *multi-*

threading en Python. Esta técnica permite que el programa de identificación facial desarrollado ejecute simultáneamente varias operaciones en el mismo espacio de proceso, es lo que se llama *Threading*. A cada flujo de ejecución que se origina durante el procesamiento se le denomina hilo o subproceso, pudiendo realizar o no una misma tarea. En Python, el módulo Threading hace posible la programación con hilos.

En el programa de identificación facial el uso de hilos puede ser de mucha utilidad, ya que se pueden realizar varias inferencias en paralelo atendiendo al número de NCS disponibles, mientras que la CPU del dispositivo experimental realiza otras tareas críticas del programa, además del control de flujo de datos entre cada uno de los hilos.

Ejecutar varios hilos o subprocesos es similar a ejecutar varios programas diferentes al mismo tiempo, pero con algunas ventajas añadidas:

- Los hilos en ejecución de un proceso comparten el mismo espacio de datos que el hilo principal y pueden, por tanto, tener acceso a la misma información o comunicarse entre sí más fácilmente que si estuvieran en procesos separados.
- Ejecutar un proceso de varios hilos suele requerir menos recursos de memoria que ejecutar lo equivalente en procesos separados.
- Permite simplificar el diseño de las aplicaciones que necesitan ejecutar varias operaciones concurrentemente.

Para cada hilo de un proceso existe un puntero que realiza el seguimiento de las instrucciones que se ejecutan en cada momento. Además, la ejecución de un hilo se puede detener temporalmente o de manera indefinida. En general, un proceso sigue en ejecución cuando al menos uno de sus hilos permanece activo, es decir, cuando el último hilo concluye su cometido termina el proceso, liberándose en ese momento todos los recursos utilizados. En este contexto, en la figura 7.16, se puede ver el esquema de funcionamiento del programa.

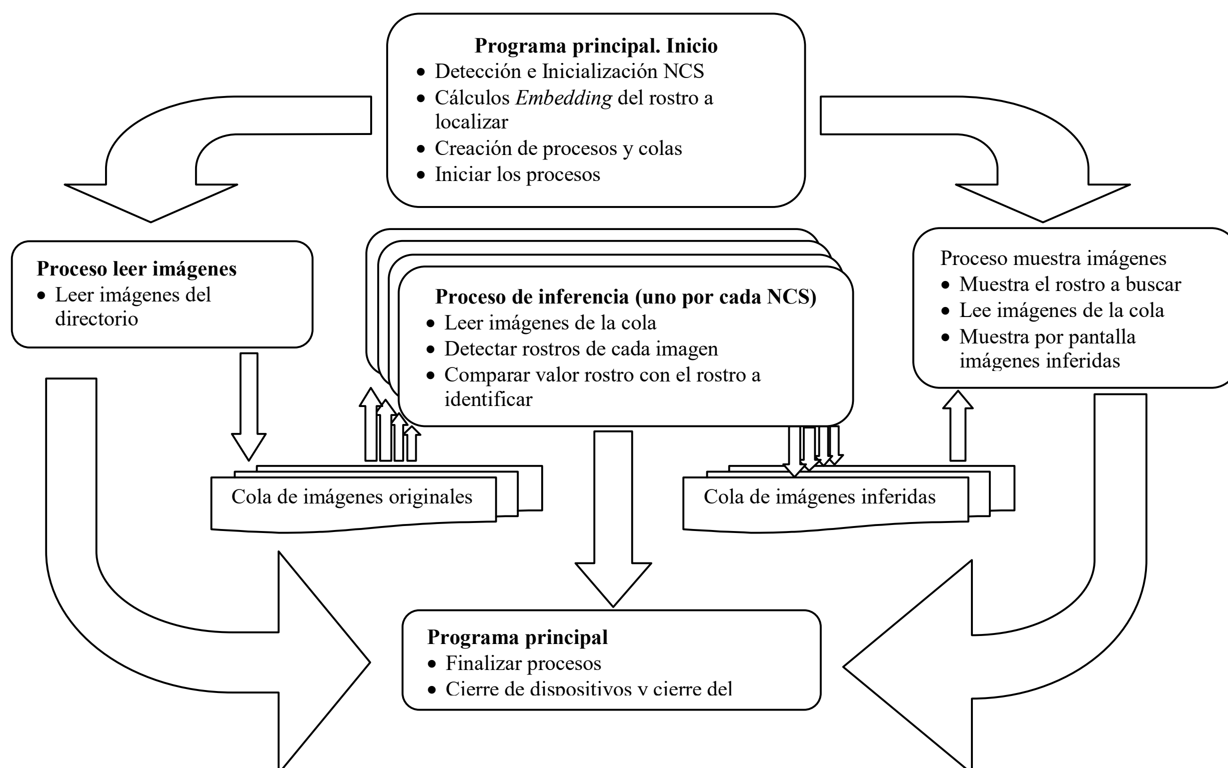


Figura 7.16: *Esquema de funcionamiento del programa de identificación facial desarrollado en lenguaje de programación Python [Fuente: elaboración propia].*

Siguiendo la estructura de la figura 7.16, a continuación, se describe el funcionamiento del programa, paso a paso:

1. Se cargan las imágenes a comparar. Por una parte, la imagen que se quiere buscar, denominada “imagen validada” y, por otra parte, la imagen a comparar del directorio “imágenes”. Se detectan los rostros que se encuentran en cada una de ellas. Si no se localiza ningún rostro en esta última se pasa a la siguiente imagen existente en el directorio “imágenes”.

2. A continuación, el programa para cada uno de los rostros calcula su “*embedding*”, vector con 128 valores que se utilizará para identificar cada rostro, según se define en la documentación de Facenet [Schro, 2015].

3. Se buscan unidades NCS conectadas al equipo. En función del número de ellas encontradas (puede que no haya ninguna), el programa realiza acciones diferentes:

- Si no se ha encontrado ninguna unidad de NCS conectada. Entonces el programa realiza todas las funciones utilizando las bibliotecas de Tensorflow tal y como se realizan en [Manda, 2017].
- Si se ha encontrado alguna unidad NCS. Entonces estas se inicializan y se cargan en ella los ficheros ".graph" correspondientes.

4. Se definen las colas que se utilizarán para pasarse la información entre los diferentes "*threads*" y se definen los procesos que se ejecutarán en paralelo. Estos son básicamente tres:

    - Thread "leer_imagenes", encargado de leer todas las imágenes con extensión ".jpg" que se encuentran en el directorio "imágenes", escalarlas a un mismo tamaño y colocarlas en la cola de imágenes leídas.

    - Thread "calcula_inferencia_caras_2", encargado de realizar las inferencias de las caras que se encuentran en las imágenes que han sido almacenadas en la cola de imágenes leídas, y compararlas con la imagen validada. En función del resultado de dicha comparación se compone una nueva imagen con un borde de color rojo o verde. También sitúa un rectángulo azul en cada una de las caras encontradas en la imagen. La nueva imagen resultante se añade a una nueva cola de imágenes inferidas.

    - Thread "muestra_imagen". Este Thread es el encargado de leer las imágenes inferidas de la cola anterior y mostrarlas por pantalla. Si la cola se encuentra vacía, espera un tiempo TIME_MAX en el que realiza 10 comprobaciones del tamaño de la cola. Si en todas el tamaño es cero, abandona el Thread, ya que se entiende que se han mostrado todas las imágenes del directorio. En función de la capacidad de proceso del entorno de ejecución puede ser necesario ajustar este tiempo para que el proceso no se abandone antes de finalizar la presentación de todas las imágenes.

5. Se arrancan todos los procesos, uno de “leer_imágenes”, uno de “muestra_imagen” y tantos “calcula_inferencia_caras_2” como dispositivos NCS se hayan encontrado. Si no hay ningún NCS, se arranca un Thread “calcula_inferencia_caras_2”, ligeramente modificado que realiza las inferencias utilizando Tensorflow.

6. Una vez que se finalizan todos los Threads, el programa libera los recursos de los NCS correspondientes y finaliza.

7.5.3 Resultados de la aceleración de sistema de identificación facial

Una vez desarrollado el sistema de identificación facial, en la figura 7.17 se puede ver el entorno de trabajo donde se han realizado las pruebas, que está formado por la placa experimental Odroid XU4 y los cuatro aceleradores NCS.

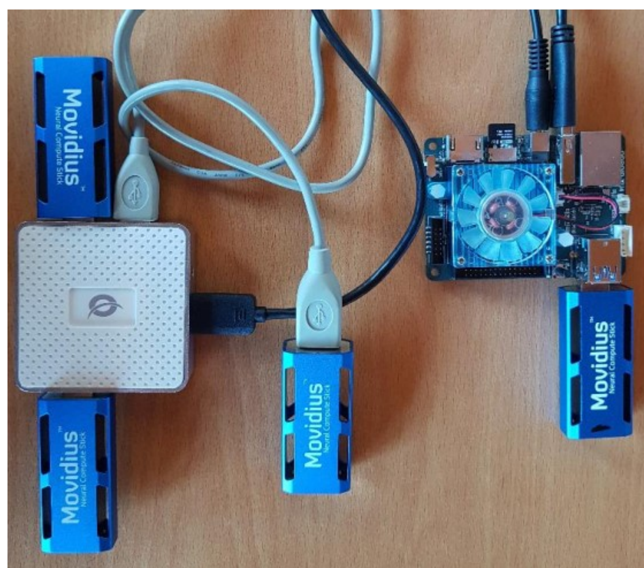


Figura 7.17*: Entorno de trabajo: Odroid XU4, y cuatro Movidius NCS.*

En estas condiciones, se ejecuta el programa de identificación facial sobre la placa experimental Odroid XU4, usando las mismas 10 imágenes de prueba de la figura 7.9, para tres resoluciones diferentes de las mismas (640×480, 340×240 y 240×160), además del rostro a identificar utilizado en la figura 7.10. De este modo se obtienen los resultados de tiempo medio de ejecución, consumo energético y rendimiento por vatio del sistema de identificación facial, bajo dos sistemas de detección diferentes. El primero, utilizando el sistema de detección facial que usa Facenet, y el segundo, el sistema de detección de rostros optimizado en el capítulo 6 de la presente memoria, que hace uso del algoritmo de Viola-Jones. Ambos se han ejecutado usando: Ningún NCS, un NCS, dos NCS, tres NCS y cuatro NCS en paralelo respectivamente. Los resultados de las pruebas se muestran en la tabla 7.4.

		Detección facial con Viola-Jones				Detección facial con Facenet			
Resolución imagen	**Número de NCS**	**Tiempo de ejecución (s)**	**Consumo potencia (w)**	**Pixeles procesados (MP/s)**	**Rendimiento por vatio ((MP/s)/w)**	**Tiempo de ejecución (s)**	**Consumo potencia (w)**	**Pixeles procesados (MP/s)**	**Rendimiento por vatio ((MP/s)/w)**
640x480	**0**	29,32	5,85	0,105	0,018	40,23	6,35	0,076 €	0,012
	1	18,46	6,32	0,166	0,026	25,39	6,94	0,121 €	0,017
	2	14,72	6,68	0,209	0,031	19,4	7,21	0,158 €	0,022
	3	13,23	6,97	0,232	0,033	15,87	7,35	0,194 €	0,026
	4	12,45	7,33	0,247	0,034	13,08	7,67	0,235 €	0,031
320x240	**0**	21,04	5,03	0,037	0,007	35,67	6,13	0,022 €	0,004
	1	13,89	5,61	0,055	0,010	23,43	6,51	0,033 €	0,005
	2	10,78	5,96	0,071	0,012	15,32	6,78	0,050 €	0,007
	3	9,21	6,18	0,083	0,013	11,21	6,99	0,069 €	0,010
	4	7,71	6,43	0,100	0,015	8,75	7,26	0,088 €	0,012
230x160	**0**	15,76	4,85	0,023	0,005	29,33	5,19	0,013 €	0,002
	1	12,34	5,58	0,030	0,005	15,83	5,43	0,023 €	0,004
	2	10,23	5,89	0,036	0,006	11,45	5,67	0,032 €	0,006
	3	8,65	6,09	0,043	0,007	10,03	5,91	0,037 €	0,006
	4	7,78	6,25	0,047	0,008	8,56	6,03	0,043 €	0,007

***Tabla 7.4**: Resultados de tiempo de ejecución, consumo de potencia y rendimiento del programa de identificación facial ejecutado sobre la placa Odroid XU4, comparando los sistemas de detección facial Facenet con el que usa el algoritmo de Viola-Jones, y haciendo uso de ningún NCS, un NCS, dos NCS, tres NCS y cuatro NCS. Y para resoluciones de las 10 imágenes de prueba de 640×480, 320×240 y 230×160.*

En la figura 7.18, se puede ver la representación gráfica de los resultados obtenidos, tanto para consumo de energía como para el tiempo de ejecución medio del programa de identificación facial de cada uno de los sistemas usados para la detección

de rostros, Viola-Jones y Facenet respectivamente, que va a permitir realizar un mejor análisis de los valores obtenidos.

Por otro lado, en la figura 7.19, se muestran las gráficas que representan el rendimiento de sistema de identificación facial para cada uno de los sistemas de detección de rostros indicados. En la gráfica de la izquierda se pueden ver los Megapíxeles procesados por segundo y en la gráfica de la derecha el rendimiento medio del sistema por vatio. Esto va a proporcionar información de interés para la utilización del programa atendiendo a las necesidades requeridas en cada caso.

