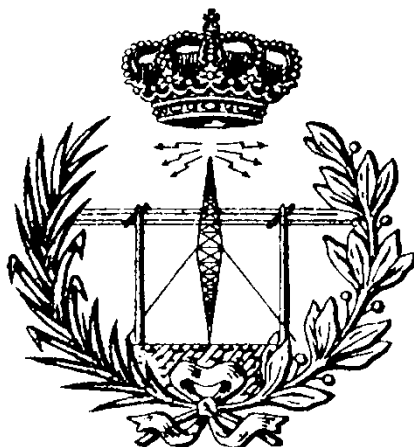


ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA



TRABAJO FIN DE GRADO

“Procesamiento de contenido visual utilizando la placa Parallella”

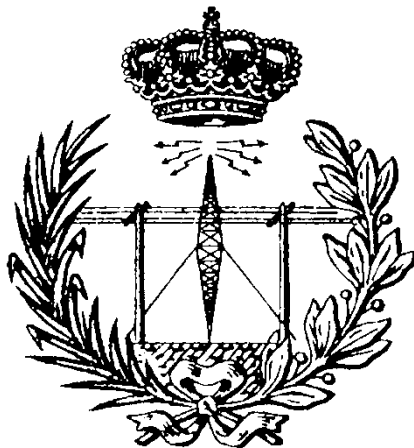
Titulación: Grado en Ingeniería en Tecnologías de la
Telecomunicación

Autor: D. Samuel Rodríguez Rodríguez

Tutores: Dr. Félix B. Tobajas Guerrero
Dr. Valentín De Armas Sosa

Fecha: Junio de 2016

ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA



TRABAJO FIN DE GRADO

**“Procesamiento de contenido visual utilizando la
placa Parallella”**

HOJA DE FIRMAS

Alumno

Fdo.: D. Samuel Rodríguez Rodríguez

Tutor/a

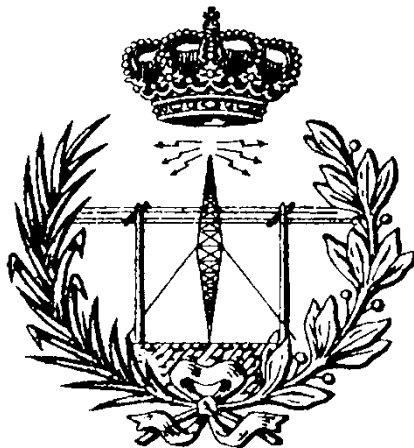
Tutor/a

Fdo.: Dr. Félix B. Tobajas Guerrero

Fdo.: Dr. Valentín De Armas Sosa

Fecha: Junio de 2016

ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA



TRABAJO FIN DE GRADO

**“Procesamiento de contenido visual utilizando la
placa Parallella”**

HOJA DE EVALUACIÓN

Calificación: _____

Presidente

Fdo.:

Vocal

Secretario/a

Fdo.:

Fdo.:

Fecha: Junio de 2016

AGRADECIMIENTOS

A mis padres, porque son ellos los que me han brindado la oportunidad de llegar hasta aquí, siempre me han mostrado su apoyo y, sobre todo, han depositado confianza plena en mí. Todo agradecimiento es poco para lo que me habéis dado.

A mis tutores Félix B. Tobajas Guerrero y Valentín de Armas Sosa también debo agradecerles especialmente su esfuerzo y dedicación. Ellos son los principales responsables de que haya podido sacar adelante este Trabajo Fin de Grado.

RESUMEN

El paralelismo computacional y el procesamiento digital de contenido visual son dos elementos con una presencia actual muy significativa en múltiples áreas de la ciencia, que van de la mano en diversas ocasiones. Es por ello que en este Trabajo Fin de Grado se hace uso de una plataforma *multicore*, como lo es la placa Parallella de *Adapteva*, con el fin de desarrollar una aplicación basada en un algoritmo de procesamiento de contenido visual, todo esto con la ayuda del paralelismo computacional que esta placa ofrece a partir de los recursos que proporciona. Además, también se define la integración del módulo *hardware Raspberry Pi Camera* con la placa Parallella como posible método de captura y obtención del contenido visual.

Para la implementación de la aplicación paralela de procesamiento de contenido visual, se toma como soporte el *framework* OpenCL y la librería OpenCV, detallándose su integración para un correcto funcionamiento sobre la placa Parallella. Así, el algoritmo finalmente desarrollado es un filtro de detección de bordes Sobel, cuya funcionalidad será validada, analizando el factor de aceleración obtenido en su ejecución sobre la placa Parallella en función de diversos parámetros. Del mismo modo, también se analiza la implementación interna y transparente al usuario que la librería OpenCV ofrece, para algunas de sus funciones, de paralelismo a través del *framework* OpenCL.

ABSTRACT

Parallel computing and digital image processing are two techniques that currently are strongly present in many areas of the Science and, in many times, they act together. So that, in this Final Project a multicore platform is used, specifically the *Adapteva's* Parallella board, for developing an application based in a visual content processing algorithm, taking advantage of the parallel computing that this board offers from its resources. In addition, it is also specified the integration of the *Raspberry Pi Camera* hardware module and the Parallella board as a possible method for getting visual content.

In order to develop the parallel application about visual content processing, the OpenCL framework and the OpenCV library are used, explaining their installation on the Parallella board for getting them working properly. The implemented algorithm is a Sobel edge detection filter, whose validity is proved, and it is also analyzed the speed-up achieved in its execution on the Parallella board according to various parameters. In the same way, the internal and transparent implementation that OpenCV offers (in some of its functions) of the OpenCL framework will be analyzed.

ÍNDICE DE CONTENIDOS

Índice de Figuras	V
Índice de Códigos	VII
Índice de Tablas.....	IX
Acrónimos	XI

MEMORIA

Capítulo 1. Introducción	1
1.1 Antecedentes	1
1.2 Objetivos	3
1.3 Petitorio	4
1.4 Estructura del documento	4
Capítulo 2. La placa Parallella	5
2.1 Introducción y características	5
2.2 SoC Zynq.....	9
2.2.1 Sistema de procesamiento (PS)	10
2.2.2 Lógica programable (PL).....	11
2.2.3 SoC Zynq de la placa Parallella.....	11
2.3 Chip Epiphany-III.....	12
2.3.1 Arquitectura	12
2.3.2 Modelo de programación.....	17
2.4 Placa Porcupine	22
Capítulo 3. Integración de la Raspberry Pi Camera Board en la placa Parallella	25
3.1 Raspberry Pi Camera Board	25
3.2 Integración de la placa Raspberry Pi Camera.....	26
3.2.1 Prerrequisitos	26
3.2.2 Generación de la imagen <i>flash</i> de arranque de la placa Parallella	28
3.2.3 Actualización de la memoria <i>flash</i> de la placa Parallella.....	31
3.2.4 Generación del ejecutable con la utilidad de la Raspberry Pi Camera	37
3.2.5 Problemas asociados a la ejecución de la utilidad de la Raspberry Pi Camera	39
Capítulo 4. Integración de la metodología de programación paralela	43
4.1 Epiphany SDK.....	43
4.2 Code::Blocks	44
4.2.1 Creación de un proyecto en Code::Blocks.....	45

4.3	Solución basada en comandos Epiphany	47
4.4	Solución basada en OpenCL	48
4.5	Comparación de las soluciones	49
4.6	OpenCL en Code::Blocks	50
4.6.1	Prerrequisitos	50
4.6.2	Depurador	51
4.6.3	Compilador	54
4.6.4	Inclusión de archivos OpenCL en un proyecto	60
4.6.5	Prueba inicial	63
Capítulo 5.	Desarrollo de la aplicación basada en paralelismo	69
5.1	OpenCV	69
5.1.1	Instalación de OpenCV en la placa Parallella.....	70
5.2	OpenCV en Code::Blocks.....	71
5.3	Filtro de detección de bordes Sobel	73
5.4	Implementación de la aplicación basada en paralelismo	74
5.4.1	Desarrollo del código <i>host</i>	75
5.4.2	Desarrollo del código <i>kernel</i>	80
5.5	Implementación OpenCV – OpenCL en la placa Parallella	84
Capítulo 6.	Pruebas y resultados	89
6.1	Definición del banco de pruebas	89
6.2	Resultados para el procesamiento de imagen QVGA (320x240 píxeles)	95
6.2.1	Resultados principales	96
6.2.2	Resultados para un único fragmento o <i>tile</i>	99
6.3	Resultados para el procesamiento de imagen VGA (640x480 píxeles)	100
6.3.1	Resultados	101
6.4	Resultados para el procesamiento de imagen 4x VGA (1280x960 píxeles)	104
6.4.1	Resultados	105
6.5	Implementación OpenCV – OpenCL en la placa Parallella	107
6.5.1	Resultados	107
6.6	Análisis general de los resultados.....	109
Capítulo 7.	Conclusiones	111
7.1	Conclusiones generales.....	111
7.2	Líneas futuras	113
Referencias.....		115

PLIEGO DE CONDICIONES

Pliego de condiciones	121
-----------------------------	-----

PRESUPUESTO

Presupuesto	125
Recursos humanos	125
Recursos <i>hardware</i>	126
Recursos <i>software</i>	127
Coste total del proyecto	127

ANEXOS

Anexo – Contenido del CD-ROM.....	131
-----------------------------------	-----

ÍNDICE DE FIGURAS

<i>Figura 1.1. Ejemplo de implementación de un programa en paralelo.....</i>	<i>2</i>
<i>Figura 2.1. Placa Parallella – Cara superior.....</i>	<i>8</i>
<i>Figura 2.2. Placa Parallella – Cara inferior.....</i>	<i>8</i>
<i>Figura 2.3. Arquitectura de la placa Parallella</i>	<i>9</i>
<i>Figura 2.4. Diagrama de bloques del SoC Zynq</i>	<i>10</i>
<i>Figura 2.5. Arquitectura Epiphany.....</i>	<i>13</i>
<i>Figura 2.6. Visión de conjunto de la CPU eCore.....</i>	<i>14</i>
<i>Figura 2.7. Mapa de direcciones local de un eCore</i>	<i>15</i>
<i>Figura 2.8. Topología de red eMesh</i>	<i>16</i>
<i>Figura 2.9. Modelo de programación del eSDK</i>	<i>18</i>
<i>Figura 2.10. Diagrama de flujo para la generación de programas Epiphany.....</i>	<i>19</i>
<i>Figura 2.11. Placa Porcupine – Conectores externos.....</i>	<i>22</i>
<i>Figura 2.12. Placa Porcupine – Interfaces con la placa Parallella</i>	<i>22</i>
<i>Figura 3.1. Raspberry Pi Camera Board</i>	<i>25</i>
<i>Figura 3.2. Acceso SSH remoto a la placa Parallella</i>	<i>28</i>
<i>Figura 3.3. Conexión laptop – placa Parallella a través del puerto serie</i>	<i>32</i>
<i>Figura 3.4. Conexión JTAG PC – placa Parallella.....</i>	<i>34</i>
<i>Figura 3.5. Conexión placa Parallella - Raspberry Pi Camera Board.....</i>	<i>37</i>
<i>Figura 3.6. Raspberry Pi Camera Board LED.....</i>	<i>39</i>
<i>Figura 3.7. Testeo de los puertos I2C.....</i>	<i>40</i>
<i>Figura 4.1. Área de inicio del IDE Code::Blocks.....</i>	<i>45</i>
<i>Figura 4.2. Selección de la plantilla predefinida</i>	<i>46</i>
<i>Figura 4.3. Selección del lenguaje de programación</i>	<i>46</i>
<i>Figura 4.4. Definición del nombre y del directorio.....</i>	<i>47</i>
<i>Figura 4.5. Selección del compilador y de las configuraciones a habilitar</i>	<i>47</i>
<i>Figura 4.6. Ejemplo de implementación de un programa paralelo con OpenCL</i>	<i>49</i>
<i>Figura 4.7. Ajustes del depurador GDB/CDB</i>	<i>52</i>
<i>Figura 4.8. Configuración Debug Host</i>	<i>53</i>
<i>Figura 4.9. Configuración BD DebugCL</i>	<i>54</i>
<i>Figura 4.10. Ajustes del compilador GNU GCC</i>	<i>55</i>
<i>Figura 4.11. Linker settings del compilador GNU GCC para OpenCL</i>	<i>55</i>
<i>Figura 4.12. Search directories/Compiler del compilador GNU GCC para OpenCL.....</i>	<i>56</i>

<i>Figura 4.13. Search directories/Linker del compilador GNU GCC para OpenCL</i>	<i>56</i>
<i>Figura 4.14. Toolchain executables del compilador GNU GCC para OpenCL.....</i>	<i>57</i>
<i>Figura 4.15. Toolchain executables del compilador BD OpenCL.....</i>	<i>58</i>
<i>Figura 4.16. Custom variables del compilador BD OpenCL</i>	<i>59</i>
<i>Figura 4.17. Opciones avanzadas del compilador BD OpenCL</i>	<i>60</i>
<i>Figura 4.18. Ajustes del proyecto.....</i>	<i>61</i>
<i>Figura 4.19. Ajustes del target Debug.....</i>	<i>61</i>
<i>Figura 4.20. Ajustes del target CL</i>	<i>62</i>
<i>Figura 4.21. Compilador relativo al target CL.....</i>	<i>62</i>
<i>Figura 4.22. Opciones de generación de archivo OpenCL</i>	<i>63</i>
<i>Figura 4.23. Proyecto “mandelbrot” en Code::Blocks.....</i>	<i>68</i>
<i>Figura 4.24. Conjunto de Mandelbrot.....</i>	<i>68</i>
<i>Figura 5.1. Search directories/Compiler del compilador GNU GCC para OpenCV</i>	<i>72</i>
<i>Figura 5.2. Search directories/Linker del compilador GNU GCC para OpenCV.....</i>	<i>72</i>
<i>Figura 5.3. Linker settings del compilador GNU GCC para OpenCV.....</i>	<i>73</i>
<i>Figura 6.1. Imagen QVGA – Aeropuerto de Gran Canaria</i>	<i>96</i>
<i>Figura 6.2. Imagen QVGA tras la aplicación del filtro Sobel.....</i>	<i>96</i>
<i>Figura 6.3. Imagen VGA – Plaza de Colón</i>	<i>100</i>
<i>Figura 6.4. Imagen VGA tras la aplicación del filtro Sobel</i>	<i>101</i>
<i>Figura 6.5. Imagen 4x VGA – Teatro Pérez Galdós</i>	<i>104</i>
<i>Figura 6.6. Imagen 4x VGA tras la aplicación del filtro Sobel.....</i>	<i>104</i>
<i>Figura 6.7. Imagen VGA tras la conversión a escala de grises</i>	<i>107</i>
<i>Figura 6.8. Imagen 4x VGA tras la conversión a escala de grises</i>	<i>108</i>

ÍNDICE DE CÓDIGOS

Código 2.1. Programación eSDK – Código host	20
Código 2.2. Programación eSDK – Código kernel	21
Código 3.1. Configuración Ethernet por defecto.....	27
Código 3.2. Configuración Ethernet estática	27
Código 3.3. Habilitación de agentes SSH externos.....	28
Código 3.4. Descarga de los repositorios parallella-flash y parallella-uboot	29
Código 3.5. Generación del fichero ejecutable u-boot.elf	29
Código 3.6. Generación del archivo binario BOOT.bin	30
Código 3.7. Generación del archivo binario env.bin	31
Código 3.8. Carga del archivo BOOT.bin en la memoria flash	33
Código 3.9. Carga y almacenamiento del archivo env.bin y otros parámetros de entorno	33
Código 3.10. Regeneración del firmware de la placa Parallella desde el XMD	35
Código 3.11. Instalación del compilador de devicetree y descompilado del devicetree.dtb	36
Código 3.12. Habilitación de la consola serie en la placa Parallella	36
Código 3.13. Compilado del nuevo devicetree.dtb desde el devicetree.dts	36
Código 3.14. Inicio de la nueva configuración de la placa Parallella desde un terminal serie	38
Código 3.15. Generación del ejecutable con la utilidad de la cámara Raspberry Pi	38
Código 3.16. Implementación de capturas con la Raspberry Pi Camera en la placa Parallella	39
Código 4.1. Instalación del IDE Code::Blocks	45
Código 4.2. Instalación de dependencias de paquetes para OpenCL en la placa Parallella	51
Código 4.3. Programa principal del host “maldelbrot”	64
Código 4.4. Kernel “mandelbrot”	67
Código 5.1. Instalación de paquetes obligatorios para instalar correctamente OpenCV	70
Código 5.2. Instalación de paquetes opcionales para instalar correctamente OpenCV	70
Código 5.3. Instalación de la librería OpenCV	71
Código 5.4. Cabeceras del código host.....	75
Código 5.5. Comprobación del número de argumentos pasados al programa	76
Código 5.6. Generación de las variables y las constantes principales del programa	77
Código 5.7. Inicialización de la configuración OpenCL	77
Código 5.8. Lectura de la imagen y cálculo de los argumentos del kernel OpenCL	78
Código 5.9. Preparación de la memoria compartida, de las ventanas para mostrar las imágenes y de la imagen en escala de grises	79

<i>Código 5.10. Ejecución del algoritmo de forma paralela en el chip Epiphany.....</i>	<i>79</i>
<i>Código 5.11. Visualización de las imágenes, del tiempo de ejecución y final del programa</i>	<i>80</i>
<i>Código 5.12. Prototipo de la función correspondiente al kernel OpenCL.....</i>	<i>81</i>
<i>Código 5.13. Inicialización de los parámetros principales del procesamiento en cada core</i>	<i>82</i>
<i>Código 5.14. Inicialización de los parámetros para el tratamiento de cada fragmento o tile</i>	<i>82</i>
<i>Código 5.15. Aplicación del algoritmo del filtro Sobel en el código kernel.....</i>	<i>83</i>
<i>Código 5.16. Evaluación y procesamiento de posibles píxeles fuera de los tiles</i>	<i>84</i>
<i>Código 5.17. Programa secuencial cvtColor</i>	<i>86</i>
<i>Código 5.18. Programa paralelizado cvtColor</i>	<i>87</i>
<i>Código 6.1. Almacenamiento de resultados en un archivo de texto.....</i>	<i>90</i>
<i>Código 6.2. Eliminación de la espera infinita en la muestra de las imágenes.....</i>	<i>90</i>
<i>Código 6.3. script.sh.....</i>	<i>91</i>
<i>Código 6.4. Asignación de permisos de ejecución</i>	<i>95</i>

ÍNDICE DE TABLAS

<i>Tabla 2.1. Modelos de la placa Parallella-16.....</i>	<i>6</i>
<i>Tabla 2.2. Zynq Z-7010 VS Zynq Z-7020</i>	<i>12</i>
<i>Tabla 4.1. Configuración del depurador para el código host</i>	<i>53</i>
<i>Tabla 4.2. Configuración del depurador para el código OpenCL</i>	<i>54</i>
<i>Tabla 4.3. Custom variables para el compilador BD OpenCL</i>	<i>58</i>
<i>Tabla 4.4. Plantilla de Command line macro.....</i>	<i>59</i>
<i>Tabla 5.1. Resoluciones estándar de imágenes.....</i>	<i>81</i>
<i>Tabla 6.1. Resultados temporales del paralelismo computacional para la imagen QVGA (1).....</i>	<i>97</i>
<i>Tabla 6.2. Resultados temporales del paralelismo computacional para la imagen QVGA (2).....</i>	<i>97</i>
<i>Tabla 6.3. Resultados temporales del paralelismo computacional para la imagen QVGA (3).....</i>	<i>97</i>
<i>Tabla 6.4. Resultados temporales de la ejecución total del programa para la imagen QVGA (1) ...</i>	<i>98</i>
<i>Tabla 6.5. Resultados temporales de la ejecución total del programa para la imagen QVGA (2) ...</i>	<i>98</i>
<i>Tabla 6.6. Resultados temporales de la ejecución total del programa para la imagen QVGA (3) ...</i>	<i>98</i>
<i>Tabla 6.7. Resultados temporales del paralelismo computacional para el uso de un solo tile</i>	<i>100</i>
<i>Tabla 6.8. Resultados temporales del paralelismo computacional para la imagen VGA (1)</i>	<i>101</i>
<i>Tabla 6.9. Resultados temporales del paralelismo computacional para la imagen VGA (2)</i>	<i>102</i>
<i>Tabla 6.10. Resultados temporales del paralelismo computacional para la imagen VGA (3)</i>	<i>102</i>
<i>Tabla 6.11. Resultados temporales de la ejecución total del programa para la imagen VGA (1) ..</i>	<i>102</i>
<i>Tabla 6.12. Resultados temporales de la ejecución total del programa para la imagen VGA (2) ..</i>	<i>103</i>
<i>Tabla 6.13. Resultados temporales de la ejecución total del programa para la imagen VGA (3) ..</i>	<i>103</i>
<i>Tabla 6.14. Resultados temporales del paralelismo computacional para la imagen 4x VGA (1) ...</i>	<i>105</i>
<i>Tabla 6.15. Resultados temporales del paralelismo computacional para la imagen 4x VGA (2) ...</i>	<i>105</i>
<i>Tabla 6.16. Resultados temporales del paralelismo computacional para la imagen 4x VGA (3) ...</i>	<i>105</i>
<i>Tabla 6.17. Resultados temporales de la ejecución total del programa para la imagen 4x VGA (1)</i> <i>.....</i>	<i>106</i>
<i>Tabla 6.18. Resultados temporales de la ejecución total del programa para la imagen 4x VGA (2)</i> <i>.....</i>	<i>106</i>
<i>Tabla 6.19. Resultados temporales de la ejecución total del programa para la imagen 4x VGA (3)</i> <i>.....</i>	<i>106</i>
<i>Tabla 6.20. Resultados temporales de la integración OpenCV-OpenCL para la imagen VGA.....</i>	<i>108</i>
<i>Tabla 6.21. Resultados temporales de la integración OpenCV-OpenCL para la imagen 4x VGA ...</i>	<i>109</i>
<i>Tabla PL.1. Condiciones hardware.....</i>	<i>121</i>

<i>Tabla PL.2. Condiciones software</i>	<i>121</i>
<i>Tabla P.1. Coste de recursos hardware (I).....</i>	<i>126</i>
<i>Tabla P.2. Coste de recursos hardware (II).....</i>	<i>126</i>
<i>Tabla P.3. Coste total de recursos hardware.....</i>	<i>126</i>
<i>Tabla P.4. Coste de recursos software</i>	<i>127</i>
<i>Tabla P.5. Coste total</i>	<i>127</i>

ACRÓNIMOS

ADC	<i>Analog-to-Digital Converter</i>
ALU	<i>Arithmetic Logic Unit</i>
AMBA	<i>Advanced Microcontroller Bus Architecture</i>
AMD	<i>Advanced Micro Devices</i>
ANSI	<i>American National Standards Institute</i>
API	<i>Application Programming Interface</i>
APU	<i>Application Processor Unit</i>
ARM	<i>Acorn RISC Machine, Advanced RISC Machine</i>
ASIC	<i>Application-Specific Integrated Circuit</i>
ASSP	<i>Application Specific Standard Product</i>
AXI	<i>Advanced Extensible Interface</i>
Bash	<i>Bourne again shell</i>
CAN	<i>Controller Area Network</i>
CD-ROM	<i>Compact Disc Read-Only Memory</i>
CLB	<i>Configurable Logic Block</i>
COPRTHR	<i>Co-Processing Threads</i>
CPU	<i>Central Processing Unit</i>
DDR3	<i>Double Data Rate type 3</i>
DHCP	<i>Dynamic Host Configuration Protocol</i>
DIP	<i>Digital Image Processing</i>
DMA	<i>Direct Memory Access</i>
DSM	<i>Distributed Shared Memory</i>
DSP	<i>Digital Signal Processor</i>
DVD	<i>Digital Versatile Disc</i>
DVI	<i>Digital Visual Interface</i>
EITE	<i>Escuela de Ingeniería de Telecomunicación y Electrónica</i>
FIFO	<i>First In, First Out</i>
FPGA	<i>Field-Programmable Gate Array</i>

FPS	<i>Frames Per Second</i>
FPU	<i>Floating-Point Unit</i>
GDB	<i>GNU Debugger</i>
GCC	<i>GNU Compiler Collection</i>
GND	<i>Ground</i>
GNU	<i>GNU's Not Unix</i>
GPIO	<i>General-Purpose Input/Output</i>
GPU	<i>Graphics Processing Unit</i>
GUI	<i>Graphical User Interface</i>
HD	<i>High Definition</i>
HDMI	<i>High-Definition Multimedia Interface</i>
I2C	<i>Inter-Integrated Circuit</i>
IALU	<i>Integer ALU</i>
IDC	<i>Insulation-Displacement Contact</i>
IDE	<i>Integrated Development Environment</i>
IGIC	Impuesto General Indirecto Canario
IO	<i>Input/Output</i>
IOP	<i>Input/Output Peripheral</i>
IP	<i>Internet Protocol</i>
ISE	<i>Integrated Synthesis Environment</i>
JTAG	<i>Join Test Action Group</i>
LED	<i>Light-Emitting Diode</i>
LSB	<i>Least Significant Bit</i>
MAC	<i>Media Access Control</i>
MIMD	<i>Multiple Instruction, Multiple Data</i>
MIO	<i>Multiple Input-Output</i>
MP4	<i>Media Player 4</i>
NoC	<i>Network-on-Chip</i>
PC	<i>Personal Computer</i>
PCB	<i>Printed Circuit Board</i>
PCI	<i>Peripheral Component Interconnect</i>

PCIe	<i>PCI Express</i>
PDF	<i>Portable Document Format</i>
PL	<i>Programmable Logic</i>
PMOD	<i>Peripheral Module</i>
PS	<i>Processing System</i>
QSPI	<i>Queued Serial Peripheral Interface</i>
QVGA	<i>Quarter Video Graphics Array</i>
RAM	<i>Random-Access Memory</i>
RGB	<i>Red, Green, Blue</i>
RISC	<i>Reduced Instruction Set Computing</i>
RJ	<i>Registered Jack</i>
Rx	<i>Reception</i>
SCP	<i>Secure Copy</i>
SD	<i>Secure Digital</i>
SDK	<i>Software Development Kit</i>
SDRAM	<i>Synchronous Dynamic Random-Access Memory</i>
SIMD	<i>Single Instruction, Multiple Data</i>
SKU	<i>Stock Keeping Unit</i>
SO	<i>Sistema Operativo</i>
SoC	<i>System-on-Chip</i>
SPDIF	<i>Sony/Philips Digital Interface Format</i>
SPI	<i>Serial Peripheral Interface</i>
SPMD	<i>Single Program, Multiple Data</i>
SSH	<i>Secure Shell</i>
TFG	<i>Trabajo Fin de Grado</i>
TTL	<i>Transistor-Transistor Logic</i>
Tx	<i>Transmission</i>
UART	<i>Universal Asynchronous Receiver-Transmitter</i>
ULPGC	<i>Universidad de Las Palmas de Gran Canaria</i>
USB	<i>Universal Serial Bus</i>
VGA	<i>Video Graphics Array</i>

MEMORIA

Capítulo 1. INTRODUCCIÓN

En este capítulo se detallarán los antecedentes y las necesidades que dan lugar a la realización de este Trabajo Fin de Grado (TFG), así como los objetivos inicialmente planteados y la estructura del presente documento.

1.1 ANTECEDENTES

Con el paso de los años, resulta posible observar cómo se está cumpliendo la Ley de Moore en cuanto a la integración de transistores en un procesador, pero ello no es suficiente, pues la frecuencia de reloj o frecuencia de trabajo de dichos procesadores no se puede aumentar sin afectar el sobrecalentamiento de los dispositivos. Debido a este escenario, surgen las arquitecturas multiprocesador, que tal y como indica su nombre, consisten en múltiples procesadores o *cores* comunicados entre sí a través de *hardware*. Con ello se consigue un procesamiento mucho más efectivo, haciendo uso del paralelismo computacional, donde cada uno de los procesadores puede ejecutar una tarea en concurrencia con los demás [1]. Dicho procesamiento paralelo se implementa con el objetivo de conseguir menores tiempos de cómputo, haciendo uso de varios procesadores que se ocupen del mismo problema [2].

Desde el punto de vista computacional, el procesamiento digital de imágenes (*Digital Image Processing*, DIP), así como la inteligencia artificial y el análisis numérico (todo ello en tiempo real), requieren de una elevada potencia de cómputo, sobre todo en el caso de imágenes cuando se trata la visualización en 3D o en alta definición (HD). Es por ello que la gran mayoría de los dispositivos existentes en el mercado son multiprocesadores [2].

A la hora de implementar un procesamiento paralelo, existen dos soluciones principales: los sistemas SIMD (*Single Instruction, Multiple Data*), donde múltiples procesadores actúan simultáneamente de forma síncrona, pero todos ellos ejecutan la misma instrucción; y los sistemas MIMD (*Multiple Instruction, Multiple Data*), donde múltiples procesadores trabajan en concurrencia de forma asíncrona y pueden ejecutar instrucciones distintas.

Un tipo de sistema SIMD es la GPU (*Graphics Processing Unit*), un coprocesador creado para grandes requerimientos computacionales, operaciones en tiempo real, tratamiento gráfico de forma paralela, ejecución de programas de propósito general, etc. Se trata de un elemento *hardware* programable utilizado actualmente para el procesamiento paralelo y la liberación de carga a las

CPU (*Central Processing Unit*) en las que se ejecutan los programas principales de diferentes aplicaciones [3].

Por otra parte, los sistemas MIMD mejoran la velocidad de cómputo con respecto a los sistemas SIMD, permitiendo una cantidad ingente de operaciones por segundo, ya que cada procesador se encarga de una parte diferente del problema a tratar (diferentes instrucciones). Este tipo de sistemas implementa memoria compartida, por lo que su programación resulta más compleja, necesitando un buen planificador para lograr las prestaciones deseadas [4]. Dentro de estos sistemas MIMD se encuentra la placa Parallella de *Adapteva* [5], la cual será utilizada para la realización de este TFG.

La placa Parallella de *Adapteva* es una plataforma *multicore* de coste no muy elevado que pretende lograr un balance óptimo entre alto rendimiento y bajo consumo de potencia. Esta placa constituye un elemento de computación de alto rendimiento basado en el chip 16-core Epiphany-III de *Adapteva* y un SoC Zynq que integra un procesador ARM Cortex A9 *dual-core*, además de otros elementos básicos que se indican en [5], lo que la hace ideal para la implementación de paralelismo computacional. Este chip es casi 5 veces más rápido que un PC AMD *dual-core*, en experimentos realizados por *Adapteva* para la misma multiplicación paralela de grandes matrices (534 ms VS 2450 ms).

En la Figura 1.1 se muestra un ejemplo de la estructura típica de procesamiento paralelo, en este caso para una FPGA (*Field Programmable Gate Array*) conectada a un *host* externo. En dicho ejemplo, hay un programa principal *host* (en esta ocasión, escrito en lenguaje de programación C/C++) y funciones previamente programadas en los *kernel* (en este caso, una función suma escrita en lenguaje de programación OpenCL).

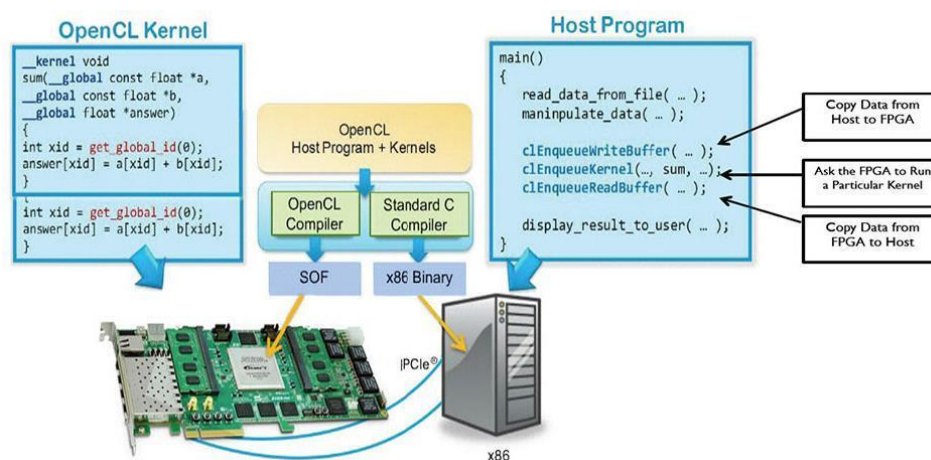


Figura 1.1. Ejemplo de implementación de un programa en paralelo

Con ello, el modo de actuación será el de ir ejecutando el programa principal del *host*, que luego irá solicitando la ejecución de los *kernel hardware* cuando sea necesario a través de funciones específicas proporcionadas por una API (*Application Programming Interface*) que controla la comunicación entre el dispositivo *host* y los *kernel*.

1.2 OBJETIVOS

El objetivo principal que se pretende alcanzar con la realización de este Trabajo Fin de Grado es el de procesar una imagen/secuencia de vídeo aprovechando el paralelismo computacional que ofrece la placa Parallella de *Adapteva*, analizando las prestaciones del procesamiento en función de los recursos de la placa utilizados y del paralelismo alcanzado.

Además, se define la integración del módulo *hardware Raspberry Pi Camera Board* en esta placa como posible método de captura y obtención del contenido visual. Este elemento permitirá obtener imágenes o secuencias de vídeo de alta resolución y es bastante utilizado para aplicaciones de seguridad en la vivienda (vídeo-vigilancia), así como para la grabación de fauna salvaje.

De cara a la implementación del paralelismo computacional en la placa Parallella, inicialmente se plantea el uso del *framework* OpenCL. Asimismo, se pretende desarrollar la correcta integración y utilización de algún módulo de la librería de tratamiento de contenido visual OpenCV en la placa Parallella, de modo que se consiga aplicar un algoritmo de procesamiento de contenido visual haciendo uso, tanto de OpenCL, como de OpenCV.

Los objetivos operativos que se plantean en el desarrollo del presente TFG son los siguientes:

- Realizar la integración del módulo *hardware Raspberry Pi Camera Board* para su correcto funcionamiento en la placa Parallella.
- Procesar de forma paralela una imagen o una secuencia de vídeo haciendo uso del *framework* OpenCL, con el fin de analizar las prestaciones en función del grado de paralelismo o factor de aceleración alcanzado.
- Adaptar la implementación de un algoritmo de procesamiento de contenido visual de la librería OpenCV para su correcto funcionamiento en la placa Parallella.

1.3 PETICIONARIO

Actúa como peticionario del presente Trabajo Fin de Grado la Escuela de Ingeniería de Telecomunicación y Electrónica (EITE) de la Universidad de Las Palmas de Gran Canaria (ULPGC), con el fin de obtener los requerimientos que incluye la asignatura Trabajo Fin de Grado en el plan de estudios del Grado en Ingeniería en Tecnologías de la Telecomunicación.

1.4 ESTRUCTURA DEL DOCUMENTO

El Capítulo 1 hace referencia a los antecedentes del trabajo a realizar, los objetivos del TFG, el solicitante del mismo y la organización del documento. Tras ello, en el Capítulo 2 se presentan las principales características de la placa Parallella, el dispositivo *hardware* entorno al cual girará todo el proyecto, destacando sus elementos y funcionalidades más significativas, así como el modelo de trabajo basado en ella. El Capítulo 3 recoge uno de los objetivos del TFG, ligado a la integración del elemento *hardware Raspberry Pi Camera Board* en la placa Parallella. Posteriormente, el Capítulo 4 introduce las alternativas existentes en cuanto a metodologías que posibilitan la programación paralela en la placa Parallella. En el Capítulo 5 se detalla el desarrollo de varias aplicaciones en la placa Parallella, relativas a diferentes implementaciones de algunos algoritmos de forma paralela. Por otra parte, en el Capítulo 6 se presentan las pruebas llevadas a cabo para la validación de las aplicaciones desarrolladas, así como los resultados obtenidos y un análisis de los mismos. Finalmente, el Capítulo 7 recoge las conclusiones obtenidas tras la realización de este TFG.