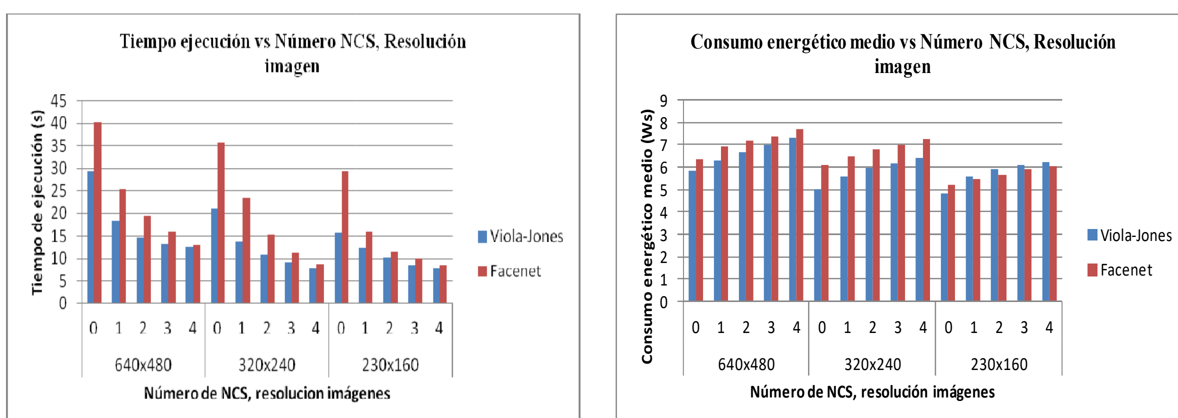


Figura 7.18: Representación gráfica de los Resultados de tiempo de ejecución medio y consumo de potncia medio durante la ejecución del programa de identificación facial sobre la placa Odroid XU4, usando los sistemas de detección facial Viola-Jones y Facenet para diferente número de NCS y para resoluciones de las imágenes de prueba de 640×480, 320×240 y 230×160.

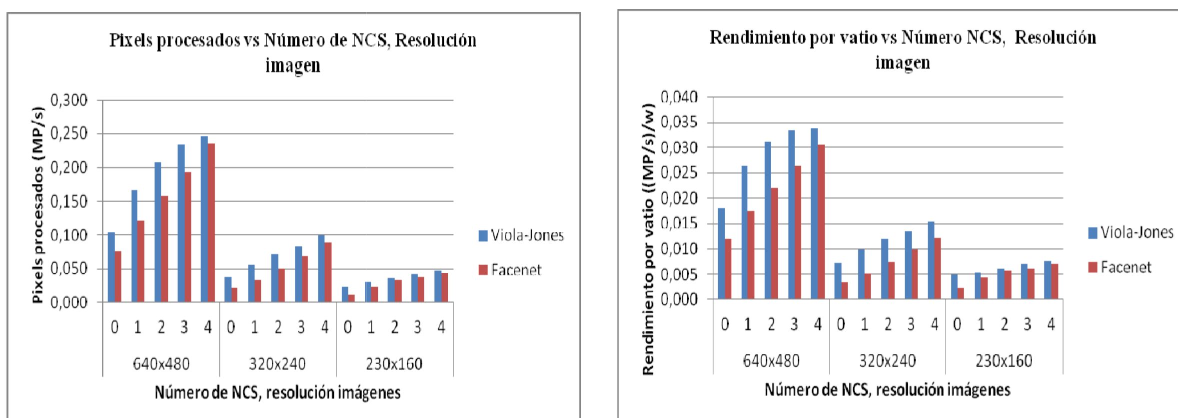


Figura 7.19: Representación gráfica de los Resultados de pixeles procesados por segundo y rendimiento por vatio del programa de identificación facial sobre la placa Odroid XU4, usando los sistemas de detección facial Viola-Jones y Facenet para diferente número de NCS y para resoluciones de las imágenes de prueba de 640×480, 320×240 y 230×160.

El análisis de los resultados muestra que el tiempo de ejecución, consumo de potencia y rendimiento por vatio del sistema de identificación facial es mejor cuando se utiliza el sistema de Viola-Jones, optimizado en los capítulos 5 y 6 de esta memoria, para la detección de rostro que cuando se usa el sistema Facenet tanto para la detección como para la identificación facial. Todo ello independientemente del número de NCS que se usen o de la resolución de las imágenes que se utilicen en las pruebas experimentales.

Como cabe esperar el consumo de potencia se ve afectado por la resolución de la imagen y por el número de NCS que se utilizan para la aceleración de sistema de identificación, siendo mayor cuanta más resolución tengan las imágenes y más dispositivos NCS se utilicen.

Por otro lado, los resultados experimentales muestran la viabilidad del uso de los NCS para la aceleración del sistema de identificación de rostros global, tanto haciendo uso del sistema de Viola-Jones como del sistema Facenet para la detección facial. Con ambos sistemas se consigue un porcentaje de reducción parecido del tiempo de ejecución respecto a su ejecución sin NCS (Viola-Jones sin NCS respecto a Viola-Jones con NCS, y Facenet sin NCS respecto a Facenet con NCS). La reducción del tiempo de ejecución sería aproximadamente de un 40% haciendo uso de un solo NCS, un 50% con dos NCS, un 53% con tres NCS y en torno a un 54% con cuatro NCS funcionando en paralelo. En cada uno de estos casos se obtiene un incremento de consumo de potencia de aproximadamente: 0,5w con un NCS, 0,8w con dos NCS, 1,1w con tres NCS y 1,14w con cuatro NCS, que supone un incremento aproximado del consumo de potencia de 0,3w por cada NCS añadido.

También se puede apreciar que el incremento de dispositivos Movidius NCS no implica una reducción lineal del tiempo de ejecución. Se observa que a partir del uso del tercer NCS la aceleración del sistema ya no es tan significativa como con uno o dos NCS en paralelo. Esto es debido al incremento de recursos de gestión que tiene que soportar la CPU con cada dispositivo NCS que se incorpora. Ocurre lo mismo que se vio en el proceso de paralelización del programa de detección facial en el capítulo 5 de esta tesis doctoral con el incremento del uso de CPU a medida que se aumenta el

número de núcleos funcionando en paralelo. Este hecho se traduce en que existe un punto de saturación que va a impedir mejorar la aceleración global del sistema de identificación de rostros. Se puede concluir, por tanto, que en el caso del dispositivo experimental utilizado, la placa Odroid XU4, los resultados óptimos de aceleración y consumo energético se podrían obtener con 2 Movidius NCS, funcionando en paralelo. Con 3 y 4 NCS, el rendimiento por vatio del sistema se incrementa, pero es poco significativo, tendiendo a una saturación del sistema a medida que añadimos NCS. Este hecho se puede ver gráficamente en la figura 7.20. En esta se muestran los resultados de tiempo de ejecución y rendimiento por vatio en función del número de NCS, para cada uno de los sistemas de detección facial utilizados, Viola-Jones y Facenet. También se visualiza como se tiende a la saturación a medida que se aumenta el número de dispositivos NCS.

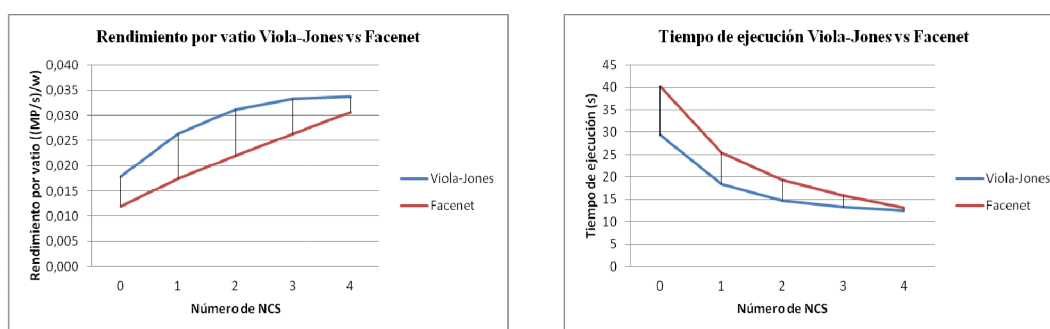


Figura 7.20: *Representación gráfica de rendimiento por vatio y tiempo de ejecución del programa de identificación facial sobre la placa Odroid XU4, usando los sistemas de detección facial Viola-Jones y Facenet para diferente número de NCS y una resolución de las 10 imágenes de prueba de 640×480 pixel.*

También se observa cómo las líneas que marcan el tiempo de ejecución y el rendimiento por vatio del sistema de identificación de rostros haciendo uso del sistema de Viola-Jones y por otro lado el de Facenet para la detección facial tienden a la convergencia a medida que incrementamos el número de dispositivos NCS, como se puede ver en la figura 7.20. Esto es debido a que la mejora en el tiempo de ejecución en el sistema de identificación que usa Facenet para la detección facial es mayor dado que implica más procesos de inferencia, tanto para la detección como para la identificación, que se pueden acelerar con los NCS. Es decir, en el caso del sistema de identificación con Viola-Jones como sistema de detección facial, sólo se puede acelerar con los NCS el proceso de identificación, no el de detección facial.

Por último, en la tabla 7.5, se puede ver el porcentaje medio de uso de la CPU de la placa Odroid XU4, durante la ejecución del sistema de identificación de rostros con diferente número de NCS, sobre las 10 imágenes de prueba a una resolución de 640×480 pixel. En la misma se puede ver, como se comentó en el apartado anterior, el incremento de uso de la CPU a medida que se aumenta el número de dispositivos NCS. En la primera columna usando el algoritmo de Viola-Jones optimizado como sistema de detección facial, y en la segunda columna haciendo uso del sistema Facenet para la detección facial. En las figuras 7.21, 7.22, 7.23 y 7.24, se muestra la monitorización de cada uno de los 8 núcleos de la CPU de la placa Odroid XU4, a partir de la cual, se ha obtenido la media de uso de CPU durante la ejecución del programa de identificación facial con el uso del sistema de detección facial de Viola-Jones, en el primer caso y con Facenet en el segundo. En ambos caso haciendo uso de ningún NCS y para 4 NCS funcionando en paralelo.

	CPU placa Odroid XU4	
	Detección facial con Viola-Jones	**Detección facial con Facenet**
	Uso medio CPU (%)	**Uso medio CPU (%)**
Sin NCS	70,6	71,3
1 NCS	47,7	40,2
2 NCS	58,5	47,4
3 NCS	66,4	58,7
4 NCS	76,2	78,1

Tabla 7.5*: Porcentaje medio de uso de CPU de la placa Odroid XU4, durante la ejecución del programa de identificación facial, comparando los sistemas de detección facial de Facenet con el que usa el algoritmo de Viola-Jones, y haciendo uso de ningún NCS, un NCS, dos NCS, tres NCS y cuatro NCS para las 10 imágenes de prueba con una resolución de 640×480 pixel.*

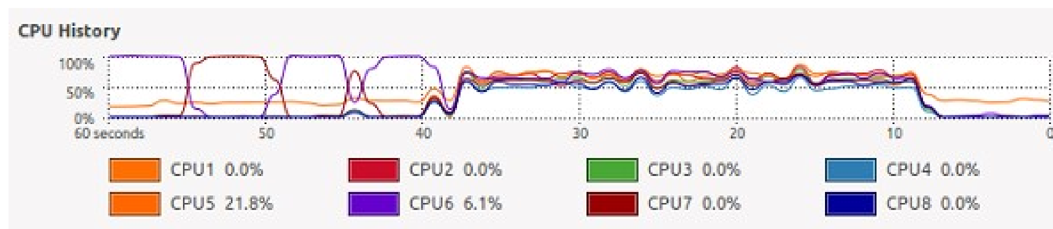


***Figura 7.21**: Uso de CPU de la Odroid XU4 con Viola-Jones y Ningún NCS.*

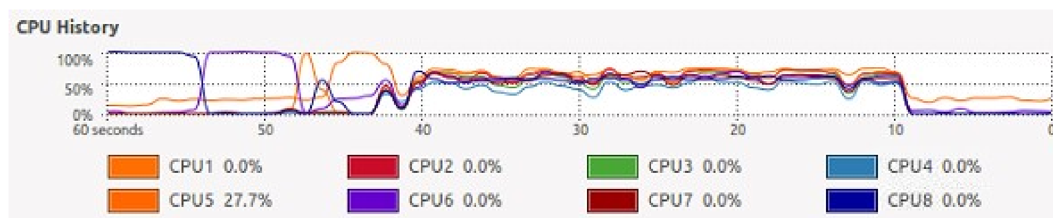


***Figura 7.22**: Uso de CPU de la Odroid XU4 con Facenet y Ningún NCS.*

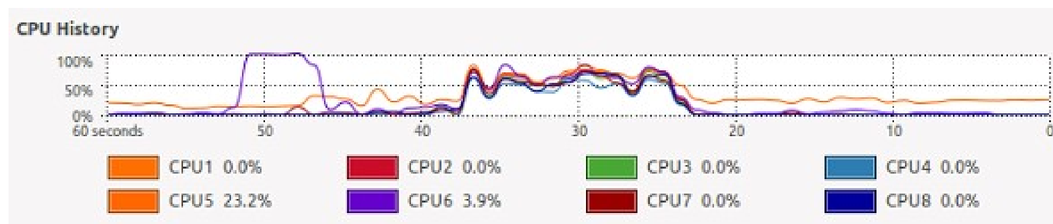


***Figura 7.23**: Uso de CPU de la Odroid XU4 con Viola-Jones y 4NCS.*

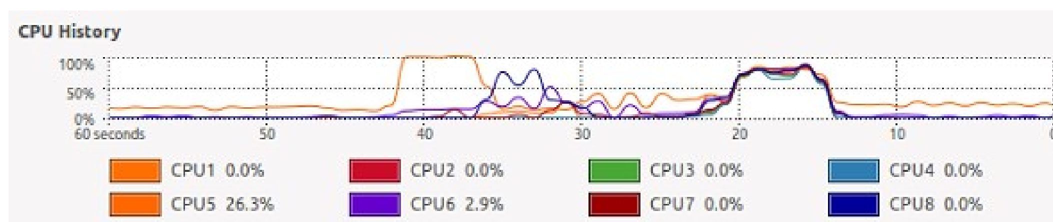


***Figura 7.24**: Uso de CPU de la Odroid XU4 con Facenet y 4 NCS.*

Gráficamente, en la figura 7.25, se puede ver la evolución que sigue el uso de la CPU de la placa experimental Odroid XU4, a medida que se van incorporando dispositivos Movidius NCS. A Partir del primer NCS, cada dispositivo añadido requiere