Capítulo 2. LA PLACA PARALLELLA

En este capítulo se explica y se define la principal herramienta *hardware* utilizada durante el desarrollo del presente TFG: la placa Parallella [5]. Se tendrán en cuenta sus características, los principales dispositivos que integra, sus utilidades más destacadas, etc. Tras una descripción inicial, se introducirán, con más detalle, los dos dispositivos en los que se fundamenta la arquitectura de la placa Parallella. Además, también se tratará otro elemento *hardware* asociado a esta placa, el cual va a servir de utilidad posteriormente.

2.1 INTRODUCCIÓN Y CARACTERÍSTICAS

La placa Parallella-16 (de la cual existe también el modelo Parallella-64, determinado por la versión del chip Epiphany utilizado) es una plataforma computacional desarrollada por la compañía *Adapteva* y compuesta por un total de 18 *cores*, la cual posee el tamaño aproximado de una tarjeta de crédito (54mm x 87mm). Se trata de un dispositivo que ofrece alto rendimiento y bajo consumo de potencia, pudiendo ser utilizado como un ordenador independiente, como un sistema empujado, o como componente de un *cluster* de servidores en paralelo [5] [6].

El procesador principal o *host* de esta placa es un ARM Cortex A9 *dual-core*, incluido en un SoC Zynq de la empresa *Xilinx* que se encuentra integrado en la placa. El siguiente componente en importancia es un chip Epiphany-III o Epiphany-16, consistente en una matriz escalable de 16 *cores* (procesadores RISC), el cual es usado como coprocesador. Los núcleos independientes que conforman esta matriz están conectados entre sí a través de una NoC (*Network-on-Chip*) integrada internamente en el dispositivo Epiphany, además de una arquitectura DSM (*Distributed Shared Memory*) [5] [6].

En cuanto a los sistemas de almacenamiento, el *host* o CPU de la placa Parallella posee 32KB de memoria caché L1 (nivel 1) en cada núcleo y 512KB de memoria caché compartida L2 (nivel 2). En el chip Epiphany, por su parte, cada *core* incluye 32KB de memoria local de un único acceso por ciclo, divididos en 4 bancos de 8 KB cada uno. Adicionalmente, la placa Parallella contiene 1 GB de memoria SDRAM DDR3, accesible tanto por la CPU ARM, como por el dispositivo Epiphany. Por otra parte, se incluyen 128MB de memoria *flash* tipo NOR (basada en puertas lógicas NOR), donde está almacenado el código que inicia el procedimiento de arranque de la placa, y la cual es accesible a través de una interfaz QSPI [6] [7].

Cabe destacar que, actualmente, existen tres versiones de la placa Parallella-16, las cuales se detallan en la Tabla 2.1, si bien es cierto que la placa utilizada para el desarrollo de este TFG no se corresponde con ninguna de ellas, sino que se trata de la versión *Kickstarter* del modelo *Embedded Platform*.

Tabla 2.1. Modelos de la placa Parallella-16

Modelos	Parallella-16 Micro Server	Parallella-16 Desktop Computer	Parallella-16 Embedded Platform
Usos	Servidor Ethernet sin procesador conectado	Una computadora personal	Sistemas integrados avanzados
Procesador de <i>host</i>	Zynq Z-7010	Zynq Z-7010	Zynq Z-7020
Coprocesador	Epiphany-III	Epiphany-III	Epiphany-III
Memoria	DDR3 de 1 GB	DDR3 de 1 GB	DDR3 de 1 GB
Ethernet	10/100/1000	10/100/1000	10/100/1000
Almacenamiento	Micro SD	Micro SD	Micro SD
USB	No	Sí	Sí
HDMI	No	Sí	Sí
Conectores de expansión	No	2 <i>eLinks</i> + 24 GPIO	2 <i>eLinks</i> + 48 GPIO
FPGA	Sí	Sí	Sí
Peso	36 gramos	38 gramos	38 gramos
SKU	P1600-DK-xx	P1601-DK-xx	P1602-DK-xx

Como sistema operativo, las distribuciones oficiales que proporciona *Adapteva* están basadas en el sistema operativo GNU/Linux, concretamente en la distribución Ubuntu. De estas distribuciones, existen dos versiones: una *headless*, sin interfaz gráfica propia, y otra *hdmi*, la cual permite la conexión de un monitor externo a través de una interfaz HDMI, así como de un teclado y un ratón mediante un puerto USB dedicado, si bien es cierto que este último tipo de distribución no es compatible con el modelo *Micro Server* [8] de la placa Parallella-16.

Además de las características ya citadas, la placa Parallella incluye una serie de periféricos que aportan diferentes funcionalidades adicionales. Entre estos periféricos, los más destacados son los siguientes [7]:

- **Interfaz Gigabit Ethernet.** Presenta soporte de 10/100/1000 Mbps a través de un conector RJ45.

- **Puerto USB 2.0.** Un conector mini USB que permite conectar dispositivos periféricos tales como un teclado, un ratón, una cámara, etc. (No disponible en el modelo *Micro Server* de la placa Parallella-16).
- **Adaptador micro SD.** Una tarjeta micro SD es el principal medio de almacenamiento de la placa Parallella y la fuente fundamental para su arranque. Esta tarjeta contiene el sistema operativo, su configuración, y el sistema de almacenamiento de archivos.
- **Puerto HDMI.** Permite una conexión de alta calidad con monitores DVI/HDMI y televisores a través de un conector micro HDMI. (No disponible en el modelo *Micro Server* de la placa Parallella-16).
- **Indicadores LED.** Un LED verde controlado mediante un pin GPIO del SoC Zynq, un LED rojo controlado por un *flag* en el chip Epiphany a través de uno de sus pines, y dos indicadores LED en el conector RJ45, que indican la presencia o ausencia de actividad y la velocidad de la conexión.
- **Puerto Serie.** Una conexión UART de 3,3V desde el SoC Zynq, utilizando 3 pines dedicados para ello.
- **Suministro de alimentación.** La placa Parallella debe ser alimentada con un suministro de potencia estable de 5V/2A. Ello puede implementarse de dos modos: bien mediante un conector coaxial de continua, o bien mediante un conector micro USB. La selección de un modo u otro se realiza según la posición de un *jumper*.
- **Parallella Expansion Connectors (PEC).** La placa Parallella posee cuatro conectores de expansión situados, tal y como se podrá visualizar posteriormente, en su cara inferior. Estos conectores son: *PEC_POWER* (para señales de potencia y de control), *PEC_FPGA* o *PEC_GPIO* (para la lógica programable del dispositivo Zynq), *PEC_NORTH* (enlace a la mitad norte del chip Epiphany) y *PEC_SOUTH* (enlace a la mitad sur del chip Epiphany). (No disponible en el modelo *Micro Server* de la placa Parallella-16).
- **Puerto I2C.** Una interfaz I2C de 5V disponible en el conector de expansión *PEC_POWER*.
- **Audio.** Es posible reproducir audio a través del conector HDMI, además del pin SPDIF presente en el conector de expansión *PEC_POWER*.
- **Entrada analógica.** El conector de expansión *PEC_POWER* presenta una entrada analógica diferencial, la cual es muestreada y digitalizada por el ADC integrado en el dispositivo Zynq.
- **Conector JTAG.** Es posible programar la FPGA contenida en el SoC Zynq, así como depurar programas ejecutándose en este SoC, a través de los pines JTAG existentes en el conector de expansión *PEC_POWER*.

En la Figura 2.1 y en la Figura 2.2 se muestra la cara superior y la cara inferior de la placa Parallella, respectivamente [7], detallando los principales componentes indicados.

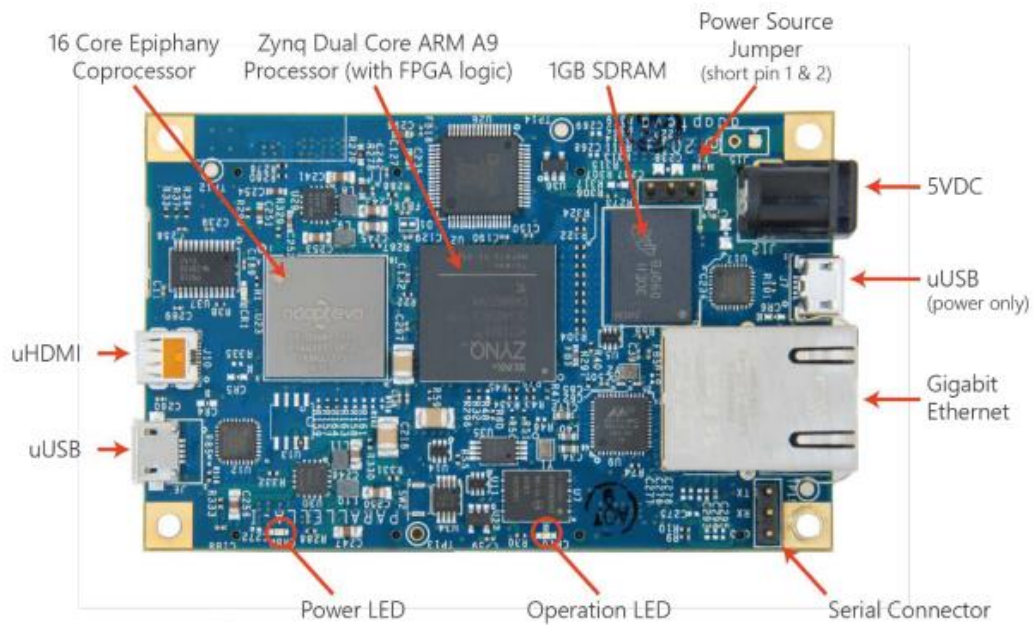


Figura 2.1. Placa Parallella – Cara superior

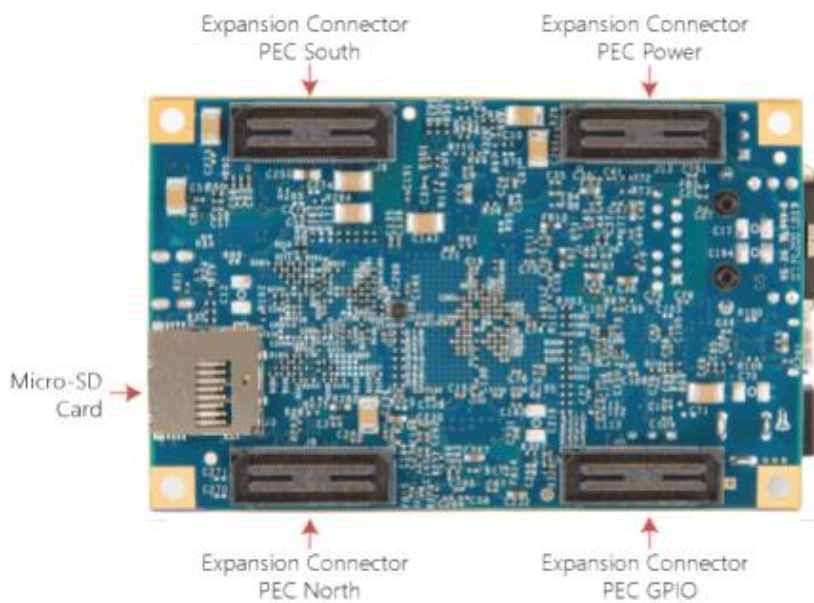


Figura 2.2. Placa Parallella – Cara inferior

Por otro lado, en la Figura 2.3 se muestra la arquitectura general de la placa Parallella desde un nivel alto de abstracción, destacando sus principales elementos.

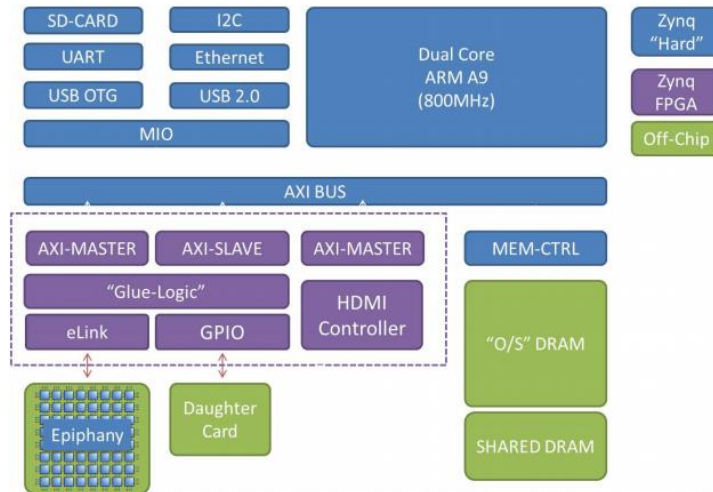


Figura 2.3. Arquitectura de la placa Parallella

Para finalizar, y tras haber desarrollado un estudio general de la placa Parallella, cabe destacar las múltiples aplicaciones para las que este dispositivo ofrece utilidad. Estas actividades incluyen diversas áreas como infraestructura computacional, aplicaciones militares y aéreas, medicina, comunicación y seguridad, instrumentación en la industria, etc. Todas estas áreas y algunas de las aplicaciones específicas que integran, están disponibles en [5].

2.2 SoC ZYNQ

El procesador central de la placa Parallella es el dispositivo Zynq, un SoC desarrollado por la compañía *Xilinx*. Este elemento representa un nuevo tipo de procesador, al integrar y combinar en un único chip las funcionalidades *software* de un Sistema de Procesamiento o PS (*Processing System*) —específicamente, el microprocesador ARM Cortex A9— y las características de la programación *hardware* de la Lógica Programable o PL (*Programmable Logic*) —específicamente, las series 7 de FPGAs de 28nm de *Xilinx*—. Por lo tanto, se consigue integrar las funcionalidades de CPU, DSP, ASSP y señal mixta en un solo dispositivo, lo que hace que el SoC Zynq ofrezca la flexibilidad y escalabilidad de una FPGA, a la vez que el rendimiento, la potencia y la facilidad de uso asociados típicamente a ASICs y ASSPs [7] [9] [10].

El esquemático interno de este dispositivo se muestra en la Figura 2.4, donde se diferencia claramente el Sistema de Procesamiento (PS) y la Lógica Programable (PL). Además, estas dos partes se encuentran en dos dominios de alimentación independientes.

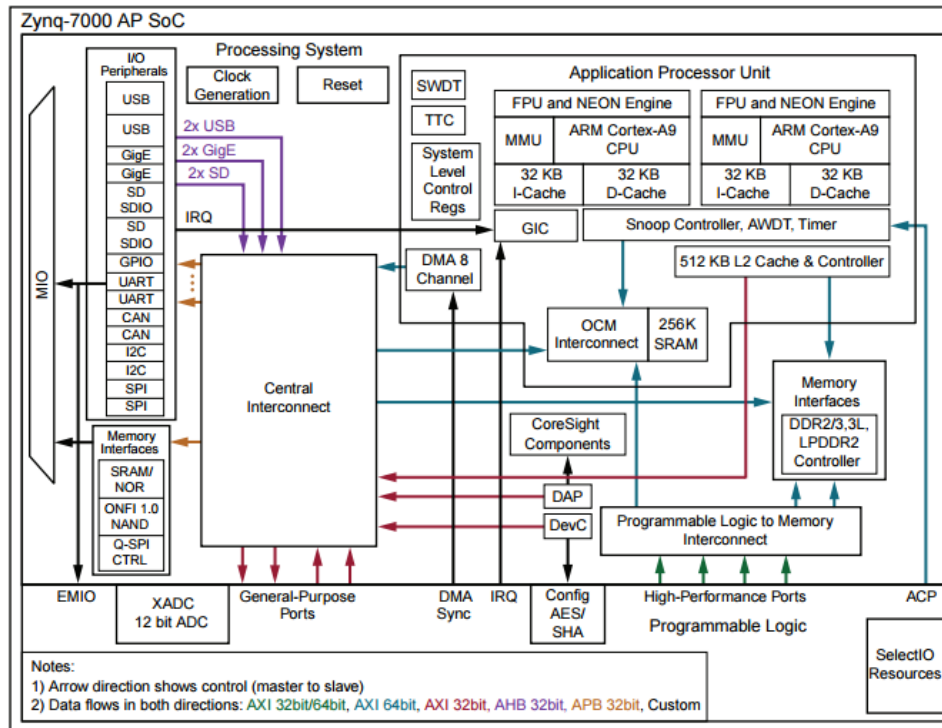


Figura 2.4. Diagrama de bloques del SoC Zynq

Así, los procesadores correspondientes a la parte PS siempre se iniciarán primero, dando lugar a un *software* específico que permite el arranque del sistema PL, así como su configuración. Por otro lado, la parte PL puede configurarse en su propio arranque, o bien ser configurada con posterioridad [9].

2.2.1 SISTEMA DE PROCESAMIENTO (PS)

El Sistema de Procesamiento (PS) se construye alrededor de un procesador ARM Cortex A9 *dual-core*, incluyendo también un conjunto de memorias y controladores. Así, como se puede observar en la Figura 2.4, este sistema consta de cuatro bloques principales: la unidad de procesamiento de aplicación (APU), las interfaces de memoria, los periféricos de entrada/salida (IOP) y el sistema de interconexión [9].

La APU, que incluye al procesador ARM Cortex A9 *dual-core* como CPU, ofrece una amplia gama de prestaciones asociadas a un elevado rendimiento, además de múltiples sub-bloques con capacidades relativas a los diferentes estándares de memoria y de periféricos que soporta.

Por otro lado, en la Figura 2.4 se aprecian dos bloques de interfaces de memoria, los cuales incluyen controladores para diversas tecnologías de acceso y almacenamiento en memoria (uno de los bloques interactúa con la memoria *flash*, mientras el otro hace lo propio con la memoria RAM).

El bloque IOP, por su parte, presenta una amplia variedad de interfaces estándar en la industria para la comunicación de datos con el sistema exterior. Entre estos periféricos se encuentran: GPIOs, controladores Gigabit Ethernet, controladores USB, controladores SD, controladores SPI, controladores UART, controladores CAN, controladores I2C y buffers MIO.

Finalmente, el sistema de interconexión tiene como objetivo implementar la comunicación entre los tres bloques anteriores y la parte de Lógica Programable (PL). Estas interconexiones tienen como fundamento el estándar AMBA de ARM, concretamente, la mayoría de ellas están basadas en las especificaciones del protocolo AXI.

2.2.2 LÓGICA PROGRAMABLE (PL)

La Lógica Programable (PL) ofrece una arquitectura con unas prestaciones bastante favorables para la flexibilidad de configuración por parte del usuario. Para lograr esas capacidades, este sistema incluye: CLBs (*Configurable Logic Block*), memoria RAM de doble puerto, bloques DSP (*Digital Signal Processor*), bloques configurables de entrada/salida, bloques para soporte PCIe, ADCs de 12 bits, una interfaz JTAG disponible para *boundary scan* y transceptores serie [9].

2.2.3 SOC ZYNQ DE LA PLACA PARALLELLA

Para el caso particular de la placa Parallella, se debe tener en cuenta el modo de programación en el procesador relativo al *host*, el cual se incluye en este SoC. Así, es posible hacer uso de todas las herramientas de programación más comunes, incluyendo todo tipo de entornos de desarrollo, compiladores, depuradores, etc. [6].

Para la inclusión del SoC Zynq en la placa Parallella, los modelos desarrollados integran diferentes versiones de éste. Mientras los modelos *Micro Server* y *Desktop Computer* utilizan el dispositivo Zynq Z-7010, el modelo *Embedded Platform* hace uso del dispositivo Zynq Z-7020 [8]. Las principales diferencias entre ambos radican, fundamentalmente, en las características de la FPGA, y pueden observarse en la Tabla 2.2.

Tabla 2.2. Zynq Z-7010 VS Zynq Z-7020

	Z-7010	Z-7020
Programmable Logic Cells	28K	85K
Look-Up Tables	17600	53200
Flip-flops	35200	106400
Extensible Block RAM	240KB	560 KB
Programmable DSP Slices	80	220
Bank-13 IO Pins	No	Yes

2.3 CHIP EPIPHANY-III

El chip Epiphany-III (referencia E16G301) es un SoC MIMD de 16 núcleos o *cores*, implementado con la tecnología de 65nm en la que se basa esta tercera generación Epiphany. Su arquitectura se fundamenta en una construcción *multicore* escalable, de memoria compartida, y de paralelismo computacional. Los 16 núcleos, denominados *eCores*, se disponen en una red bidimensional de 16 nodos, denominada *eMesh*, incluyendo cada uno de ellos una CPU RISC de punto flotante, memoria local, un motor DMA multicanal, y una interfaz de red sin retardo con los nodos adyacentes [11].

2.3.1 ARQUITECTURA

La red bidimensional *eMesh* que conforma el chip Epiphany toma ventaja del área espacial y la abundancia de cables cortos y punto a punto que existe dentro del chip, de modo que es posible realizar transacciones completas (incluyendo direcciones de origen y destino, así como datos de la comunicación) en un único ciclo de reloj. Los diferentes nodos se dividen en una parte interna de procesamiento (en la que se encuentran los *eCores*) y en otra de comunicación con el exterior o *router*, a la que se accede a través de una interfaz de red.

Cada enlace puede transmitir hasta 8 bytes de datos por ciclo de reloj, permitiendo el flujo de 64 bytes de datos a través de cada nodo *router* en cada ciclo de reloj. Por lo tanto, es posible alcanzar un ancho de banda efectivo de 64GB/s con una frecuencia de operación en la red de 1GHz.

En la Figura 2.5 se representa el esquemático de la arquitectura del chip Epiphany, incluyendo la composición de los diferentes nodos que la forman y las interconexiones entre los mismos. Como

se puede ver, la parte de procesamiento de cada nodo incluye: un núcleo CPU RISC (*eCore*), una memoria local, un motor DMA y una interfaz de red con la parte del nodo que actúa como *router*.

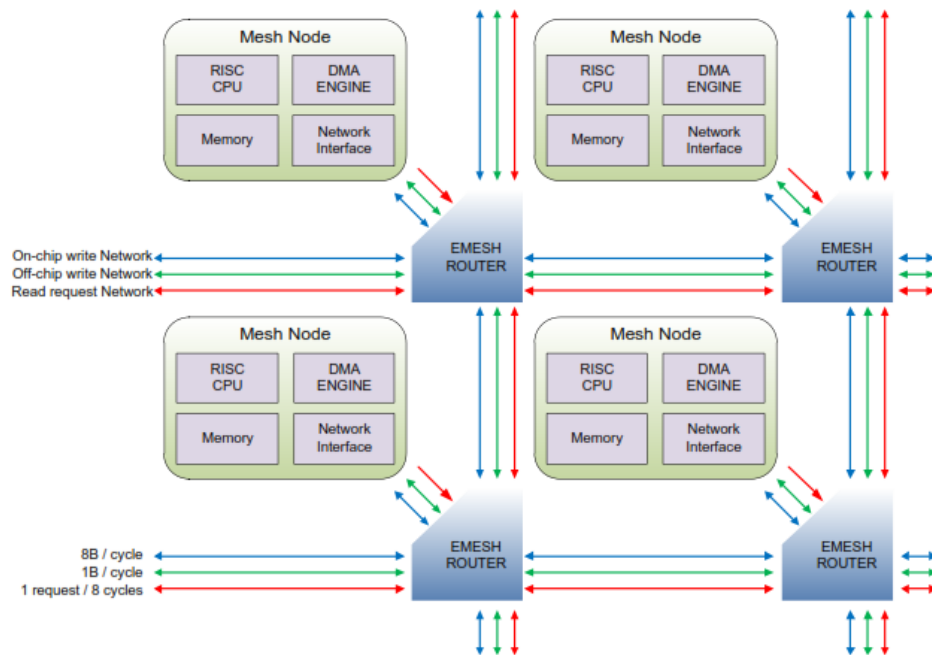


Figura 2.5. Arquitectura Epiphany

2.3.1.1 CPU *eCore*

El corazón de cada nodo procesador es la CPU *eCore*, un microprocesador RISC de punto flotante diseñado específicamente para procesamiento *multicore* y mejorado para lograr un balance favorable entre rendimiento, eficiencia energética y facilidad de uso para múltiples aplicaciones en tiempo real.

En la Figura 2.6 se representa la arquitectura general de la CPU *eCore*. Como se puede observar, los principales componentes incluidos en este procesador son: un secuenciador de programa de propósito general, un largo archivo de registros de propósito general, una ALU de enteros (IALU), una unidad de punto flotante (FPU), una unidad de depuración y un controlador de interrupciones.

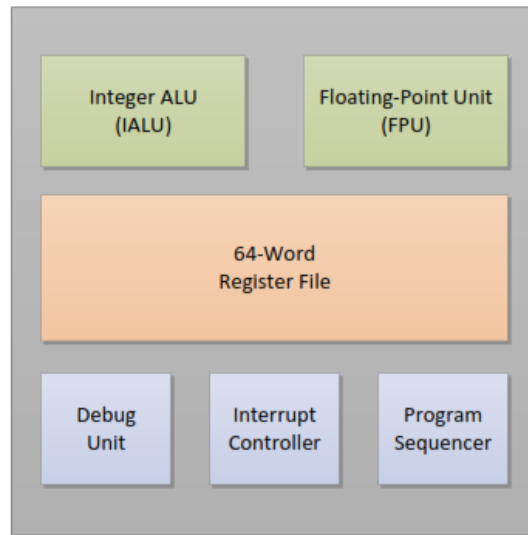


Figura 2.6. Visión de conjunto de la CPU eCore

2.3.1.2 MEMORIA

La arquitectura del dispositivo Epiphany usa un espacio de direcciones único consistente en 2^{32} bytes. Las direcciones de cada byte se tratan como números sin signo, que van desde la dirección 0 hasta la dirección $2^{32}-1$. Este espacio de direcciones también puede considerarse como un conjunto de 2^{30} direcciones para el caso de almacenamiento en palabras o *words* de 32 bits.

Cada eCore de la red tiene un rango de memoria local accesible de 32KB que se mapea dentro del espacio global de direcciones de memoria. Este rango de memoria local comienza en la dirección 0x00000000 y termina en la dirección 0x00007FFF, dividido en cuatro bancos de 8KB. Además, la memoria local se completa con un espacio reservado y determinados registros mapeados en memoria. Por otra parte, los diferentes nodos presentan un identificador de direccionamiento global que permite la comunicación con el resto de nodos que componen el chip Epiphany. Con esto, en la Figura 2.7 se muestra el mapa de direcciones local de cada eCore.

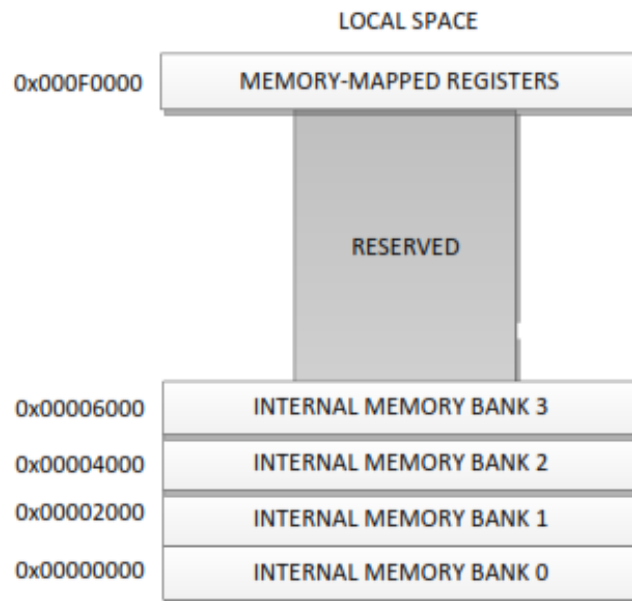


Figura 2.7. Mapa de direcciones local de un eCore

Para concluir este aspecto, cabe destacar que la arquitectura de memoria en el chip Epiphany es *little-endian* y que la misma supone que todos los accesos a memoria estén adecuadamente alineados en límites de media palabra o *half-word*.

2.3.1.3 RED INTERNA

La NoC (*Network-on-Chip*) del dispositivo Epiphany que, como ya se ha comentado, tiene como denominación *eMesh*, presenta una topología de red bidimensional, donde cada nodo implementa conexiones directas solamente con sus nodos adyacentes. Esta red interna consiste en tres estructuras de red separadas y ortogonales, cada una de ellas utilizada para un tipo de tráfico distinto. El esquemático de estas tres redes, las cuales se describen a continuación, se detalla en la Figura 2.8.

- **cMesh**. Usada para las transacciones de escritura destinadas a otro nodo dentro del chip Epiphany.
- **rMesh**. Usada para efectuar todas las peticiones de lectura.
- **xMesh**. Usada para las transacciones de escritura destinadas a recursos fuera del chip y para transacciones destinadas a otro chip en el caso de tratarse de un sistema en configuración *multi-chip*.

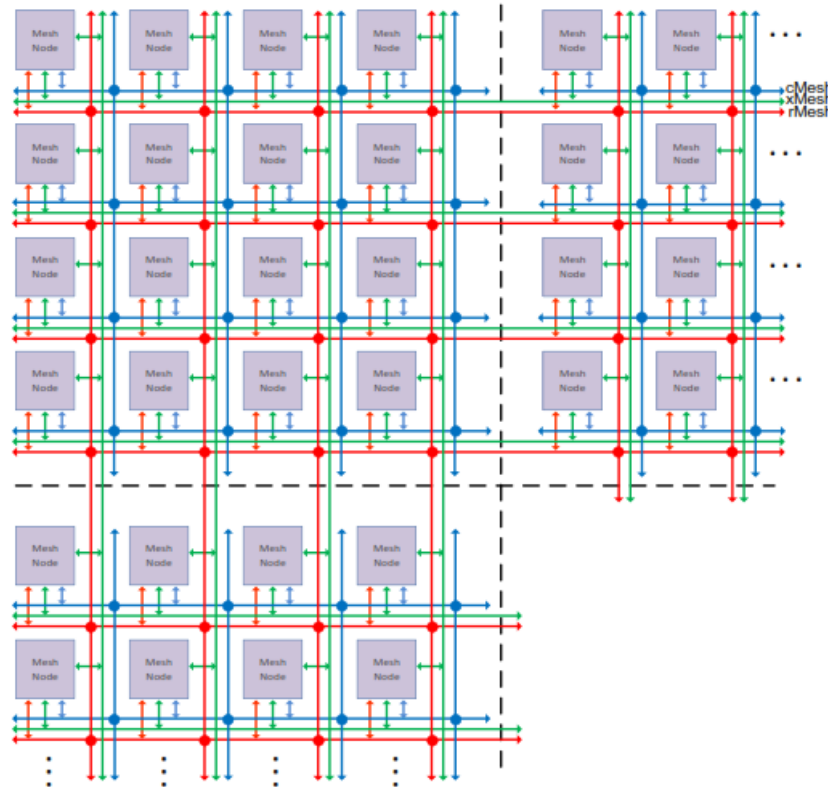


Figura 2.8. Topología de red *eMesh*

La red *cMesh* posee un ancho de banda significativamente más alto y una latencia notablemente más baja que la red *xMesh*, por lo que las tareas con una cantidad significativa de comunicaciones entre sí deben situarse juntas en el mismo chip para lograr un rendimiento óptimo.

Como aspecto destacable de la NoC que presenta el chip Epiphany, se tiene la optimización de las transacciones de escritura sobre las transacciones de lectura. Por otro lado, también existe separación entre el tráfico interno en el chip y el tráfico con el exterior, incluyendo este último un mayor retardo al utilizarse la red *xMesh*.

Otra característica que se incluye es la posibilidad de operaciones libres de puntos muertos, pues el tráfico se encuentra dividido en tres redes distintas en función del propósito de cada comunicación, además de incorporar un esquema de encaminamiento fijo con transacciones de movimiento, en primer lugar, a lo largo de filas y, posteriormente, a lo largo de columnas. En cuanto a la parte de los nodos que actúa como *router*, se lleva a cabo arbitraje del tipo *Round-Robin* en cada una de las tres redes.

Para finalizar, tratando exclusivamente la red *eMesh*, se soportan transacciones a nivel de *byte*, *half-word*, *word* y *double-word*, o lo que es lo mismo, transacciones de tamaño 8, 16, 32 y 64 bits,

respectivamente. Se debe tener en cuenta que los datos de estas transacciones siempre son alineados al bit menos significativo (LSB). Esta red permite la posibilidad de *broadcasting* o difusión eficiente de datos a múltiples *eCores*, todo ello a través de un modo especial de enrutamiento denominado “*multicast*”.

2.3.2 MODELO DE PROGRAMACIÓN

El modelo de programación de la arquitectura Epiphany es neutral y compatible con la mayor parte de modelos de programación de paralelismo más populares. Estos modelos incluyen: SIMD, SPMD, programación maestro-esclavo, MIMD, etc. Las características que permiten este soporte efectivo para la programación paralela son:

- Procesadores de propósito general que soportan la programación a nivel de tarea en ANSI C/C++ en cada nodo. El mapa de memoria compartida minimiza el *overhead* de las interfaces entre las tareas.
- Tecnología distribuida de *routing* que separa las tareas.
- Paso de mensajes entre *cores* sin coste de puesta en marcha.
- Soporte *hardware* integrado para compartir datos de manera eficiente entre múltiples *cores*.

2.3.2.1 ESDK (EPIPHANY SOFTWARE DEVELOPMENT KIT)

El *eSDK*, proporcionado por la compañía *Adapteva*, es el entorno de desarrollo *software* por defecto para la programación en la arquitectura *multicore* que presenta el chip Epiphany [12]. Con este propósito, el *eSDK* incluye: un compilador ANSI C/C++ basado en el compilador GCC, un depurador *multicore* basado en el depurador GDB, librerías de utilidad *hardware* para la programación del código a ejecutar en los *eCores* y otras para la comunicación del *host* con el dispositivo Epiphany, así como un simulador rápido y funcional con capacidad de seguimiento de instrucciones. Además de ello, se sirve de la ayuda de un ensamblador y un *linker* para la generación y la combinación de ficheros objeto.

De entre todas las librerías citadas anteriormente [13], las cuales implican la programación Epiphany, las más significativas son las siguientes:

- **eHAL (Epiphany Hardware Abstraction Layer)** [14]. Incluye las capacidades necesarias para realizar una comunicación efectiva entre el dispositivo *host* y el chip Epiphany, la cual se implementa a través de la memoria compartida existente entre el *host* y los *eCores*.
- **eLib (Epiphany Hardware Utility)** [15]. Ofrece las características que implican la programación interna de los *eCores* que conforman el dispositivo Epiphany, pues existen algunas tareas relativas a la programación que no son posibles mediante el uso de los lenguajes C/C++.

Estas librerías se encuentran dentro del paquete *tools* de la estructura del *eSDK*, cuyos paquetes principales son los siguientes:

- **tools**. Cadena de herramientas GNU y librerías en tiempo real de Epiphany, además de *software* específico del *host*.
- **bsps**. Paquetes de soporte de la placa Parallella, con archivos específicos para la plataforma como *scripts* de enlace por defecto y archivos de definición de la misma.
- **docs**. Documentación del *eSDK*, la cual incluye su arquitectura de referencia.
- **examples**. Algunos ejemplos de código para trabajar con la estructura del *software*.

En la Figura 2.9 se muestra el modelo de programación del *eSDK* proporcionado por *Adapteva*.

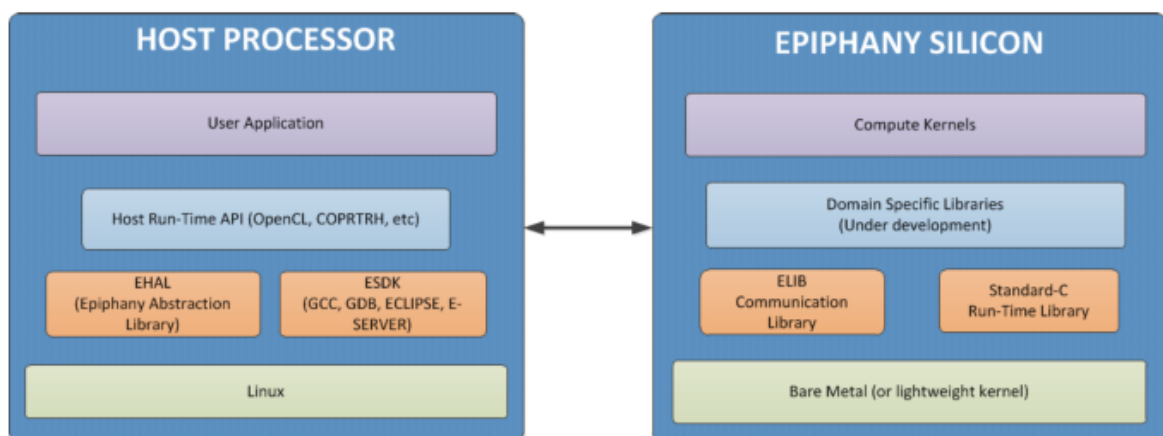


Figura 2.9. Modelo de programación del *eSDK*

Finalmente, cada uno de los nodos procesadores integrados en el chip Epiphany puede ejecutar programas independientes, por lo que es posible configurar y desarrollar proyectos *multicore* con diferentes tareas. El flujo de trabajo para que esto sea posible se refleja en la Figura 2.10.

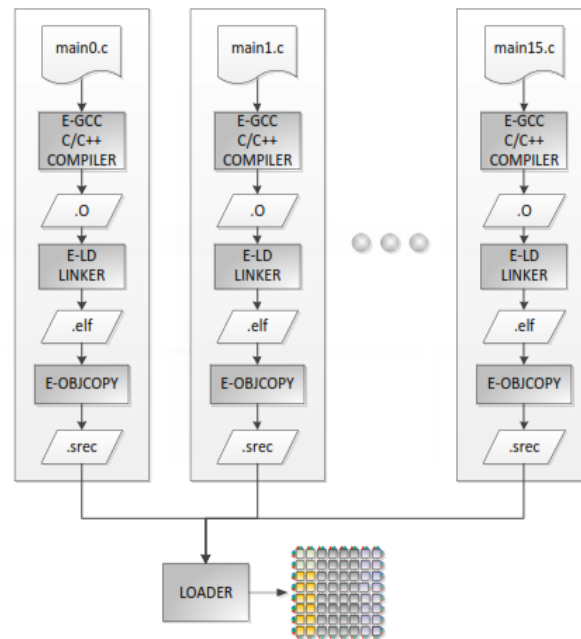


Figura 2.10. Diagrama de flujo para la generación de programas Epiphany

2.3.2.2 EJEMPLO DE PROGRAMACIÓN

Tras haber explicado la metodología de programación por defecto para el chip Epiphany y, por tanto, para la placa Parallella, se visualizará uno de los ejemplos más sencillos de programación que ofrece *Adapteva* en su documentación. Se trata del conocido programa “*Hello World*”, solo que en este caso se implementa en cada uno de los *eCores*. En Código 2.1 se muestra el código correspondiente al dispositivo *host*, mientras en Código 2.2 se indica el código a ejecutar en cada *eCore*. En el primer caso, se observa la utilización de la cabecera `e-hal.h`, así como en el *kernel* se hace uso de la cabecera `e_lib.h`.

```

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#include <e-hal.h>

#define _BufSize    (128)
#define _BufOffset  (0x01000000)
#define _SeqLen     (20)

int main(int argc, char *argv[])
{
    unsigned row, col, coreid, i;
    e_platform_t platform;
    e_epiphany_t dev;
    e_mem_t emem;
    char emsg[_BufSize];
    srand(1);

    // Initialize system from default HDF. Then, reset the platform and get
    // the actual system parameters.
    e_init(NULL);
    e_reset_system();
    e_get_platform_info(&platform);

    // Allocate a buffer in shared external memory for message passing from
    // eCore to host.
    e_alloc(&emem, _BufOffset, _BufSize);

    for (i=0; i<_SeqLen; i++)
    {
        // Draw a random core
        row = rand() % platform.rows;
        col = rand() % platform.cols;
        coreid = (row + platform.row) * 64 + col + platform.col;
        fprintf(stderr, "%3d: Message from eCore 0x%03x (%2d,%2d): ", i,
                coreid, row, col);

        // Open the single-core workgroup and reset the core.
        e_open(&dev, row, col, 1, 1);
        e_reset_group(&dev);

        // Load and launch the device program onto the selected eCore.
        e_load("e_hello_world.srec", &dev, 0, 0, E_TRUE);

        // Wait for core execution, then read message from shared buffer.
        usleep(10000);
        e_read(&emem, 0, 0, 0x0, emsg, _BufSize);

        // Print the message and close the workgroup.
        fprintf(stderr, "\"%s\"\n", emsg);
        e_close(&dev);
    }

    // Release the allocated buffer and finalize the e-platform connection.
    e_free(&emem);
    e_finalize();

    return 0;
}

```

Código 2.1. Programación eSDK – Código host


```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "e_lib.h"

char outbuf[128] SECTION("shared_dram");

int main(void) {
    e_coreid_t coreid;

    // Who am I? Query the CoreID from hardware.
    coreid = e_get_coreid();

    // The PRINTF family of functions do not fit in the internal memory, so
    // we link against the FAST.LDF linker script, where these functions
    // are placed in external memory.
    sprintf(outbuf, "Hello World from core 0x%03x!", coreid);

    return EXIT_SUCCESS;
}

```

Código 2.2. Programación eSDK – Código kernel

En este ejemplo, el código *host*, en primer lugar, inicializa el sistema Epiphany (sentencias `e_init`, `e_reset_system` y `e_get_platform`) y crea un *buffer* de memoria compartida para la obtención de resultados desde los *eCores* (función `e_alloc`). Posteriormente, se entra en un bucle de 20 iteraciones. En cada una de las iteraciones de este bucle, se comienza seleccionando un *eCore* específico al azar (`coreid`, escogido mediante la función `rand`), tras lo cual se abre la comunicación con el chip Epiphany (sentencias `e_open` y `e_reset_group`) y se indica la ejecución del programa *kernel* especificado en el *eCore* seleccionado (función `e_load`). Tras esperar la finalización de la ejecución, se recibe el resultado a través del *buffer* creado anteriormente (sentencia `e_read`), mostrando el mensaje en la consola. Finalmente, se libera el *buffer* utilizado y finaliza la conexión (funciones `e_free` y `e_finalize`, respectivamente).

En cuanto al código relativo al *kernel*, representado en Código 2.2, el *eCore* seleccionado simplemente obtiene su identificador dentro del chip Epiphany (función `e_get_coreid`) y envía un mensaje “Hello World” al *buffer* en memoria compartida. Dentro del mensaje mencionado, cada *eCore* hace referencia a su identificación.

2.4 PLACA PORCUPINE

La placa Porcupine es una “*breakout board*” creada por *Adapteva* para la placa Parallella. Este elemento *hardware* permite acceder a cada uno de los pines existentes en los conectores de expansión de un modo más sencillo. Por lo tanto, no incluye lógica, sino que simplemente se trata de meras conexiones físicas que facilitan la labor del usuario a la hora de acceder individualmente a los pines citados.

En la Figura 2.11 se puede visualizar la placa Porcupine por la cara en la que se encuentran sus conexiones externas, mientras la Figura 2.12 muestra la cara correspondiente a la interfaz con los conectores de expansión de la placa Parallella.



Figura 2.11. Placa Porcupine – Conectores externos

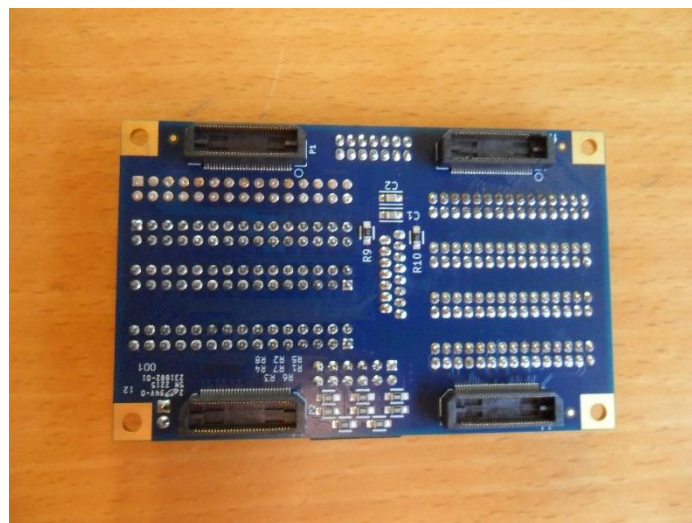


Figura 2.12. Placa Porcupine – Interfaces con la placa Parallella

Esta placa, en su versión 2, incluye las siguientes conexiones [16]:

- Todos los pines individuales de los conectores de expansión *PEC_FPGA* (*PEC_GPIO*) y *PEC_POWER*.
- Conectores IDC para los conectores de expansión *PEC_NORTH* y *PEC_SOUTH*.
- Un conector PMOD para dar soporte a una amplia variedad de periféricos analógicos.
- Un conector JTAG para la programación de la FPGA y para reprogramar la memoria *flash* de la placa, en la cual se encuentra su sistema de arranque.
- Un conector de cable plano dedicado para una *Raspberry Pi Camera*.

Capítulo 3. INTEGRACIÓN DE LA RASPBERRY PI CAMERA BOARD EN LA PLACA PARALLELLA

Una vez introducida la placa Parallella y sus módulos más destacados de los que se va a hacer uso, se procede a detallar en este capítulo el procedimiento a seguir para la consecución de uno de los objetivos inicialmente establecidos en el presente TFG, como es lograr la conectividad del módulo *Raspberry Pi Camera Board*, que constituye un módulo externo con una cámara integrada, preparado para ser integrado en placas *Raspberry Pi*, en la placa Parallella de *Adapteva*. En primer lugar, se presentan los diferentes elementos integrados en la placa *Raspberry Pi Camera* y sus características, para pasar posteriormente a la explicación del procedimiento llevado a cabo para su integración con la placa Parallella.

3.1 RASPBERRY PI CAMERA BOARD

El módulo *Raspberry Pi Camera Board* [17] es el primero de los periféricos comercializados por la *Fundación Raspberry Pi*. Se trata de una cámara provista de un sensor de 5 megapíxeles montada sobre una pequeña PCB y cuyo conexionado es accesible a través de un cable plano, tal y como se puede observar en la Figura 3.1. Esta cámara permite capturar vídeos de alta definición, así como simples fotografías, soportando resoluciones de 1080p (Full HD), 720p (HD) y 480p (VGA), para 30, 60 y 90 fps, respectivamente, en lo que respecta a los diferentes modos de vídeo en lugar de a imágenes individuales. Dicho dispositivo, además, es bastante utilizado en aplicaciones de seguridad en la vivienda (vídeo-vigilancia) y para la grabación de fauna salvaje [17] [18].

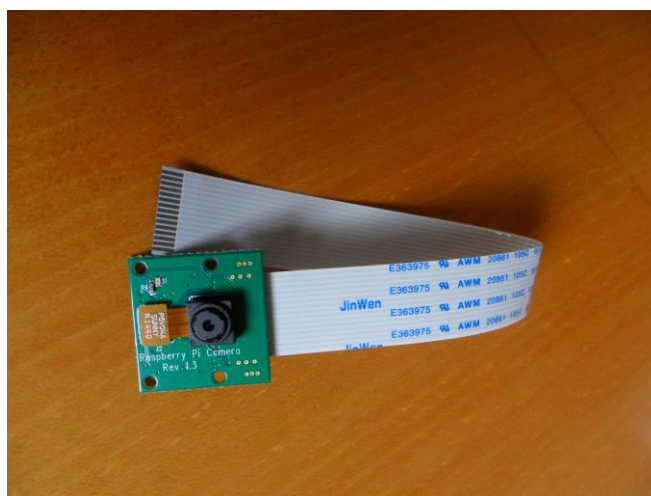


Figura 3.1. Raspberry Pi Camera Board

3.2 INTEGRACIÓN DE LA PLACA RASPBERRY PI CAMERA

En este apartado se detalla, paso a paso, el desarrollo de las diferentes acciones llevadas a cabo de cara a lograr la correcta integración del módulo hardware *Raspberry Pi Camera Board* con la placa Parallella. Cabe destacar que, para los distintos pasos implementados en la placa Parallella, será necesario descargar e instalar, desde los repositorios de Ubuntu, los paquetes que el sistema operativo solicite en cada caso, pudiendo así finalizar las acciones que se estén realizando.

3.2.1 PRERREQUISITOS

Antes de comenzar con el procedimiento desarrollado, es necesario disponer en la tarjeta micro SD que se insertará en la placa Parallella, la versión 2015.1 de la imagen *headless* que proporciona *Adapteva* para la placa Parallella que posee el dispositivo Zynq Z-7020 (la cual se distribuye bajo el sistema operativo Ubuntu 14.04), aunque en el momento de redactar esta memoria ya se encuentra disponible la versión 2016.3 de dicha imagen [19], que usa como sistema operativo Ubuntu 15.04. Si bien es cierto que en la página web de *Adapteva* solamente se encuentra la versión más reciente, en el repositorio *pubuntu* en el *GitHub* de Parallella se distribuyen ambas accediendo a la pestaña “*releases*” [20].

Así, una vez obtenida la versión 2015.1 *headless*, será necesario cargarla en una tarjeta micro SD de 8 o 16 GB, preferiblemente de 16GB. Para ello, se introduce ésta en un ordenador utilizando el adaptador pertinente y se formatea, eliminando además las posibles particiones que contenga y dejándola en estado “de fábrica”. Para realizar esta acción, se ha usado la versión 7.6 de la herramienta *software MiniTool Partition Wizard Home Edition Free* [21], aunque existen muchas otras que ofrecen soluciones para el mismo propósito. De cara a almacenar la imagen en la tarjeta micro SD, se hace uso de la versión 0.9.5 del *software* de código abierto *Win32 Disk Imager* [22], que permite la lectura de todo lo contenido en un dispositivo de almacenamiento externo a un archivo imagen con extensión *.img*, así como el proceso inverso de escribir el contenido de un fichero imagen *.img* a un dispositivo externo de almacenamiento.

Con la imagen 2015.1 *headless* ya en la tarjeta micro SD, será necesario configurar las características del sistema operativo para que sea posible acceder y administrar la placa Parallella remotamente desde un PC o *laptop*. Para ello, se introduce la tarjeta en un PC o *laptop* con una distribución de GNU/Linux (en este caso, Ubuntu 14.04 LTE), ya que otros sistemas operativos reconocen únicamente como disco la primera partición de una unidad externa, mientras que los sistemas

basados en GNU/Linux permiten la posibilidad de operar en todas ellas. Una vez hecho esto, se accede a la partición *rootfs*, donde se realizan los siguientes cambios:

- Para asignar a la placa una dirección IP conocida y que, por lo tanto, se pueda acceder a ella remotamente desde un PC o *laptop*, se modifica el archivo `eth0` que contiene la configuración de la interfaz de mismo nombre, y que se encuentra en el directorio `/etc/network/interfaces.d/`. Así, será necesario modificar la opción `dhcp` por `static` de que se le asigne por defecto una dirección IP dinámicamente mediante un servidor DHCP, la cual se caracteriza por las sentencias indicadas en Código 3.1.

```
auto eth0
iface eth0 inet dhcp
```

Código 3.1. Configuración Ethernet por defecto

Esta característica se sustituirá por una asignación manual, definiendo, en este caso, la dirección IP estática, la máscara de subred y la puerta de enlace predeterminada, tal y como se puede ver en Código 3.2. Otros parámetros opcionales que complementarían la configuración pueden ser la dirección de red y la dirección de difusión o *broadcast*.

```
auto eth0
iface eth0 inet static
address 192.168.0.11
netmask 255.255.255.0
gateway 192.168.0.1
network 192.168.0.0
broadcast 192.168.0.255
```

Código 3.2. Configuración Ethernet estática

En el caso particular del presente TFG, la subred en la que se opera es la 192.168.0.0, con máscara de subred 255.255.255.0. Por tanto, la puerta de enlace predeterminada será la dirección IP del *router* al que se conectará la placa Parallella (192.168.0.1), mientras la dirección IP estática a asignar será una cualquiera que esté dentro del rango de la subred (no se ha escogido una de las direcciones más bajas, sino la dirección 192.168.0.11, para evitar posibles interferencias con otros dispositivos).

- Para permitir que un cliente externo pueda tomar el control de la máquina a través del protocolo SSH, será necesario habilitar la posibilidad de agentes externos en la

configuración SSH de la distribución. Ello se realiza en el archivo `ssh_config` que se encuentra en el directorio `/etc/ssh/`. En este fichero, lo que se precisa hacer es añadir al final del mismo las líneas de código indicadas en Código 3.3.

```
ForwardAgent    yes
ForwardX11      yes
```

Código 3.3. Habilitación de agentes SSH externos

Teniendo ya la tarjeta micro SD con la configuración correcta para la placa Parallella, se inserta dicha tarjeta en la placa y se conecta ésta al *router* de trabajo mediante un cable Ethernet. Tras esto, ya es posible encenderla conectándola a la alimentación. Una vez encendida, existen dos opciones para acceder remotamente a la misma: bien desde una consola *Bash* de Linux, o bien desde una herramienta *software* que gestione un cliente SSH. En ambos casos, el modo de acceso consiste en seleccionar la dirección IP de la máquina con la que se pretende establecer la conexión (en este caso, la placa Parallella), así como el identificador de usuario de la cuenta a la que se desea acceder, tras lo cual se solicitará la contraseña de dicho usuario. En relación a la imagen 2015.1 *headless* de la placa Parallella, se puede ver en la Figura 3.2 tanto el nombre de usuario como la contraseña a utilizar, además del modo de acceso desde la consola *Bash* de Linux.

```
Usuario:      parallella
Contraseña:   parallella

Acceso desde consola: $ ssh parallella@192.168.0.11
                      Password: parallella
```

Figura 3.2. Acceso SSH remoto a la placa Parallella

3.2.2 GENERACIÓN DE LA IMAGEN *FLASH* DE ARRANQUE DE LA PLACA PARALLELLA

Operando ya remotamente sobre la placa Parallella, se debe crear una imagen de arranque a almacenar en la memoria *flash* de la misma, que difiere de la que incluye por defecto la placa Parallella en su estado “de fábrica”. En esta imagen, que se sitúa en memoria *flash*, es donde se encuentra la información con la secuencia de acciones a llevar a cabo para una correcta ejecución del inicio o arranque de la placa.

Para la generación de la nueva imagen a introducir en memoria *flash*, va a resultar necesario tener en cuenta dos de los repositorios en *GitHub* de Parallella, que son: `parallella/parallella-`

`flash` y `parallella/parallella-uboot`. Además, para dar a la imagen *flash* que se va a crear la funcionalidad requerida, también será necesario utilizar algunos archivos que se encuentran en el repositorio `parallella/parallella-examples`, concretamente en el directorio `/rpi-camera/proto/` de dicho repositorio.

Así, en primer lugar, se descargarán los dos repositorios indicados mediante la ejecución de los comandos que se muestran en Código 3.4, en la consola *Bash* de Linux. Para el caso del repositorio `parallella-uboot`, será necesario descargar el correspondiente al “*branch*” o rama `parallella-next`.

```
$ git clone https://github.com/parallella/parallella-flash
$ git clone https://github.com/parallella/parallella-uboot
  --branch parallella-next
```

Código 3.4. Descarga de los repositorios *parallella-flash* y *parallella-uboot*

Tras la descarga, el directorio `parallella-uboot` creado localmente servirá para construir el fichero ejecutable `u-boot.elf`, el cual es necesario para la generación posterior del archivo binario de arranque a almacenar en memoria *flash*. Por tanto, se generará dicho ejecutable con la configuración específica definida para el propósito de este TFG y se almacenará en el directorio `parallella-flash`, que también se ha creado localmente. Todo ello se llevará a cabo mediante los comandos indicados en Código 3.5, en la consola *Bash* de Linux.

```
$ cd path/to/parallella-uboot
$ make adapteva_parallella_defconfig
$ make
$ cp u-boot path/to/parallella-flash/u-boot.elf
```

Código 3.5. Generación del fichero ejecutable *u-boot.elf*

Al realizar esta acción, además, se crea la utilidad `mkenvimage`, que será necesaria más adelante para generar el fichero binario con el entorno de arranque para la nueva configuración de la placa Parallella.

Posteriormente, se reemplazan los archivos `elink2_top_wrapper.bit` y `FSBL.elf` por los dos archivos de mismo nombre que se pueden encontrar en el directorio `/rpi-camera/proto/` del repositorio de *GitHub* `parallella/parallella-examples` y en [23], respectivamente. El primero de los ficheros se corresponde con el *bitstream* creado desde el software *Xilinx Vivado*,

mientras el segundo se trata del ejecutable *First Stage Boot Loader* generado con la herramienta *Xilinx SDK*, respectivamente, ambos para la configuración de la FPGA contenida en el SoC Zynq Z-7020 que contiene la placa Paralella.

Llegados a este punto, se ejecutará el fichero `mkbootflash.sh` que se encuentra en el directorio `parallella-flash`, con el fin de generar el archivo binario `BOOT.bin` con la nueva imagen de arranque de la placa Paralella a partir de los tres archivos mencionados en el procedimiento anterior. Para la correcta ejecución de este archivo, es necesario tener instalada la versión *2015.1* del *Xilinx SDK*, pues se hace uso del comando `bootgen`, cuya ejecución implica la disponibilidad del entorno SDK. *Xilinx SDK* es un entorno de desarrollo para la creación de aplicaciones que se integran en plataformas *Xilinx*, como lo es la Zynq Z-7020 que tiene incorporada la placa Paralella [24].

Por tanto, al tener en el PC o *laptop* un mejor soporte para instalar dicho SDK, se copiará todo el directorio `parallella-flash` desde la placa Paralella al PC o *laptop*, donde se instalará y configurará el entorno *Xilinx SDK 2015.1*. Para copiar el directorio mencionado se hará uso del protocolo SCP, que se basa en el protocolo SSH, disponiendo de dos opciones: bien utilizar la consola *Bash* de Linux, o bien usar una herramienta *software* que gestione un cliente SCP. El modo de operación, en este caso, es muy similar al ya explicado para el acceso remoto usando el protocolo SSH. Tras ello, se ejecutará el fichero `mkbootflash.sh` y, habiendo generado el binario `BOOT.bin`, se almacenará de nuevo el directorio en la placa Paralella. En Código 3.6 se muestran los comandos a ejecutar, según el procedimiento descrito, desde una consola *Bash* de Linux en un PC o *laptop* conectado a la misma red local que la placa Paralella.

```
$ scp -r parallella@192.168.0.11:/path/to/parallella-flash
/path/to/parallella-flash-PC
Password: parallella
$ cd path/to/parallella-flash-PC
$ ./mkbootflash.sh
$ scp -r /path/to/parallella-flash-PC
parallella@192.168.0.11:/path/to/parallella-flash
Password: parallella
```

*Código 3.6. Generación del archivo binario **BOOT.bin***

De vuelta a la placa Paralella, y para concluir con la generación de los archivos necesarios para sobrescribir el sistema de arranque de la misma, se generará el archivo binario con el entorno de arranque `env.bin`, que complementa al `BOOT.bin` generado previamente. El entorno

`env.bin` se crea a partir del archivo `env.txt`, en el cual se definen los parámetros del entorno de arranque que se desea establecer en la placa Parallella. Para realizar esto, tal y como se indicó con antelación, se hará uso de la utilidad `mkenvimage` creada tras el desarrollo del procedimiento relativo al directorio `parallella-uboot`. Sin embargo, para hacer uso de esta utilidad, previamente es necesario exportar el directorio en el que se generó dentro del *path* por defecto de búsqueda de comandos, que en este caso es la carpeta `/tools` del directorio `parallella-uboot`. En Código 3.7 se muestran los comandos en consola *Bash* de Linux en la placa Parallella con los que se consigue realizar esta exportación, así como la generación del archivo binario `env.bin`.

```
$ cd path/to/parallella-uboot/tools
$ export PATH=$PATH:`pwd`
$ cd
$ cd path/to/parallella-flash
$ mkenvimage -s 131072 -o env.bin env.txt
```

Código 3.7. Generación del archivo binario `env.bin`

3.2.3 ACTUALIZACIÓN DE LA MEMORIA *FLASH* DE LA PLACA PARALLELLA

Una vez generados los archivos que contienen la nueva configuración de arranque de la placa Parallella, se está en condiciones de llevar a cabo el procedimiento para sobrescribir la memoria *flash* de la placa con los archivos generados. Este punto del procedimiento es crítico, pues una mala gestión en el mismo puede dar lugar a la pérdida de la configuración de la memoria *flash* o a la corrupción de ciertas partes del *firmware* relativo al dispositivo Zynq, quedando la placa inoperativa o inservible. En caso de que esto suceda, resultaría necesario regenerar dicho *firmware* a su estado “de fábrica”.

Así, para llevar a cabo la actualización de la memoria *flash*, será necesario acceder a ella a través de su puerto serie, conectado a los tres pines situados en la esquina que se encuentra a un lado del adaptador Ethernet RJ45 en la placa Parallella. Con el fin de acceder a este puerto serie desde un PC o *laptop*, será necesario usar un cable convertidor de USB a UART TTL. El conexionado se realizará tal y como se muestra en la Figura 3.3, teniendo en cuenta que la disposición de los tres pines del puerto serie en la placa Parallella es, desde el adaptador Ethernet RJ45 hasta el extremo de la placa: Tx, Rx y GND.

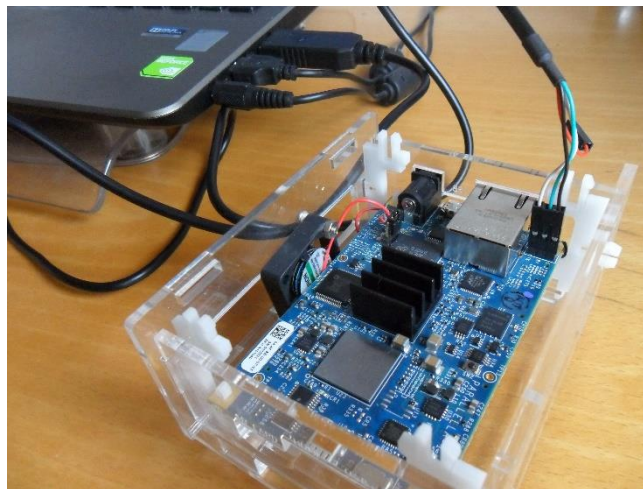


Figura 3.3. Conexión *laptop* – placa Parallella a través del puerto serie

De cara a gestionar la conexión serie abierta desde el ordenador, es posible usar diversas herramientas *software* que permiten ejecutar un terminal serie. En este caso, se ha optado por utilizar la versión 2.7 de herramienta libre *Minicom* desde un terminal Linux, la cual deberá configurarse con los parámetros que por defecto tiene el puerto serie de la placa Parallella para la comunicación. Estos parámetros son los siguientes:

- Baudrate: 115200 bps
 - Data bits: 8
 - Parity: None
 - Stop bits: 1
- } **115200 8N1**

Habiendo definido los parámetros de la comunicación serie, y con el cable adaptador conectado como se comentó anteriormente a la placa Parallella en ausencia de alimentación, se selecciona el dispositivo detectado en el PC o *laptop* (en este caso ha sido el dispositivo serie `/dev/ttyUSB0`) y se abre la comunicación serie.

Previamente a iniciar el contacto con la placa Parallella, se deberán mover los archivos binarios `BOOT.bin` y `env.bin` generados en el punto anterior hasta la partición *boot* de la tarjeta micro SD en la que se encuentra la imagen *2015.1 headless* para la placa Parallella. Ya con todo listo, se conectará la alimentación a la placa Parallella, que deberá encontrarse sin ninguna tarjeta micro SD insertada en la ranura habilitada para ello. La ausencia de tarjeta micro SD provocará que la placa Parallella no consiga finalizar su arranque por defecto y, por lo tanto, se deberá mostrar el siguiente *prompt* a la espera de comandos en la consola del terminal serie del PC o *laptop* donde se ha llevado a cabo la comunicación serie.

```
zynq-uboot>
```

Llegados a este punto, se comenzará a modificar la configuración de arranque de la memoria *flash* QSPI de la placa Parallella. Se inserta la tarjeta micro SD en la ranura correspondiente de la placa Parallella y se comprueba su detección, tras lo cual se cargará el archivo `BOOT.bin`, se borrará dicha memoria *flash* y se escribirá con la configuración del archivo cargado. El modo de actuación para ello en la consola serie es el que se muestra en Código 3.8.

```
zynq-uboot> mmcinfo
zynq-uboot> fatload mmc 0 0x4000000 BOOT.bin
zynq-uboot> sf probe 0 0 0
zynq-uboot> sf erase 0 0x1000000
zynq-uboot> sf write 0x4000000 0 0x$filesize
```

Código 3.8. Carga del archivo *BOOT.bin* en la memoria *flash*

Seguidamente, se volcará el contenido del archivo `env.bin` con los parámetros del entorno de arranque y, finalmente, se añadirá a dichos parámetros la dirección MAC y el SKU de la placa, los cuales están impresos en la etiqueta de referencia de cada placa Parallella, culminando con el almacenamiento de estas variables de entorno. Los comandos para llevar a cabo este proceso son los que se pueden ver en Código 3.9.

```
zynq-uboot> fatload mmc 0 0x4000000 env.bin
zynq-uboot> sf write 0x4000000 0x4e0000 0x$filesize
zynq-uboot> env import -t -d 0x4000000
zynq-uboot> setenv ethaddr 04:4f:8b:00:07:e5
zynq-uboot> setenv AdaptevaSKU SKUA101040
zynq-uboot> saveenv
```

Código 3.9. Carga y almacenamiento del archivo *env.bin* y otros parámetros de entorno

Finalizado este proceso, se apaga la placa Parallella, retirando la alimentación. Si todo ha ido correctamente, la placa estará ahora preparada para su uso.

3.2.3.1 RECUPERACIÓN DE LA PLACA PARALLELLA TRAS UNA PROGRAMACIÓN ERRÓNEA DE LA MEMORIA FLASH

Tal y como se comentó en el apartado anterior, una incorrecta realización del procedimiento descrito para actualizar la memoria *flash* de la placa Parallella podría dar lugar a que ésta quedara en un estado inservible o inoperativo. Si esto pasara, como en el caso del desarrollo de este TFG, se tendrá que reprogramar la placa Parallella a su estado “de fábrica” mediante una conexión JTAG, tanto la memoria *flash* QSPI como el dispositivo Zynq Z-7020. Para ello, será necesario usar la herramienta software *Xilinx SDK* correspondiente al entorno *Xilinx ISE Design 14.4*, además del dispositivo *Xilinx Platform USB Cable II* [25] y de la placa Porcupine de *Adapteva* [16].

Ante esta situación, lo primero que se hará será descargar los archivos `ps7_init.tcl`, `stub.tcl`, `u-boot.elf` y `parallella.7020.flash.bin`, que se encuentran en el repositorio de *GitHub* `parallella/parallella-hw`, específicamente en el directorio `/archive/firmware` [26]. Estos archivos contienen la configuración por defecto, tanto del *firmware* del dispositivo Zynq, como de la memoria *flash*. Los tres primeros archivos se almacenarán en el PC o *laptop*, mientras que el fichero `parallella.7020.flash.bin` deberá almacenarse en la partición *boot* de la tarjeta micro SD a insertar en la placa Parallella.

Para preparar el procedimiento, al igual que en el punto anterior, se conecta el PC o *laptop* al puerto serie de la placa Parallella a través del cable convertidor de USB a UART TTL, y se inicia la conexión serie. En este caso, además, se establecerá la conexión desde el PC o *laptop* a la placa mediante el *Xilinx Platform USB Cable II*, utilizando la placa Porcupine como interfaz y a través del adaptador dedicado que ofrece la misma para dicho cable. En la Figura 3.4 se puede ver una imagen de la conexión realizada.

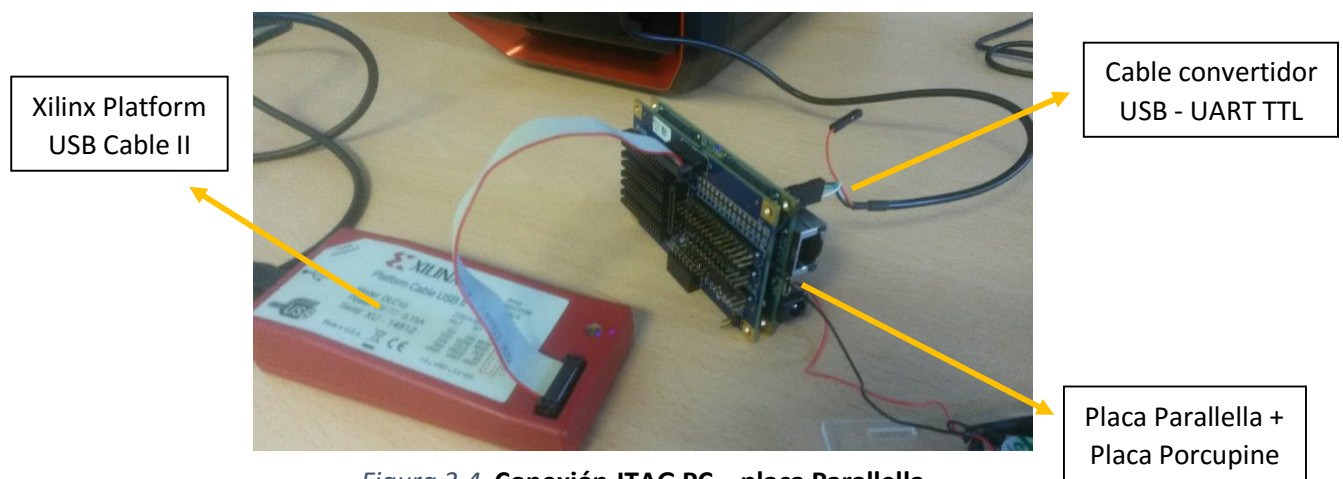


Figura 3.4. Conexión JTAG PC – placa Parallella

Con todo preparado, se conectará la placa Parallella a la alimentación sin tener insertada la tarjeta micro SD y, seguidamente, se ejecutará la aplicación *Xilinx Microprocessor Debugger* (XMD) desde el *software Xilinx SDK* del entorno *Xilinx ISE 14.4*. Con ello, se introduce en el XMD la secuencia de comandos mostrada en Código 3.10, para ejecutar la aplicación de arranque a través de la conexión JTAG [27].

```
XMD% connect arm hw
XMD% source /path/to/ps7_init.tcl
XMD% ps7_init
XMD% init_user
XMD% source /path/to/stuv.tcl
XMD% target 64
XMD% dow /path/to/u-boot.elf
XMD% con
```

Código 3.10. Regeneración del *firmware* de la placa Parallella desde el XMD

Tras esto, se inserta la tarjeta micro SD en la ranura correspondiente de la placa Parallella y se actúa del mismo modo que se describió para actualizar la memoria *flash* de la placa Parallella. En este caso, se cambia el archivo binario `BOOT.bin` por el archivo `parallella.7020.flash.bin` y se obvian las dos líneas de comandos relativas a la inclusión del fichero de entorno `env.bin`.