un mayor esfuerzo computacional por parte de la CPU para la gestión de cada uno de los hilos que se generan para que cada NCS funcione en paralelo. Es decir, en el primer caso, sin NCS, la CPU es la que soporta todo el peso de la ejecución de programa y de realizar las inferencias para identificación facial, por lo que el uso medio de la CPU de la placa Odroid XU4, es muy elevado, en torno al 90%. Cuando se añade un Movidius NCS, éste soportará computacionalmente los procesos de inferencia del programa liberando a la CPU de esta tarea, con lo que se produce una disminución muy abrupta del uso de la CPU de en torno al 40%, como se puede ver en la figura 7.25. Si se añade un segundo NCS, se observa como el porcentaje de uso de CPU aumenta debido al incremento de la gestión interna de hilos que tiene que soportar la CPU para que los dos NCS funcionen en paralelo. Y por este motivo, seguirá aumentando a medida que se vayan aumentando el número de NCS para la aceleración del sistema de identificación facial.

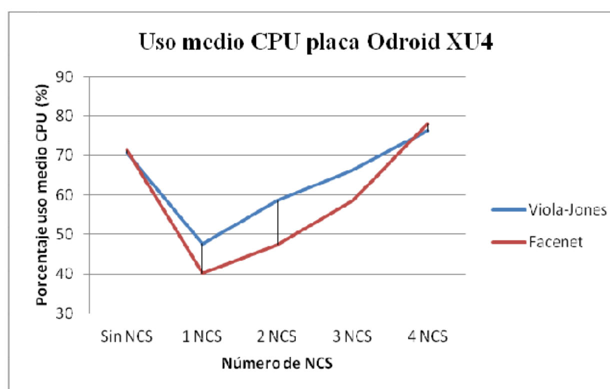


***Figura 7.25**: Representación del porcentaje medio de uso de CPU durante la ejecución del programa de identificación facial en la placa Odroid XU4, usando Viola-Jones y el sistema Facenet para la detección facial, sin NCS, un NCS, dos NCS, tres NCS y cuatro NCS.*

Atendiendo a los resultados experimentales obtenidos se puede decir que el sistema hibrido de identificación facial compuesto por el detector facial basado en el algoritmo de Viola-Jones, objeto de la presente memoria y el sistema de identificación Facenet, presenta un mejor rendimiento que el sistema de identificación Facenet puro que usa como sistema de detección de rostros el sistema presentado en el trabajo de [Zhang, 2016].

7.6 Conclusiones

En este capítulo, se ha llevado a cabo una experiencia práctica de uso del sistema de detección facial objeto de esta tesis doctoral, integrándolo como parte de un sistema de identificación facial como es Facenet, método que ha demostrado una gran fiabilidad en la identificación de rostros a través del uso de Redes Neuronales Convolucionales.

En este sentido, se ha podido verificar la mejora de rendimiento que supone incluir el sistema de detección facial optimizado en los capítulos 5 y 6, y que usa el algoritmo de Viola-Jones, en el programa de identificación facial Facenet respecto a la detección facial que realiza el propio Facenet. Se consigue una disminución del tiempo de ejecución de aproximadamente un 40%. También se consigue una leve mejora en el consumo energético del sistema de identificación global. Además de una mejora en el rendimiento por vatio del sistema global de identificación de en torno al 10%.

Por otro lado, dado el elevado tiempo de procesamiento del sistema de identificación de rostros global y el interés actual que despierta en la comunidad científica el uso de aceleradores neuronales como el dispositivo de Intel Movidius, Neural compute Stick (NCS). Se ha procedido a acelerar el sistema con hasta cuatro NCS en paralelo, consiguiendo unos resultados satisfactorios. Se ha reducido el tiempo de ejecución del sistema de identificación de rostros sin NCS, aproximadamente, un 40% con 1 NCS, un 50% con dos NCS, un 53% con tres NCS y un 54% con cuatro NCS, para cada uno de los dos sistemas de detección facial utilizados: Viola-Jones y Facenet. Es decir, prácticamente, se ha obtenido el mismo porcentaje de aceleración para ambos sistemas de detección facial respecto a su ejecución sin NCS.

Con el tercer y cuarto NCS, se ha podido comprobar en la placa Odroid XU4, que la aceleración del sistema ya no es tan significativa, y existe un punto de saturación, debido al consumo de recursos de gestión de hilos por parte de la CPU para el uso de varios NCS en paralelo (para cada NCS se crea un proceso o hilo que se ejecuta en paralelo aprovechando al máximo los recursos computacionales de la placa Odroid XU4). De esto se concluye que el número de dispositivos NCS óptimos para la aceleración y rendimiento energético del sistema de identificación en la placa

experimental Odroid XU4, puede estar en torno a 2 Movidius NCS funcionando en paralelo. Es decir, añadir un tercer y cuarto NCS aumenta el rendimiento por vatio del sistema de identificación respectivamente, pero no es tan significativo como al añadir uno y dos NCS.

En definitiva, se puede concluir que se ha conseguido que el sistema hibrido de identificación facial formado por el sistema de detección facial optimizado y el sistema Facenet de identificación de rostros, tiene mejor rendimiento que el sistema de identificación facial Facenet puro, basado en Redes Neuronales Convolucionales. Además se ha conseguido acelerar el sistema de identificación facial, con muy buenos resultados, con los nuevos dispositivos de Intel Movidius Neural Compute Stick. Esto demuestra la viabilidad de este dispositivo para aceleración de placas reducidas de baja potencia de cómputo, como es la Odroid XU4. Que se ha analizado y optimizado para aprovechar todo los recursos disponibles en la misma y descritos en la presente memoria.

CAPÍTULO 8.

8 CONCLUSIONES Y TRABAJOS FUTUROS

En este capítulo se presentan las principales conclusiones de esta tesis doctoral y las posibles investigaciones futuras. Para ello, se comenzará con el resumen de las más importantes conclusiones y tareas realizadas para mejorar las prestaciones del sistema de detección facial desarrollado. Se finalizará con una propuesta de trabajos a realizar en el futuro en el marco de la investigación llevada a cabo.

8.1 Conclusiones

Atendiendo a los objetivos que se marcaron en el capítulo 1, se fijaba como objetivo principal de esta tesis doctoral el demostrar la posibilidad de construir un sistema de detección facial eficiente, que funcionara en los mini dispositivos de bajo coste multi-núcleo (Raspberry Pi 2 B y la placa Odroid XU4), que disponen de tecnología similar a los actuales teléfonos móviles inteligentes y sistemas empotrados, aprovechando todos los recursos y prestaciones disponibles en los mismos. Se puede decir que se han conseguido íntegramente los objetivos propuestos, dado que se ha desarrollado un sistema de detección facial más rápido gracias a las técnicas de paralelización del software, y más eficiente energéticamente gracias a las técnicas y políticas de ahorro de energía que se han aplicado para sacar el máximo partido a las ventajas que nos brinda la CPU multi-núcleo asimétrica de la placa Odroid XU4.

También en el capítulo 1, se plantearon varios subobjetivos para alcanzar el objetivo principal, que son los siguientes:

1. Proponer un algoritmo eficiente para la detección facial, con una tasa de detección de rostros alta, una tasa de error mínima, que ofrezca una detección de caras robusta, y permita procesar imágenes de manera muy rápida consiguiendo una razón de detección alta.

2. Desarrollar, implementar y optimizar el algoritmo de Viola-Jones en lenguaje de programación C++, usando técnicas de optimización y depuración de código para conseguir un software óptimo acorde al objetivo de esta tesis doctoral.

3. Preparar, configurar y adecuar los entornos tecnológicos experimentales elegidos: la placa Raspberry Pi 2 B y la placa Odroid XU4. Ejecutar en ambas placas el software desarrollado y evaluar los tiempos de ejecución secuencial en cada una. Analizar los resultados obtenidos a través de una herramienta de *profiling*.

4. A partir de los resultados del *profiling del* programa, extraer el paralelismo a nivel de tareas usando la API de OpenMP y OmpSs.

5. Desarrollar técnicas de reducción del consumo energético que exploten de manera eficiente todos los recursos computacionales que ofrecen la arquitectura asimétrica de la placa Odroid XU4.

6. Establecer unas reglas óptimas para la ejecución del programa en función de los parámetros configurables del algoritmo implementado, que permitan una ejecución más eficiente con el menor impacto posible en el consumo de energía.

7. Realizar una comparativa de rendimiento energético entre las arquitecturas multi-núcleo simétrica y asimétrica experimentales.

8. Llevar a cabo una experiencia práctica de uso eficiente del sistema de detección facial desarrollado usándolo, por ejemplo, para la identificación de personas dentro de un conjunto de imágenes almacenadas.

En el proceso de investigación de la presente tesis doctoral, se puede concluir que se han alcanzado y cubierto cada uno de los subobjetivos indicados, como se indica a continuación:

- El sub-objetivo 1, se consiguió a partir de la selección del Algoritmo de Viola-Jones, como el mejor algoritmo que se adaptaba a los requerimientos establecidos (capítulo 2).

- Los sub-objetivos 2 y 3, se lograron a partir de la adecuación del software y de la ejecución secuencial del mismo en cada una de las placas experimentales, tal y como se detalla en los capítulos 3 y 4.

- En cuanto al sub-objetivo 4, se alcanza al conseguir acelerar de forma sustancial la ejecución del programa con la paralelización del software (capítulo 5).

- En el capítulo 6, se consigue obtener un ahorro del consumo energético a partir de varias técnicas como son la DVFS, la asignación optima de tareas y la selección adecuada de los parámetros del algoritmo desarrollado, consiguiendo con ello cubrir los sub-objetivos 5 y 6.

- El sub-objetivo 7 se ha podido completar, en el capítulo 6, gracias a la reducción del consumo de energía que se ha podido experimentar en la placa asimétrica Odroid XU4, respecto a la ejecución paralela. Se ha verificado la ventaja que aporta las arquitectura asimétrica de la placa Odroid XU4 frente a la simétrica de la placa Raspberry Pi 2 B, sobre el ahorro en el consumo de energía. Se ha conseguido prácticamente igualar el consumo energético de la ejecución secuencial en la placa Raspberry Pi 2 B (ejecución con menor consumo) con la ejecución en paralelo de la Odroid XU4, que contiene un mayor número de núcleos y es más rápida.

- Finalmente, en el capítulo 7, se ha alcanzado el sub-objetivo 8, al integrar el sistema de detección facial objeto de esta tesis doctoral en el sistema de identificación Facenet, logrando con un ello una mejora en el rendimiento temporal y energético del mismo. Además, se ha conseguido acelerar satisfactoriamente dicho sistema de identificación facial con varios dispositivos Intel Movidius, Neural Compute Stick (NCS), proponiendo a continuación un número optimo de dichos dispositivos, funcionando en paralelo, para obtener el mejor rendimiento temporal del sistema.

En consecuencia, se puede concluir que en el presente trabajo se ha conseguido en líneas generales, lo siguiente:

I. Desarrollar un sistema de detección facial basado en el algoritmo de Viola-Jones, más rápido y eficiente energéticamente que otros algoritmos del mismo tipo existentes en la actualidad, como puede ser el que nos proporciona la biblioteca OpenCV, a través de la función "detectMultiScale", muy utilizada por los desarrolladores para la creación de sistemas de detección facial. Además, se ha adaptado a dispositivos

empotrados de bajo coste como son la placa Raspberry Pi 2 B y la placa Odroid XU4, cuyas características son similares a los actuales *smartphones*.

II. Se ha llevado a cabo un exhaustivo estudio en el marco del problema general de la detección facial, analizando los principales algoritmos que existen en la actualidad, que dan una solución viable en tiempo real a este problema. Seguidamente se han definido unos requerimientos para el sistema basados en: fiabilidad, precisión, rendimiento en tiempo real, error de detección, tiempo de aprendizaje y facilidad de implementación, que han permitido determinar cuál es el mejor algoritmo para la implementación del sistema de detección facial. En este caso, se concluye que el algoritmo que tiene mejores prestaciones para el desarrollo de dicho sistema es el de Viola-Jones.

III. A continuación, se ha llevado a cabo un exhaustivo estudio del algoritmo de Viola-Jones, analizando cada una de sus partes con objeto de ver posibles mejoras en el marco de los objetivos de esta tesis doctoral y de las características de los dispositivos experimentales utilizados. En este sentido, se ha podido concluir que los mayores esfuerzos para la implementación del algoritmo están en el aprendizaje previo a través del algoritmo Adaboost, la selección de las características que van a determinar si una imagen es un rostro, y en el cálculo de las características en la imagen.

IV. Se han estudiado varias plataformas hardware existentes en la actualidad, como son las CPUs multi-núcleo, las GPUs, los DSPs, las ASICs y las FPGAs, para ver las posibilidades que ofrece cada una de cara a la implementación del sistema de detección de rostros. El análisis de las diferentes plataformas se ha llevado a cabo teniendo en cuenta los requerimientos fijados para el sistema de detección facial basados en fiabilidad, rendimiento, coste, tamaño, consumo energético, flexibilidad, facilidad de uso y tiempo de desarrollo. En este sentido, se concluye que la mejor opción para la implementación del sistema es usar una CPU multi-núcleo. Las plataformas de este tipo elegidas para implementar el desarrollo

experimental son una Raspberry Pi 2 B de cuatro núcleos simétricos y una placa Odroid XU4 de 8 núcleos, que posee una arquitectura asimétrica formada por dos *cluster* de 4 núcleos cada uno, denominados Big y LITTLE respectivamente.

V. La implementación del sistema de detección facial se ha desarrollado en lenguaje de programación C++, a partir de una versión simplificada pre-entrenada, que se ha adaptado a las plataformas experimentales Raspberry Pi 2 B y Odroid XU4. En este punto se ha analizado detalladamente la ejecución secuencial del programa con unas imágenes de muestra observando los tiempos de ejecución, el error de detección, la resolución y el número de rostros en la imagen. En este contexto, se ha podido encontrar una relación directa entre la Imagen Integral de una imagen y la velocidad de detección, lo que permite determinar cuál es la mejor resolución y tonalidad de las imágenes a procesar para que el rendimiento del sistema de detección de rostros sea óptimo. Se concluye también que, además del formato y la resolución, la tonalidad de la imagen puede influir en la velocidad de ejecución del programa de detección.

VI. Con el fin de mejorar el tiempo de ejecución del programa de detección facial, se ha procedido a realizar un estudio detallado para la paralelización del mismo y aprovechar todos los recursos que nos ofrecen las arquitecturas multi-núcleo. Para ello, se ha llevado a cabo el *profiling* de la ejecución secuencial del programa a través de la herramienta Gperftools que se ha visto, en comparación a otras, es una de las mejores herramientas para la paralelización de programas por su fiabilidad. Con ésta se obtienen las funciones del sistema de detección facial que mayor carga computacional soportan y por tanto, sobre las que se tienen que centrar los esfuerzos para la optimización y aceleración del mismo.

VII. A continuación, se ha adecuado el código fuente siguiendo las técnicas de mapeo estudiadas con la idea de optimizar dicha paralelización en las plataformas experimentales utilizadas. Como entorno de programación

paralela se ha hecho uso de la API que ofrece OpenMP, que es una de las mejores opciones dentro de las tecnologías de programación en paralelo en arquitecturas de memoria compartida.

VIII. El resultado de la paralelización ha sido sumamente satisfactorio, ya que se ha conseguido reducir el tiempo de ejecución considerablemente en ambas plataformas de bajo coste. Dicha reducción ha sido de más de la mitad (50%) en el caso de la placa Raspberry Pi 2 B y aproximadamente el 65% en la placa Odroid XU4, lo que guarda una relación directa con el número de núcleos que tiene cada una. En contraposición a estos buenos resultados, se han producido incrementos de aproximadamente el 50% en el consumo energético de ambos dispositivos respecto a la ejecución secuencial.

IX. En vista del incremento del consumo de energía que supone la paralelización del programa, se han estudiado diferentes opciones para optimizar dicho consumo. Todas basadas en técnicas y políticas para aprovechar los recursos que proporciona la arquitectura asimétrica de la placa Odroid XU4. Dichas técnicas y políticas aplicadas son fundamentalmente tres: el planificador de tareas de OmpSs (consciente de la asimetría de la CPU), las técnicas de escalado de frecuencias aplicadas a dicha placa y la selección óptima de los parámetros configurables del sistema "step" y "scalaFactor". Con todo ello, se ha conseguido una reducción del consumo energético de, aproximadamente, el 55,3%, respecto a la ejecución en paralelo en la placa Odroid XU4, sin aplicar políticas de reducción del consumo, y el 23% respecto a la ejecución secuencial en la placa Odroid XU4. Con este hecho también se ha podido verificar la utilidad del uso de la arquitectura multi-núcleo asimétrica (Odroid XU4) frente a la arquitectura simétrica (Raspberry Pi 2 B) para la optimización del consumo energético. Prácticamente se ha conseguido igualar el consumo energético de la ejecución secuencial del programa en la placa Raspberry Pi 2 B (menor consumo energético) con la ejecución paralela en la Odroid XU4.

X. Seguidamente, se ha realizado una comparativa entre el sistema de detección facial y la función de OpenCV "detectMultiScale", muy usada por los

desarrolladores para la realización de sistemas de detección facial. De los resultados obtenidos se comprueba que el sistema de detección facial desarrollado en esta tesis doctoral presenta una mejor precisión y velocidad de ejecución que la función "detectMultiScale" de OpenCV, que se ha ejecutado en las mismas condiciones sobre la placa Odroid XU4.

XI. Finalmente, se ha llevado a cabo una experiencia práctica de uso del sistema de detección facial optimizado en el capítulo 6, sobre un sistema de identificación de rostros como es Facenet (capítulo 7). Para ello, se ha intercambiado el sistema de detección de rostros optimizado en el capítulo 6, con el propio que usa Facenet, consiguiendo una mejora del rendimiento de aproximadamente el 40% en el tiempo de ejecución y una leve mejora en el consumo energético del sistema de identificación facial global de en torno al 24%. También se ha conseguido acelerar el sistema de identificación a través de varios dispositivos Intel Movidius, Neural Compute Stick (NCS). En este sentido, se ha adaptado el programa para que se pueda hacer uso de varios NCS en paralelo. Con ello, se ha verificado la mejora del rendimiento por vatio del programa de identificación, variando el número de NCS empleados entre 1 y 4. La reducción del tiempo de ejecución que se obtiene respecto a la ejecución del sistema sin NCS, son, aproximadamente, de un 40% con un NCS, un 50% con dos NCS, un 53% con tres NCS y un 54% con cuatro NCS funcionando en paralelo, tanto haciendo uso del sistema de Viola-Jones como del sistema Facenet para la detección facial en el sistema de identificación desarrollado. Esta experiencia ha demostrado la utilidad de los NCS para la aceleración de este tipo de algoritmos en placas de bajo coste y de características similares a la Odroid XU4.

En este contexto, las principales aportaciones de la tesis doctoral se pueden resumir en los siguientes puntos:

1. Elaboración de una metodología eficiente y sencilla para la paralelización de un programa secuencial.

2. Desarrollo de un sistema de detección facial con mejor precisión y velocidad de ejecución que la función "detectMultiScale" de OpenCV.

3. Se ha obtenido una relación entre el valor de la Imagen Integral de una imagen y la velocidad de ejecución del software, que puede permitir hacer una selección más óptima de las características de la imagen de entrada al sistema de detección facial para que la ejecución del programa sea más eficiente.

4. Se han aplicado algunas técnicas y políticas novedosas para el ahorro del consumo energético que han permitido reducir a más de la mitad el consumo de energía respecto a una ejecución paralela normal, que permite demostrar la utilidad de la arquitectura asimétrica de la placa Odroid XU4 respecto a la arquitectura simétrica de la placa Raspberry Pi 2 B, para conseguir un ahorro del consumo de energía.

5. Se ha creado un sistema híbrido de identificación facial con mejor rendimiento, integrando el sistema de detección facial optimizado en la tesis doctoral, basado en el algoritmo de Viola-Jones, y el sistema de identificación de rostros de Facenet, que hace uso de Redes Neuronales Convolucionales para la identificación de rostros. Además, se ha conseguido acelerar el programa de identificación facial con varios dispositivos Intel Movidius, Neural Compute Stick, unidad USB pensada para trabajar con sistemas de inteligencia artificial y aprendizaje profundo.

6. Esta tesis doctoral ha originado la publicación, en la revista *International Journal of Circuit Theory and Application*, de un artículo titulado: "Acceleration and energy consumption optimization in cascading classifiers for face detection on low cost ARM big. LITTLE asymmetric architectures." [Corpa, 2018].

8.2 Trabajos futuros

De las posibles líneas de investigación futuras para la mejora de las prestaciones del sistema de detección facial y de las líneas abiertas en el trabajo de investigación de esta tesis doctoral, se pueden destacar, sin ánimo de exhaustividad, las siguientes:

- El incremento del consumo energético que se produce con la ejecución en paralelo del programa de detección facial respecto a la ejecución secuencial es, en general, uno de los principales inconvenientes que se producen en la paralelización de un programa informático. Este punto es un tema de sumo interés en el ámbito de los dispositivos experimentales de prueba que se están usando, ya que este tipo de dispositivos suelen depender de baterías cuya autonomía se ve muy afectada por el tipo de procedimientos que se ejecuten en los mismos. Por tanto, cualquier nueva técnica, además de las consideradas en esta tesis doctoral, que consiga optimizar el consumo será de gran interés. Así por ejemplo, una línea interesante sería utilizar para la implementación del sistema GPUs o FPGAs de última generación que presentan cada vez mejores prestaciones.

- También puede ser sumamente interesante para la mejora de los tiempos de ejecución y del consumo energético en cualquier dispositivo, llevar a cabo un análisis detallado de las características de los clasificadores en cascada usadas en la detección de rostros. Bien a través de la inclusión de nuevas características más definitorias en el proceso de detección, o bien mejorando el método de aprendizaje Adaboost, utilizado en el algoritmo de Viola-Jones, que requiere un gran esfuerzo computacional y tiempo para su entrenamiento.

- Otro campo de estudio que surge en base al análisis que se ha llevado a cabo en esta tesis doctoral, es el referente al formato, resolución y tonalidad que debe tener la imagen de entrada al sistema de detección facial, para que el rendimiento del sistema sea óptimo, tanto en tiempo de ejecución como en consumo energético. Como se ha podido comprobar, el formato, la resolución y la tonalidad de la imagen influyen en la velocidad de ejecución del programa de

detección. Por tanto, una línea de trabajo futuro podría ser investigar la influencia de estos parámetros con objeto de encontrar las características que debe tener la imagen de entrada para que el proceso de detección facial sea óptimo.

- Por último, sería interesante aplicar todas las técnicas vistas en el presente trabajo de investigación al reconocimiento e identificación de personas a través del rostro. Este área de investigación amplia enormemente el campo de análisis ante el abanico de posibilidades que se abren. Se podría, por ejemplo, explorar dentro del conjunto de características que determina si una imagen es un rostro, establecer un criterio de selección que permita la clasificación y por tanto la identificación de los rostros detectados de forma sencilla, como se ha conseguido con el sistema Facenet, pero con un rendimiento mucho mayor.

En definitiva, la detección y reconocimiento facial es un tema de investigación de sumo interés en el campo de la visión artificial actualmente. Por tanto, las líneas de investigación son innumerables, al igual que las aplicaciones prácticas en el mundo digital actual.

9 BIBLIOGRAFIA

- [Abyan, 2011] P.K. Aby , J. Anumol , J. Bibin, L.D Dinu, J. Jomon, G. Sabarinath. "*Implementation and optimization of embedded Face Detection system*". 2011 International Conference on Signal Processing, Communication, Computing and Networking Technologies, Thuckafay (India), pp. 250-253. 2011.
- [Acasa, 2011] L. Acasandrei, A. Barriga: "*Accelerating Viola-Jones Face Detection for Embedded and SoC Environments*", Fifth ACM/IEEE International Conference on Distributed Smart Cameras (ICDSC'2011), Ghent, Belgium. 2011.
- [Adria, 2016] A. García, A. sanz. "*Soporte de sistema operativo para ahorro de energía en plataformas móviles con procesadores multicore asimétricos*". Trabajo fin de grado. Universidad Complutense de Madrid, 2016.
- [Alexe, 2001] A. Kalinov and A. L. Lastovetsky. "*Heterogeneous distribution of computations solving linear algebra problems on networks of heterogeneous computer*s". Journal of Parallel and Distributed Computing. Vol 61 (4), pp. 520-535. 2001.
- [Alons, 2012] P. Alonso, R. M. Badia, J. Labarta, M. Barreda, M. F. Dolz, R. Mayo, E. S. Quintana- Ortí, and R. Reyes. Tools for power-energy modelling and analysis of parallel scientific applications. In 41st Int. Conf. on Parallel Processing (ICPP'2012), Pittsburg (USA), pages 420–429, 2012.
- [Angel, 2008] A. L. Calvo, A. Cortés, D. Giménez, and C. Pozuelo. "*Using metaheuristics in a parallel computing course".* In Marian Bubak, G. Dick van Albada, J. Dongarra, and P. M. A. Sloot. Editors, ICCS (2), vol. 5102 of Lecture Notes in Computer Science, pp. 659-668. Springer, 2008.
- [Annav, 2005] M. Annavaram, E. Grochowski, and J. Shen. "*Mitigating Amdahl's Law through EPI Throttling".* In Proceedings of ISCA 05, IEEE Computer Society, Washington, DC, USA, pp. 298-309. 2005.
- [Anton, 2003] A. J. Dorta, J. A. González, C. Rodríguez, and F. Sande. "*A parallel skeletal language. Parallel".* Processing Letters. Vol 13(3), pp. 437-448. 2003.