Con ello, la placa Parallella está lista para su uso por defecto, pero sigue habiendo un problema: las claves relativas al protocolo SSH ya no son útiles y, por lo tanto, no se puede acceder remotamente a la placa Parallella a través de este protocolo. El problema es que para regenerar sus claves SSH, es necesario, en primero lugar, poder acceder a ella. Para ello, se utilizará el cable serial USB a UART TTL que se ha venido usando en diversas ocasiones hasta el momento. La identificación y posterior resolución de este problema representó un esfuerzo considerable y una inversión de tiempo significativa, al no aparecer claramente documentado en ninguno de los repositorios de la compañía *Adapteva*.

Para poder acceder a la placa Parallella y administrarla mediante una conexión serie, será necesario habilitar dicha opción en el archivo que contiene las prestaciones de los periféricos de la placa. Este archivo se denomina `devicetree.dtb` y se encuentra en la partición *boot* de la tarjeta micro SD que se ha venido utilizando. Para modificarlo, se debe introducir la tarjeta micro SD en el PC o *laptop* en el que se esté ejecutando un sistema operativo Linux, donde se instalará el *software* compilador de *devicetree* y se descompilará el citado archivo (tras copiarlo localmente) mediante los comandos indicados en Código 3.11 [28].


```
$ sudo apt-get install device-tree-compiler
$ cd path/to/devicetree
$ dtc -I dtb -O dts -o devicetree.dts devicetree.dtb
```

Código 3.11. Instalación del compilador de *devicetree* y descompilado del *devicetree.dtb*

En el archivo `devicetree.dts` creado, se modifica su contenido añadiendo el fragmento de código que se encuentra subrayado en Código 3.12 [28].

```
chosen {
    bootargs = "console=ttyPS0,115200 root=/dev/mmcblk0p2 rw
               earlyprintk rootfstype=ext4 rootwait";
    linux,stdout-path = "/amba@0/serial@e0001000";
}
```

Código 3.12. Habilitación de la consola serie en la placa Parallella

Una vez modificado y guardado, se vuelve a compilar con el comando que se muestra en Código 3.13, almacenándolo de nuevo en la partición *boot* de la tarjeta micro SD, reemplazando el ya existente [28].

```
$ cd path/to/devicetree
$ dtc -I dts -O dtb -o devicetree.dtb devicetree.dts
```

Código 3.13. Compilado del nuevo *devicetree.dtb* desde el *devicetree.dts*

Tras habilitar la consola serie, se conecta la placa Parallella a la alimentación una vez introducida la tarjeta micro SD en su ranura correspondiente y realizada la conexión serie entre el PC o *laptop* y la placa. De este modo, la placa Parallella se iniciará del modo que lo hace normalmente y en la consola del terminal serie en el PC o *laptop* se verá la consola *Bash* de Linux de la placa Parallella a la espera de órdenes. Finalmente, para regenerar las claves SSH y tener la placa Parallella totalmente operativa, se ejecuta el siguiente comando en dicha consola *Bash* del terminal serie.

```
$ dpkg-reconfigure openssh-server
```


3.2.4 GENERACIÓN DEL EJECUTABLE CON LA UTILIDAD DE LA RASPBERRY PI CAMERA

Con la memoria *flash* QSPI de la placa Parallella finalmente configurada con los archivos binarios generados, se reemplazará el `devicetree.dtb` de la partición *boot* de la tarjeta micro SD por el archivo de mismo nombre que se encuentra en el directorio `/rpi-camera/proto/` del repositorio de *GitHub* `parallella/parallella-examples`. Este nuevo fichero con la configuración de los periféricos de la placa Parallella contiene la definición de un nuevo puerto I2C, denominado *i2c-1*, a través del cual se implementa la conexión con el dispositivo *Raspberry Pi Camera*. En el mismo directorio, también se encuentra el archivo `elink2_top_wrapper.bit.bin`, que deberá añadirse a la partición de la tarjeta micro SD a la que se ha hecho referencia anteriormente.

Previamente a encender la placa Parallella, es necesario conectar la *Raspberry Pi Camera*. Para ello, se usará la placa Porcupine, cuya versión 2 ofrece un conector dedicado para el cable plano a través del cual se establece la conexión con dicha cámara. La Figura 3.5 representa el estado final de la placa Parallella conectada a la placa *Raspberry Pi Camera*, preparada para su inicio.

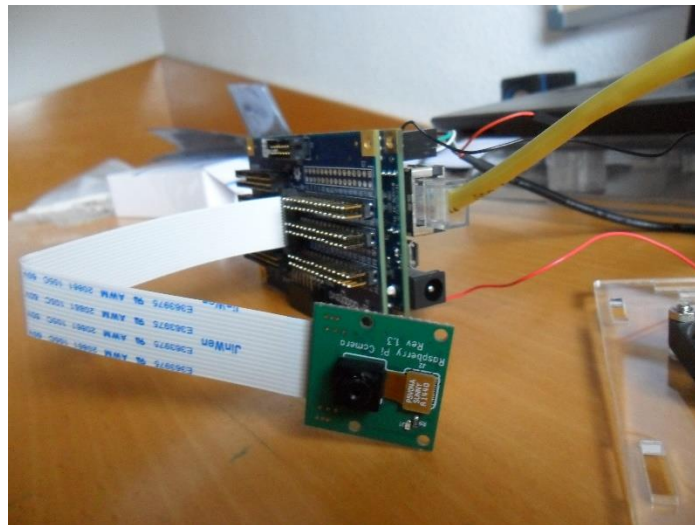


Figura 3.5. Conexión placa Parallella - *Raspberry Pi Camera Board*

Con la nueva configuración de arranque de la placa Parallella, se debe conectar a la alimentación con el puerto serie conectado al PC o *laptop* mediante el cable adaptador de USB a UART TTL. En el terminal serie abierto, se verá el mismo *prompt* que en el procedimiento de programación de la memoria *flash* de la placa, esta vez con el nuevo entorno de arranque que se ha definido. En este caso, con el objetivo de culminar el inicio de la placa Parallella, se insertará por dicho terminal serie las líneas de comandos que se muestran en Código 3.14.

```
zynq-uboot> load mmc 0 0x4000000 elink2_top_wrapper.bit.bin
zynq-uboot> fpga load 0 0x4000000 $filesize
zynq-uboot> load mmc 0 0x3000000 ${kernel_image}
zynq-uboot> load mmc 0 0x2A00000 ${old_devicetree_image}
zynq-uboot> bootm 0x3000000 - 0x2A00000
```

Código 3.14. Inicio de la nueva configuración de la placa Parallella desde un terminal serie

Con ello, la placa Parallella está lista para un uso general y se puede acceder a ella remotamente mediante un cliente SSH. Tras acceder a la placa Parallella desde un PC o *laptop*, se introducirá el directorio `/rpi-camera/proto/rpi_cam/` del repositorio de *GitHub* `parallella/parallella-examples`, bien descargándolo directamente desde la propia placa, o bien descargándolo en un PC o *laptop* y realizando la copia mediante el uso de un cliente SCP. Este directorio contiene los archivos fuente escritos en código C a partir de los que se generará el ejecutable con la utilidad de la cámara *Raspberry Pi*. Una vez se encuentre dicho directorio con sus archivos en la placa Parallella, se accede a él y se crea el ejecutable mencionado mediante los comandos indicados en Código 3.15.

```
$ cd path/to/rpi_cam
$ make
```

Código 3.15. Generación del ejecutable con la utilidad de la cámara Raspberry Pi

Finalmente, se intentará capturar una imagen/vídeo con la cámara Raspberry Pi desde la placa Parallella, para lo cual será necesario el acceso a dos consolas *Bash* de Linux. Por lo tanto, se accederá remotamente a la placa Parallella desde el PC o *laptop* con dos clientes SSH distintos, gestionando así dos consolas diferentes en la placa. En una de las consolas se creará un *buffer* FIFO para almacenar la captura y se iniciará la conversión del vídeo desde ese *buffer* FIFO a un archivo con formato MP4. Después de ello, se comenzará la captura en la otra consola, haciendo uso del ejecutable creado con la utilidad de la cámara y almacenando el contenido de dicha captura en el *buffer* FIFO ya citado. Todo este proceso deberá ejecutarse con privilegios de súper-usuario. A continuación, en Código 3.16 se muestran las líneas de comandos de ambas consolas para llevar a cabo esta acción.

```
$ sudo bash
$ mkfifo /tmp/image.raw
$ avconv -s 1280x960 -f rawvideo -pix_fmt rgba -r 3 -i
  /tmp/image.raw -vf scale=640:480 -g 4 -coder 0 -y
  /tmp/out.mp4
```

```
$ cd path/to/rpi_cam
$ sudo bash
$ ./rpi_cam rgbz32 -o 1280x960 -p rgbz -f /tmp/image.raw
```

Código 3.16. Implementación de capturas con la Raspberry Pi Camera en la placa Parallella

3.2.5 PROBLEMAS ASOCIADOS A LA EJECUCIÓN DE LA UTILIDAD DE LA RASPBERRY PI CAMERA

En el caso concreto del presente TFG, la ejecución de los comandos que permiten iniciar la captura de contenido visual en la placa Parallella utilizando la *Raspberry Pi Camera Board* dan lugar a un error relacionado con la imposibilidad de encontrar el sensor de la cámara: *Unable to find sensor*. En cambio, durante dicha ejecución, la placa Parallella sí que está accediendo a la cámara, pues se enciende el LED rojo que contiene la misma, tal y como se puede observar en la Figura 3.6. Este LED indica actividad en la cámara, si bien no se accede a él través del bus I2C, sino mediante un pin GPIO independiente.

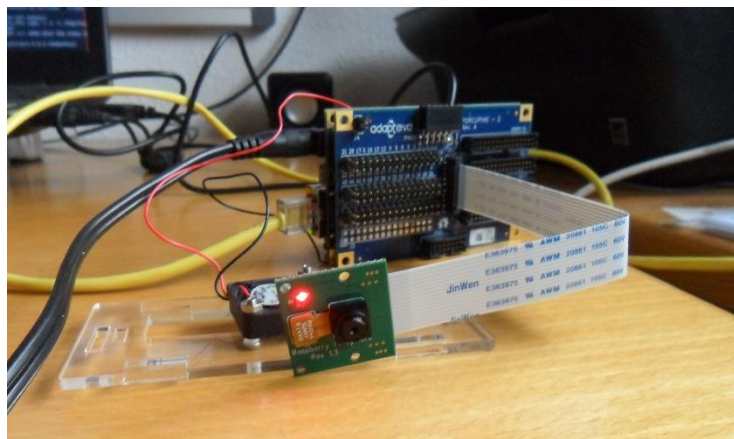


Figura 3.6. Raspberry Pi Camera Board LED

Tras analizar el código fuente en lenguaje de programación C desde el cual se generó el ejecutable con la utilidad de la cámara y el *datasheet* del sensor que utiliza dicha cámara, se introdujeron algunas sentencias de *debugging* en el código C. Estas sentencias realizaban la lectura de las direcciones *0x300A* y *0x300B* del sensor, las cuales contienen identificadores de sólo lectura con

valores por defecto *0x56* y *0x47*, respectivamente, según la hoja de especificaciones del fabricante [29]. Sin embargo, en varias ejecuciones, se comprobó que estos valores no solamente no coincidían con los correctos, sino que los mismos eran distintos en cada una de las ejecuciones y variaban aleatoriamente, sin seguir ningún patrón.

Por tanto, esta situación llevó a testear posibles fallos en el *hardware*. Así, en lo referente a la placa Porcupine, ya se había validado su correcto funcionamiento, al ser usada para facilitar la conexión JTAG entre el PC o *laptop* y la placa Parallella en fases previas del procedimiento descrito en este capítulo. En cuanto a la placa *Raspberry Pi Camera*, por otra parte, se comprobó su validez al conectarla a una plataforma *Raspberry Pi*, cuya conectividad y utilidad está definida e implementada por defecto mediante la descarga y la ejecución de un paquete desde los repositorios de Linux asociados a la *Fundación Raspberry Pi* [30].

Habiendo descartado estos dos componentes como posibles responsables del error obtenido, solamente quedaba por analizar la conexión I2C creada. Para esto, se precisó del uso de las herramientas que proporciona el paquete de Linux *i2c-tools*, en concreto del comando *i2cdetect* [31]. En primer lugar, se detectaron los puertos I2C existentes, encontrando el puerto *i2c-1* que se había definido con la nueva configuración, tal y como se puede ver en la Figura 3.7.

Seguidamente, una vez verificada la existencia del adaptador *i2c-1*, se realizó la lectura de sus direcciones. Esta lectura permite ver el o los dispositivos que se encuentran conectados al adaptador de ese bus, si es que los hubiera. En el caso particular de este TFG, esta lectura ha sido fallida, no detectando ningún dispositivo, como también se muestra en la Figura 3.7.

```

parallella@parallella: ~
parallella@parallella:~$ sudo i2cdetect -l
i2c-0  i2c          Cadence I2C at e0004000          I2C adapter
i2c-1  i2c          Cadence I2C at e0005000          I2C adapter
parallella@parallella:~$ sudo i2cdetect -r 1
WARNING! This program can confuse your I2C bus, cause data loss and worse!
I will probe file /dev/i2c-1 using read byte commands.
I will probe address range 0x03-0x77.
Continue? [Y/n] y
    0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:  -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
10:  -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20:  -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30:  -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40:  -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50:  -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60:  -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70:  -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
parallella@parallella:~$

```

Figura 3.7. Testeo de los puertos I2C

Finalmente, tras encontrar la causa del error indicado a la hora de ejecutar la utilidad de la *Raspberry Pi Camera*, se decidió estudiar con detenimiento el proyecto en Vivado desde el que se generó el *bitstream* o *bitfile* pre-compilado utilizado para el dispositivo Zynq. En el mismo, ha sido posible detectar evidencias de que el problema puede radicar en la compatibilidad del *pinout* definido para la generación del *bitfile* y la versión del *hardware* utilizada, lo que explicaría que no se encontraran dispositivos conectados al adaptador I2C, ya que el mismo podría estar definido en otros pines diferentes del dispositivo Zynq. Esto se debe a que el procedimiento se ha llevado a cabo con un modelo *Kickstarter* de la placa Parallella, y no con el modelo *Embedded Platform* [8] de la misma, ambas basadas en un dispositivo Zynq Z-7020. La placa Parallella *Kickstarter* fue la primera versión en distribuirse tras la finalización del proyecto de creación por parte de *Adapteva*, mientras la placa Parallella *Embedded Platform* fue una de las versiones que comenzó a comercializarse en serie y la cual sigue estando actualmente en el mercado, si bien en el momento de escribir el presente documento no hay *stock* disponible.

Por lo tanto, se ha llegado a un punto en el que las circunstancias impiden culminar el procedimiento propuesto. Sin embargo, sí que se ha conseguido completar con éxito la realización de las diversas acciones y la resolución de los problemas surgidos, de cara a la posibilidad de implementar la conectividad que se pretendía de forma mucho más ágil y únicamente con un cambio en el *hardware* a utilizar. En definitiva, se ha contribuido de manera significativa en traducir un proceso largo en cuanto a los problemas que se han tenido que afrontar, a otro relativamente corto en cuanto a que, gracias al trabajo realizado en este TFG, se conoce de antemano el modo de abordar las diferentes acciones a efectuar, además del modo de afrontar los posibles problemas que puedan surgir en el proceso de integración de la placa *Raspberry Pi Camera* con la placa Parallella *Embedded Platform*.

Capítulo 4. INTEGRACIÓN DE LA METODOLOGÍA DE PROGRAMACIÓN PARALELA

En este capítulo se discutirán las posibles metodologías de programación a utilizar para la implementación del paralelismo computacional en el desarrollo de aplicaciones para la placa Parallella, como alternativa al uso de las librerías de comandos proporcionadas por *Adapteva*. Esto dará lugar a secciones del código que se ejecutarán en el dispositivo *host* o CPU de forma secuencial (en este caso, en uno de los procesadores ARM que se encuentran en el dispositivo Zynq de la placa Parallella), y a otras partes del mismo que se gestionarán de forma paralelizada en cada uno de los *cores* incluidos en el chip Epiphany-III (a modo de aceleración *hardware*). Así, se verá tanto el entorno de desarrollo utilizado, como la metodología de programación que finalmente se ha escogido y las características que han llevado a ello.

Cabe indicar que, desde este momento y durante el desarrollo de los objetivos del TFG relacionados con el procesamiento de contenido visual, se hará uso de la versión 2014.6 de la imagen con *display* que proporciona *Adapteva* para la placa Parallella basada en el dispositivo Zynq Z-7020 (la cual se distribuye bajo el sistema operativo Ubuntu 14.04) [19], que se almacenará en la tarjeta micro SD que se inserta en la placa siguiendo el procedimiento ya descrito en el capítulo anterior. Este tipo de imagen ofrece la posibilidad de tener una GUI del sistema operativo en un monitor externo mediante una conexión HDMI, además de evitar la condición de gestionar la placa Parallella remotamente, pues es posible conectar un teclado y un ratón directamente a través de un conector USB dedicado. Para esta imagen 2014.6 con *display*, el inicio de sesión se realizará con el usuario *linaro*, cuya contraseña es también *linaro*.

4.1 EPIPHANY SDK

Epiphany SDK (*eSDK*) es el entorno de desarrollo *software* que proporciona *Adapteva* y ha sido creado específicamente para la arquitectura *multicore* de Epiphany. El entorno *eSDK* está basado en herramientas de desarrollo estándar, incluyendo un compilador C optimizado, un simulador funcional y un depurador *multicore*, además de varias librerías de utilidad *hardware* [32]. Aunque en la documentación también se define un IDE basado en *Eclipse* como entorno de desarrollo asociado al *eSDK*, éste pronto quedó obsoleto e inestable, por lo que el *eSDK* es, actualmente, un SDK basado en líneas de comandos y *scripts*.

Esta última condición, ligada a la interacción con el entorno *eSDK* basado en líneas de comandos, sin incluir una GUI, conlleva una desventaja para muchos usuarios, debido a la comodidad que implica el trabajo con un entorno gráfico. Además, al tratarse de un entorno diseñado específicamente para la placa Parallella y su chip Epiphany, se dificulta la inclusión directa de otras librerías externas.

Así, al buscar un entorno de desarrollo externo que proporcionase una interfaz gráfica para el desarrollo del programa paralelo, se decidió utilizar *Code::Blocks*. Si bien en un principio se comenzó a trabajar con el IDE *Eclipse*, el mismo fue descartado debido al lastre temporal que suponía su ejecución al tratarse de un entorno bastante más pesado.

4.2 CODE::BLOCKS

Code::Blocks es un entorno de programación multiplataforma y de código abierto orientado, principalmente, al desarrollo de programas en lenguaje C/C++, aunque también admite soporte para otros lenguajes de programación como Fortran y D. *Code::Blocks* está diseñado para ofrecer múltiples extensiones mediante la instalación de *plugins*, así como para ser totalmente configurable por parte del usuario programador [33].

Si bien es cierto que soporta múltiples compiladores, el desarrollo de este TFG se centrará en el soporte del compilador GCC y el depurador GDB de las distribuciones GNU/Linux en todas sus plataformas, pues estos son los que, por defecto, están integrados en la distribución del sistema operativo Ubuntu 14.04 instalado en la placa Parallella. Además, se trata de un *software* ligero y, por lo tanto, bastante rápido [34].

Los tres parámetros más importantes a tener en cuenta a la hora de utilizar un IDE son, desde un punto de vista general, el compilador, el depurador y la interfaz gráfica. En cuanto a los dos primeros de ellos, las prestaciones del compilador en *Code::Blocks* permiten la integración de un sistema de creación de ejecutables muy rápido (sin la necesidad de *makefiles*) y la posibilidad de crear diferentes configuraciones de compilación para los distintos archivos de un mismo proyecto. Por otro lado, su sistema de depuración ofrece un soporte completo de puntos de ruptura o *breakpoints*, junto con varias ventanas para mostrar los diferentes parámetros que se deseen observar en cada momento [33].

En cuanto a la interfaz gráfica que presenta *Code::Blocks*, en la Figura 4.1 se puede observar el área principal de trabajo que se obtiene al iniciar el programa, con las diferentes funcionalidades que se

incluyen a modo de ventanas. En esta vista principal, hay tanto ventanas para mostrar los proyectos del *workspace* que se encuentren abiertos ①, como otras para trabajar con los archivos en los que se desarrolla el código fuente ② y consolas para la recepción de mensajes por parte del *software* ③. Además, se tiene, en la parte superior, múltiples botones y pestañas desplegables que representan accesos directos a muchas de las funcionalidades que integra *Code::Blocks* ④.

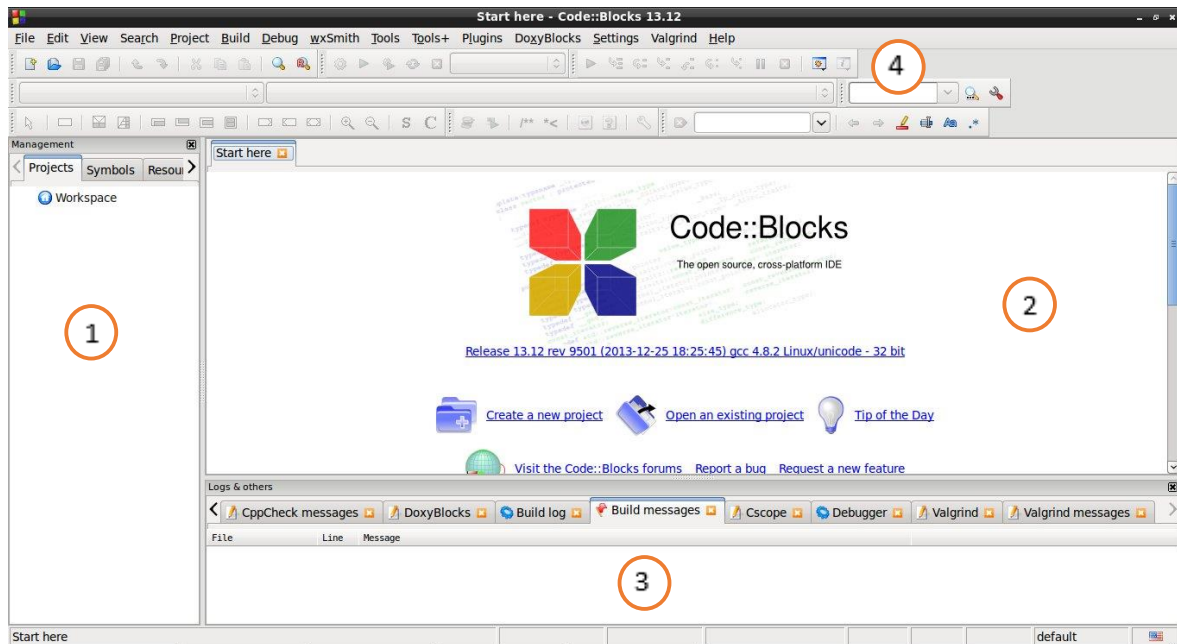


Figura 4.1. Área de inicio del IDE *Code::Blocks*

Para instalar *Code::Blocks* correctamente en la placa Parallella resulta necesario descargar e instalar los paquetes correspondientes desde los repositorios de Linux. Ello se lleva a cabo mediante la ejecución, en la consola *Bash* de Linux, de los comandos que se indican en Código 4.1. Con ello, se instalará la versión más reciente de este *software* que, para el caso particular de este TFG, se trata de la versión 13.12.

```
$ sudo apt-get install codeblocks
$ sudo apt-get install codeblocks-contrib
```

Código 4.1. Instalación del IDE *Code::Blocks*

4.2.1 CREACIÓN DE UN PROYECTO EN CODE::BLOCKS

Para crear un nuevo proyecto en *Code::Blocks* desde su área de inicio, se accede a la pestaña *File* → *New* → *Project...*, lo que dará lugar a una nueva ventana con las plantillas predefinidas para nuevos proyectos. En el caso particular de este TFG, los proyectos a crear se generarán a partir de

la plantilla *Console application*, por lo que se seleccionará esta opción y se hará *click* en el botón *Go*.

Tras esto, es necesario seleccionar ciertos parámetros de la configuración del nuevo proyecto, como es el lenguaje de programación a utilizar, el título y el directorio del proyecto, así como el compilador a utilizar y las configuraciones a habilitar. Al finalizar el procedimiento, se creará un proyecto con un archivo `main.c` o `main.cpp`, en función del lenguaje de programación seleccionado. En la Figura 4.2, en la Figura 4.3, en la Figura 4.4 y en la Figura 4.5, se muestran ayudas visuales de los diferentes pasos citados para la creación de un proyecto en el entorno *Code::Blocks*.

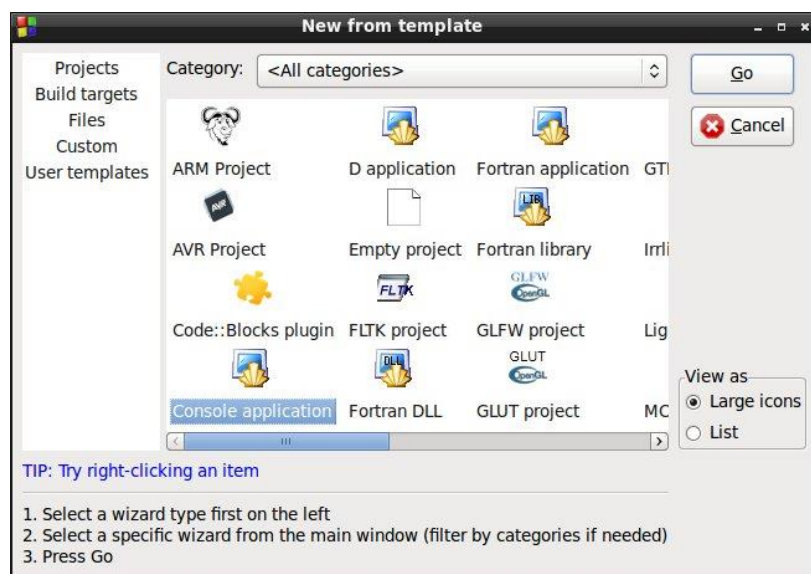


Figura 4.2. Selección de la plantilla predefinida



Figura 4.3. Selección del lenguaje de programación

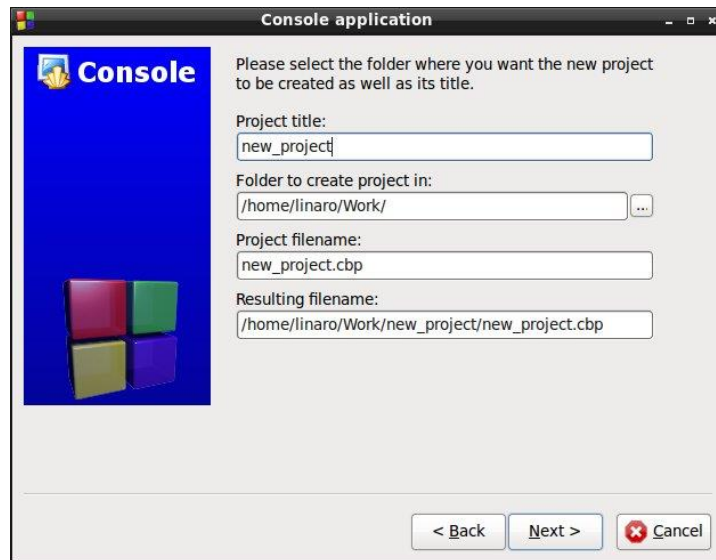


Figura 4.4. Definición del nombre y del directorio

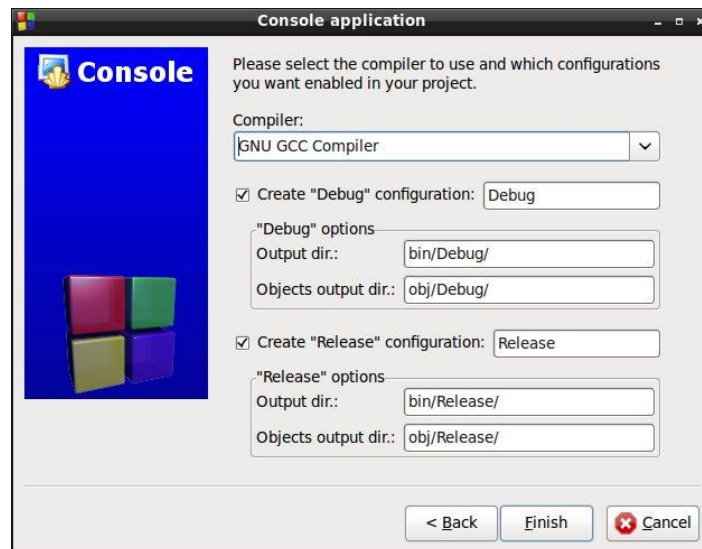


Figura 4.5. Selección del compilador y de las configuraciones a habilitar

4.3 SOLUCIÓN BASADA EN COMANDOS EPIPHANY

A la hora de implementar un programa paralelo, existen múltiples metodologías para afrontar la generación del código. La metodología por defecto que proporciona *Adapteva* en el caso concreto de la placa Parallella es la utilización de los comandos Epiphany. Estos comandos interactúan directamente con el chip Epiphany-III y la memoria compartida existente entre éste y el dispositivo Zynq integrado en la placa Parallella.

Para ello, se necesita hacer uso de las librerías Epiphany (*epiphany-libs*) relativas al *eSDK* [13], concretamente de las librerías *eHAL* (*Epiphany Hardware Abstraction Layer library*) [14] y *eLib*

(*Epiphany Hardware Utility library*) proporcionadas por *Adapteva* [15]. La primera de estas librerías contiene los tipos y las funciones necesarios para la transferencia de datos entre el *host* (dispositivo Zynq) y el chip Epiphany, que se realiza mediante escrituras y lecturas en los *buffers* de memoria compartida, los cuales deben definirse tanto en la aplicación implementada en el *host* como en la que se ejecutará en el chip Epiphany. La librería *eLib*, por su parte, se encarga de las funciones y los parámetros específicos con los que trabajan internamente los diferentes *cores* que componen la arquitectura Epiphany, automatizando algunas tareas de programación que no son proporcionadas por los lenguajes C/C++.

Así, el modelo de programación del código fuente para esta solución será la generación de dos archivos, ambos escritos en lenguaje de programación C/C++. En uno de los archivos se implementará el código a ejecutar en el dispositivo *host* o CPU (incluyendo funciones de la librería *eHAL*), mientras que en el otro se desarrollará el código que se ejecutará en cada uno de los *cores* incluidos en el chip Epiphany (utilizando las prestaciones que ofrecen las funciones de la librería *eLib*). En ambos archivos, se haría uso de notificaciones en memoria compartida para coordinar la comunicación entre los procesos secuencial y paralelo.

4.4 SOLUCIÓN BASADA EN OPENCL

Otra de las posibles soluciones en el momento de implementar el código fuente es el *framework* OpenCL (*Open-source Computing Language*). OpenCL es un estándar de programación abierto y multiplataforma, diseñado por la compañía *Khronos Group* para la programación paralela en sistemas heterogéneos [35]. Según estas características, se podrá escribir un código portable a diversos sistemas, como CPUs, GPUs y otros dispositivos aceleradores de computación. En el presente TFG, como ya se comentó en el punto anterior, se tendrá como elemento acelerador el chip Epiphany que se encuentra integrado en la placa Parallella.

Al igual que en la solución basada en comandos Epiphany, se necesitarán dos archivos: uno con el programa principal del dispositivo *host*, y otro con el código a ejecutar en los *cores* del chip Epiphany (en este último código únicamente se definen funciones específicas o *kernel*). En este caso, la comunicación entre ambos archivos se realizará mediante una serie de parámetros y funciones específicas definidas por una API que proporciona OpenCL y que, en el caso particular de la placa Parallella, se trata de la librería STDCL distribuida por *Brown Deer Technology* en su SDK COPRTHR v1.6 [36]. Es gracias a esta distribución que es posible usar OpenCL en la placa Parallella, aunque en

la documentación no se contemple ningún procedimiento para su integración en un entorno de desarrollo gráfico.

El código correspondiente al dispositivo *host*, escrito en lenguaje de programación C/C++, es el que ejecutará el programa principal, desde el cual se envían los datos hacia la memoria compartida con los *kernel* aceleradores y se solicita su ejecución con la ayuda de la API a la que se ha hecho referencia. El código *kernel*, por su parte, está escrito en lenguaje de programación OpenCL C, una versión extendida del lenguaje C para permitir la programación del paralelismo. Este código *kernel* toma los datos que necesite de la memoria compartida, los manipula según la función implementada y devuelve los resultados al *host* a través de dicha memoria compartida [35]. En la Figura 4.6 se muestra un ejemplo de la implementación de este proceso para el caso de una FPGA conectada a un *host* externo.

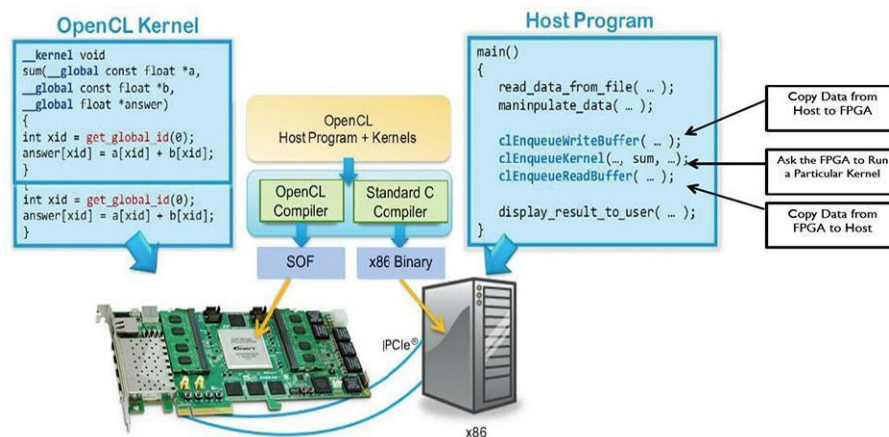


Figura 4.6. Ejemplo de implementación de un programa paralelo con OpenCL

4.5 COMPARACIÓN DE LAS SOLUCIONES

Si bien las características y el modelo de trabajo de las soluciones planteadas ya han sido expuestos en los puntos anteriores, llegado este momento se pretende analizar las ventajas y desventajas que implican cada una de ellas.

Para el caso de la solución basada en comandos Epiphany, según la descripción realizada, se deberá trabajar a un nivel de descripción más bajo que en la solución basada en OpenCL. Ello se debe a que las funciones que ofrece la API STDCL y el lenguaje de programación OpenCL C invocan, desde un mayor nivel de abstracción, las funciones de nivel de descripción más bajo, que en el caso de trabajar con comandos Epiphany, tendrían que implementarse manualmente. Entre esas funciones se encuentran, por ejemplo: la inicialización del chip Epiphany y de todas sus características, la

asignación de *buffers* de memoria compartida en direcciones específicas de la región de memoria dedicada a ello –así como la lectura y escritura en los mismos–, la gestión del identificador de cada *core* del chip Epiphany dentro del grupo de trabajo al que pertenece, la sincronización de la comunicación, etc.

Sin embargo, la solución basada en OpenCL también ofrece ciertos aspectos negativos, pues su puesta a punto en la placa Parallella conlleva un *overhead* temporal significativo en los procesos de comunicación entre el programa principal del *host* y los *kernel* aceleradores. Estos procesos son abordados por algunas de las funciones de la API STDCL, cuya implementación en el caso particular de la placa Parallella no está optimizada.

Así, aun teniendo en cuenta las repercusiones negativas del *overhead* temporal, uno de los objetivos del TFG comprende el uso del *framework* OpenCL en la placa Parallella. Ello se debe al mayor beneficio potencial de la aplicación que suponen las contribuciones que resultan del TFG. Por tanto, se deberá configurar el IDE a utilizar para integrar correctamente las librerías y el compilador que ofrece *Brown Deer Technology* con su SDK COPRTHR.

4.6 OPENCL EN CODE::BLOCKS

Una vez definida OpenCL como metodología a utilizar en el desarrollo de programación paralela en la placa Parallella, será necesario configurar adecuadamente el entorno IDE *Code::Blocks*, en el que se llevará a cabo la integración del código fuente, para soportar la implementación de OpenCL. En este caso, ni la compañía *Adapteva*, ni la compañía *Brown Deer Technology*, ni el mismo *Code::Blocks*, proporcionan información acerca del modo de integrar el compilador para su correcto funcionamiento en un entorno de desarrollo gráfico como lo es *Code::Blocks*. Por lo tanto, para llevar a cabo esta integración de forma eficaz, se ha partido de una propuesta recogida en [37], logrando así desarrollar la configuración del entorno para que se soporte el uso de OpenCL.

4.6.1 PRERREQUISITOS

Como requisitos previos para el soporte de las librerías de OpenCL correspondientes al SDK COPRTHR de la compañía *Brown Deer Technology* en la placa Parallella, existen las siguientes dependencias de paquetes [38]:

- Compilador GCC, versión 4.6 o superior. Instalado por defecto en sistemas GNU/Linux, se obtiene su versión más reciente simplemente actualizando el sistema.

- Paquete `libelf-0.8.13`.
- Paquete `libconfig-1.4.8` o superior.
- Paquete `libevent-2.0.18` o superior.

En Código 4.2 se indican los comandos a ejecutar en la consola *Bash* de Linux con el fin de instalar las dependencias de paquetes mencionadas, cumpliendo así los prerequisites establecidos para la utilización de las librerías OpenCL del SDK COPRTHR de *Brown Deer Technology* en la placa Parallella.

```
$ sudo apt-get update
$ sudo apt-get upgrade

$ wget www.mr511.de/software/libelf-0.8.13.tar.gz
$ tar -zxvf libelf-0.8.13.tar.gz
$ cd libelf-0.8.13
$ ./configure
$ sudo make install
$ cd

$ wget www.hyperrealm.com/libconfig/libconfig-1.4.8.tar.gz
$ tar -zxvf libconfig-1.4.8.tar.gz
$ cd libconfig-1.4.8
$ ./configure
$ sudo make install
$ cd

$ wget github.com/downloads/libevent/libevent/
    libevent-2.0.21-stable.tar.gz
$ tar -zxvf libevent-2.0.21-stable.tar.gz
$ cd libevent-2.0.21-stable
$ ./configure
$ sudo make install
```

Código 4.2. Instalación de dependencias de paquetes para OpenCL en la placa Parallella

4.6.2 DEPURADOR

Con los paquetes requeridos ya instalados, se está en disposición de comenzar con la configuración del entorno *software Code::Blocks*, de cara a una inclusión y un uso adecuados de las librerías OpenCL en el desarrollo de la aplicación paralela. En primer lugar, se establecerá la configuración de los depuradores a utilizar, tanto para el programa correspondiente al dispositivo *host*, como para el código a ejecutar como aceleración *hardware* en el dispositivo Epiphany. Este último no

tendrá un impacto significativo, pues los *kernel* OpenCL se compilan en tiempo de ejecución, lo que dificulta bastante la depuración de su código.

Para comenzar, desde el área de inicio de *Code::Blocks* que se reproduce en la Figura 4.1, se accede a la pestaña *Settings*, lo que hará que se visualice un menú desplegable, en el cual se escogerá la opción *Debugger...*, abriéndose una nueva ventana. Llegados a aquí, se seleccionará la opción *GDB/CDB debugger* en la lista que aparece a la izquierda de la ventana y, con la ayuda del botón *Create Config*, se crearán dos nuevas configuraciones que, en el caso particular de este TFG, tienen como nombre *Debug Host* y *BD DebugCL (Brown Deer DebugCL)*. La primera de ellas abordará la configuración del depurador del programa del dispositivo *host*, mientras que la segunda hará lo propio con el código escrito en lenguaje OpenCL C. En la Figura 4.7 se puede ver la nueva ventana de trabajo y las configuraciones para la integración del depurador creadas.

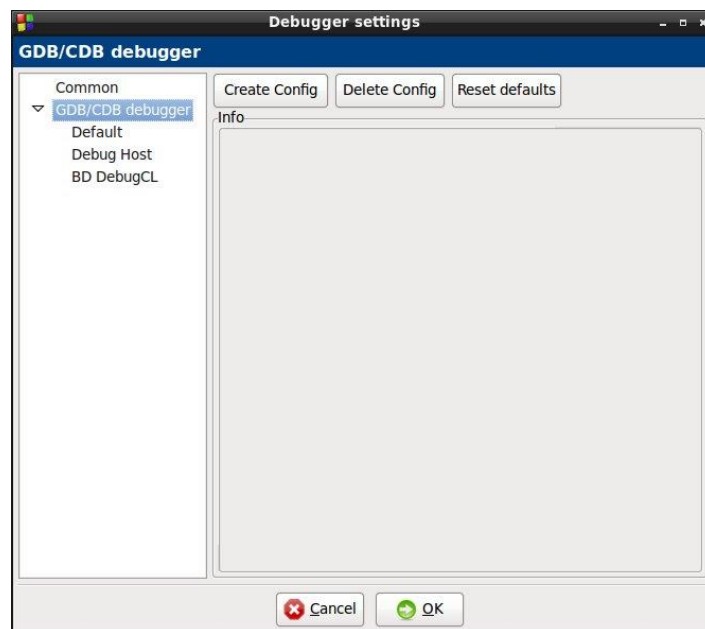


Figura 4.7. Ajustes del depurador GDB/CDB

Tras ello, se accede a la configuración *Debug Host*, en la cual se incluirán las características que se muestran en la Tabla 4.1. De igual modo, es posible observar el estado final en la ventana de esta configuración en la Figura 4.8.

Tabla 4.1. Configuración del depurador para el código *host*

Executable path	sudo -E LD_LIBRARY_PATH=/opt/adapteva/esdk/tools/host/lib EPIPHANY_HDF=/opt/adapteva/esdk/bsps/current/platform.hdf /usr/bin/gdb
Debugger Type	GDB
Debugger initialization commands	cd bin/Debug

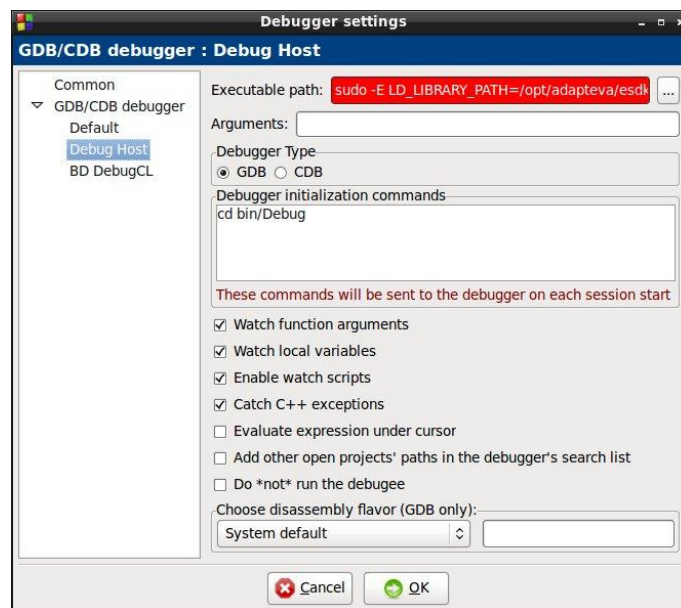


Figura 4.8. Configuración *Debug Host*

En el *Executable path* se necesita lanzar el depurador GDB como súper-usuario, por lo que se deben pasar las variables de entorno `LD_LIBRARY_PATH` y `EPIPHANY_HDF`. Aunque dicho *path* se vea en color rojo, a modo de error, ya que se espera un único *path*, la presencia del comando `sudo` implica la lectura de todo el campo. Finalmente, como comando de inicialización, se tiene el acceso al directorio `/bin/Debug` del proyecto en el que se esté trabajando, directorio donde se crea el ejecutable final resultante de la compilación del proyecto.

Seguidamente, se actuará de forma análoga con la configuración *BD DebugCL*, incluyendo en la misma las características indicadas en la Tabla 4.2. Así, la Figura 4.9 muestra el estado final en la ventana correspondiente a esta configuración.

Tabla 4.2. Configuración del depurador para el código OpenCL

Executable path	/opt/adapteva/esdk/tools/e-gnu/bin/e-gdb
Debugger Type	GDB
Debugger initialization commands	load

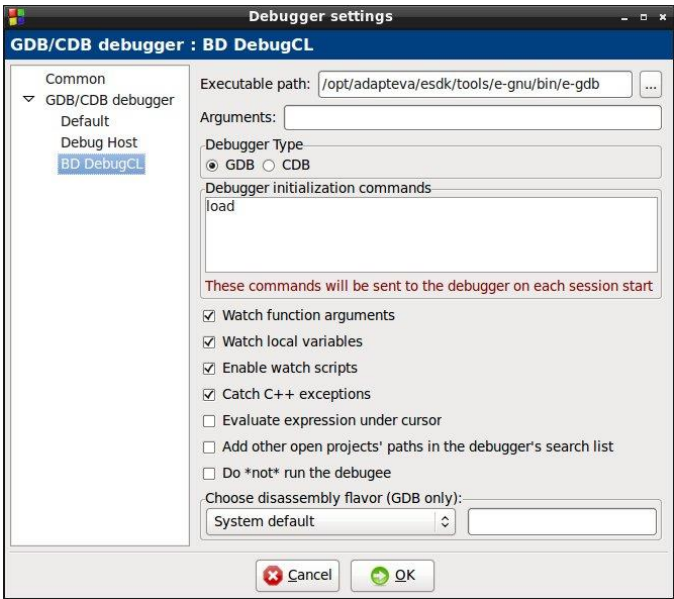


Figura 4.9. Configuración BD DebugCL

En este caso, se tiene como *Executable path* el directorio correspondiente al depurador *eGDB* (Epiphany GDB) y, como comando de inicialización, el comando `load`, que carga el código en el chip Epiphany para luego introducir *breakpoints*, etc., desde la línea de comandos del *eGDB*.

4.6.3 COMPILADOR

Con las configuraciones de depuración correctamente especificadas, es necesario continuar con las configuraciones relativas al proceso de compilación. Para ello, al igual que en el procedimiento llevado a cabo para el depurador, se accede a la pestaña *Settings* desde el área de inicio de *Code::Blocks*, escogiendo la opción *Compiler...*, lo que abrirá una nueva ventana. Tal y como se ve en la Figura 4.10, el compilador GNU GCC es el compilador por defecto para el programa del dispositivo *host*, el cual va a ser modificado para dar soporte a las librerías OpenCL que proporciona *Brown Deer Technology*.

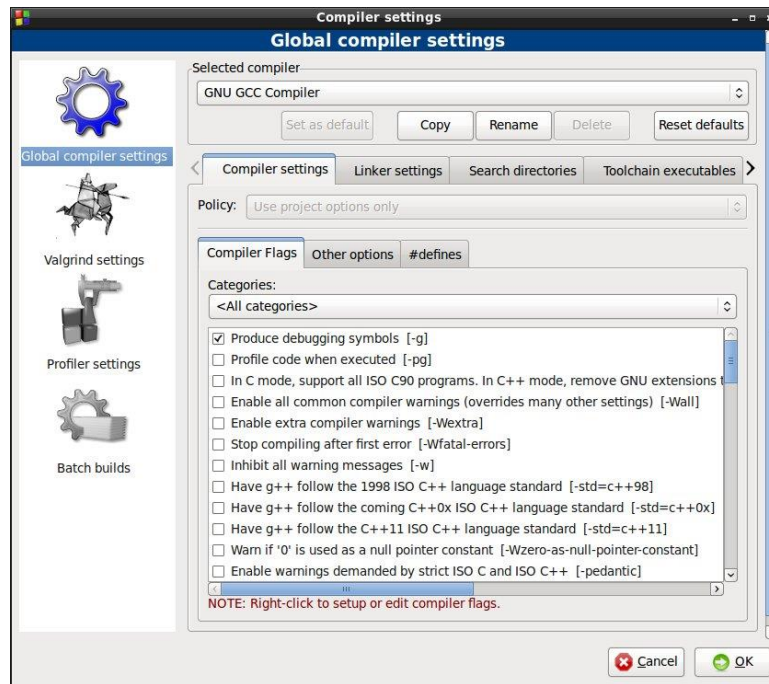


Figura 4.10. Ajustes del compilador GNU GCC

En esta ventana se tendrán que añadir los directorios y las librerías que implementan características relativas al compilador, así como el depurador asociado, que ya ha sido creado anteriormente. En primer lugar, se entra en la pestaña *Linker settings* y, en la lista *Link libraries*, se añadirán las dos librerías de OpenCL que ofrece *Brown Deer Technology*, que son STDCL y OCL. Dichas librerías se agregarán del modo en que se muestra en la Figura 4.11.

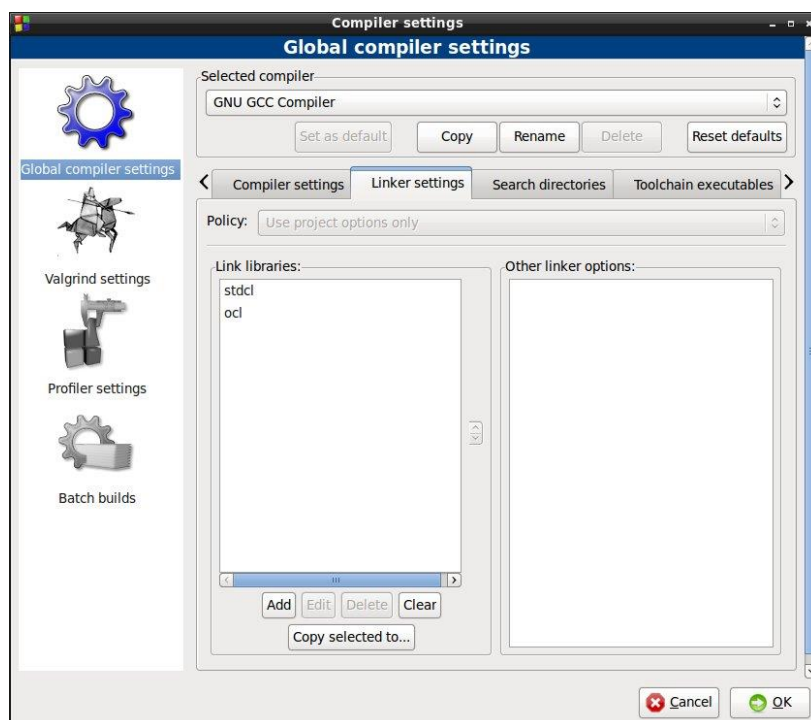


Figura 4.11. *Linker settings* del compilador GNU GCC para OpenCL

Después de esto, se accede a la pestaña *Search directories*, en cuya sub-pestaña *Compiler* se incluirá el directorio `/usr/local/browndeer/include/`. Dentro de la misma pestaña *Search directories*, pero ahora en la sub-pestaña *Linker*, se añadirá el directorio `/usr/local/browndeer/lib/`. Todo ello se puede ver, con mayor detalle, en la Figura 4.12 y en la Figura 4.13.

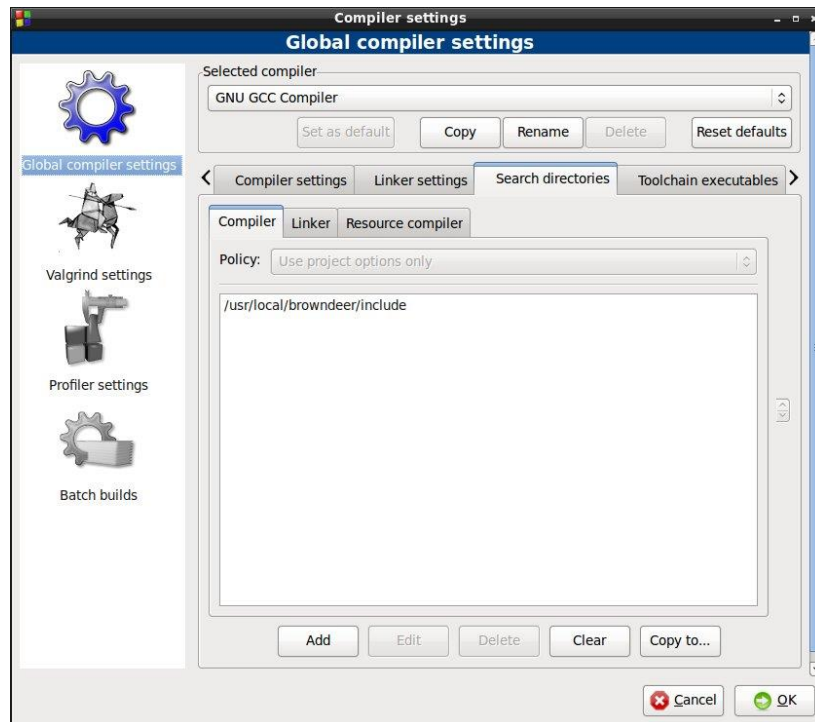


Figura 4.12. *Search directories/Compiler* del compilador GNU GCC para OpenCL

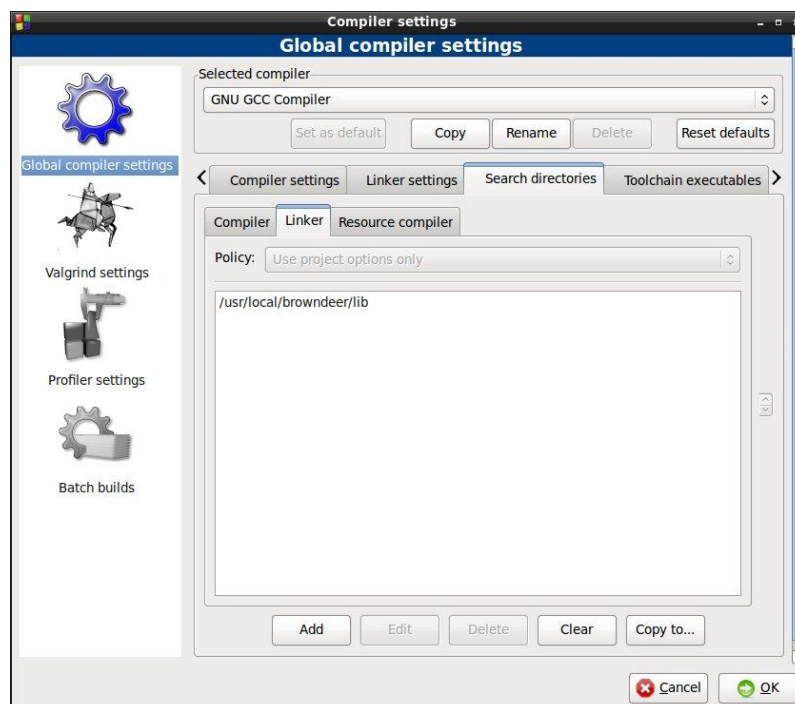


Figura 4.13. *Search directories/Linker* del compilador GNU GCC para OpenCL

Para finalizar con esta configuración del compilador, se accede a la pestaña *Toolchain executables*. En ella, tal y como se observa en la Figura 4.14, se selecciona como *Debugger* la configuración *Debug Host* que ha sido creada en pasos previos.

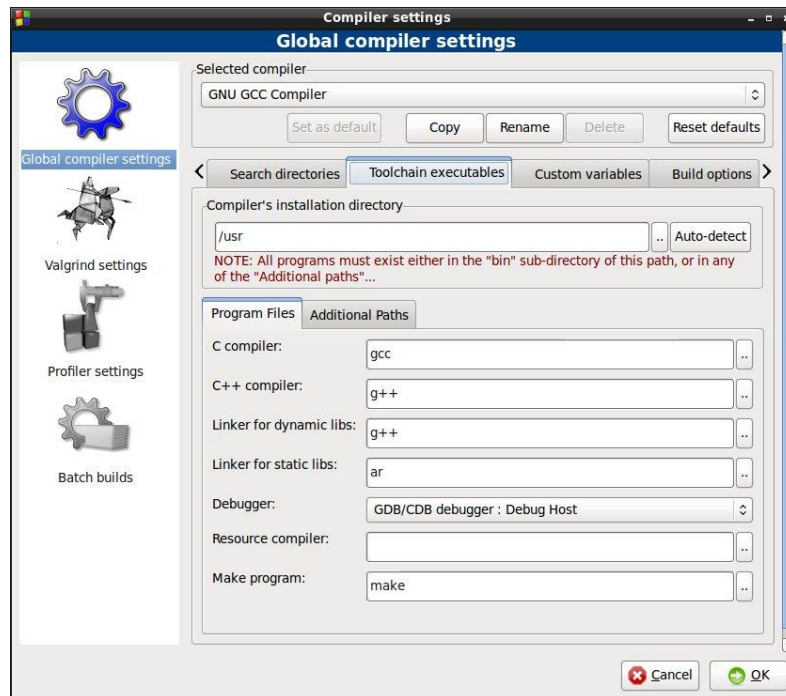


Figura 4.14. *Toolchain executables* del compilador GNU GCC para OpenCL

Llegados a este punto, con el compilador GNU GCC correctamente configurado, es posible abandonar la ventana de ajustes de compilador mediante el botón *OK*, guardando las modificaciones llevadas a cabo. Tras esto, se va a crear una nueva configuración de compilador, en este caso para los archivos OpenCL. Para ello, se vuelve a entrar en la pestaña *Settings* desde el área de inicio de *Code::Blocks*, escogiendo la opción *Compiler....* Aquí, con el compilador GNU GCC seleccionado, se pulsa el botón *Copy*, añadiendo una configuración de compilador que será una copia íntegra del compilador GNU GCC que se ha estado tratando. En el caso particular de este TFG, esa nueva configuración tiene como nombre *BD OpenCL (Brown Deer OpenCL)*.

La primera acción a realizar una vez seleccionada esta nueva configuración en el menú desplegable *Selected compiler*, será deseleccionar las opciones marcadas por defecto en la pestaña *Compiler settings*, sub-pestaña *Compiler Flags*. Además, en esta configuración *BD OpenCL*, se eliminará todo lo añadido anteriormente para la configuración del compilador GNU GCC en las pestañas *Linker settings* y *Search directories*.

El punto crítico para este compilador se encuentra en la pestaña *Toolchain executables*, que deberá modificarse para quedar del modo en que se muestra en la Figura 4.15.

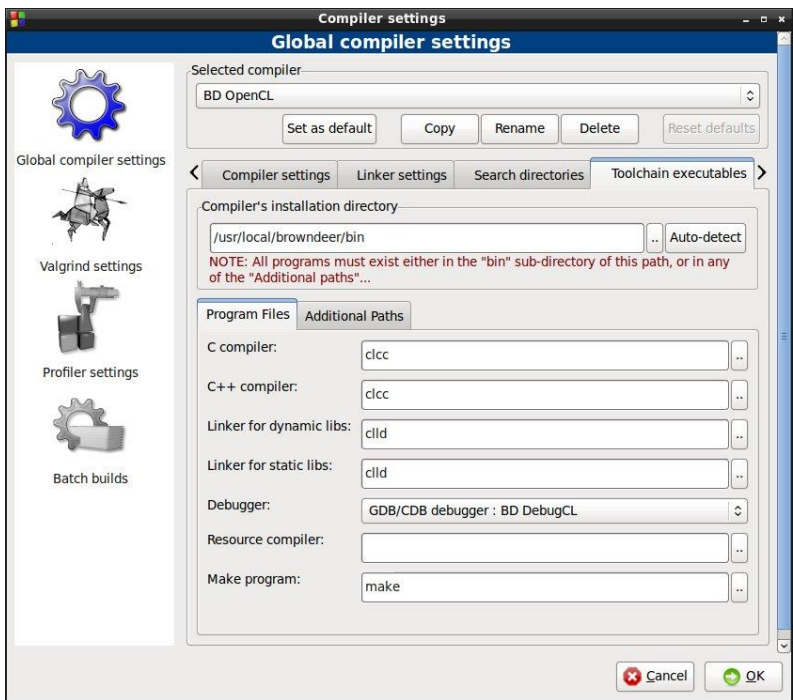


Figura 4.15. *Toolchain executables* del compilador *BD OpenCL*

En la siguiente pestaña, denominada *Custom variables*, se definirán las variables `LD_LIBRARY_PATH`, `PATH` y `EPIPHANY_HDF`, que se corresponden simplemente con copias locales de las variables con el mismo nombre del entorno *Bash*. El valor que deben tener estas variables se muestra en la Tabla 4.3, mientras que en la Figura 4.16 se indica el estado que debe tener la pestaña *Custom variables* tras llevar acabo esta acción.

Tabla 4.3. *Custom variables* para el compilador *BD OpenCL*

LD_LIBRARY_PATH	/usr/local/browndeer/lib:/usr/local/lib:/opt/adapteva/esdk/tools/host/lib
PATH	/usr/local/browndeer/bin:/opt/adapteva/esdk/tools/host/bin: /opt/adapteva/esdk/tools/e-gnu/bin:/usr/local/sbin:/usr/local/bin: /usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games
EPIPHANY_HDF	/opt/adapteva/esdk/bsps/current/platform.hdf

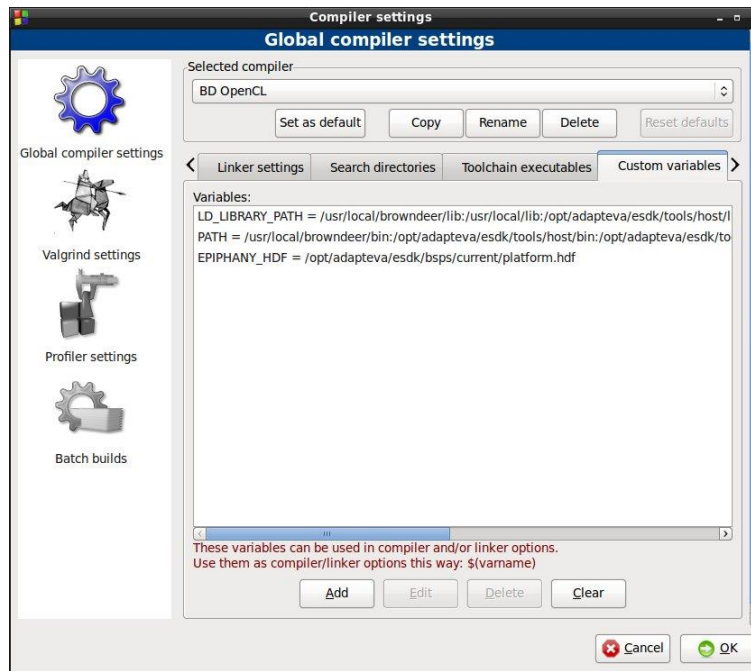


Figura 4.16. Custom variables del compilador BD OpenCL

Para finalizar, en la pestaña *Other settings*, se accederá al botón *Advanced options...*, el cual se encuentra en la esquina inferior derecha de la pantalla. Ello dará lugar a una ventana emergente, en la que se escoge *Yes* para confirmar que se desea acceder a las opciones avanzadas. Ya en la ventana de opciones avanzadas, se introducirá una plantilla en la línea de comandos del compilador, incluyendo todos los parámetros del mismo. *Code::Blocks*, posteriormente, pasa la cadena o *string* resultante de esa línea de comandos a un proceso *shell* (intérprete de órdenes).

Por tanto, en función de la opción seleccionada en el menú desplegable *Command*, se introducirá en el parámetro *Command line macro*, una plantilla diferente. En este caso, se actualizará dicho parámetro en dos de las opciones del menú desplegable *Command*, los cuales se indican en la Tabla 4.4. En la Figura 4.17 también se puede observar el estado que presenta la ventana de opciones avanzadas.

Tabla 4.4. Plantilla de *Command line macro*

Command	Command line macro
Compile single file to object file	sudo EPIPHANY_HDF=\$EPIPHANY_HDF LD_LIBRARY_PATH=\$LD_LIBRARY_PATH PATH=\$PATH bash -c "\$compiler \$options \$includes \$file -o \$object"
Link object files to console executable	sudo EPIPHANY_HDF=\$EPIPHANY_HDF LD_LIBRARY_PATH=\$LD_LIBRARY_PATH PATH=\$PATH:/bin bash -c "\$linker \$link_options \$libdirs \$link_objects \$libs -o \$exe_output \$object"

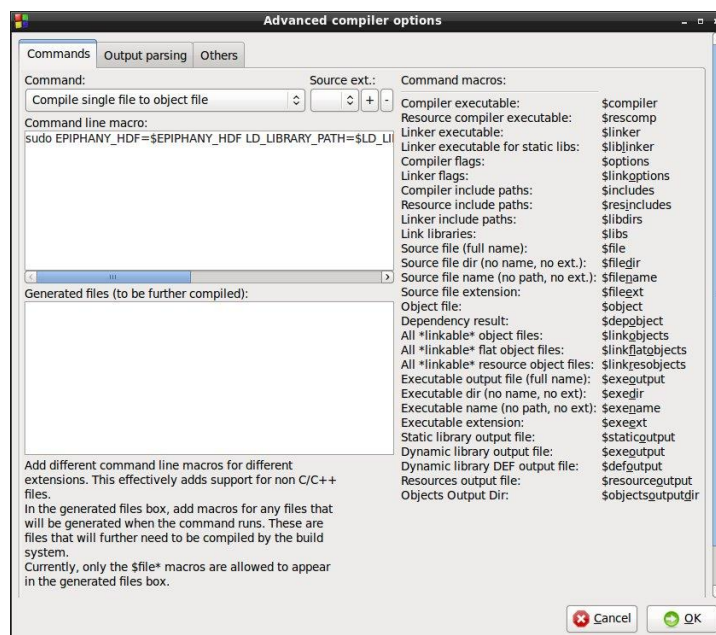


Figura 4.17. Opciones avanzadas del compilador BD OpenCL

Con todas las configuraciones del compilador ya realizadas, se deben cerrar las ventanas abiertas y guardar la configuración con la ayuda del botón *OK*.

4.6.4 INCLUSIÓN DE ARCHIVOS OPENCL EN UN PROYECTO

Teniendo un proyecto *Code::Blocks* ya creado, el modo de añadir un archivo OpenCL al mismo puede realizarse de dos formas: o bien incluir un archivo ya existente haciendo *click* con el botón derecho del ratón en el proyecto y seleccionando la opción *Add files...*, o bien generar un nuevo archivo en la pestaña *File* → *New* → *Empty file*. En este último caso, se preguntará si se desea añadir el nuevo archivo al proyecto activo y, tras contestar afirmativamente, se añadirá como un fichero de tipo C/C++, con la salvedad de que en el nombre del mismo se incluirá la extensión *.cl*.

Para concluir con la configuración del entorno *Code::Blocks* de cara al correcto funcionamiento con OpenCL sobre la placa Paralella, se hará *click* con el botón derecho del ratón en el proyecto en cuestión, escogiendo la opción *Properties...*. Dentro de la ventana de propiedades, la pestaña *Project settings* deberá configurarse tal y como se muestra en la Figura 4.18.

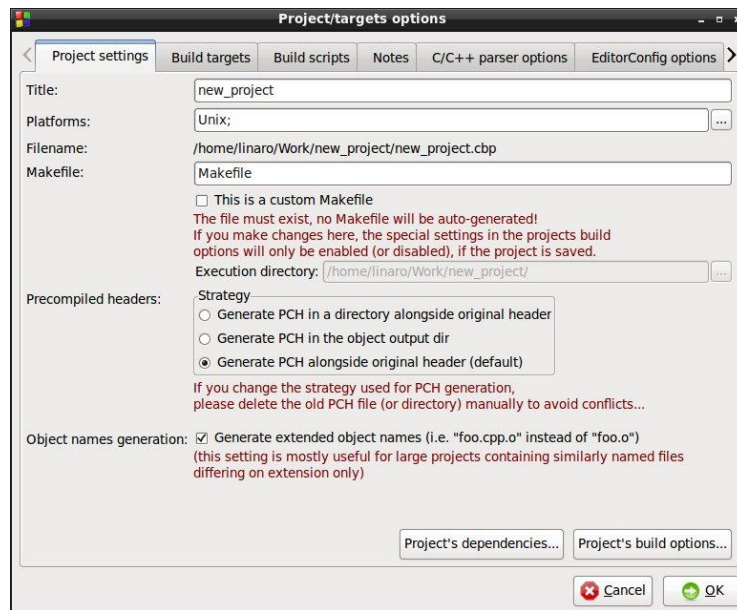


Figura 4.18. Ajustes del proyecto

Posteriormente, en la pestaña *Build targets*, se renombra el *target Release* por el nuevo nombre *CL*, haciendo así referencia a la dedicación del mismo. En dicha pestaña, se asignarán los ficheros C/C++ al *target Debug* y los ficheros OpenCL al *target CL*. Además, para ambos casos, se seleccionará la plataforma Unix, pues se trata de la adecuada para el sistema operativo que se gestiona en la placa Paralella. La Figura 4.19 y la Figura 4.20 reflejan el resultado del procedimiento explicado en este párrafo.