- [Archi, 2018]. Archit Gajjar , Xiaokun Yang , Lei Wu , Hakduran Koc , Ishaq Unwala , Yunxiang Zhang. "*An FPGA Synthesis of Face Detection Algorithm using HAAR Classifier"*. Proceedings of the 2018 2nd International Conference on Algorithms, Computing and Systems. Beijing, China. PP. 133-137. Jul. 2018.
- [Armbe, 2016] ARM. "*Benefits of the big.LITTLE Architectu*re". [Online] Available: http://www.arm.com/files/downloads/Benefits_of_the_big.LITTLE_architecture.pdf . Last visit: Oct. 2016.
- [Armco, 2017] ARM Cortex-A Series Programmer's Guide for ARMv8-A. [Online] Available: http://infocenter.arm.com/help/topic/com.arm.doc.den0024a/DEN0024A_v8_architecture_PG.pdf. Last visit: Mar. 2017.
- [Bhati, 2016] A. R. Bhatia, N. M. Patel, and N. C. Chauhan. "*Parallel Implementation of Face Detection Algorithm on GPU*". 2016 2nd International Conference on Next Generation Computing Technologies (NGCT), pp. 674-677. 2016.
- [Binko, 2010] L. B. Kong, P. Jin-ye, X. Zhao-qiang, L. Yu, and F. Shao-chong. "*Realization and Optimization of Face Detection Algorithm based on DSP. 2010 International Conference of Information Science and Management Engineering"*.Volumen 1, pp. 246-249. 2010.
- [BioID, 2018]. BioID Facial Recognition. [Online]. Available: https://itunes.apple.com/us/app/bioid-facial-recognition-authenticator/id1054317153?mt=8. Last visit: Nov. 2018.
- [Biome, 2017] "*Biometría*". [Online]. Available: https://es.wikipedia.org/wiki/Biometr%C3%ADa. Last visit: Marzo 2017.
- [Brani, 2009] B. Kisacanin, S. S. Bhattacharyya, and S. Chai. "Embedded Computer Vision". Ed. Springer-Verlag London Ltd., pp. 139-162, 2009.
- [Brian, 2013] B. Jeff. "*Big.LITTLE Technology Moves Towards Fully Heterogeneous Global Task Scheduling Improving Energy Efficiency and Performance in Mobile Devices"*. [Online] Available: https://www.arm.com/files/pdf/big_LITTLE_technology_moves_towards_fully_heterogeneous_Global_Task_Scheduling.pdf. Nov. 2013.

- [Brune, 1993] Roberto Brunelli y T. Poggio. "*Face recognition: features versus templates*". IEEE Transactions on Pattern Analysis and Machine Intelligence. 1993
- [Calte, 2017] Face Database. [Online] Available: http://www.vision.caltech.edu/html-files/archive.html. Last visit: Jun 2017.
- [Calvo, 2018] Diego Calvo. Red Neuronal Convolucional. [Online] Available: http://www.diegocalvo.es/red-neuronal-convolucional/. Last visit: Mar. 2018.
- [Capot, 2016] V. Capote. "*FPGA y Familia Xilinx*". [Online]. Available: http://fpgayxilinx.blogspot.com/2016/08/cuando-se-aborda-el-diseno-de-un.html. Last visit: Oct. 2016.
- [CBSRD, 2018]. "*CASIA 3D Face Database*". [Online] Available: http://www.cbsr.ia.ac.cn/english/3DFace%20Databases.asp. Last visit: Mar. 2018.
- [Chang, 2008] G. Changjian and L. Shih-Lien, "*Novel FPGA based Haar classifier face detection algorithm acceleration*". In International Conference on Field Programmable Logic and Applications. FPL 2008. Heidelberg (Germany), pp. 373-378, 8-10 Sept. 2008.
- [Cheng, 2011] B. Cheng, C. Lai, C. H. Chiang, and G. R. Li. "*Data Locality Optimization for A Parallel Object detection on Embedded Multi-Core Systems*". 2011 IEEE 2nd International Conference on Software Engineering and Service Science, pp. 576-579. 2011.
- [Chobe, 2009] J. Cho, B. Benson, S. Mirzaei and R. Kastner, "*Parallelized Architecture of Multiple Classifiers for Face Detection*". Presented at the 20th IEEE International Conference on Application-specific Systems, Architectures and Processors, Boston (MA-USA), pp. 75-82. 2009.
- [Chobr, 2009] J. Cho, B. Benson, and R. Kastner. "*Hardware Acceleration of Multi-view Face Detection*". 2009 IEEE 7th Symposium on Application Specific Processors, pp. 66-69. 2009.
- [Chouc, 2015] A. Chouchene, F. E. Sayadi, H. Bahri, J. Dubois, J. Miteran, and M. Atri. "*Optimized parallel implementation of face detection based on GPU component*". Microprocess. Microsyst. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0141933115000526?via%3Dihub. Last visit: Mar. 2017.
- [Chris, 2014] C. Kyrkou, "*Real-Time Hardware Acceleration of Object Detection for Intelligent*". Doctoral Thesis. University of Cyprus 2014.

- [Chron, 2015] K. Chronaki, A. Rico, R. M. Badia, E. Ayguadé, J. Labarta, and M. Valero. "*Criticalityaware dynamic task scheduling for heterogeneous architectures"*. In Proceedings of the 29th ACM on International Conference on Supercomputing, ICS '15, pp. 329-338, New York, NY, USA. ACM. 2015.
- [Corpa, 2018] A. Corpas, L. Costero, G. Botella, F. D. Igual, C. García, and M. Rodríguez. "Acceleration and energy consumption optimization in cascading classifiers for face detection on low cost ARM big. LITTLE asymmetric architectures". International Journal of Circuit Theory and Application. DOI: 10.1002/cta.2552. 2018.
- [Crist, 2010] C. Grozea, Z. Bankovic, and P. Laskov, "*FPGA vs. Multi-Core CPUs vs. GPUs: Hands-on Experience with a Sorting Application*". In Facing the Multi-Core Challenge: Conf. for Young Scientists at the Heidelberger Akademie der Wissenschaften, Heidelberg (Germany), 2010.
- [Danmo, 1992] D. I. Moldovan. "*Parallel Processing: From Applications to Systems*". Ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. 1992.
- [Dasat, 2012] S. Das, A. Jariwala and P. Engineer. "*Modified Architecture for Real-Time Face Detection using FPGA*". 2012 Nirma University International Conference on Engineering (NUiCONE), pp. 1-5. 2012.
- [David, 2013] David L. González-Alvarez. "*Metaheurísticas, Optimización Multi-objetivo y Paralelismo para Descubrir Motifs en Secuencias de ADN*". Tesis doctoral. Universidad de Extremadura, 2013.
- [DDiaz, 2018] Diana Diaz. Fotografían a 4 Hermanas. [Online]. Available: https://www.recreoviral.com/fotografia/fotografian-a-hermanas-misma-forma-por-40-anos/. Last visit: Feb. 2018.
- [Denna, 1972] R. H. Dennard, F. H. Gaensslen, L. Kuhn, and H. N. Yu, "*Design of micron MOS switching devices*", International Electron Devices Meeting, pp. 168-170, 1972.
- [Dobai, 2013] R. Dobai and L. Sekanina, "*Towards evolvable systems based on the Xilinx Zynq platform*". In IEEE International Conference on Evolvable Systems, Singapore, 2013, pp. 89-95.
- [Duran, 2011] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. "*OmpSs: a proposal for programming heterogeneous*

multi-core architectures". Parallel Processing Letters, vol 21(02), pp. 173-193. 2011.

- [Ebenu, 2012] E. Upton and G. halfacree, "Raspberry Pi User's Guide". Ed. John Wiley & Sons Inc. 2012.
- [Face2, 2018]. Face2Gene. [Online]. Available: https://www.face2gene.com/. Last visit: Nov. 2018.
- [Faced, 2018] APP. Face Detection Screen Loc. [Online]. Available: https://play.google.com/store/apps/details?id=com.aiappstudio.facedetectionscreenlock&hl=en. Last visit: Nov. 2018.
- [Facep, 2018]. Face Phi. [Online]. Available: https://www.facephi.com/en/. Last visit: Nov. 2018.
- [Fenla, 2017] J. Fenlason and R. Stallman. "*The GNU Profiler*" [Online]. Available: https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_mono/gprof.html. Last visit: Feb.2017.
- [Findb, 2018] FindBiometrics, Facial recognition, [Online]. Available: http://findbiometrics.com/solutions/facial-recognition/. Last visit: feb 2018.
- [Findf, 2018]. APP FindFace. [Online]. Available: https://play.google.com/store/apps/details?id=ru.trinitydigital.findface&hl=en. Last visit: Nov. 2018.
- [Frana, 2004] F. Almeida, R. Andonov, L. M. Moreno, V. Poirriez, M. Pérez, and Casiano Rodríguez. "*On the parallel prediction of the RNA secondary structure*". In Advances in Parallel Computing, vol 13, pp. 525-532, 2004.
- [Freund, 1996] Y. Freund, and R. E. Schapire. "*Experiments with a New Boosting Algorithm*". In Proc. of the 13th International Conference on Machine Learning, Bari (Italy), pp. 148-156, 1996.
- [Garci, 2003] O. R. Garcia, E. Cruz. "*Introduccción a OpenMP*". Trabajo de la Universidad Nacional Autónoma de México. 2003.
- [Garct, 1999] C. Garcia, G. Tziritas. "*Face Detection Using Quantized Skin Color Regions Merging and Wavelet Packet Analysis*", IEEE Transactions on Multimedia, Vol. 1, pp. 264-277, 1999.
- [Golds, 1971] A. J. Goldstein, L. D. Harmon, and A. B. Lesk, "*Identification of Human Faces*". In Proc. IEEE Conference on Computer Vision and Pattern Recognition, vol. 59, pp 748-760, May 1971.

- [Gopal, 2013] J. Gopal, A. Karmakar, and C. Shektar. "*Platform-Based Design Approach for Embedded Vision Applications*". Journal of Image and Graphics, vol. 1, no. 1, pp. 1-6, March 2013.
- [Gperf, 2018] CPU Profiler. [Online]. Available: https://gperftools.github.io/gperftools/cpuprofile.html. Last visit: Mar. 2018.
- [Grego, 2009] G. Quintana-Ortíz, E. S. Quintana-Ortíz, R. A. van de Geijn, F. G. Van Zee, and E. Chan. "*Programming matrix algorithms-by-blocks for thread-level parallelism*". ACM Transactions on Mathematical Software. Vol. 36(3), 2009.
- [Gschw, 2007] M. Gschwind. "*The Cell Broadband Engine: exploiting multiple levels of parallelism in a chip multiprocessor*". International Journal of Parallel Programming. Vol. 35(3), pp. 233-262, 2007.
- [Gueva, 2008] M.L. Guevara, J. D. Echeverry, W. Ardila. "*Detección de rostros en imágenes digitales usando clasificadores en cascada*". Scientia et Technica Año XIV, Nº 38, Junio 2008.
- [Guosh, 2015] Hu, Guosheng, et al. "*When face recognition meets with deep learning: an evaluation of convolutional neural networks for face recognition*". Proceedings of the IEEE International Conference on Computer Vision Workshops. 2015.
- [Gupta, 2010] B. Gupta, S. Gupta, and A. Tiwari, "*Face Detection using Gabor Feature Extraction and Artificial Neural Networks*". Proceedings from ISCET, pp. 18-23, 2010.
- [Haipe, 2012] H. Jia, Y. zhang, W. Wang, H. Jia, and J. Xu. "*Accelerating Viola-Jones Facce Detection Algorithm On GPUs*", 2012 IEEE 14th International Conference on High Performance Computing and Communications, pp.396-403. 2012.
- [Hardk, 2016] Hardkernel. Descarga Sistema operativo. [Online] Available: http://dn.odroid.com/5422/ODROID-XU3/Ubuntu/. Last visit: Sept. 2016
- [Harre, 2009] P. Harrell, Y. Hanafy, A. Bayoumi, G. Refai-Ahmed and M. Chu, "*Scientific and Engineering Computing Using ATI Stream Technology*". In Computing in Science & Engineering, doi:10.1109/MCSE.2009.204, vol. 11, pp. 92-97, 2009.

- [Haupm, 1997] Hauptmann, A., y Witbrock, M. "*Informedia news on demand: Multimedia information acquisition and retrieva"l*, M. T., Ed, Intelligent Multimedia Information Retrieval, AAAI Press/MIT Press, Menlo Park, CA. 1997.
- [Hillm, 2008] M. D. Hill and M. R. Marty. "*Amdahl's Law in the Multicore Era*". IEEE Computer Society, vol. 41(7), pp. 33-38, 2008
- [Hsuan, 2012]. C. H. Chiang, C. H. Kao, G. R. Li, and B. Cheng, and C. Lai. "*Multi-Level Parallelism Analysis of Face Detection on a Shared Memory Multi-Core System*". Proceedings of 2011 International Symposium on VLSI Design, Automation and Test. Papers (7), pp. 1-4. 2011.
- [Hungc, 2007] L. Hung-Chih, M. Savvides and C. Tsuhan, "*Proposed FPGA Hardware Architecture for High Frame Rate Face Detection Using Feature Cascade Classifiers*". In First IEEE International Conference on Biometrics: Theory, Applications, and Systems. BTAS 2007, Washington DC (USA), pp. 1-6, 27-29 Sept. 2007.
- [Incit, 2004] InterNational Committee for Information Technology Standards (INCITS). "*Face Recognition Format Data Interchange*". Document 385ANSI INCITS, 2004.
- [Intel, 2018]. "*Movidius Neural Compute Stick*" [Online]. Available: https://software.intel.com/en-us/neural-compute-stick. Last visit. Jun. 2018.
- [Jacob, 2017]. Tom Jacobs. "*Running Yolo on Odroid*". [Online] Available: https://hackernoon.com/running-yolo-on-odroid-yolodroid-5a89481ec141. Last visit: Jun. 2018.
- [Jerem, 2012] J. Fowers, G. Brown, P. Cooke, and G. Stitt, "*A performance and energy comparison of FPGAs, GPUs, and multicores for sliding-window applications*". In Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays (FPGA '12), pp. 47-56. 2012.
- [Jerom, 2000] J. Friedman, T. Hastie, and R. Tibshirani. "*Additive logistic regression: A statistical view of boosting*". The Annals of Statistics, 28(2), pp. 337-374, 2000.
- [Jianf, 2008] J. Ren, N. Kehtarnavaz, and L. Estevez. "*Real-Time Optimization of Viola -Jones Face Detection for Mobile Platforms"*. 008 IEEE Dallas Circuits and Systems Workshop: System-on-Chip - Design, Applications, Integration, and Software. Paper 10. 2008.

- [Jians, 2018]. Infinistysky, "*How to work with Tensorflow on Odroid*". [Online] Available: https://www.jianshu.com/p/375cacb4c0f2. Last visit: Nov. 2018.
- [Johnl, 2019] John L. Hennessy and David A. Patterson. 2019. *"A new golden age for computer architecture"*. *Commun. ACM* 62, pp. 48-60. Jan. 2019. DOI: https://doi.org/10.1145/3282307.
- [Johnm, 2009] J. Maindonald. "*Modern multivariate statistical techniques: Regression, classification and manifold learning*". Journal of Statistical Software, Book Reviews, 29(11), pp. 1-3. 2009.
- [Juanp, 2009] J. P. Martinez. "*Autooptimización en esquemas algorítmicos paralelos iterativos"*. Tesis Doctoral. Universidad de Murcia, 2009.
- [Karnr, 2012] K.N. Debbarma, M. K. Saha, A, and D. R. Pal. "*Study of implementing automated attendance system using face recognition technique*". International Journal of computer and communication engineering, Vol. 1, Nº 2, pp. 100-103, 2012.
- [Kassn, 2017] Nora Kassner. "Real-time face detection with Haar cascades". [Online] Available: http://www.mathematik.uni-muenchen.de/~deckert/light-and-matter/teaching/SS17/ATML/media/FaceDetection.pdf. Last visit: Feb. 2017.
- [Kimab, 2017] M. Kim, A. Mohanty, D. Kadetotad, N. Suda, Luning Wei, Pooja Saseendran, X. He, Y. Cao, and J. Sun-Seo. "*A real-time 17-scale object detection accelerator with adaptive 2000-stage classification in 65nm CMOS"*. 22nd Asia and South Pacific Design Automation Conference (ASP-DAC), pp. 21-22. 2017.
- [Kimse, 2013] M. Kim and S. Han. "*Real-time Face Detection on Reconfigurable Device*". 2013 International Conference on ICT Convergence (ICTC), pp. 726-727. 2013.
- [Kjaer, 2010] A. Kjaer-Nielsen, "*Real-time Vision using FPGAs, GPUs and Multi-core CPUs*". University of Southern Denmark, Doctoral Dissertation 2010.
- [Krist, 2012] K. Reese, Y. Zheng, A. Elmaghraby. "*A Comparison of Face Detection Algorithms in Visible and Thermal Spectrums*". Proc. of International conference on Advances in Computer Science and Application. 2012.
- [Kumar, 2003]. Kumar et al. "*Single-ISA Heterogeneous Multi-Core Architectures: the Potential for Processor Power Reduction*". Proceedings. 36th Annual

IEEE/ACM International Symposium on Microarchitecture, MICRO 36, San Diego (California - USA). 2003.

- [Lienh, 2003] R. Lienhart, A. Kuranov, and V. Pisarevsky. "*Empirical analysis of detection cascades of boosted classifiers for rapid object detection*". In DAGM 25th Pattern Recognition Symposium, Magdeburg (Germany)pp. 297-304. 2003.
- [Lopez, 2018] Raúl E López Briega. "*Red Neuronal Convolucional con Tensorflow*". [Online] Available: https://relopezbriega.github.io/blog/2016/08/02/redes-neuronales-convolucionales-con-tensorflow/. Last visit: Mar. 2018.
- [Luisc, 2016] L. M. Costero Valero. "*Evaluación y optimización de rendimiento y consumo energético de aplicaciones paralelas a nivel de tareas sobre arquitecturas asimétricas*". Trabajo Fin de máster, Universidad Complutense de Madrid, 2016.
- [Luxan, 2018]. Detect and Recognize Faces with Luxand FaceSDK. [Online]. Available: https://www.luxand.com/facesdk/. Last visit: Nov. 2018.
- [Manda, 2017] Arun Mandal. "*Face match in Python using Facenet and their pretrained model*". [Online]. Available: https://github.com/arunmandal53/facematch. Last Visit: May. 2018.
- [Manda, 2018] Arun Mandal. "*MTCNN Face Detection and Matching using Facenet Tensorflow*". [Online]. Available: https://www.python36.com/face-detection-matching-using-facenet/. Last visit. Jun. 2018.
- [Masek, 2013] J. Masek, R. Burget, V. Uher, and S. Guney. "*Speeding up Viola–Jones Algorithm using Multi–Core GPU Implementation*". 2013 36th International Conference on Telecommunications and Signal Processing (TSP), pp. 808-812. 2013.
- [Matai, 2011]. J. Matai, A. Irturk and R. Kastner. "*Design and Implementation of an FPGA-Based Real-Time Face Recognition System"*. 2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines, pp. 97-100. 2011.
- [Mattr, 2012] M. Richardson, and S. Wallace, "*Getting started with Raspberry Pi*". Ed: O'Reilly. 2012.
- [Mense, 1999] B. Menser, F. Muller. "*Face detection in color images using principal components analysis*". Seventh International Conference on Image Processing and Its Applications, Manchester (United Kingdom) pp. 620-624, 1999.

- [Mercu, 2018] Mercurium compiler. [Online] Available: https://pm.bsc.es/mcxx. Last visit: Feb. 2018.
- [Mielk, 2011] R. Mielke, M. Schafer, A. Bruck, "*ASIC implementation of a gaussian pyramid for use in autonomous mobile robotics*", in IEEE 54th International Midwest Symposium on Circuits and Systems (MWSCAS), Seoul (South Korea), pp. 1–4. 2011.
- [Mitta, 2016] S. Mittal. "*A survey of techniques for architecting and managing asymmetric multicore processors*". Journal ACM Computing Surveys, Article Nº45, pp. 38-45, 2016.
- [Molon, 2014] D. Moloney, B. Barry, R. Richmond, F. Connor, C. Brick, and D. Donohoe, "*Myriad 2: Eye of the computational vision storm*" in IEEE HotChips Symposium (HCS), Aug. 2014, pp. 1–18.
- [Morel, 2011] A. Morelli y S. Padovani. "*Detección y Reconocimiento de Caras".* Tesis de licenciatura en ciencias de la computación. Universidad de Buenos Aires. 2011.
- [Moren, 2005] L. M. Moreno. "*Computación en paralelo y entornos heterogéneos".* Ph. D Thesis.Universidad de La Laguna, 2005.
- [Nanos, 2018] Runtime Nanos++. [Online] Available: https://pm.bsc.es/nanox. Last visit: Feb. 2018.
- [Natio, 2016] faces evolution cognition social recognition [Online] Available: https://news.nationalgeographic.com/news/2014/09/140916-faces-evolution-cogniti on-social-recognition-genetics/. Last visit: Mar. 2016.
- [Ncapp, 2018]. "*Neural Compute Application Zoo (NC App Zoo)*". [Online]. Available: https://github.com/movidius/ncappzoo. Last visit. Jun. 2018.
- [Odro2, 2017] ODROID Forum [Online] Available: http://forum.odroid.com. Last visit: Mar 2017.
- [Odro3, 2017] ODROID Magazine [Online] Available: http://magazine.odroid.com. Last visit: Mar 2017.
- [Odroi, 2016] R. Roy, V. Bommakanti. Manual de Usuario Odroid XU4, Hardkernel. [Online]. Available: https://magazine.odroid.com/wp-content/uploads/odroid-xu4-user-manual-espanol.pdf. Last visit: Sept. 2016
- [Ohyar, 2000] J. Ohya, R. Nakatsu, J. Tang, S. Kawato. "*Locating human faces in a complex background including non-face skin colors*". International Conference on

Imaging Science, Systems, and Technology, Las Vegas (Nevada-USA) Vol. 12. 2000.

- [Ompss, 2018] OmpSs project home page. [Online]. Available: http://pm.bsc.es/ompss. Last visit: febrero 2018.
- [Openm, 2017] Blaise Barney, Lawrence Livermore National Laboratory. "*OpenMP*". [Online] Available: https://computing.llnl.gov/tutorials/openMP. Last visit: Abr. 2017.
- [Opeor, 2016] The OpenMP API specification for parallel programming. [Online] Available: http://openmp.org. Last visit: Oct. 2016.
- [Owens, 2008] J. D. Owens, M. Houston, D. Luebke, J. E. Stone, and J. C. Phillips, "*GPU Computing*". Proceedings of the IEEE, vol. 96, Nº 5, pp. 879-899, 2008.
- [Papag, 1998] C. Papageorgiou, M. Oren, and T. Poggio. "*A general framework* for object detection". In Sixth International Conference on Computer Vision, Bombay (India), pp. 555-562. 1998.
- [Peers, 1999] P. Peer, F. Solina. "*An Automatic Human Face Detection Method*". Proceedings of Computer Vision Winter Workshop, Ed. N. Brändle, pp. 122-130, 1999.
- [Pentl, 1991] A. Pentland y M. Turk. "*Face Recognition Using Eigenfaces*". Vision and Modeling Group, The Media Laboratory Massachusetts Institute of Technology. 1991.
- [Phamw, 2001] T. V. Pham, M. Worring, A. W. M. Smeulders. "*Face detection by aggregated Bayesian network classifiers*". Pattern Recognition Letters, Vol. 23, págs. 451-461, 2001.
- [Plane, 2009]. J. Planells. "*Implementación del algoritmo de detección facial de Viola-Jones*". Proyecto Fin de Carrera. Universidad de Valencia. 2009.
- [Qiong, 2017]. Qiong Cao, Li Shen, Weidi Xie, Omkar M. Parkhi, Andrew Zisserman. VGGFace2: "*A dataset for recognising faces across pose and age*". IEEE Conference on Automatic Face and Gesture Recognition. Oct 2017.
- [Raine, 2002] R. Lienhart and J. Maydt. "*An extended set of haar-like features for rapid object detection*". In Proceedings of International Conference on Image Processing, Rochester (New York-USA), pp. 900-903, 2002.

- [Ranja, 2017]. A. Ranjan, and S. Malik. "*Parallelizing a Face Detection and Tracking System for Multi-Core Processors*". 2012 Ninth Conference on Computer and Robot Vision, pp. 290-297. 2012.
- [Rasb2, 2017] Raspberry pi Foundation. [Online] Available: https://www.raspberrypi.org/blog/raspberry-pi-3-model-bplus-sale-now-35/. Last visit: Mar. 2018.
- [Raspb, 2017] Raspberry Pi Foundation. [Online]. Available: https://www.raspberrypi.org. Last visit: Jun. 2016.
- [Roger, 2002] S. Rogerson, "*Smart CCTV*". ETHIcol in the IMIS Journal, vol. 12, nº 1, February 2002.
- [Ronen, 2001] R. Ronen, A. Mendelson, K. Lai, S. L. Lu, F. Pollack, and J.P. Shen "*Coming challenges in microarchitecture and architecture*". Proceedings of the IEEE, vol. 89, no. 3, pp. 325-340, Mar. 2001.
- [Ronsa, 2010] R. Sass and A. G. Schmidt. "*Embedded Systems Design with Platform FPGAs*", 1st ed. USA: Morgan Kaufmann - Elsevier, 2010.
- [Rowle, 1996] H. A. Rowley, S. Baluja, T. Kanade. "*Neural Network-Based Face Detection*". IEEE Transactions On Pattern Analysis and Machine Intelligence, Vol. 20, pp. 23-38, 1996.
- [Sabem, 1998] E. Saber, A. Murat Tekalp. "*Frontal-View Face Detection and Facial Feature Extraction using Color*". Shape and Symmetry Based Cost Functions. 1998.
- [Sahoo, 2008] H. Sahoolizadeh, D. Sarikhanimoghadam, and H. Deghani, "*Face Detection using Gabor Wavelets and Neural Networks*". World Academy of Science, Engineering and Technology, vol. 45, pp. 552-554, 2008.
- [Sandb, 2018] David Sandberg. "*Facenet*". [Online]. Available: https://github.com/davidsandberg/facenet. Last visit: Jun. 2018.
- [Savat, 2018]. Savath Saypadith and Supavadee Aramvith. "*Real-Time Multiple Face Recognition using Deep Learning on Embedded GPU System*". Proceedings, APSIPA Annual Summit and Conference 2018. Hawaii. Nov. 2018.
- [Schap, 1990] R. E. Schapire. "*The strength of weak learnability. Machine Learning*". Kluwer Academic Publishers, Boston, pp:197-227. 1990.