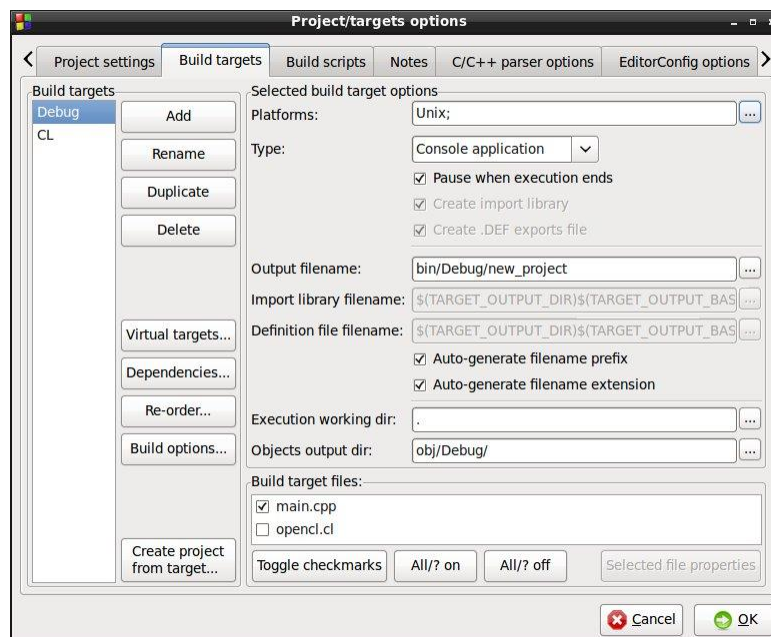


Figura 4.19. Ajustes del target Debug

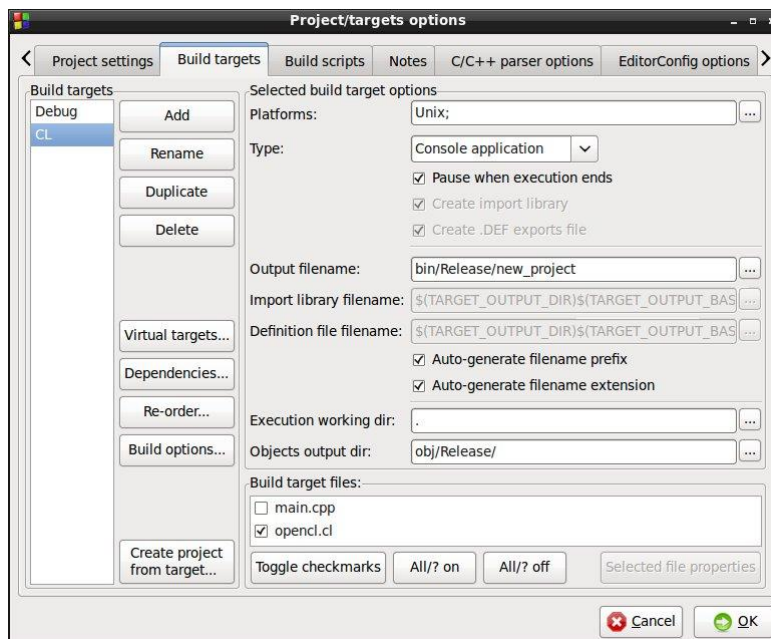


Figura 4.20. Ajustes del target CL

Tras haber asignado los diferentes archivos a su correspondiente *target*, se seleccionará el compilador *BD OpenCL* como compilador del *target CL*. Para esto, se debe hacer *click* con el botón derecho del ratón en el proyecto y seleccionar la opción *Build options...* que, aplicando las modificaciones citadas, debería tener la configuración que se muestra en la Figura 4.21.

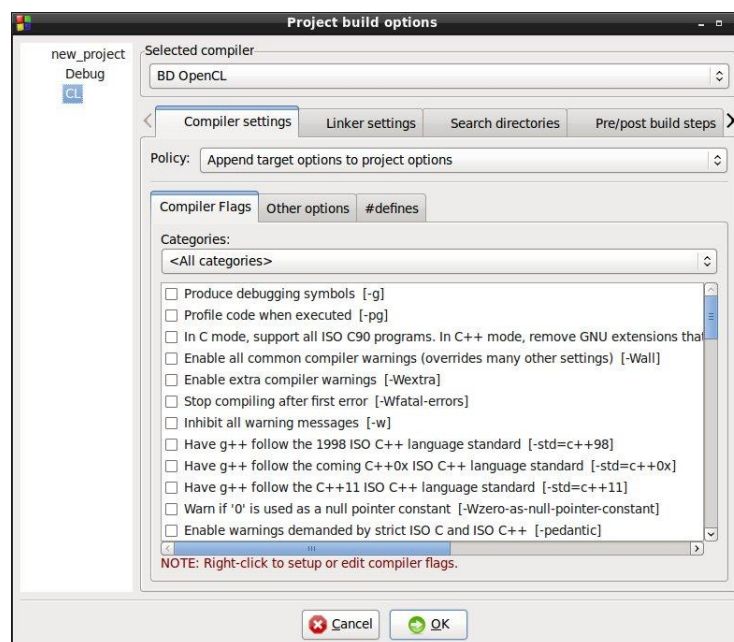


Figura 4.21. Compilador relativo al target CL

Finalmente, en el archivo OpenCL creado, se pueden habilitar las opciones *Compile file* y *Link file*. Se hace *click* con el botón derecho del ratón en el archivo y se escoge la opción *Properties....* En la nueva ventana, se accede a la pestaña *Build* y se habilitan las opciones mencionadas, como se muestra en la Figura 4.22.

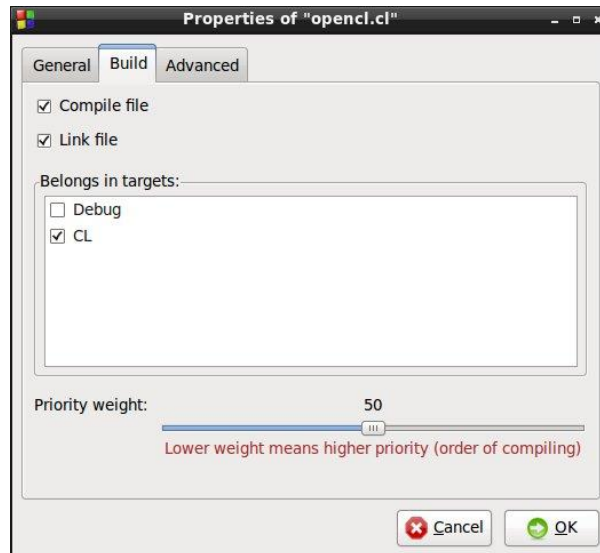


Figura 4.22. Opciones de generación de archivo OpenCL

4.6.5 PRUEBA INICIAL

Con el fin de comprobar que la configuración había sido implementada correctamente, se realizó una prueba inicial a partir de la modificación de un ejemplo disponible en [37], el cual es propiedad de *Brown Deer Technology*. Este ejemplo se corresponde con la creación en paralelo, en los 16 *cores* del dispositivo Epiphany-III, de la imagen correspondiente al conjunto de *Mandelbrot*, uno de los conjuntos fractales más conocidos y estudiados.

En Código 4.3 se reproduce el programa principal del dispositivo *host*, mientras en Código 4.4 se muestra el *kernel* a ejecutar en cada uno de los *cores*, ambos correspondientes al ejemplo mencionado. Además, en la Figura 4.23 puede verse el entorno con el proyecto creado y compilado para esta prueba, mientras que en la Figura 4.24 se refleja la imagen correspondiente al conjunto de *Mandelbrot* resultante de la correcta ejecución del programa a partir del entorno *Code::Blocks* configurado siguiendo el procedimiento descrito en este TFG.

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdcl.h>

#define OUTFILE "../mandelbrot.bmp"

#define WIDTH 1024
#define HEIGHT 768

#define CENTRE_X -0.5
#define CENTRE_Y 0
#define ZOOM 300

#define ITERATIONS 1024 // Higher is more detailed, but slower...

// Plotting functions and parameters...
#define bailoutr(n) (5*n)
#define bailoutg(n) (20*n)
#define bailoutb(n) 0

#define min(a,b) ((a<b)?a:b)

// Colours for the set itself...
#define IN_SET_R 0
#define IN_SET_G 0
#define IN_SET_B 0

void drawbmp(int width, int height, unsigned char* pixels, char* filename);

//////////////////// MAIN PROGRAM //////////////////////
int main (void)
{
    float startx; float endx;
    float starty; float endy;
    float dx; float dy;
    float dx_over_width, dy_over_height;

    char kern[] = "/home/linaro/Work/mandelbrot/mandelbrot.cl";
    void* openHandle;

    int iterations = ITERATIONS;
    int width      = WIDTH;
    int height     = HEIGHT;

    // About to malloc pixels
    cl_uchar* pixels = (cl_uchar*) clmalloc(stdacc, width * height * 3, 0);

    startx = CENTRE_X - ((float) WIDTH / (ZOOM * 2));
    endx   = CENTRE_X + ((float) WIDTH / (ZOOM * 2));

    starty = CENTRE_Y - ((float) HEIGHT / (ZOOM * 2));
    endy   = CENTRE_Y + ((float) HEIGHT / (ZOOM * 2));

    dx = endx - startx;
    dy = endy - starty;
    dx_over_width = dx / width;
    dy_over_height = dy / height;

```

Código 4.3. Programa principal del host "mandelbrot"

```

// Opening kernel
openHandle = clopen(stdacc, kern, CLLD_NOW);

// Getting the kernel with clsym
cl_kernel krn = clsym(stdacc, openHandle, "mandel_kern", CLLD_NOW);

// Calling clndrange
clndrange_t ndr = clndrange_init1d(0, height, 16);

// Calling clforka
clforka(stdacc, 0, krn, &ndr, CL_EVENT_WAIT, iterations, width, startx,
        starty, dx_over_width, dy_over_height, pixels);

// Transferring memory contents from the Epiphany using clmsync
clmsync(stdacc, 0, pixels, CL_MEM_HOST|CL_EVENT_WAIT);

// Calling drawbmp
drawbmp(width, height, pixels, OUTFILE);

clfree(pixels);

return 0;
}

void drawbmp (int width, int height, unsigned char* pixels, char* filename) {
unsigned int headers[13];
FILE* outfile;
int extrabytes;
int paddedsize;
int x; int y; int n;

extrabytes = 4 - ((width * 3) % 4); // How many bytes of padding to add to
                                   // each horizontal line - the size of
                                   // which must be a multiple of 4 bytes.

if (extrabytes == 4)
    extrabytes = 0;

paddedsize = ((width * 3) + extrabytes) * height;

// Headers...
headers[0] = paddedsize + 54; // bfSize (whole file size)
headers[1] = 0;              // bfReserved (both)
headers[2] = 54;              // bfOffbits
headers[3] = 40;              // biSize
headers[4] = width;           // biWidth
headers[5] = height;          // biHeight
                                // 6 will be written directly...
headers[7] = 0;               // biCompression
headers[8] = paddedsize;      // biSizeImage
headers[9] = 0;               // biXPelsPerMeter
headers[10] = 0;              // biYPelsPerMeter
headers[11] = 0;              // biClrUsed
headers[12] = 0;              // biClrImportant

outfile = fopen (filename, "wb");

```

Código 4.3. Programa principal del host “maldelbrot”

```

// Headers begin...
// When printing ints and shorts, write out 1 character at time to
// avoid endian issues.
fprintf (outfile, "BM");

for (n = 0; n <= 5; n++)
{
    fprintf(outfile, "%c", headers[n] & 0x000000FF);
    fprintf(outfile, "%c", (headers[n] & 0x0000FF00) >> 8);
    fprintf(outfile, "%c", (headers[n] & 0x00FF0000) >> 16);
    fprintf(outfile, "%c", (headers[n] & (unsigned int) 0xFF000000) >> 24);
}

// These next 4 characters are for the biPlanes and biBitCount fields.
fprintf(outfile, "%c", 1);
fprintf(outfile, "%c", 0);
fprintf(outfile, "%c", 24);
fprintf(outfile, "%c", 0);

for (n = 7; n <= 12; n++)
{
    fprintf(outfile, "%c", headers[n] & 0x000000FF);
    fprintf(outfile, "%c", (headers[n] & 0x0000FF00) >> 8);
    fprintf(outfile, "%c", (headers[n] & 0x00FF0000) >> 16);
    fprintf(outfile, "%c", (headers[n] & (unsigned int) 0xFF000000) >> 24);
}

// Headers done, now write the data...
for (y = height - 1; y >= 0; y--) // BMPs are written bottom to top.
{
    for (x = 0; x <= width - 1; x++)
    {
        // Also, it's written in (b,g,r) format...
        fprintf(outfile, "%c", pixels[(x * 3) + 2 + (y * width * 3)]);
        fprintf(outfile, "%c", pixels[(x * 3) + 1 + (y * width * 3)]);
        fprintf(outfile, "%c", pixels[(x * 3) + 0 + (y * width * 3)]);
    }
    if (extrabytes) // BMP lines must be of lengths divisible by 4
    {
        for (n = 1; n <= extrabytes; n++)
        {
            fprintf(outfile, "%c", 0);
        }
    }
}

fclose (outfile);
return;
}

```

Código 4.3. Programa principal del host "maldelbrot"

```

#define set_red(n) (5*n)
#define set_green(n) (20*n)
#define set_blue(n) 0

__kernel void mandel_kern(
    int iterations,
    int width,
    float startx,
    float starty,
    float dx,
    float dy,
    __global uchar* pixels
)
{
    int threeXwidth = 3 * width;
    unsigned char line[threeXwidth];
    int i, j, n, m, pixelBase;
    float x, y, r, s, nextr, nexts;

    j = get_global_id(0);

    for (i = 0; i < width; i++)
    {
        x = startx + i*dx;
        y = starty + j*dy;

        r = x;
        s = y;

        for (n = 0; n < iterations; n++)
        {
            nextr = ((r * r) - (s * s)) + x;
            nexts = (2 * r * s) + y;
            r = nextr;
            s = nexts;

            if ((r * r) + (s * s) > 4) break;
        }

        if (n == iterations) n=0;

        line[i * 3 + 0] = min(255,set_red(n));
        line[i * 3 + 1] = min(255,set_green(n));
        line[i * 3 + 2] = min(255,set_blue(n));
    }

    pixelBase = j * threeXwidth;
    for (m =0; m < threeXwidth; m++)
        pixels[pixelBase + m] = line[m];
}

```

Código 4.4. Kernel “mandelbrot”

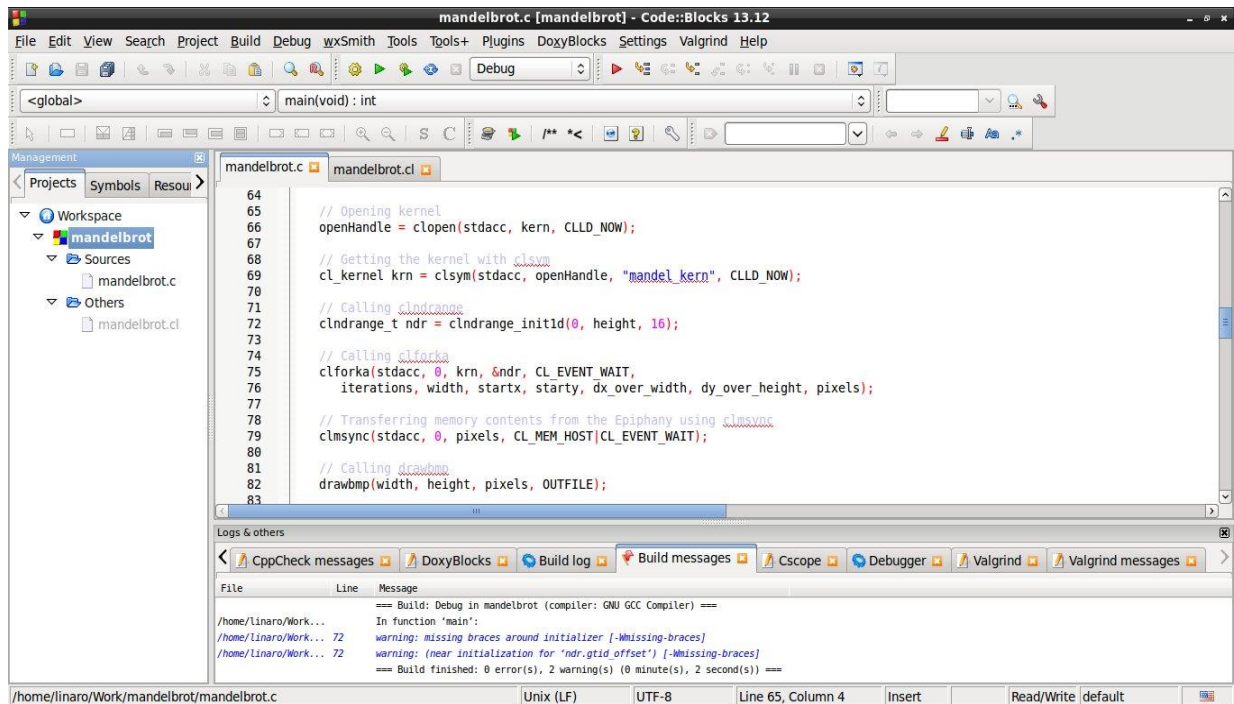


Figura 4.23. Proyecto “mandelbrot” en Code::Blocks

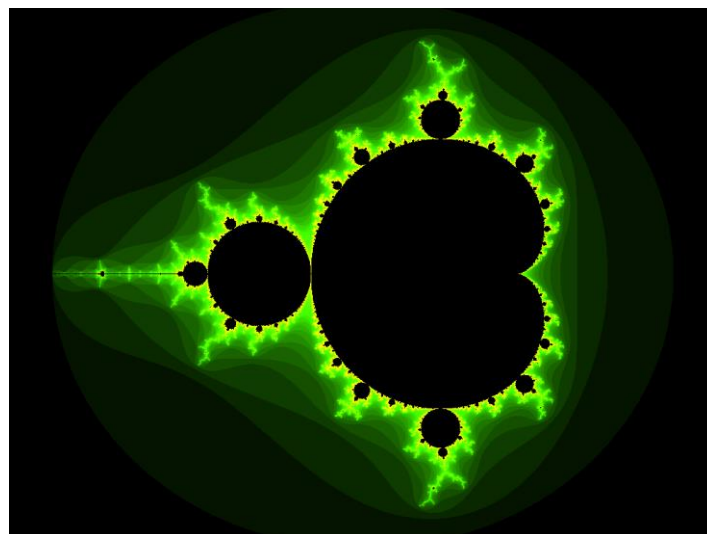


Figura 4.24. Conjunto de Mandelbrot

Con la implementación de este ejemplo, se comprueba la correcta actuación, tanto del compilador, como también del depurador, al analizar la ejecución mediante el proceso de depuración, haciendo uso de puntos de ruptura o *breakpoints* durante la misma.

Capítulo 5. DESARROLLO DE LA APLICACIÓN BASADA EN PARALELISMO

Una vez definida la metodología propuesta para la programación de aplicaciones paralelas en la placa Parallella, además de haber configurado el entorno de programación seleccionado, es el momento de comenzar, en este capítulo, la descripción del procedimiento seguido para el desarrollo de una aplicación de procesamiento de imágenes basada en OpenCL sobre la placa Parallella. Concretamente, la aplicación desarrollada se corresponde con la implementación del filtro de detección de bordes Sobel mediante OpenCL sobre la placa Parallella, que más adelante se explicará con mayor detalle.

En los objetivos del TFG, se propone el uso de la librería OpenCV (*Open-source Computer Vision*), por lo que, en primer lugar, se realizará una introducción a este recurso y a su integración en la placa Parallella.

5.1 OPENCV

OpenCV es una librería de visión artificial creada originalmente por *Intel Corporation* y que actualmente se encuentra bajo el mantenimiento y soporte de *Itseez Inc.* Esta librería está desarrollada en lenguaje de programación C y C++, y además presenta soporte para poder ejecutarse bajo sistemas, tanto Linux, como Windows y Mac OS X. OpenCV se diseñó para ofrecer eficiencia computacional, especialmente pensada para aplicaciones en tiempo real [39].

Uno de los propósitos de OpenCV es proporcionar una infraestructura de visión artificial fácil de utilizar, ayudando así a los usuarios a desarrollar programas para aplicaciones basadas en el uso de contenido visual más fácilmente y con mayor rapidez. Esta librería contiene más de 500 funciones que abarcan múltiples áreas en las que se hace uso del tratamiento de contenido visual de forma digital, como pueden ser: inspección de productos de fábrica, imágenes médicas, seguridad, calibración de cámaras, robótica, etc. [39].

La arquitectura de OpenCV es uno de sus aspectos más útiles, pues posee una infraestructura dividida en módulos, en los cuales se encuentran las diferentes funciones que contiene, clasificadas a partir del propósito de cada una de ellas [40]. En [41], es posible observar la distribución de los diversos módulos que integra la versión 3.1.0 de OpenCV (una de las más recientes), que se agrupan, a su vez, en módulos principales y módulos adicionales.

OpenCV, además, implementa algunas de sus funciones en *kernels* OpenCL para poder ser ejecutadas en paralelo en algún tipo de acelerador. Ello lo hace mediante una API interna y transparentemente al usuario, la cual efectúa la comunicación entre la CPU y el acelerador [42].

5.1.1 INSTALACIÓN DE OPENCV EN LA PLACA PARALLELLA

Se considera relevante destacar que OpenCV es una librería para la cual, por defecto y según la documentación proporcionada por *Adapteva*, no se ofrece ningún soporte específico de cara a su configuración, su instalación y, más importante aún, su uso sobre la placa Parallella.

En el caso particular de este TFG, se decidió instalar la versión de OpenCV más reciente en el momento de su realización, que es la 3.1.0, si bien es cierto que, en los días previos a la redacción de esta memoria, se publicó una nueva versión, la 2.4.13. Éstas y todas las versiones existentes de la librería OpenCV pueden descargarse desde [43]. Para el caso de la versión utilizada en este TFG, la propia documentación de OpenCV ofrece una pequeña guía con el procedimiento a seguir para lograr una correcta instalación en máquinas gestionadas por sistemas operativos basados en distribuciones GNU/Linux [44].

Previamente a la instalación de OpenCV, existen ciertos requerimientos de paquetes que deben ser instalados. Si bien algunos de estos paquetes son opcionales, se recomienda la instalación de todos ellos para no restringir ninguna funcionalidad. En Código 5.1 se indica la instalación de los paquetes obligatorios, mientras que en Código 5.2 se muestra el comando que instala los paquetes opcionales.

```
$ sudo apt-get install build-essential cmake git
libgtk2.0-dev pkg-config libavcodec-dev libavformat-dev
libswscale-dev
```

Código 5.1. Instalación de paquetes obligatorios para instalar correctamente OpenCV

```
$ sudo apt-get install python-dev python-numpy libtbb2
libtbb-dev libjpeg-dev libpng-dev libtiff-dev
libjasper-dev libdc1394-22-dev
```

Código 5.2. Instalación de paquetes opcionales para instalar correctamente OpenCV

Aunque la instalación se muestre en un único comando `sudo apt-get install`, también es posible y recomendable instalar los paquetes de forma independiente, ya que, en el caso particular

de este TFG, no era viable la instalación de algunos paquetes de este modo. Para solucionar, fue necesario instalar el paquete `aptitude` y realizar la instalación del modo `sudo aptitude install nombre_del_paquete`. De esta manera, como `aptitude` trabaja con un mayor nivel de abstracción que `apt-get`, se ofrecen algunas soluciones al problema. Una de las soluciones era reducir la versión de uno o varios de los paquetes que había instalados en el sistema, que fue la utilizada finalmente para solventar el problema.

Habiendo ya cumplido los prerequisites, se descarga la versión que se desee de OpenCV, en este caso la 3.1.0, desde [43], tras lo cual se descomprime el archivo descargado, quedando un directorio con todos los ficheros necesarios para su instalación. Con ello, se completa la instalación desde la consola *Bash* de Linux, utilizando los comandos que se indican en Código 5.3. En este código, el comando final `ldconfig` sirve para sincronizar las librerías generadas, con las librerías compartidas ya existentes.

```
$ cd /path/to/OpenCV/directory
$ mkdir build
$ cd build
$ cmake -D CMAKE_BUILD_TYPE=Release -D
    CMAKE_INSTALL_PREFIX=/usr/local ..
$ make
$ sudo make install
$ sudo ldconfig
```

Código 5.3. Instalación de la librería OpenCV

Al finalizar el procedimiento se generan varias librerías, cada una de ellas correspondiente a uno de los diferentes módulos en los que se organiza OpenCV. Una forma de validar si la instalación se ha realizado correctamente, es ejecutar un ejemplo que ofrece la documentación de OpenCV en [45]. Además, también es posible comprobar la versión de OpenCV instalada mediante la ejecución del comando `pkg-config --modversion opencv` en la consola *Bash* de Linux.

5.2 OPENCV EN CODE::BLOCKS

Con la librería OpenCV instalada en la placa Parallella, es el momento de configurar el IDE *Code::Blocks* en el que se va a implementar el desarrollo de la programación paralela sobre la placa Parallella, con el fin de poder usar las funcionalidades que proporciona. Al igual que en el caso del procedimiento propuesto para configurar OpenCL, se parte desde el área de inicio del entorno

Code::Blocks, donde se accede a la pestaña *Settings*, escogiendo la opción *Compiler...* en el menú desplegable, lo que abrirá la ventana correspondiente a los ajustes del compilador.

Una vez seleccionado el compilador GNU GCC, se accede a la pestaña *Search directories*, en cuya sub-pestaña *Compiler* se añadirán las entradas `/usr/local/include/opencv` y `/usr/local/include/opencv2` a las ya existentes de la configuración para OpenCL. Del mismo modo, en la sub-pestaña *Linker* se agrega la entrada `/usr/local/lib`. Estos pasos se representan en la Figura 5.1 y en la Figura 5.2, respectivamente.

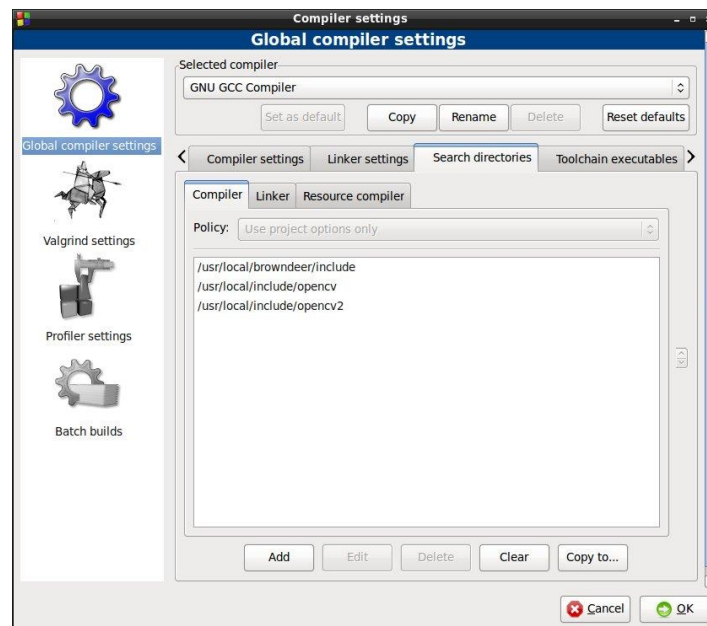


Figura 5.1. *Search directories/Compiler* del compilador GNU GCC para OpenCV

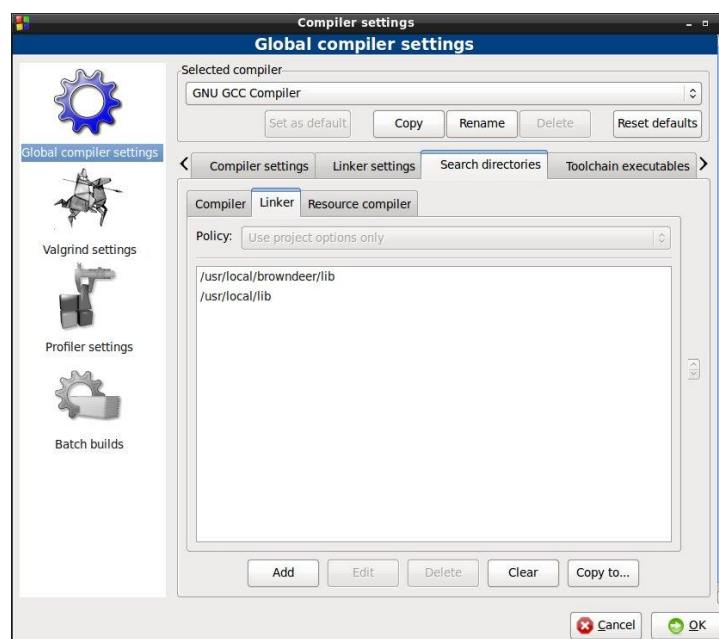


Figura 5.2. *Search directories/Linker* del compilador GNU GCC para OpenCV

Tras esto, para finalizar con el proceso de configuración, se accede a la pestaña *Linker settings*, incluyendo en la lista *Link libraries* todas las librerías `/usr/local/lib/libopencv_*****.so` existentes, tal y como se muestra en la Figura 5.3. En esta figura también es posible observar que se mantienen las librerías STDCL y OCL de la configuración ya realizada para OpenCL.

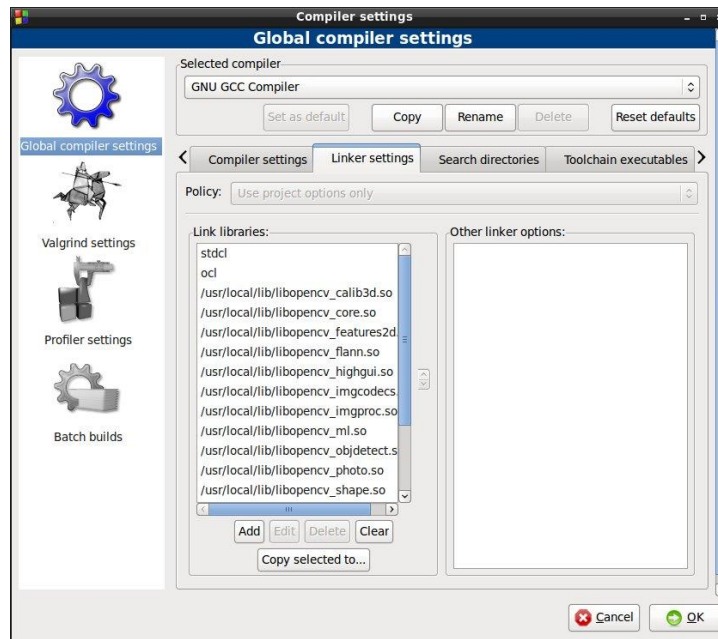


Figura 5.3. *Linker settings* del compilador GNU GCC para OpenCV

5.3 FILTRO DE DETECCIÓN DE BORDES SOBEL

El filtro de detección de bordes Sobel es uno de los algoritmos de procesamiento de imagen contenidos entre las múltiples funciones de OpenCV. Aunque bien es cierto que, en el caso particular de la versión de la librería utilizada en este TFG, no se incluye ningún *kernel* dedicado para la ejecución paralela de este algoritmo mediante la API OpenCL incluida en OpenCV, siendo éste uno de los motivos por los que se seleccionó este algoritmo de procesamiento de imágenes.

Este filtro consiste en la combinación de dos gradientes (uno para la orientación horizontal y otro para la vertical), los cuales se calculan a partir de la convolución entre dos matrices específicas de tamaño 3x3 y la imagen que, a su vez, es un conjunto de píxeles almacenados en una matriz bidimensional [46]. Como indicación, se pueden ver, a continuación, las matrices de referencia y el cálculo de cada gradiente, donde G_x es el gradiente horizontal, G_y es el gradiente vertical, y A es la imagen de datos.

$$G_x = \begin{bmatrix} -1 & 0 & -1 \\ -2 & 0 & -2 \\ -1 & 0 & -1 \end{bmatrix} * A \quad G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * A$$

Tras el cálculo de ambos gradientes, éstos se combinan para obtener la magnitud total del gradiente en cada punto $|G|$, de forma que [46]:

$$|G| = \sqrt{(G_x)^2 + (G_y)^2}$$

Sin embargo, comúnmente se utiliza una magnitud aproximada como la que se muestra a continuación, al tratarse de un cálculo mucho más rápido en cuanto a tiempo de computación. [46]

$$|G| = |G_x| + |G_y|$$

Para una mayor eficacia de este filtro, se suele aplicar a imágenes en escala de grises, pues únicamente contienen la componente de luminancia, sin tener en cuenta las componentes de color o croma. Así, solamente se cuantifica un parámetro, como es la intensidad luminosa de la imagen, en una escala desde 0 hasta 255, en el caso particular de una codificación de 8 bits por píxel.

5.4 IMPLEMENTACIÓN DE LA APLICACIÓN BASADA EN PARALELISMO

Con todo en el entorno ya configurado, y los aspectos teóricos definidos, se está en condiciones de comenzar con el desarrollo del código que implementará la aplicación planteada, que se basa en el paralelismo computacional. Como ya se ha comentado, la metodología de programación implica la existencia de dos archivos: uno con el código referente al programa principal del dispositivo *host*, y otro con la implementación del *kernel* a ejecutar en paralelo en el chip Epiphany.

Si bien en un principio se pretendía aplicar el algoritmo de procesamiento a contenido visual capturado a partir de la placa *Raspberry Pi Camera*, las circunstancias no lo han permitido. Por lo tanto, se ha desarrollado el programa paralelo del filtro de detección de bordes Sobel para aplicarlo a imágenes leídas desde un archivo almacenado en la misma placa Parallella, como se habría dispuesto en el caso de haber sido posible la integración de la *Raspberry Pi Camera*.

El procedimiento seguido en el desarrollo del programa ha surgido de una modificación del código disponible en [47]. Así, en primer lugar, se carga la imagen y se convierte a escala de grises en la CPU, enviándose posteriormente al chip Epiphany, donde se aplica el algoritmo Sobel de forma paralela en un número determinado de *cores*, para finalmente, recibir los resultados desde el chip

Epiphany y mostrar en pantalla, tanto la imagen inicial, como la imagen resultante del filtro de detección de bordes Sobel.

Debido a la escasa memoria local de los *cores*, el algoritmo se aplica en cada uno de ellos dividiendo el total de píxeles que le corresponde en fragmentos de un determinado número de píxeles. El tamaño de estos fragmentos deberá ser soportado por la memoria local de los *cores*, existiendo en consecuencia un límite máximo a un valor concreto.

5.4.1 DESARROLLO DEL CÓDIGO *HOST*

En este apartado, se va a comenzar con el desarrollo del código correspondiente al dispositivo *host*. El archivo con el código *host* se ha desarrollado en lenguaje de programación C y tiene como nombre, en el caso de este TFG, `sobel_parallel.cpp`. Como se puede ver, el archivo tiene una extensión relativa al lenguaje de programación C++, algo que se ha decidido por convenio, pues el mismo incluye cabeceras o *headers* C++ relativas a librerías de OpenCV. Mientras las cabeceras C (extensión *.h*) son compatibles para código fuente tanto en lenguaje C como en lenguaje C++, las cabeceras C++ (extensión *.hpp*) únicamente son compatibles para código desarrollado en lenguaje C++. Por lo tanto, aunque se programe en lenguaje C en un archivo *.cpp*, el compilador no presentará problema alguno a la hora de realizar su función.

Dicho esto, lo primero a incluir en el archivo `sobel_parallel.cpp` serán las cabeceras de las librerías que van a utilizarse. Estas cabeceras se muestran en Código 5.4 y comprenden: la librería estándar de propósito general del lenguaje de programación C, la librería estándar del lenguaje de programación C para operaciones de entrada y salida, la librería que define las estructuras para el cálculo del tiempo, la librería *STDCL* ya citada, aportada por *Brown Deer Technology*, y dos de los módulos de OpenCV (concretamente, uno para tratar la interfaz del contenido visual con el usuario y otro que ofrece algunas funciones para el procesamiento de imágenes).

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

#include <stdcl.h>

#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>
```

Código 5.4. Cabeceras del código *host*

Tras esto, se comienza con el cuerpo principal del programa o función *main*. Se ha establecido pasar, como argumentos al mismo, el número de *cores* del chip Epiphany para la ejecución del código paralelizable, el tamaño de los fragmentos para la ejecución del algoritmo en cada *core*, y el directorio con la imagen a la que se pretende aplicar el algoritmo. Por tanto, para poder seguir con la ejecución, en Código 5.5 se puede observar cómo se comprueba que el número de argumentos sea cuatro, ya que, como es sabido, el primer argumento se reserva para contener el nombre del programa. Si esto no ocurriera, se finaliza la ejecución con un mensaje de error.

```
int main(int argc, char *argv[])
{
    if (argc != 4) {
        fprintf(stderr, "The amount of arguments is not correct.\n");
        exit(1);
    }
}
```

Código 5.5. Comprobación del número de argumentos pasados al programa

A continuación, se crean las variables necesarias para medir el tiempo de ejecución del paralelismo, así como el tiempo de ejecución total del programa. Además, utilizando una clase de OpenCV, se generan también las variables en las que se almacenarán las imágenes: una para la imagen en color, otra para la imagen en escala de grises, y otra para la imagen resultado de la aplicación del filtro Sobel. Esta clase será *IplImage*, que es la única en la que se consiguió el soporte necesario para gestionar sus datos en *buffers* compartidos. Finalmente, se obtiene desde los argumentos pasados al programa, el número de *cores* y el tamaño de los fragmentos (o *tiles*) para la ejecución del algoritmo en cada *core*, comprobando que el número de *cores* se encuentra dentro de los límites del chip Epiphany. Todo esto se muestra en Código 5.6.


```
// Variables for the time measurement
struct timeval* start = (struct timeval*)malloc(sizeof(struct timeval));
struct timeval* end   = (struct timeval*)malloc(sizeof(struct timeval));
double* elapsed       = (double*)malloc(sizeof(double));

struct timeval* total_start = (struct timeval*)malloc(sizeof(struct
    timeval));
struct timeval* total_end   = (struct timeval*)malloc(sizeof(struct
    timeval));
double* total_elapsed       = (double*)malloc(sizeof(double));

// OpenCV image variables
IplImage* colorImage;
IplImage* grayImage;
IplImage* outputImage;

// Get the number of cores for the execution and the tile size
const int CORES      = atoi(argv[1]);
const int TILE_SIZE  = atoi(argv[2]);

if (CORES < 1 || CORES > 16){
    fprintf(stderr, "The number of cores is not correct.\n");
    exit(2);
}
```

Código 5.6. Generación de las variables y las constantes principales del programa

Tras haber creado los parámetros principales del programa, en Código 5.7 se inicializa todo lo relativo a la configuración de OpenCL, con las variables y las funciones que contiene la API STDCL. Como contexto de OpenCL se escoge `stdacc`, que incluye a todos los aceleradores que no son ni CPUs, ni GPUs. Se elige el dispositivo número 0, que se corresponde con el chip Epiphany. Y para concluir, se define el *kernel* a ejecutar en paralelo desde el archivo OpenCL, así como el grupo de trabajo en el chip Epiphany en función del número de *cores* a utilizar. En este caso, como archivo OpenCL se tiene el fichero `sobel_kernel.cl`, cuyo *kernel* interno se denomina `sobel_kern`.

```
// OpenCL variables
CLCONTEXT* cp      = stdacc;
unsigned int devnum = 0;

// Opening the kernel file and getting the kernel
const char kern[] = "/home/linaro/Work/sobel_parallel/sobel_kernel.cl";
void* openHandle  = clopen(cp, kern, CLLD_NOW);
cl_kernel krn     = clsym(cp, openHandle, "sobel_kern", CLLD_NOW);

// Define the computational domain and workgroup size
clndrange_t ndr = clndrange_init1d(0, CORES, CORES);
```

Código 5.7. Inicialización de la configuración OpenCL

Seguidamente, comienza la ejecución de la funcionalidad asociada al programa. Se carga, en una de las variables `IplImage`, la imagen leída desde un archivo cuya ubicación se encuentra en uno de los argumentos recibidos por la función *main*, comprobándose instantáneamente si la lectura ha sido exitosa. Con ello, se calculan los parámetros a enviar al *kernel* OpenCL como argumentos, que son: el número total de píxeles y el número de píxeles de una línea, además del número de *cores* y el tamaño de los *tiles* para la ejecución del algoritmo en cada *core*, anteriormente obtenidos. Previamente a todo, se almacena el instante temporal de comienzo de la ejecución, pudiendo así calcular luego su duración, como se muestra en Código 5.8.

```
gettimeofday(total_start, NULL);

// Get an image from a file
colorImage = cvLoadImage(argv[3], CV_LOAD_IMAGE_UNCHANGED);
if (!colorImage) {
    fprintf(stderr, "Error reading the image file.\n");
    exit(3);
}

// Calculating the kernel arguments
cl_uint width      = colorImage->width;
cl_uint height     = colorImage->height;
cl_uint n          = height*width;
cl_uint line       = width;
cl_uint cores      = CORES;
cl_uint tile_size  = TILE_SIZE;
```

Código 5.8. Lectura de la imagen y cálculo de los argumentos del kernel OpenCL

Como último paso previo a la ejecución paralela, tal y como se indica en Código 5.9, se crean dos *buffers* de memoria compartida, ambos del tamaño de la imagen cargada: uno para la imagen en escala de grises a enviar al acelerador, y otro para la imagen resultante de aplicar el algoritmo, la cual se recibe del acelerador. Además, se ajustan las variables de estas imágenes para soportar el tamaño de la imagen cargada. Tras ello, se ligan los datos de esas dos variables a los *buffers* creados, se generan dos ventanas para la posterior visualización de las imágenes y, haciendo uso de una función ofrecida por la librería OpenCV, se aplica la conversión de la imagen inicial a escala de grises.

```
// Allocate OpenCL device-sharable memory
cl_char* aa = (cl_char*)clmalloc(cp, n*sizeof(cl_char), 0);
cl_char* bb = (cl_char*)clmalloc(cp, n*sizeof(cl_char), 0);

// Create OpenCV IplImage headers to hold the data
grayImage = cvCreateImageHeader(cvGetSize(colorImage),
    colorImage->depth, 1);
outputImage = cvCreateImageHeader(cvGetSize(colorImage),
    colorImage->depth, 1);

// Use the shared buffers for the image data
cvSetData(grayImage, aa, line);
cvSetData(outputImage, bb, line);

// Creating the windows for showing the images
cvNamedWindow("Input", CV_WINDOW_AUTOSIZE);
cvNamedWindow("Output", CV_WINDOW_AUTOSIZE);

// Converting the image to grayscale
cvCvtColor(colorImage, grayImage, CV_BGR2GRAY);
```

Código 5.9. Preparación de la memoria compartida, de las ventanas para mostrar las imágenes y de la imagen en escala de grises

Una vez se ha preparado todo para la ejecución de la parte paralela de la aplicación, se procede a ello con el uso de las funciones que ofrece la API STDCL. Asimismo, se toma nota del instante temporal de inicio y de finalización de su ejecución mediante las variables generadas para ese propósito. Como se muestra en Código 5.10, en primer lugar se sincronizan los datos en memoria compartida de la imagen en escala de grises, desde el *host* hacia el acelerador; posteriormente, se ejecuta el *kernel* OpenCL para aplicar el algoritmo Sobel; a continuación, se realiza la sincronización inversa con el fin de recoger los datos de la imagen resultante en el dispositivo *host*; y finalmente, se espera a que finalice la ejecución de estas funciones.

```
gettimeofday(start, NULL);

// Non-blocking sync vector aa to device memory (copy to Epiphany)
clmsync(cp, devnum, aa, CL_MEM_DEVICE | CL_EVENT_NOWAIT);

// Non-blocking fork of the OpenCL kernel to execute on the Epiphany
clforka(cp, devnum, krn, &ndr, CL_EVENT_NOWAIT, n, line, cores, tile_size,
    aa, bb);

// Non blocking sync vector bb to host memory (copy back to host)
clmsync(cp, devnum, bb, CL_MEM_HOST | CL_EVENT_NOWAIT);

// Block waiting of completion of operations in command queue
clwait(cp, devnum, CL_ALL_EVENT);

gettimeofday(end, NULL);
```

Código 5.10. Ejecución del algoritmo de forma paralela en el chip Epiphany

Después de realizar la ejecución paralela, se muestra al usuario en las ventanas creadas anteriormente, tanto la imagen original, como la imagen resultante de la aplicación del filtro de detección de bordes Sobel. Asimismo, mediante el uso de la función `cvSaveImage`, la imagen obtenida como resultado también podría almacenarse en un nuevo archivo en la placa Parallella. Como última acción, se almacena el instante temporal de finalización de la ejecución del programa y se calculan los tiempos de ejecución total y el correspondiente a la ejecución del procesamiento paralelo. Estos tiempos serán volcados al usuario mediante la consola *Bash* de Linux en la que se ha ejecutado el programa. Finalmente, se espera indefinidamente hasta que sea pulsada una tecla, tras lo cual se liberan los *buffers* de memoria compartida y se cierra el programa. Todo ello se recoge en Código 5.11.

```
// Showing the images
cvShowImage("Input", colorImage);
cvShowImage("Output", outputImage);

gettimeofday(total_end, NULL);

// Getting and showing the computing times
*elapsed = end->tv_sec + (end->tv_usec/1000000.0) - start->tv_sec
          - (start->tv_usec/1000000.0);
*total_elapsed = total_end->tv_sec + (total_end->tv_usec/1000000.0)
                - total_start->tv_sec - (total_start->tv_usec/1000000.0);
fprintf(stdout, "This frame took %.6lf seconds of parallel
               computing and a total time of %.6lf seconds.\n", (*elapsed),
               (*total_elapsed));

cvWaitKey(0);

cfree(aa);
cfree(bb);

return (0);
}
```

Código 5.11. Visualización de las imágenes, del tiempo de ejecución y final del programa

5.4.2 DESARROLLO DEL CÓDIGO *KERNEL*

Con el programa principal correspondiente al dispositivo *host* ya desarrollado, se va a entrar de lleno en el código relativo al *kernel* o función que implementa el paralelismo. Esta función, como se puede ver en Código 5.12, recibe como parámetros el número total de píxeles de la imagen, el número de píxeles de cada línea de la imagen, el número de *cores* utilizados, y el tamaño de los fragmentos para la ejecución del algoritmo en cada *core*, además de punteros a las direcciones de

memoria compartida en las que se encuentran los *buffers* creados por el dispositivo *host* para la transferencia de los datos de las imágenes con el chip Epiphany.

```
__kernel void sobel_kern(
    uint n,
    uint line,
    uint cores,
    uint tile_size,
    __global char* gaa,
    __global char* gbb
)
{
```

Código 5.12. Prototipo de la función correspondiente al kernel OpenCL

Antes de comenzar con su desarrollo, resulta conveniente comentar que este *kernel* ha sido creado para poder soportar el procesamiento de imágenes con tamaños o resoluciones estándar, como los que se indican en la Tabla 5.1. Del mismo modo, únicamente se admite un número de *cores* que se corresponda con la unidad o con una potencia de 2. Es decir, que teniendo el chip Epiphany-III 16 *cores*, las posibilidades consideradas son: 1, 2, 4, 8 y 16 *cores*.

Tabla 5.1. Resoluciones estándar de imágenes

ESTÁNDAR	RESOLUCIÓN (píxeles)
VGA (Video Graphics Array)	640x480
QVGA (Quarter VGA)	320x240
SVGA (Super VGA)	800x600
XGA (Extended Graphics Array)	1024x768
WXGA (Wide XGA)	1280-1366 x 720-768
SXGA (Super XGA)	1280x1024
WSXGA (Wide SXGA)	1600-1900 x 900-1080
UXGA (Ultra XGA)	1600x1200
QXGA (Quadruple XGA)	2048x1536
DVD Standard	720x480
HD	1280x720
Full HD	1920x1080

Habiendo aclarado esto, se va a comenzar con la implementación del *kernel*. La primera acción a realizar consiste en obtener el identificador individual que define a cada *core*, así como el número de píxeles a procesar en cada uno de ellos. A partir de estos valores, se calculan ciertos parámetros importantes, como lo son el píxel inicial y el píxel final del conjunto de píxeles que le corresponde a cada *core*. Si, además, se tiene en cuenta el tamaño definido para cada fragmento o *tile*, es posible calcular otros parámetros, como el número de *tiles* a procesar, o el posible resto para un *tile* final si quedasen píxeles sin incluir en los fragmentos anteriores. Esto se puede ver en Código 5.13.

```
int k = get_global_id(0);

int n_core = n/cores;
int m_core = n%cores;

int ifirst = k*n_core + ((k>m_core)? 0:k);
int iend   = ifirst + n_core + ((k<m_core)? 1:0);

int tiles_number = n_core/tile_size;
int remainder    = n_core%tile_size;
```

Código 5.13. Inicialización de los parámetros principales del procesamiento en cada *core*

Adicionalmente, según el procedimiento ya expuesto sobre el que se basa la aplicación correspondiente al filtro Sobel, y debido a las matrices utilizadas para la convolución, son necesarias tanto la línea anterior como la línea posterior para el cálculo de cada uno de los píxeles de la imagen resultante. Por tanto, se define el tamaño del *buffer* de entrada para cada fragmento o *tile*, incluyendo las líneas adicionales citadas, además de definir también una variable con el límite final para el bucle que recorre cada *tile*. Tras esto, se crean los *buffers* de entrada y salida para cada fragmento y, con el fin de culminar la preparación para la ejecución, se definen: una variable *i* para recorrer distintos bucles, una variable *tile* para ejecutar un bucle con tanta iteraciones como *tiles* se vayan a procesar, y una variable *tmp* para almacenar temporalmente el resultado del procesamiento de cada píxel, como está recogido en Código 5.14.

```
int entry      = tile_size + 2*line + 2;
int tile_final = entry - line - 1;

char aa[entry]; // tile_size pixels, plus 1 line before and 1 line after,
                // plus 2 extra border pixels
char bb[tile_size];

int i, tile, tmp;
```

Código 5.14. Inicialización de los parámetros para el tratamiento de cada fragmento o *tile*

Con todo preparado, se está en condiciones de realizar la implementación de lo que sería el algoritmo correspondiente a la funcionalidad del filtro de detección de bordes Sobel. Como se indica en Código 5.15, el procedimiento a seguir para cada uno de los *tiles* es: en primer lugar, se almacena el mismo localmente desde la imagen de entrada en memoria compartida; posteriormente, para cada píxel del fragmento, se aplican los múltiples productos con las matrices de referencia que conforman la convolución, almacenando localmente el resultado; finalmente, se transfiere el contenido de dicho resultado al *buffer* en memoria compartida dedicado a la imagen resultado.

```
for (tile=0;tile<tiles_number;tile++)
{
    for(i=0;i<entry;i++)
        aa[i] = gaa[ifirst+tile*tile_size+i-line-1];

    for(i=line+1; i<tile_final; i++) {

        tmp = abs(-    aa[i-line-1]
                  - 2 * aa[i-1]
                  -    aa[i+line-1]
                  +    aa[i-line+1]
                  + 2 * aa[i+1]
                  +    aa[i+line+1])
        + abs(      aa[i-line-1]
                  + 2 * aa[i-line]
                  +    aa[i-line+1]
                  -    aa[i+line-1]
                  - 2 * aa[i+line]
                  -    aa[i+line+1]);

        bb[i-line-1] = tmp > 255 ? 255 : tmp;
    }

    for(i=(ifirst+tile*tile_size); i<(ifirst+(tile+1)*tile_size); i++)
        gbb[i] = bb[i-ifirst-tile*tile_size];
}
```

Código 5.15. Aplicación del algoritmo del filtro Sobel en el código kernel

Para concluir, se evalúa la posibilidad de que la división del número de píxeles correspondientes a cada *core* entre el tamaño de *tile* no sea exacta, lo que daría lugar a que algunos píxeles quedasen fuera de los *tiles* generales. Si esto sucediera, se procesarían los píxeles restantes como si de un único fragmento se tratase, y cerrando, a continuación, la función relativa al *kernel*, como se muestra en Código 5.16.

```

if (remainder != 0)
{
    int last_entry = remainder+2*line+2;
    int last_tile_final = last_entry-line-1;
    char aa[last_entry];
    char bb[remainder];

    for(i=0;i<last_entry;i++)
        aa[i] = gaa[ifirst+tiles_number*tile_size+i-line-1];

    for(i=line+1; i<last_tile_final; i++) {

        tmp = abs(-    aa[i-line-1]
                  - 2 * aa[i-1]
                  -    aa[i+line-1]
                  +    aa[i-line+1]
                  + 2 * aa[i+1]
                  +    aa[i+line+1])
        + abs(
            aa[i-line-1]
            + 2 * aa[i-line]
            +    aa[i-line+1]
            -    aa[i+line-1]
            - 2 * aa[i+line]
            -    aa[i+line+1]);

        bb[i-line-1] = tmp > 255 ? 255 : tmp;
    }

    for(i=(ifirst+tiles_number*tile_size); i<iend; i++)
        gbb[i] = bb[i-ifirst-tiles_number*tile_size];
}
}

```

Código 5.16. Evaluación y procesamiento de posibles píxeles fuera de los tiles

5.5 IMPLEMENTACIÓN OPENCV – OPENCL EN LA PLACA PARALLELLA

Como se comentó en la introducción a la librería OpenCV, ésta contiene una implementación propia de una API interna y transparente al usuario dedicada a la ejecución de paralelismo computacional, así como funciones desarrolladas en código *kernel* OpenCL. Si bien el filtro de detección de bordes Sobel no está implementado en la versión de OpenCV considerada en el presente TFG, sí que lo está una de las funciones que se utiliza para convertir la imagen en color a escala de grises (`cvtColor`). Por lo tanto, al estar uno de los objetivos del proyecto ligado con la adaptación de un algoritmo de OpenCV para su correcto funcionamiento en la placa Parallella, también se propone la evaluación de esta implementación. Tal y como se comenta en [42], el paralelismo se realiza internamente mediante el uso de la clase `UMat`.

De un modo similar al planteado en el código correspondiente al dispositivo *host* para la implementación paralela del filtro de detección de bordes Sobel, en Código 5.17 y en Código 5.18 se indican los programas creados para la aplicación secuencial y paralelizada de la función `cvtColor`, respectivamente. En este caso, al utilizar las clases actuales `Mat` y `UMat`, en lugar de la clase `IplImage` usada anteriormente, las funciones de OpenCV utilizadas son las más recientes y adecuadas a estas clases.

Ambas implementaciones recogen, al igual que en el caso del filtro Sobel: las cabeceras y los `namespace` necesarios para el uso de determinadas clases y funciones, las variables creadas para la medición de intervalos de tiempo, las variables generadas para almacenar las imágenes, y la ejecución del programa. Dentro de esta ejecución, se comienza con la lectura de la imagen, desde un archivo que se encuentra en la placa *Parallella*, a una de las variables creadas, comprobando que esta acción se ha llevado a cabo correctamente. A continuación, realiza la implementación de la función `cvtColor` a analizar, tomando nota del instante temporal de comienzo y de finalización. Finalmente, se muestran las imágenes anterior y posterior a la ejecución del algoritmo mediante ventanas dedicadas a ello, tras lo cual se calcula el retardo temporal de ejecución y se presenta a través de la consola *Bash* de Linux. Para culminar, se realiza una espera indefinida, hasta que se pulse una tecla, lo que llevaría a la culminación del procedimiento implementado en la aplicación.

```

#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <sys/time.h>

#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/core/utility.hpp>

using namespace std;
using namespace cv;

int main()
{
    struct timeval* start = (struct timeval*)malloc(sizeof(struct
        timeval));
    struct timeval* end    = (struct timeval*)malloc(sizeof(struct
        timeval));
    double* elapsed        = (double*)malloc(sizeof(double));

    Mat colorImage, grayImage;

    colorImage = imread("/home/linaro/Work/Images/*****.jpg",
        CV_LOAD_IMAGE_UNCHANGED);

    if (colorImage.empty()) {
        fprintf(stderr, "Error reading the image file.\n");
        exit(1);
    }

    gettimeofday(start, NULL);

    cvtColor(colorImage, grayImage, CV_BGR2GRAY);

    gettimeofday(end, NULL);

    namedWindow("Input", CV_WINDOW_AUTOSIZE);
    namedWindow("Output", CV_WINDOW_AUTOSIZE);

    imshow("Input", colorImage);
    imshow("Output", grayImage);

    *elapsed = end->tv_sec + (end->tv_usec/1000000.0) - start->tv_sec
        - (start->tv_usec/1000000.0);
    printf("This frame took %.6lf seconds of parallel computing.\n",
        (*elapsed));

    cvWaitKey(0);

    return (0);
}

```

Código 5.17. Programa secuencial *cvtColor*

```
#include <stdlib.h>
#include <iostream>
#include <sys/time.h>

#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/core/utility.hpp>
#include <opencv2/core/ocl.hpp>

using namespace std;
using namespace cv;

int main()
{
    ocl::setUseOpenCL(true);

    struct timeval* start = (struct timeval*)malloc(sizeof(struct
        timeval));
    struct timeval* end   = (struct timeval*)malloc(sizeof(struct
        timeval));
    double* elapsed       = (double*)malloc(sizeof(double));

    Mat img;
    UMat colorImage, grayImage;

    img = imread("/home/linaro/Work/Images/*****.jpg",
        CV_LOAD_IMAGE_UNCHANGED);

    if (img.empty()) {
        fprintf(stderr, "Error reading the image file.\n");
        exit(1);
    }

    colorImage = img.getUMat(ACCESS_READ);

    gettimeofday(start, NULL);

    cvtColor(colorImage, grayImage, CV_BGR2GRAY);

    gettimeofday(end, NULL);

    namedWindow("Input",  CV_WINDOW_AUTOSIZE);
    namedWindow("Output", CV_WINDOW_AUTOSIZE);

    imshow("Input",  colorImage);
    imshow("Output", grayImage);

    *elapsed = end->tv_sec + (end->tv_usec/1000000.0) - start->tv_sec
        - (start->tv_usec/1000000.0);
    printf("This frame took %.6lf seconds of parallel computing.\n",
        (*elapsed));

    waitKey(0);

    return (0);
}
```

Código 5.18. Programa paralelizado cvtColor

Las diferencias presentes entre las dos aplicaciones implican, fundamentalmente, la utilización de las clases `Mat` y `UMat`, pues es esta última la que es capaz de implementar, internamente, el uso de OpenCL. La habilitación de esta opción se llevará a cabo previamente al comienzo, a través de la línea de código `ocl::setUseOpenCL(true)` que se indica en Código 5.18. Por lo tanto, el programa secuencial hará uso de la clase `Mat` y se ejecutará solamente en el dispositivo *host*, mientras la aplicación paralelizada incluirá la clase `UMat` y llevará la ejecución de la función `cvtColor` al chip Epiphany.

Análogamente a lo comentado en el filtro de detección de bordes Sobel, si se deseara almacenar la imagen resultante en un archivo contenido en la placa Parallella, el modo de hacerlo sería, en este caso, mediante la función `imread` de la librería OpenCV, que es la que corresponde a las clases utilizadas en esta ocasión.

Capítulo 6. PRUEBAS Y RESULTADOS

Una vez implementada la aplicación basada en el paralelismo, en este capítulo se procederá a la validación de su ejecución en función de algunos parámetros a tener en cuenta. Asimismo, como ya se comentó anteriormente, también se analizará el factor de aceleración o *speed-up* que se obtiene al utilizar la implementación interna y transparente de OpenCL que ofrece la librería OpenCV para algunas de sus funciones. Tras definir las pruebas a realizar, se mostrarán los resultados obtenidos y se propondrá un análisis de los mismos, con la extracción de las conclusiones correspondientes.

6.1 DEFINICIÓN DEL BANCO DE PRUEBAS

En el caso del programa creado de forma paralela para la aplicación del filtro de detección de bordes Sobel en el chip Epiphany, se tendrán en cuenta para su análisis los tres parámetros que incluye como argumentos la función principal *main*. Estos parámetros son: la imagen a procesar (que se define por su tamaño o resolución), el número de *cores* utilizados para el procesamiento, y el tamaño de los fragmentos o *tiles* usados para la ejecución del algoritmo en los *cores*. Así, los distintos valores considerados para cada uno de estos parámetros son los siguientes:

- Tamaño o resolución de la imagen: 320x240 (QVGA), 640x480 (VGA) y 1280x960 (4x VGA).
- Número de *cores*: 1, 2, 4, 8 y 16.
- Tamaño de los fragmentos o *tiles*: 800 píxeles, 1600 píxeles y 3200 píxeles.

Con ello, se realizarán pruebas orientadas, fundamentalmente, a obtener los retardos temporales en todas y cada una de las 45 posibles combinaciones existentes entre los valores dados para los parámetros a analizar. Además, el resultado final para cada caso será una media ponderada de 5 ejecuciones del algoritmo en las mismas condiciones. Con el objetivo de llevar a cabo las pruebas definidas de manera automática, sin necesidad de ejecutar el programa manualmente en cada iteración, se decidió generar un *script* en el que se ejecutasen de manera continuada todas las configuraciones definidas.

A la hora de implementar este *script*, se modificó ligeramente el código relativo al programa principal del dispositivo *host*, de modo que los resultados no fueran mostrados en la consola *Bash* de Linux, sino que fueran volcados y almacenados en un archivo, denominado por defecto `results.txt`. Así, una vez finalizada la ejecución de este *script*, se introducirían manualmente

los resultados desde el fichero `results.txt` a una hoja de Excel. Esta modificación se representa en Código 6.1.

```
fprintf(stdout, "This frame took %.6lf seconds of parallel computing and  
a total time of %.6lf seconds.\n", (*elapsed), (*total_elapsed));
```



```
FILE* fp;  
fp = fopen("/home/linaro/Work/sobel_parallel/results.txt", "a");  
fprintf(fp, "This frame took %.6lf seconds of parallel computing and a  
total time of %.6lf seconds.\n", (*elapsed), (*total_elapsed));  
fclose(fp);
```

Código 6.1. Almacenamiento de resultados en un archivo de texto

Del mismo modo, se eliminó la espera indefinida al mostrar las imágenes hasta que fuera pulsada una tecla, sustituyéndola por un pequeño intervalo de 30ms en el que se visualizan las imágenes. Este cambio se indica en Código 6.2.

```
cvWaitKey(0);
```



```
cvWaitKey(30);
```

Código 6.2. Eliminación de la espera infinita en la muestra de las imágenes

Posteriormente, se generó el citado *script*, de nombre `script.sh`, cuyo contenido se indica en Código 6.3. Además, una vez creado el mismo, es necesario asignar los permisos de ejecución necesarios, lo cual se realiza desde la consola *Bash* de Linux, haciendo uso de los comandos indicados en Código 6.4.

```
#!/bin/bash

cd /home/linaro/Work/sobel_parallel/bin/Debug

echo "320x240 picture:" >> /home/linaro/Work/sobel_parallel/results.txt

echo "CORES=1, TILE_SIZE=800:" >> /home/linaro/Work/sobel_parallel/results.txt
for i in `seq 1 5`; do
    ./sobel_parallel 1 800 /home/linaro/Work/Images/320x240.jpg
done

echo "CORES=2, TILE_SIZE=800:" >> /home/linaro/Work/sobel_parallel/results.txt
for i in `seq 1 5`; do
    ./sobel_parallel 2 800 /home/linaro/Work/Images/320x240.jpg
done

echo "CORES=4, TILE_SIZE=800:" >> /home/linaro/Work/sobel_parallel/results.txt
for i in `seq 1 5`; do
    ./sobel_parallel 4 800 /home/linaro/Work/Images/320x240.jpg
done

echo "CORES=8, TILE_SIZE=800:" >> /home/linaro/Work/sobel_parallel/results.txt
for i in `seq 1 5`; do
    ./sobel_parallel 8 800 /home/linaro/Work/Images/320x240.jpg
done

echo "CORES=16, TILE_SIZE=800:" >> /home/linaro/Work/sobel_parallel/results.txt
for i in `seq 1 5`; do
    ./sobel_parallel 16 800 /home/linaro/Work/Images/320x240.jpg
done

echo "CORES=1, TILE_SIZE=1600:" >> /home/linaro/Work/sobel_parallel/results.txt
for i in `seq 1 5`; do
    ./sobel_parallel 1 1600 /home/linaro/Work/Images/320x240.jpg
done

echo "CORES=2, TILE_SIZE=1600:" >> /home/linaro/Work/sobel_parallel/results.txt
for i in `seq 1 5`; do
    ./sobel_parallel 2 1600 /home/linaro/Work/Images/320x240.jpg
done

echo "CORES=4, TILE_SIZE=1600:" >> /home/linaro/Work/sobel_parallel/results.txt
for i in `seq 1 5`; do
    ./sobel_parallel 4 1600 /home/linaro/Work/Images/320x240.jpg
done

echo "CORES=8, TILE_SIZE=1600:" >> /home/linaro/Work/sobel_parallel/results.txt
for i in `seq 1 5`; do
    ./sobel_parallel 8 1600 /home/linaro/Work/Images/320x240.jpg
done

echo "CORES=16, TILE_SIZE=1600:" >> /home/linaro/Work/sobel_parallel/results.txt
for i in `seq 1 5`; do
    ./sobel_parallel 16 1600 /home/linaro/Work/Images/320x240.jpg
done

echo "CORES=1, TILE_SIZE=3200:" >> /home/linaro/Work/sobel_parallel/results.txt
for i in `seq 1 5`; do
    ./sobel_parallel 1 3200 /home/linaro/Work/Images/320x240.jpg
done
```

Código 6.3. *script.sh*

```

echo "CORES=2, TILE_SIZE=3200:" >> /home/linaro/Work/sobel_parallel/results.txt
for i in `seq 1 5`; do
    ./sobel_parallel 2 3200 /home/linaro/Work/Images/320x240.jpg
done

echo "CORES=4, TILE_SIZE=3200:" >> /home/linaro/Work/sobel_parallel/results.txt
for i in `seq 1 5`; do
    ./sobel_parallel 4 3200 /home/linaro/Work/Images/320x240.jpg
done

echo "CORES=8, TILE_SIZE=3200:" >> /home/linaro/Work/sobel_parallel/results.txt
for i in `seq 1 5`; do
    ./sobel_parallel 8 3200 /home/linaro/Work/Images/320x240.jpg
done

echo "CORES=16, TILE_SIZE=3200:" >> /home/linaro/Work/sobel_parallel/results.txt
for i in `seq 1 5`; do
    ./sobel_parallel 16 3200 /home/linaro/Work/Images/320x240.jpg
done

echo "640x480 picture:" >> /home/linaro/Work/sobel_parallel/results.txt

echo "CORES=1, TILE_SIZE=800:" >> /home/linaro/Work/sobel_parallel/results.txt
for i in `seq 1 5`; do
    ./sobel_parallel 1 800 /home/linaro/Work/Images/640x480.jpg
done

echo "CORES=2, TILE_SIZE=800:" >> /home/linaro/Work/sobel_parallel/results.txt
for i in `seq 1 5`; do
    ./sobel_parallel 2 800 /home/linaro/Work/Images/640x480.jpg
done

echo "CORES=4, TILE_SIZE=800:" >> /home/linaro/Work/sobel_parallel/results.txt
for i in `seq 1 5`; do
    ./sobel_parallel 4 800 /home/linaro/Work/Images/640x480.jpg
done

echo "CORES=8, TILE_SIZE=800:" >> /home/linaro/Work/sobel_parallel/results.txt
for i in `seq 1 5`; do
    ./sobel_parallel 8 800 /home/linaro/Work/Images/640x480.jpg
done

echo "CORES=16, TILE_SIZE=800:" >> /home/linaro/Work/sobel_parallel/results.txt
for i in `seq 1 5`; do
    ./sobel_parallel 16 800 /home/linaro/Work/Images/640x480.jpg
done

echo "CORES=1, TILE_SIZE=1600:" >> /home/linaro/Work/sobel_parallel/results.txt
for i in `seq 1 5`; do
    ./sobel_parallel 1 1600 /home/linaro/Work/Images/640x480.jpg
done

echo "CORES=2, TILE_SIZE=1600:" >> /home/linaro/Work/sobel_parallel/results.txt
for i in `seq 1 5`; do
    ./sobel_parallel 2 1600 /home/linaro/Work/Images/640x480.jpg
done

```

Código 6.3. *script.sh*


```

echo "CORES=4, TILE_SIZE=1600:" >> /home/linaro/Work/sobel_parallel/results.txt
for i in `seq 1 5`; do
    ./sobel_parallel 4 1600 /home/linaro/Work/Images/640x480.jpg
done

echo "CORES=8, TILE_SIZE=1600:" >> /home/linaro/Work/sobel_parallel/results.txt
for i in `seq 1 5`; do
    ./sobel_parallel 8 1600 /home/linaro/Work/Images/640x480.jpg
done

echo "CORES=16, TILE_SIZE=1600:" >> /home/linaro/Work/sobel_parallel/results.txt
for i in `seq 1 5`; do
    ./sobel_parallel 16 1600 /home/linaro/Work/Images/640x480.jpg
done

echo "CORES=1, TILE_SIZE=3200:" >> /home/linaro/Work/sobel_parallel/results.txt
for i in `seq 1 5`; do
    ./sobel_parallel 1 3200 /home/linaro/Work/Images/640x480.jpg
done

echo "CORES=2, TILE_SIZE=3200:" >> /home/linaro/Work/sobel_parallel/results.txt
for i in `seq 1 5`; do
    ./sobel_parallel 2 3200 /home/linaro/Work/Images/640x480.jpg
done

echo "CORES=4, TILE_SIZE=3200:" >> /home/linaro/Work/sobel_parallel/results.txt
for i in `seq 1 5`; do
    ./sobel_parallel 4 3200 /home/linaro/Work/Images/640x480.jpg
done

echo "CORES=8, TILE_SIZE=3200:" >> /home/linaro/Work/sobel_parallel/results.txt
for i in `seq 1 5`; do
    ./sobel_parallel 8 3200 /home/linaro/Work/Images/640x480.jpg
done

echo "CORES=16, TILE_SIZE=3200:" >> /home/linaro/Work/sobel_parallel/results.txt
for i in `seq 1 5`; do
    ./sobel_parallel 16 3200 /home/linaro/Work/Images/640x480.jpg
done

echo "1280x960 picture:" >> /home/linaro/Work/sobel_parallel/results.txt

echo "CORES=1, TILE_SIZE=800:" >> /home/linaro/Work/sobel_parallel/results.txt
for i in `seq 1 5`; do
    ./sobel_parallel 1 800 /home/linaro/Work/Images/1280x960.jpg
done

echo "CORES=2, TILE_SIZE=800:" >> /home/linaro/Work/sobel_parallel/results.txt
for i in `seq 1 5`; do
    ./sobel_parallel 2 800 /home/linaro/Work/Images/1280x960.jpg
done

echo "CORES=4, TILE_SIZE=800:" >> /home/linaro/Work/sobel_parallel/results.txt
for i in `seq 1 5`; do
    ./sobel_parallel 4 800 /home/linaro/Work/Images/1280x960.jpg
done

```

Código 6.3. *script.sh*

```

echo "CORES=8, TILE_SIZE=800:" >> /home/linaro/Work/sobel_parallel/results.txt
for i in `seq 1 5`; do
    ./sobel_parallel 8 800 /home/linaro/Work/Images/1280x960.jpg
done

echo "CORES=16, TILE_SIZE=800:" >> /home/linaro/Work/sobel_parallel/results.txt
for i in `seq 1 5`; do
    ./sobel_parallel 16 800 /home/linaro/Work/Images/1280x960.jpg
done

echo "CORES=1, TILE_SIZE=1600:" >> /home/linaro/Work/sobel_parallel/results.txt
for i in `seq 1 5`; do
    ./sobel_parallel 1 1600 /home/linaro/Work/Images/1280x960.jpg
done

echo "CORES=2, TILE_SIZE=1600:" >> /home/linaro/Work/sobel_parallel/results.txt
for i in `seq 1 5`; do
    ./sobel_parallel 2 1600 /home/linaro/Work/Images/1280x960.jpg
done

echo "CORES=4, TILE_SIZE=1600:" >> /home/linaro/Work/sobel_parallel/results.txt
for i in `seq 1 5`; do
    ./sobel_parallel 4 1600 /home/linaro/Work/Images/1280x960.jpg
done

echo "CORES=8, TILE_SIZE=1600:" >> /home/linaro/Work/sobel_parallel/results.txt
for i in `seq 1 5`; do
    ./sobel_parallel 8 1600 /home/linaro/Work/Images/1280x960.jpg
done

echo "CORES=16, TILE_SIZE=1600:" >> /home/linaro/Work/sobel_parallel/results.txt
for i in `seq 1 5`; do
    ./sobel_parallel 16 1600 /home/linaro/Work/Images/1280x960.jpg
done

echo "CORES=1, TILE_SIZE=3200:" >> /home/linaro/Work/sobel_parallel/results.txt
for i in `seq 1 5`; do
    ./sobel_parallel 1 3200 /home/linaro/Work/Images/1280x960.jpg
done

echo "CORES=2, TILE_SIZE=3200:" >> /home/linaro/Work/sobel_parallel/results.txt
for i in `seq 1 5`; do
    ./sobel_parallel 2 3200 /home/linaro/Work/Images/1280x960.jpg
done

echo "CORES=4, TILE_SIZE=3200:" >> /home/linaro/Work/sobel_parallel/results.txt
for i in `seq 1 5`; do
    ./sobel_parallel 4 3200 /home/linaro/Work/Images/1280x960.jpg
done

echo "CORES=8, TILE_SIZE=3200:" >> /home/linaro/Work/sobel_parallel/results.txt
for i in `seq 1 5`; do
    ./sobel_parallel 8 3200 /home/linaro/Work/Images/1280x960.jpg
done

echo "CORES=16, TILE_SIZE=3200:" >> /home/linaro/Work/sobel_parallel/results.txt
for i in `seq 1 5`; do
    ./sobel_parallel 16 3200 /home/linaro/Work/Images/1280x960.jpg
done

```

Código 6.3. *script.sh*

```
$ cd /path/to/script.sh  
$ sudo chmod +x script.sh
```

Código 6.4. Asignación de permisos de ejecución

Por otro lado, se tendrá en cuenta que es en los accesos a memoria compartida donde se invierte la mayor parte del tiempo de ejecución referente al paralelismo en los *cores* del dispositivo Epiphany, especialmente en las acciones de lectura [48]. Por lo tanto, también se analizarán casos en los que únicamente se haga uso de un fragmento o *tile*, lo que implicaría un único acceso a memoria compartida para lectura de datos, y otro para escritura. Sin embargo, como se verá posteriormente, ello solamente es posible para los casos en los que el tamaño de la memoria local de los *cores* lo permita, correspondientes a la menor de las imágenes que se utiliza (320x240 píxeles o QVGA). Este análisis se llevará a cabo introduciendo manualmente el tamaño de *tiles* en cada ejecución, haciendo que se corresponda con el total de píxeles a procesar en cada *core*.