- [Schro, 2015] Florian Schroff, Dmitry Kalenichenko, James Philbin. "*FaceNet: A Unified Embedding for Face Recognition and Clustering*". Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition. 2015.
- [Shife, 2018]. Shifeng Zhang Xiangyu Zhu Zhen Lei Hailin Shi Xiaobo Wang Stan Z. Li. "*FaceBoxes: A CPU Real-time Face Detector with High Accurac*y". 2018 IEEE International Joint Conference on Biometrics (IJCB). Denver, CO, USA. PP. 1-9. 2018.
- [Sirov, 1987] L. Sirovich and M. Kirby. "*Low Dimensional procedure for the characterization of human faces*". Journal of the optical Society of America. Vol 4, pag 519, March 1987.
- [Stein, 2005] G. P. Stein, E. Rushinek, G. Hayun, and A. Shashua, "*A Computer Vision System on a Chip: a case study from the automotive domain*". 2005 IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit. Vol. 3, pp. 130-130, 2005.
- [Steph, 2017] Face Database. [Online] Available: https://github.com/StephenMilborrow/muct/blob/master/muct-a- jpg-v1.tar.gz. Last visit: Jun 2017.
- [Strat, 2019]. Intel FPGAs and Programmable Device. [Online]. Available: https://www.intel.com/content/www/us/en/products/programmable/stratix-series.html. Last visit: Feb. 2019.
- [Sused, 2015] V. Suse, and D. Ionescu, "*A Real-Time Reconfigurable Architecture for Face Detection*". 2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig). Papers (2), pp. 1-6. 2015.
- [Tensor, 2018]. "*An open source machine learning library for research and production*". [Online] Available: https://www.tensorflow.org/?hl=es. Last visit: Mar. 2018.
- [Tulls, 2004] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. "*Single-ISAHeterogeneous Multi-Core Architectures for Multithreaded Workload Performance*". Proceedings 31st Annual International Symposium on Computer Architecture, München (Germany), pp. 64–75. 2004.
- [Valgr, 2017] Valgrind. [Online]. Available: http://valgrind.org/. Last visit: Feb. 2017.
- [Valia, 1984] L. G. Valiant. "*A theory of the learnable. Commun*". Association for computing Machinery (ACM), Vol. 27, pp. 1134-1142. 1984.

- [Villeg, 2005] C. Villegas. "*Reconocimiento de rostros utilizando análisis de componentes principales*". Tesis Doctoral Universidad Iberoamamericana 2005.
- [Viola, 2001] P. Viola and M. Jones. "*Rapid Object detection using a boosted cascade of simple features*". Proceedings of Computer Vision and Pattern Recognition. Vol. 1 pp. 511–518, 2001.
- [Viola, 2004] P. Viola and M Jones. "*Robust real-time face detection*". International Journal of Computer Vision, vol 57, pp. 137-154, 2004.
- [Wongy, 2015] A. Wong, Y. Wai, S. Mohd Tahir, and Y. Choon Chang. "*GPU acceleration of real time Viola-Jones face detection*". 2015 IEEE International Conference on Control System, Computing and Engineering, 27 - 29 November 2015, Penang, Malaysia, pp. 183-188. 2015.
- [Xilin, 2018] Xilinx, "*Zynq-7000 All Programmable SoC*" 2014. [Online]. Available: http://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html. Last visit: Jan-2017.
- [Yangh, 1994] G. Yang, T.S. Huang. "*Human Face Detection in a Complex Background*". Pattern Recognition, pp. 53-63, Vol. 27, 1994.
- [Yangq, 2018] Yangqing Jia, Evan Shelhamer, "*CAFFE*", [Online] Available: http://caffe.berkeleyvision.org/. Last visit: Jun. 2018.
- [Young, 2016] Y. Lee, C. Jang, and H. Kim. "*Accelerating a computer vision algorithm on a mobile SoC using CPU-GPU co-processing. A case study on face detection*". 2016 IEEE/ACM International Conference on Mobile Software Engineering and Systems. Austin (USA), pp. 70-76. 2015.
- [Zhang, 2016] Kaipeng Zhang, Zhanpeng Zhang , Zhifeng Li ; Yu Qiao, "*Joint Face Detection and Alignment Using Multitask Cascaded Convolutional Networks*". IEEE Signal Processing Letters. Volume: 23, Issue: 10, pp. 1499 – 1503. Oct. 2016.
- [Zheny, 2017] Y. Zhenyu. "*5KK73 GPU Assignment 2012".* [Online] Available: https://sites.google.com/site/5kk73gpu2012/assignment/viola-jones-face-detection. Last visit: March 2017.
- [Zhouy, 2011] W. Zhou, Y. Zou, L. Dai, and X. Zeng. "*A High Speed Reconfigurable Face Detection Architecture*". 9th IEEE International Conference on ASIC. Xiamen (China), pp. 83-86. 2011.
-

APÉNDICE A

PRINCIPALES DIRECTIVAS DE OPENMP Y OMPSS

1.-Principales directivas de OpenMP

La mayoría de las construcciones en OpenMP son directivas de compilación o pragmas, que atendiendo al lenguaje de programación utilizado siguiendo una estructura como la que se muestra en la siguiente tabla:

Lenguaje	Centinela	Directiva	Cláusula
C/C++	#pragma omp	Justo después del centinela y antes de las opciones	Opciones aplicables a la directiva
Fortran	!$OMP C$OMP *$OMP	Justo después del centinela y antes de las opciones	Opciones aplicables a la directiva

Estructura de OpenMP en C/C++ y Fortran.

Las directivas son tomadas por comentarios por aquellos compiladores que no están preparados para interpretarlas. A continuación se describe brevemente los constructores OpenMP que se han aplicado a lo largo de esta tesis doctoral:

- **Constructor PARALLEL**: definen una región del programa que puede ser ejecutada por múltiples hilos en paralelo. El hilo que encuentra el constructor pasa a ser el hilo maestro del nuevo equipo paralelo, asignándose el identificador número cero durante todo el bloque paralelo. Todos los hilos, incluido el hilo maestro, ejecutarán el contenido del bloque paralelo manteniéndose constante el número de hilos hasta el final del bloque paralelo. Al final del bloque paralelo solo el hilo maestro continuará con su ejecución. Su sintaxis es la siguiente:

  #pragma omp parallel [cláusula[[,]cláusula]…]

  Donde cláusula puede ser una de las siguientes: if ,num threads (entero), default (shared | none), private (lista), firstprivate (lista), shared (lista), copyin (lista),reduction (lista).

- **Constructor SINGLE**: especifica que un bloque de código determinado solo será ejecutado por uno de los hilos definido en el equipo de hilos (no necesariamente ha de ser el hilo maestro). Su notación dentro del programa es la siguiente:

  #pragma omp single [cláusula[[,]cláusula]…]

  Donde cláusula puede ser una de las siguientes: private (lista), firstprivate (lista), copyprivate (lista), nowait.

- **Constructor TASK**: Cuando un hilo se encuentra un constructor *task*, se genera una tarea que ejecutará el código correspondiente al bloque paralelo definido. Su sintaxis es la siguiente:

  #pragma omp task [cláusula[[,]cláusula]…]

Donde cláusula puede ser una de las siguientes: if (expresión escalar), final (expresión escalar), untied default (shared | none), mergeable, private (lista), firstprivate (lista), shared (lista).

- **Constructor TASKWAIT**: especifica la región donde cada hilo espera que los demás hilos del equipo terminen su ejecución.

#pragma omp taskwait

Funciones OpenMP

Para poder utilizar las funciones OpenMP, es necesario incluir el archivo omp.h, mediante: #include <omp.h> en el código del programa. Las principales funciones que ofrece son las siguientes:

- **void omp_set_num_threads(int num_threads).** Establece el número de hilos a utilizar en la siguiente región paralela.
- **int omp_get_num_threads(void)**. Obtiene el número de hilos que se están utilizando en una región paralela.
- **int omp_get_max_threads(void)**. Obtiene la máxima cantidad posible de hilos.
- **int omp_get_thread_num(void)**. Devuelve el número del hilo.
- **int omp_get_num_procs(void)**. Devuelve el máximo número de procesadores que pueden asignarse al programa.
- **int omp_in_parallel(void)**. Devuelve valor distinto de cero si se ejecuta dentro de una región paralela.
- **int omp_set_dynamic(void)**. Permite gestionar si el número de hilos se quiere ajustar dinámicamente en las regiones paralelas.
- **int omp_get_dynamic(void)**. Permite conocer si el número de hilos se ajusta dinámicamente en las regiones paralelas.
- **int omp_set_nested(void)**. Gestiona el paralelismo anidado.
- **int omp_get_nested(void)**. Devuelve un valor distinto de cero si está permitido el paralelismo anidado.
- **void omp_init_lock(omp_lock_t *lock)** y **void omp init nest lock(omp nest lock t *lock)**. Para inicializar una llave que puede ser simple o

anidada. Una llave se inicializa como no bloqueada, y una llave anidada se inicializa a cero.

- **void omp_init_destroy(omp_lock_t *lock)** y **void omp init nest destroy(omp nest lock_t *lock)**. Para destruir una llave.
- **void omp_set_lock(omp_lock_t *lock)** y **void omp set nest lock(omp nest lock t *lock)**. Para pedir una llave.
- **void omp_unset_lock(omp_lock_t *lock**) y **void omp unset nest lock(omp nest lock t *lock)**. Para soltar una llave.
- **int omp_test_lock(omp_lock_t *lock)** y **int omp unset test lock(omp nest lock t *lock)**. Intenta pedir una llave pero no se bloquea.

Compilación y ejecución OpenMP

Existen multitud de compiladores que soportan las directivas OpenMP, como son: GNU, intel, HP, MS, Cray... etc. A continuación se muestra en la tabla las opciones de compilación de los dos compiladores más comunes:

Desarrollado	**Compilador**	**Opción**
GNU	gcc	Compilar con la opción **-fopenmp**
Intel	icc	Compilar con la opci´on **-openmp**

Compiladores comunes OpenMP.

Para compilar un programa es suficiente con indicarle al compilador que interprete las directivas OpenMP. Si no se hace, el compilador creerá que son comentarios e ignorará las líneas. La sintaxis para compilar en GNU un programa cualquiera es la siguiente:

- **gcc –fopenmp {source.c} opciones -o ejecutable**

En el caso del programa de detección facial la compilación se puede hacer por línea de comando usando la instrucción:

- **make –j CXXFLAGS=-fopenmp LDFLAGS=-fopenmp**

O bien añadiendo directamente al fichero Makefile la opción fopenmp.

2.- Principales directivas de OmpSs

Una vez instalado OmpSs en las placas experimentales: Odroid XU4 y Raspberry Pi 2 B, se puede comenzar a compilar programas OmpSs. Para ello, se debe utilizar el compilador Mercurium C/C++, previamente instalado como se describe en las instrucciones de instalación del compilador que se describen en [Ompss, 2018]. Para compilar un programa C++ en Mercurium se usa la instrucción: **mcxx --ompss.** Así por ejemplo, para compilar el programa: prueba.cpp se puede hacer a través de la instrucción:

$ mcxx –o prueba –ompss prueba.cpp

Como en el caso de OpenMP, las directivas OmpSs se especifican en el programa usando el comando #pragma. En la siguiente tabla, se puede ver la sintaxis que siguen las principales directivas y una breve definición de su función junto con las cláusulas complementarias que se pueden usar .

Directiva	Declaración	Descripción	Cláusulas
Task	#pragma omp task [Cláusulas]	Esta construcción puede aparecer dentro de cualquier bloque de código del programa, que marca la siguiente declaración como una tarea	private(<list>) firstprivate(<list>) shared(<list>) depend priority(<value>) if(<scalar-expression>) final(<scalar-expresion>) label(<string>) tied
Loop	#pragma omp for [Cláusulas]	Cuando una tarea encuentra una construcción de bucle, comienza a crear tareas para cada uno de los trozos en los que se divide el espacio de iteración del bucle	Dynamic Guided Static Nowait
Target	#pragma omp target [Cláusulas]	Para apoyar la heterogeneidad, se introduce una nueva construcción: la construcción objetivo. La intención de la construcción de destino es especificar que un elemento determinado se puede ejecutar en un conjunto de dispositivos	device(target-device) copy_in(list-of-variables) copy_out(list-of-variables) copy_inout(list-of-variables) copy_deps implements(function-name)
Taskwait	#pragma omp taskwait [Cláusulas]	Es una directiva independiente (sin código bloque asociado) y especifica una espera en la finalización de todas las tareas directas de descendientes	on(list-of-variables)
Taskyield	#pragma omp taskyield	Especifica que la tarea actual puede suspenderse y el tiempo de ejecución del planificador puede programar una tarea diferente	
Atomic	#pragma omp atomic	La construcción atómica asegura que la siguiente expresión se ejecute atómicamente	

Critical	#pragma omp critical	Permite a los programadores especificar regiones de código que se ejecutarán en exclusión mutua hasta que no haya ningún subproceso en el equipo que esté ejecutándola	

Principales directivas de OmpSs.

APÉNDICE B

INSTALACIÓN SOFTWARE DE INTEL MOVIDIUS NEURAL STICK (NCS) EN LA PLACA ODROID XU4.

Para poder usar uno o más NCS se requiere la instalación del software de Movidius en la placa experimental. En este caso se ha usado la versión 1.12, que es sobre la que están desarrollados los ejemplos que se suministran en el Ncappzoo [Ncapp, 2018], y que son los que se utilizan en este trabajo. La instalación de este software en la placa Odroid XU4 implica la instalación del Tensorflow y una serie de dependencias, y pasos previos que se tiene que realizar.

Además, se tienen que añadir los siguientes ficheros al sistema de identificación facial:

- **run.py:** Programa perteneciente al Zoo de aplicaciones de ejemplos del NCS.
- **Makefile**: pertenece al conjunto de ficheros proporcionados con el NCS y que se encarga de bajarse los diferentes modelos y su compilación para generar el fichero .graph necesario para cargar en el NCS.
- **Inception_resnet_v1.py**. Adaptacion Facenet en Tensorflow.
- **Facenet_celeb_ncs.graph**: fichero .graph que utiliza el programa (y que general el Makefile) para ser cargado en el NCS.
- **Convert_facenet.py:** Módulo utilizado por el makefile para convertir el fichero del modelo de facenet obtenido, al formato que necesita el NCS para compilarlo.

A continuación se describen detalladamente, los pasos que se han seguido, durante el desarrollo de la tesis doctoral, para instalar el software que utiliza el dispositivo Intel Movidius Neural Stick, sobre la placa Odroid XU4.

1. El sistema operativo instalado en la placa Odroid XU4 es Ubuntu 16.04.3-4.14 mate XU4. Descargado de la web oficial a través del siguiente enlace: https://odroid.in/ubuntu_16.04lts/ubuntu-16.04.3-4.14-mate-odroid-xu4-20171212.img.xz

2. Se instala la librería: "Libgstreamer", previo a la instalación de OpenCV, a partir de la instrucción en línea de comandos: "sudo apt-get install libgstreamer-plugins-base1.0-dev".

3. Se instala OpenCV y Python3 sobre Ubuntu 16.04 , para ello se sigue el procedimiento descrito en el fichero: ".sh", descargado a partir del siguiente enlace: https://gist.github.com/Mahedi-1/804a663b449e4cdb31b5fea96bb9d561

4. Se instala el software para el uso del Movidius Neural Stick (NCS): NCSDK 1.12, descargado a partir del enlace: https://ncs-forum-uploads.s3.amazonaws.com/ncsdk/ncsdk-01_12_00_01-full/ncsdk-1.12.00.01.tar.gz.

   Aunque la versión 2.05 del NCSDK ya está disponible, se ha utilizado la 1.12 debido a que desde el mes de agosto de 2018, la 2.05 introduce el NCAPI v2, que no es compatible con NCAPI v1, sobre el que están desarrollados los ejemplos que se suministran en el Ncappzoo y que son en los que se ha basado la realización del programa de identificación facial.

   Se descomprime y edita el fichero: "ncsdk.conf", con la instrucción: "sudo nano ncsdk.conf" y modificamos para que no se instale Tensorflow ni la herramienta Toolkit. Para ello, solo hay que cambiar en la línea correspondiente: Tensorflow=yes por Tensorflow=no. Y del mismo modo para Toolkit.

Finalmente, se instala el NCSDK, ejecutando dentro del directorio donde se ha descomprimido la instrucción: “make install”.

5. Se instala Tensorflow 1.11.0 para python 3.5. Para ello es necesario previamente instalar un conjunto de paquetes dependientes, a través de las siguientes instrucciones:

```
#sudo add-apt-repository ppa:ubuntu-toolchain-r/test
#sudo apt-get update
#sudo apt-get -y upgrade
#sudo apt-get install -y pkg-config zip g++ zlib1g-dev unzip wget curl -y
#sudo apt-get install -y gcc-4.8 g++-4.8 -y
#sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-4.8 100
#sudo update-alternatives --install /usr/bin/g++ g++ /usr/bin/g++-4.8 100
#sudo apt-get install -y libstdc++6
#sudo apt-get install -y libhdf5-serial-dev hdf5-tools
#sudo apt-get install python3-pip python3-numpy swig python3-dev
#curl https://bootstrap.pypa.io/get-pip.py -o get-pip.py python3 get-pip.py
#Sudo pip install –U scikit-learn
#Sudo  pip install imutils
```

Se descarga la versión de Tensorflow correspondiente, a partir de la instrucción:
#Wget https://github.com/lhelontra/tensorflow-on-arm/releases/download/v1.11.0/tensorflow-1.11.0-cp35-none-linux_armv7l.whl

Se instala Tensorflow 1.11.0, con la instrucción:
#pip3 install ./tensorflow-1.11.0-cp35-none-linux_armv7l.whl

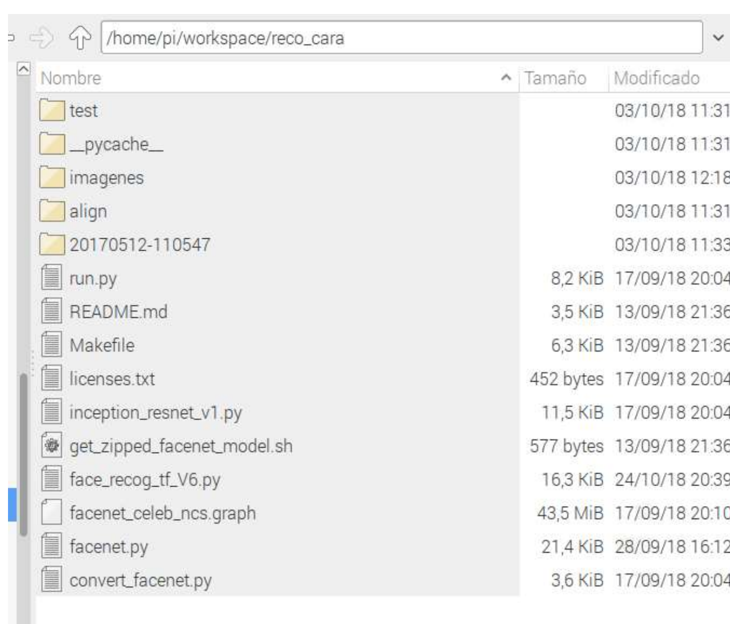
Para comprobar la correcta instalación del software se puede ejecutar la siguiente instrucción por línea de comandos: “**python3 -c 'import tensorflow as tf; print(tf.__version__)**'”. Con lo que se obtiene la versión de Tensorflow que se ha instalado.

NOTA: La instalación de se debe realizar con la versión de Python 3.5. Una versión superior o inferior provoca un error de instalación como el que se indica a continuación: “tensorflow-1.11.0-cp35-none-linux_armv7l.whl is not a supported wheel on this platform”.

APÉNDICE C

MÓDULOS DEL PROGRAMA FACENET TENSORFLOW PARA LA IDENTIFICACIÓN FACIAL

El sistema de identificación facial descrito en el CAPÍTULO 7 consta de los módulos que se muestran en la siguiente figura:



Módulos del programa de identificación de rostros.

A continuación, se describe brevemente el papel de cada uno de los componentes del programa de identificación facial:

Directorios:

- **Imágenes**: Directorio donde se almacenan los ficheros con imágenes de las que se quiere identificar una cara. Pueden ser imágenes de una persona o de grupos.
- **Test**: Directorio similar al anterior utilizado para probar con un menor nº de ficheros. Se trabaja sobre uno u otro cambiando una variable del programa.
- **20170512-110547**: directorio que contiene el modelo de Facenet.
- **Align**: Este directorio contiene la librería de detección de caras.

Ficheros:

- **Licenses.txt:** Fichero de licencia de las fotos utilizadas por Arun Mandal en sus ejemplos [Manda, 2017].
- **Get_zipped_facenet_model.sh:** Script utilizado por el fichero Makefile para bajarse el modelo de facenet.
- **Face_recog_tf_V6.py**. Módulo principal del programa de identificación de rostros.
- **Facenet.py** Implementación de la CNN para la identificación de rostros.

APÉNDICE D

EXTENDER PYTHON CON C++.

Para poder usar el programa de detección facial basado en el algoritmo de Viola-Jones objeto de esta tesis doctoral, escrito en lenguaje de programación C++, sobre el programa de identificación facial escrito en lenguaje de programación Python, es necesario llamar a funciones C++ desde Python. Este hecho, que a primera vista parece sencillo se complica cuando las funciones llamadas utilizan objetos de programación complejos como pueden ser una imagen o una vector de estructuras. En este caso, el tratamiento de cada objeto en cada lenguaje es diferente y por tanto requiere de una transformación adecuada que permita a estos objetos ser entendido por ambos lenguajes. En este sentido, a continuación se describe el procedimiento que se ha seguido durante el desarrollo de la tesis doctoral para poder utilizar código C++ en Python, a través de la API Python/C.

Es necesario crear una librería dinámica ".so" que contenga la función en C++ que se va a llamar desde Python. Para ello es necesario modificar el código fuente del programa de detección facial. En este caso se crea un librería llamada: "caras.so", que contiene la función para detección de rostros desarrollada en C++ que se denomina: "detectorcaras". Para que pueda ser llamada desde Python esta función se debe redefinir para que funcione como un método de Python. En este sentido, se modifica la función siguiendo la indicaciones de la API Python/C, para ello, se cambian los argumentos de entrada y de salida de la función a partir de objetos Python definidos en el fichero de cabecera "python.h", que se debe incluir en el código fuente del programa.

En este contexto, la función: "detectorcaras" se redefine del siguiente modo:

static PyObject* detectorcaras(PyObject* self, PyObject* args)

La función recibe como argumento de entrada un objeto Python, (en este caso una imagen) y devuelve también un objeto Python (que será un vector de estructuras).

Para transformar un vector tipo: "std::vector" de C++ a un objeto Python es necesario crear una función que transforme los elementos del vector en elementos de una lista de Python. La función que realiza esta operación es la siguiente:

```
PyObject* vectorToList_Rect(vector<MyRect> data) {
PyObject *dict = NULL;
PyObject *list= Py_BuildValue("[]");
int i = 0;
int contador=0;
for (i; i < data.size(); i++) {
      dict = Py_BuildValue("(i,i,i,i)", data[i].x,data[i].y,data[i].width,data[i].height);
      PyList_Append(list, dict);
   }
return list;
}
```

Finalmente, para definir el método: "detectorcaras", que se llamará desde Python es necesario hacer el "mapeo" del método. Para ello se añade al código fuente de programa de detección facial lo siguiente (el código esta desarrollado para python3.5, para Python 2.7 se requiere otra codificación):

```
static PyMethodDef caras_methods[] =
{
  {"detectorcaras", detectorcaras, METH_VARARGS, "En imagen detecta las caras sobre ella"},
  {NULL, NULL, 0, NULL} //Siempre añadir esta línea después de los métodos definidos
};

// Definimos el modulo, para ello definimos la estructura
static struct PyModuleDef caras_definition = {
```

```
 PyModuleDef_HEAD_INIT,
    "caras",
    "A Python module que busca rostros",
    -1,
    caras_methods
};

// Inicializar el modulo
PyMODINIT_FUNC PyInit_caras(void) {
    Py_Initialize();
    return PyModule_Create(&caras_definition);
}
```

Con esto ya se puede crear la librería: “caras.so”, compilando la función a partir del siguiente Makefile:

```
CFLAGS = `pkg-config --cflags opencv`
LIBS =  `pkg-config --libs opencv`
GCC ?=  mcxx  -ompss -Wall -fPIC -I/usr/local/include -I/usr/include/python3.5
HEADERS := image.h haar.h
all: build
build: caras.so
image.o: image.c $(HEADERS)
$(GCC) -o $@ -c $<
caras.o: caras.cpp $(HEADERS)
$(GCC) -o $@ -c $<
haar.o: haar.cpp $( HEADERS)
$(GCC) -o $@ -c $<
rectangles.o: rectangles.cpp $(HEADERS)
$(GCC) -o $@ -c $<
caras.so: caras.o haar.o image.o  rectangles.o
$(GCC) -shared -o $@ $+ $(LDFLAGS) $(CFLAGS) $(LIBS) -L/usr/lib
#$(GCC) -shared -o caras.so $+ -L/usr/lib -lboost_python -
L/usr/include/python3.5/config
```

Una vez compilada la librería, ya se puede llamar desde Python como sigue:

```
#from caras import detectorcaras
#detectorcaras(imagen)
```

Donde el argumento es la imagen sobre la que se van a localizar los rostros que pueda contener.