Para finalizar, se efectuará el análisis de las prestaciones correspondientes a la implementación de OpenCL que ofrece OpenCV internamente, concretamente para la función `cvtColor`. En este caso, también se obtendrá como resultado una media ponderada de 5 ejecuciones del programa realizadas manualmente, y se hará uso de dos de los tamaños o resoluciones de imagen considerados inicialmente: 640x480 píxeles (VGA) y 1280x960 píxeles (4x VGA). Para la selección de imagen, se modifica el código fuente en la función de OpenCV `imread`, detallando el *path* asociado a la ubicación de la imagen deseada, y volviendo a compilar el proyecto.

6.2 RESULTADOS PARA EL PROCESAMIENTO DE IMAGEN QVGA (320x240 PÍXELES)

Como imagen de resolución QVGA, se ha escogido una imagen correspondiente al Aeropuerto de Gran Canaria, la cual se reproduce en la Figura 6.1. Además, una vez ejecutado el algoritmo, en la Figura 6.2 se muestra la imagen obtenida como resultado de la aplicación del filtro de detección de bordes Sobel en la placa Parallella.



Figura 6.1. Imagen QVGA – Aeropuerto de Gran Canaria



Figura 6.2. Imagen QVGA tras la aplicación del filtro Sobel

6.2.1 RESULTADOS PRINCIPALES

Habiendo realizado todas las pruebas descritas anteriormente, se recopilaron los datos obtenidos en el fichero de texto con los resultados. A partir de los mismos, en la Tabla 6.1, en la Tabla 6.2 y en la Tabla 6.3, se recogen los resultados correspondientes al tiempo invertido en la ejecución del paralelismo computacional, mientras que en la Tabla 6.4, en la Tabla 6.5 y en la Tabla 6.6, se muestran los resultados análogos relativos al tiempo total de ejecución, para tamaños de *tile* de 800, 1600 y 3200 píxeles, respectivamente.

Tabla 6.1. Resultados temporales del paralelismo computacional para la imagen QVGA (1)

TILE_SIZE (píxeles)	800				
CORES	1	2	4	8	16
1 (s)	0,576761	0,443294	0,372385	0,360939	0,357252
2 (s)	0,556228	0,431681	0,360551	0,335963	0,357418
3 (s)	0,541427	0,437856	0,374708	0,37255	0,364091
4 (s)	0,550919	0,417027	0,373417	0,35351	0,343778
5 (s)	0,527326	0,418281	0,37397	0,343354	0,337271
AVERAGE (s)	0,5505322	0,4296278	0,3710062	0,3532632	0,351962
FRAME RATE (fps)	1,81642418	2,32759612	2,69537275	2,83075056	2,84121581

Tabla 6.2. Resultados temporales del paralelismo computacional para la imagen QVGA (2)

TILE_SIZE (píxeles)	1600				
CORES	1	2	4	8	16
1 (s)	0,478941	0,403628	0,362801	0,353529	0,342912
2 (s)	0,511124	0,422963	0,367593	0,365786	0,364201
3 (s)	0,489595	0,396033	0,365417	0,349124	0,332111
4 (s)	0,48766	0,3969	0,366247	0,348202	0,36127
5 (s)	0,485264	0,399149	0,364699	0,351078	0,350248
AVERAGE (s)	0,4905168	0,4037346	0,3653514	0,3535438	0,3501484
FRAME RATE (fps)	2,03866616	2,47687466	2,73709092	2,82850385	2,85593194

Tabla 6.3. Resultados temporales del paralelismo computacional para la imagen QVGA (3)

TILE_SIZE (píxeles)	3200				
CORES	1	2	4	8	16
1 (s)	0,466739	0,421986	0,350856	0,350008	0,348239
2 (s)	0,483752	0,397822	0,390375	0,337843	0,332645
3 (s)	0,47708	0,385453	0,367795	0,357769	0,342875
4 (s)	0,4672	0,390983	0,362506	0,343815	0,346414
5 (s)	0,472251	0,40453	0,373177	0,35316	0,341253
AVERAGE (s)	0,4734044	0,4001548	0,3689418	0,348519	0,3422852
FRAME RATE (fps)	2,1123589	2,49903287	2,7104546	2,86928403	2,92154028

Tabla 6.4. Resultados temporales de la ejecución total del programa para la imagen QVGA (1)

TILE_SIZE (píxeles)	800				
CORES	1	2	4	8	16
1 (s)	0,719683	0,581203	0,514847	0,537629	0,511455
2 (s)	0,696036	0,574936	0,503696	0,485245	0,505355
3 (s)	0,68551	0,60087	0,545187	0,511788	0,512303
4 (s)	0,694652	0,554163	0,506221	0,490296	0,513612
5 (s)	0,660516	0,564043	0,523897	0,508028	0,495327
AVERAGE (s)	0,6912794	0,575043	0,5187696	0,5065972	0,5076104
INCREASE (s)	0,1407472	0,1454152	0,1477634	0,153334	0,1556484
INCREASE (%)	25,5656617	33,8467855	39,8277441	43,4050306	44,2230695

Tabla 6.5. Resultados temporales de la ejecución total del programa para la imagen QVGA (2)

TILE_SIZE (píxeles)	1600				
CORES	1	2	4	8	16
1 (s)	0,624519	0,550163	0,507064	0,504599	0,483789
2 (s)	0,660424	0,558642	0,517926	0,509152	0,527418
3 (s)	0,626344	0,535344	0,520358	0,497595	0,484821
4 (s)	0,643743	0,520709	0,500784	0,488599	0,511345
5 (s)	0,619376	0,535233	0,506922	0,497558	0,519713
AVERAGE (s)	0,6348812	0,5400182	0,5106108	0,4995006	0,5054172
INCREASE (s)	0,1443644	0,1362836	0,1452594	0,1459568	0,1552688
INCREASE (%)	29,4310817	33,7557395	39,7588185	41,2839371	44,3437126

Tabla 6.6. Resultados temporales de la ejecución total del programa para la imagen QVGA (3)

TILE_SIZE (píxeles)	3200				
CORES	1	2	4	8	16
1 (s)	0,626049	0,559508	0,485172	0,490903	0,47872
2 (s)	0,622472	0,544891	0,553721	0,475181	0,463439
3 (s)	0,620851	0,541684	0,502092	0,51328	0,470112
4 (s)	0,608574	0,589701	0,516414	0,506664	0,477633
5 (s)	0,633182	0,575452	0,533095	0,497299	0,478149
AVERAGE (s)	0,6222256	0,5622472	0,5180988	0,4966654	0,4736106
INCREASE (s)	0,1488212	0,1620924	0,149157	0,1481464	0,1313254
INCREASE (%)	31,4363787	40,5074236	40,4283277	42,5074099	38,3672446

A partir de estos resultados, es posible observar que el tiempo de ejecución paralela en los *cores* del chip Epiphany está directamente relacionado, tanto con el tamaño de los fragmentos procesados en cada *core*, como con el número de *cores* utilizados. Como reflejan los resultados, en el caso de una imagen con un número de píxeles relativamente pequeño, como es el caso de la resolución QVGA, al llegar al uso de 8 y 16 *cores* para la implementación del paralelismo, se aprecia que la variación temporal es poco significativa. Ello demuestra que se está cerca del límite de dispositivos para paralelizar de manera eficiente en el que, aunque se sigan incluyendo más *cores* para la ejecución del código paralelizable, no se obtendrán mejoras temporales.

Para el tamaño de cada fragmento o *tile*, en el caso de la resolución de imagen QVGA, no se refleja una mejora evidente, ya que el número de accesos a memoria compartida no es significativo. Es en el uso de un único *core* del dispositivo Epiphany en el cual se aprecia notablemente el factor de aceleración alcanzado, pues en esta particularidad la cantidad de accesos a memoria compartida es mayor. Sin embargo, se muestra cómo el impacto que implica doblar el número de *cores* a utilizar es bastante mayor con respecto al que se obtiene doblando el tamaño de cada *tile*.

Analizando los tiempos totales se aprecia que, tal y como era de esperar, el tiempo invertido en las operaciones secuenciales desarrolladas en el dispositivo *host* es constante para todos los casos (con variaciones poco significativas), pues depende fundamentalmente del tamaño de la imagen.

6.2.2 RESULTADOS PARA UN ÚNICO FRAGMENTO O *TILE*

En el caso del estudio realizado para la ejecución del algoritmo haciendo uso de un solo fragmento o *tile* en cada *core*, las únicas posibilidades que han podido abordarse debido a la reducida memoria local de cada *core* del chip Epiphany, como ya se comentó, están relacionadas con la imagen de resolución QVGA. Estas posibilidades son dos y se corresponden con el uso de 8 y de 16 de los *cores* integrados en el dispositivo Epiphany. En estos dos casos a analizar, el único fragmento o *tile* tendrá un tamaño de 9600 píxeles y de 4800 píxeles, respectivamente, tal como se muestra en la Tabla 6.7.

Tabla 6.7. Resultados temporales del paralelismo computacional para el uso de un solo *tile*

CORES	8	16
1 (s)	0,337917	0,346101
2 (s)	0,329531	0,327263
3 (s)	0,345584	0,351686
4 (s)	0,339041	0,332368
5 (s)	0,358437	0,33705
AVERAGE (s)	0,342102	0,3388936
FRAME RATE (fps)	2,9231048	2,95077865

Al utilizar un único fragmento o *tile* en cada uno de los *cores* del chip Epiphany utilizados, el número de accesos a memoria compartida por parte de cada *core* se reduce a la unidad. Si se toma como referencia el caso en el que se utilizan *tiles* de 3200 píxeles, al utilizar 8 *cores* se realizan 3 accesos de este tipo, mientras que al usar 16 *cores* tienen lugar 2 accesos. Por lo tanto, estos resultados demuestran que, como ya se adelantó, los accesos a memoria compartida ralentizan la ejecución, pues con un único fragmento se obtiene un mejor factor de aceleración. Visto esto, si cada *core* ofreciera una mayor capacidad de memoria local, las mejoras temporales que podrían conseguirse tendrían un impacto significativo.

6.3 RESULTADOS PARA EL PROCESAMIENTO DE IMAGEN VGA (640x480 PÍXELES)

Para el caso del análisis de la imagen de tamaño VGA, se eligió una imagen de la Casa de Colón en Las Palmas de Gran Canaria, que se representa en la Figura 6.3. Por otro lado, en la Figura 6.4 es posible observar la imagen obtenida como resultado tras la aplicación del filtro de detección de bordes Sobel en la placa Parallella.



Figura 6.3. Imagen VGA – Plaza de Colón



Figura 6.4. Imagen VGA tras la aplicación del filtro Sobel

6.3.1 RESULTADOS

Tras llevar a cabo las pruebas pertinentes, y volcar los datos obtenidos en el archivo de texto que contiene los resultados, en la Tabla 6.8, en la Tabla 6.9 y en la Tabla 6.10, se incluyen los resultados obtenidos referentes al tiempo invertido en la ejecución del paralelismo computacional para tamaños de *tile* de 800, 1600 y 3200 píxeles, respectivamente. Por otra parte, en la Tabla 6.11, en la Tabla 6.12 y en la Tabla 6.13, se muestran los resultados obtenidos, relativos al tiempo total de ejecución.

Tabla 6.8. Resultados temporales del paralelismo computacional para la imagen VGA (1)

TILE_SIZE (píxeles)	800				
CORES	1	2	4	8	16
1 (s)	1,595193	0,955118	0,681714	0,479882	0,402761
2 (s)	1,618675	0,942086	0,643467	0,505834	0,413194
3 (s)	1,571249	0,980259	0,664848	0,505704	0,439902
4 (s)	1,621367	0,993788	0,663521	0,483845	0,421581
5 (s)	1,597146	1,005972	0,683096	0,482738	0,410484
AVERAGE (s)	1,600726	0,9754446	0,6673292	0,4916006	0,4175844
FRAME RATE (fps)	0,62471653	1,02517355	1,49851078	2,03417164	2,39472547

Tabla 6.9. Resultados temporales del paralelismo computacional para la imagen VGA (2)

TILE_SIZE (píxeles)	1600				
CORES	1	2	4	8	16
1 (s)	1,241922	0,812969	0,581018	0,46285	0,407222
2 (s)	1,266087	0,809153	0,553647	0,452141	0,392495
3 (s)	1,222623	0,822719	0,582493	0,438888	0,404825
4 (s)	1,273883	0,778243	0,558568	0,442667	0,406392
5 (s)	1,224928	0,792619	0,552412	0,461892	0,38748
AVERAGE (s)	1,2458886	0,8031406	0,5656276	0,4516876	0,3996828
FRAME RATE (fps)	0,80263998	1,245112	1,76794767	2,21391953	2,50198407

Tabla 6.10. Resultados temporales del paralelismo computacional para la imagen VGA (3)

TILE_SIZE (píxeles)	3200				
CORES	1	2	4	8	16
1 (s)	1,073175	0,707943	0,526533	0,457412	0,395812
2 (s)	1,065987	0,698634	0,511382	0,43498	0,385029
3 (s)	1,099533	0,772234	0,532063	0,442943	0,382799
4 (s)	1,075204	0,7242	0,518902	0,440123	0,404733
5 (s)	1,047186	0,711352	0,521778	0,425875	0,384918
AVERAGE (s)	1,072217	0,7228726	0,5221316	0,4402666	0,3906582
FRAME RATE (fps)	0,93264703	1,38336963	1,91522597	2,27135104	2,55978244

Tabla 6.11. Resultados temporales de la ejecución total del programa para la imagen VGA (1)

TILE_SIZE (píxeles)	800				
CORES	1	2	4	8	16
1 (s)	1,790518	1,135292	0,887546	0,652038	0,579359
2 (s)	1,800177	1,127605	0,832396	0,684294	0,616464
3 (s)	1,747221	1,160931	0,850958	0,671078	0,612648
4 (s)	1,797634	1,195178	0,843659	0,653697	0,603248
5 (s)	1,789081	1,19328	0,867731	0,654857	0,595654
AVERAGE (s)	1,7849262	1,1624572	0,856458	0,6631928	0,6014746
INCREASE (s)	0,1842002	0,1870126	0,1891288	0,1715922	0,1838902
INCREASE (%)	11,5072911	19,172037	28,3411546	34,9047987	44,0366546

Tabla 6.12. Resultados temporales de la ejecución total del programa para la imagen VGA (2)

TILE_SIZE (píxeles)	1600				
CORES	1	2	4	8	16
1 (s)	1,442187	1,000626	0,762059	0,642896	0,591285
2 (s)	1,457799	1,015833	0,72807	0,642158	0,575176
3 (s)	1,410079	0,998654	0,758981	0,640757	0,580281
4 (s)	1,473578	0,979909	0,7310,8	0,627634	0,608353
5 (s)	1,409655	0,974711	0,733931	0,663539	0,56948
AVERAGE (s)	1,4386596	0,9939466	0,74576025	0,6433968	0,584915
INCREASE (s)	0,192771	0,190806	0,18013265	0,1917092	0,1852322
INCREASE (%)	15,4725711	23,7574841	31,84651	42,4428742	46,3448014

Tabla 6.13. Resultados temporales de la ejecución total del programa para la imagen VGA (3)

TILE_SIZE (píxeles)	3200				
CORES	1	2	4	8	16
1 (s)	1,255415	0,882458	0,70691	0,647411	0,60874
2 (s)	1,262971	0,873481	0,681566	0,631431	0,560559
3 (s)	1,299799	0,956279	0,72573	0,639743	0,573609
4 (s)	1,26017	0,918456	0,702984	0,611358	0,602511
5 (s)	1,24747	0,903176	0,705951	0,60651	0,574659
AVERAGE (s)	1,265165	0,90677	0,7046282	0,6272906	0,5840156
INCREASE (s)	0,192948	0,1838974	0,1824966	0,187024	0,1933574
INCREASE (%)	17,9952379	25,4398078	34,9522228	42,4797157	49,4952877

Analizando los resultados obtenidos para el procesamiento de la imagen de resolución VGA, las mejoras temporales que introduce un aumento del número de *cores* utilizados, así como en el tamaño de cada *tile*, se hacen más significativas. Además, es posible advertir que se ha visto reducida la importancia que representaba un aumento en el número de *cores* con respecto a un incremento proporcional en el tamaño de cada fragmento o *tile*.

Al igual que en el caso anterior, el aumento del tiempo de ejecución correspondiente al código secuencial efectuado en el dispositivo *host* es constante, incluyendo pequeñas variaciones. Igualmente, este tiempo presenta un mayor retardo para una imagen VGA que para una imagen QVGA, pues el número de píxeles a procesar aumenta en un factor de 4.

6.4 RESULTADOS PARA EL PROCESAMIENTO DE IMAGEN 4x VGA (1280x960 PÍXELES)

La imagen de resolución 4x VGA escogida se corresponde con el Teatro Pérez Galdós en Las Palmas de Gran Canaria, la cual se reproduce en la Figura 6.5. En este caso, la imagen obtenida como resultado de la ejecución del programa correspondiente al filtro de detección de bordes Sobel en la placa Parallella puede verse en la Figura 6.6.



Figura 6.5. Imagen 4x VGA – Teatro Pérez Galdós



Figura 6.6. Imagen 4x VGA tras la aplicación del filtro Sobel

6.4.1 RESULTADOS

Al igual que en los casos anteriores, en la Tabla 6.14, en la Tabla 6.15 y en la Tabla 6.16, se muestran los resultados temporales asociados a la ejecución del paralelismo computacional para tamaños de *tile* de 800, 1600 y 3200 píxeles, respectivamente. Por otro lado, la Tabla 6.17, la Tabla 6.18 y la Tabla 6.19, contienen los resultados obtenidos relativos al tiempo total de ejecución.

Tabla 6.14. Resultados temporales del paralelismo computacional para la imagen 4x VGA (1)

TILE_SIZE (píxeles)	800				
CORES	1	2	4	8	16
1 (s)	8,203786	4,325475	2,392808	1,435626	0,996774
2 (s)	8,238899	4,308333	2,432383	1,431792	0,993475
3 (s)	8,16788	4,312407	2,434888	1,446501	0,989272
4 (s)	8,181833	4,347667	2,431442	1,447146	0,996406
5 (s)	8,221647	4,330655	2,382707	1,448123	1,011243
AVERAGE (s)	8,202809	4,3249074	2,4148456	1,4418376	0,997434
FRAME RATE (fps)	0,12190946	0,23121882	0,41410515	0,69355939	1,0025726

Tabla 6.15. Resultados temporales del paralelismo computacional para la imagen 4x VGA (2)

TILE_SIZE (píxeles)	1600				
CORES	1	2	4	8	16
1 (s)	5,404146	2,926013	1,711905	1,070908	0,810223
2 (s)	5,375537	2,974453	1,679262	1,073784	0,762999
3 (s)	5,44569	2,925331	1,739664	1,06383	0,780012
4 (s)	5,4752	2,930695	1,685769	1,116178	0,781321
5 (s)	5,394559	2,934308	1,74783	1,061804	0,818757
AVERAGE (s)	5,4190264	2,93816	1,712886	1,0773008	0,7906624
FRAME RATE (fps)	0,18453499	0,34034906	0,58381001	0,92824585	1,26476231

Tabla 6.16. Resultados temporales del paralelismo computacional para la imagen 4x VGA (3)

TILE_SIZE (píxeles)	3200				
CORES	1	2	4	8	16
1 (s)	4,056495	2,225628	1,341493	0,900762	0,716716
2 (s)	3,975854	2,267561	1,322821	0,877887	0,65447
3 (s)	3,984517	2,220577	1,346672	0,949219	0,721859
4 (s)	4,033805	2,243397	1,327632	0,923027	0,67576
5 (s)	4,024441	2,170793	1,36691	0,939376	0,70492
AVERAGE (s)	4,0150224	2,2255912	1,3411056	0,9180542	0,694745
FRAME RATE (fps)	0,24906461	0,44931881	0,74565344	1,08926031	1,43937704

Tabla 6.17. Resultados temporales de la ejecución total del programa para la imagen 4x VGA (1)

TILE_SIZE (píxeles)	800				
CORES	1	2	4	8	16
1 (s)	8,602234	4,689325	2,702432	1,783883	1,339612
2 (s)	8,596004	4,691132	2,803016	1,774151	1,337014
3 (s)	8,527971	4,721268	2,814646	1,822462	1,370616
4 (s)	8,577663	4,735904	2,819365	1,828139	1,390835
5 (s)	8,59488	4,682654	2,72232	1,771883	1,381952
AVERAGE (s)	8,5797504	4,7040566	2,7723558	1,7961036	1,3640058
INCREASE (s)	0,3769414	0,3791492	0,3575102	0,354266	0,3665718
INCREASE (%)	4,59527218	8,76664319	14,8046815	24,5704509	36,7514843

Tabla 6.18. Resultados temporales de la ejecución total del programa para la imagen 4x VGA (2)

TILE_SIZE (píxeles)	1600				
CORES	1	2	4	8	16
1 (s)	5,72949	3,283118	2,026967	1,396144	1,146775
2 (s)	5,718153	3,336756	2,056071	1,398338	1,076383
3 (s)	5,83859	3,277183	2,145688	1,392918	1,093525
4 (s)	5,89219	3,331134	2,069711	1,423257	1,121284
5 (s)	5,722817	3,320148	2,088733	1,441543	1,220522
AVERAGE (s)	5,780248	3,3096678	2,077434	1,41044	1,1316978
INCREASE (s)	0,3612216	0,3715078	0,364548	0,3331392	0,3410354
INCREASE (%)	6,66580255	12,6442331	21,2826773	30,9235081	43,1328719

Tabla 6.19. Resultados temporales de la ejecución total del programa para la imagen 4x VGA (3)

TILE_SIZE (píxeles)	3200				
CORES	1	2	4	8	16
1 (s)	4,378135	2,571728	1,731075	1,240559	1,067351
2 (s)	4,394632	2,67047	1,652811	1,209426	0,974103
3 (s)	4,350635	2,603875	1,718338	1,260299	1,043279
4 (s)	4,385841	2,607541	1,684608	1,272169	1,026266
5 (s)	4,391849	2,544376	1,703629	1,330101	1,048163
AVERAGE (s)	4,3802184	2,599598	1,6980922	1,2625108	1,0318324
INCREASE (s)	0,365196	0,3740068	0,3569866	0,3444566	0,3370874
INCREASE (%)	9,09574004	16,8048292	26,6188285	37,5202902	48,5195863

En cuanto a los resultados correspondientes al procesamiento de la imagen de resolución 4x VGA, sí que se aprecian diferencias temporales significativas entre los distintos escenarios, algo que se hace más visible en tanto que la imagen a procesar es mayor. Además, los resultados que se ofrecen para esta imagen muestran la influencia real del tamaño de cada fragmento o *tile*.

Como ya se ha comentado, el aumento del tiempo relativo al código secuencial efectuado en el dispositivo *host* es, aproximadamente, constante. Análogamente, este tiempo representa un mayor retardo que en los casos anteriores, al haber aumentado considerablemente el número de píxeles a procesar.

6.5 IMPLEMENTACIÓN OPENCV – OPENCL EN LA PLACA PARALLELLA

En el desarrollo de este TFG, se llevó a cabo un análisis del factor de aceleración de la implementación interna de OpenCL que ofrece OpenCV en algunas de sus funciones, concretamente para la función `cvtColor` en las imágenes de resolución VGA y 4x VGA. Por tanto, se reutilizarán las imágenes seleccionadas previamente con estas resoluciones, mostrando los resultados obtenidos de la ejecución de los programas secuencial y paralelo generados para este estudio.

6.5.1 RESULTADOS

En cuanto a la ejecución para la imagen de resolución VGA, en la Figura 6.7 se muestra el resultado de su transformación a escala de grises mediante la ejecución de esta función. Por otra parte, en la Tabla 6.20 se recogen los resultados temporales de la ejecución, tanto en serie como en paralelo, de esta conversión.



Figura 6.7. Imagen VGA tras la conversión a escala de grises

Tabla 6.20. Resultados temporales de la integración OpenCV-OpenCL para la imagen VGA

	SERIES	PARALLEL
1 (s)	0,011446	0,007686
2 (s)	0,011576	0,007704
3 (s)	0,011428	0,007815
4 (s)	0,011612	0,007041
5 (s)	0,011464	0,007262
AVERAGE (s)	0,0115052	0,0075016
FRAME RATE (fps)	86,91722004	133,304895
DECREASE (s)	0,0040036	
DECREASE (%)	34,79817822	

Del mismo modo, la Figura 6.8 representa el resultado de la conversión a escala de grises para la imagen de resolución 4x VGA, y análogamente, la Tabla 6.21 contiene los resultados referentes a su tiempo de ejecución, tanto serie como paralela.



Figura 6.8. Imagen 4x VGA tras la conversión a escala de grises

Tabla 6.21. Resultados temporales de la integración OpenCV-OpenCL para la imagen 4x VGA

	SERIES	PARALLEL
1 (s)	0,023004	0,019373
2 (s)	0,022948	0,018193
3 (s)	0,022838	0,018727
4 (s)	0,022229	0,018321
5 (s)	0,023483	0,01882
AVERAGE (s)	0,0229004	0,0186868
FRAME RATE (fps)	43,66735952	53,51371021
DECREASE (s)	0,0042136	
DECREASE (%)	18,39967861	

Como se observa, los resultados temporales de ambas imágenes informan acerca de la validez de la implementación interna de OpenCL que OpenCV realiza de manera transparente al usuario. Se trata de una funcionalidad bastante simple y que ofrece múltiples ventajas en sistemas con acceso a aceleradores externos, como es el caso de la placa Parallella y su dispositivo Epiphany. En esta ocasión, como se muestra en la Tabla 6.20 y en la Tabla 6.21, se alcanza un factor de aceleración del 34,8% y del 18,4%, respectivamente, para cada una de las imágenes, valores bastante significativos al tener en cuenta que se trata de un algoritmo que no conlleva demasiada carga computacional.

6.6 ANÁLISIS GENERAL DE LOS RESULTADOS

Tras haber analizado y comentado individualmente cada uno de los escenarios considerados en este análisis, se cree oportuno realizar un comentario general acerca de los resultados obtenidos, especialmente para los casos en los que se ha variado el número de *cores* a utilizar y el tamaño de cada fragmento o *tile* en el procesamiento paralelo ejecutado en el chip Epiphany.

Como se ha podido ver, estos dos parámetros presentan un impacto directo en el factor de aceleración que se alcanza, si bien es cierto que el número de *cores* es una característica con una repercusión más significativa que el tamaño de los *tiles*. Sin embargo, los resultados muestran que cuanto mayor sea el tamaño de la imagen a procesar a través del filtro de detección de bordes Sobel implementado, mayor es la importancia que tiene dicho tamaño de los *tiles*, ya que se incrementa el número de accesos a memoria compartida por parte de cada *core* del chip Epiphany. En todo caso, existe una limitación trascendental para este parámetro, la cual ha sido mencionada en varias ocasiones, y se trata de la memoria local de los *cores*.

Para concluir, dadas las imágenes resultantes, se destaca la validez de la aplicación paralela desarrollada para el filtro de detección de bordes Sobel, así como de la implementación conjunta de OpenCL y OpenCV en la placa Parallella.

Capítulo 7. CONCLUSIONES

En el procedimiento llevado a cabo en este TFG se han contextualizado y detallado los diferentes pasos llevados a cabo durante su realización. Llegado este punto, se expondrán las conclusiones generales a las que se llega a raíz de las acciones implementadas a lo largo del procedimiento descrito, así como la propuesta de futuras ampliaciones a realizar sobre el TFG desarrollado.

7.1 CONCLUSIONES GENERALES

Una vez obtenidos los resultados en los diferentes escenarios analizados para las aplicaciones de procesamiento de contenido visual basadas en el paralelismo que se han desarrollado, es posible confirmar la validez de las implementaciones realizadas, tanto para el filtro de detección de bordes Sobel generado, como para los programas que hacen uso del paralelismo interno basado en OpenCL que ofrece OpenCV. Por lo tanto, se consiguen varios de los objetivos principales del TFG, logrando integrar correctamente y utilizar conjuntamente el *framework* OpenCL y la librería OpenCV en la placa Parallella.

Además, el análisis de los resultados temporales efectuado ofrece una visión general del impacto directo sobre el factor de aceleración que representa cada uno de los parámetros examinados en la implementación del filtro de detección de bordes Sobel. Con ello, se han estudiado los patrones que sigue la respuesta temporal para modificaciones determinadas de estos parámetros, logrando establecer conclusiones de validez.

Por otra parte, teniendo en cuenta la integración de la placa *Raspberry Pi Camera* en la placa Parallella como un posible método de obtención directa de contenido visual en esta placa, se valora positivamente el procedimiento desarrollado, así como las soluciones que se plasman para diferentes problemas surgidos. Con este procedimiento y disponiendo de los recursos *hardware* necesarios, sería posible esta integración, si bien las circunstancias específicas de este TFG han impedido que esto se consiguiese.

También cabe destacar la consecución de la integración de las librerías OpenCL, distribuidas por la compañía *Brown Deer Technology* a través del SDK COPRTHR, en un entorno de desarrollo gráfico como *Code::Blocks*, algo que no se encuentra especificado por este distribuidor. Al instalar también la librería OpenCV en la placa Parallella y configurar el entorno *Code::Blocks* para su uso, se logra

implementar funciones de la librería OpenCV de forma paralela en la placa Parallella, para la cual no existe ningún soporte identificado en la documentación que proporciona *Adapteva*.

Finalizado el procedimiento llevado a cabo durante el desarrollo del presente TFG, y tras la experiencia adquirida con ello, se considera establecer las siguientes conclusiones:

- La placa Parallella es un dispositivo *hardware* flexible y que ofrece múltiples recursos orientados, fundamentalmente, al desarrollo de aplicaciones basadas en el paralelismo computacional. Si bien, uno de los principales problemas obtenidos a la hora de trabajar con esta plataforma, es la escasa memoria local de la que disponen los núcleos que integran su chip *multicore* Epiphany.
- Durante el procedimiento relativo a la integración de la placa *Raspberry Pi Camera* en la placa Parallella, se encontraron bastantes dificultades de diferente índole, como problemas de alineamiento a nivel de procesador y otros relacionados con la inexistencia de algunos comandos ligados a herramientas *software* específicas. Además, fue posible detectar y solucionar la corrupción de la memoria *flash* de la placa Parallella, quedando la misma inoperativa, algo que supuso una inversión considerable de esfuerzo y tiempo.
- Se ha logrado integrar correctamente el entorno de desarrollo *Code::Blocks* en la placa Parallella, tanto para OpenCL, como para OpenCV. Se trata de un entorno muy sencillo e intuitivo, pero también especialmente rápido a la hora de utilizarse en esta placa con respecto a otros entornos como *Eclipse*, que fue una de las opciones consideradas inicialmente.
- Tras analizar las diferentes metodologías de programación del paralelismo en la placa Parallella, se destaca la comodidad que ofrece el uso del *framework* OpenCL sobre las implementaciones basadas en comandos Epiphany. Así, se evita la necesidad de programar en detalle cada uno de los aspectos relativos a la comunicación entre el dispositivo Zynq y el chip Epiphany, así como la inicialización de este último.
- Se ha demostrado que la librería OpenCV puede ser instalada, de forma correcta, en la placa Parallella para su uso, representando un conjunto de funciones amplio y que cubre una gran cantidad de necesidades. En este caso, la documentación asociada a la placa Parallella no detalla la posibilidad de integrar y utilizar OpenCV en esta plataforma, algo que se ha

logrado con el desarrollo de este TFG. La organización de esta librería, estructurada en módulos, hace que resulte sencillo su uso. Además, la documentación que incluye sobre los diferentes módulos, clases y funciones que contiene, se encuentra bastante detallada y resulta de elevada utilidad.

- Se destaca la validez de la placa *Parallella* para efectuar un análisis de eficiencia del paralelismo, pues se ha llevado a cabo el análisis del factor de aceleración alcanzado, en función del número de *cores* del chip *Epiphany* utilizados y de la cantidad de accesos a memoria compartida de los mismos, durante la implementación de un filtro de detección de bordes Sobel que ha sido desarrollado.
- La documentación y el soporte que ofrece *Adapteva* para la placa *Parallella* no cubre todas las necesidades requeridas. En algunos casos, únicamente se ciñe al uso de los foros de *Parallella*, en los cuales no siempre se recibe una respuesta de utilidad, o simplemente una respuesta.

7.2 LÍNEAS FUTURAS

A continuación se destacan las líneas futuras de trabajo más importantes que se proponen tras la realización del presente TFG:

- Adaptar la aplicación para, una vez integrada la placa *Raspberry Pi Camera* en el modelo adecuado de la placa *Parallella*, aplicar el programa en tiempo real a fotogramas recogidos, en instantes de tiempo periódicos, desde la secuencia de vídeo que se capture con esta cámara.
- Integrar el uso del motor DMA que incluye cada *core* del chip *Epiphany* con el fin de agilizar las operaciones de lectura y escritura a memoria compartida, mejorando así el factor de aceleración.
- Extender el uso con la creación de más *kernels* relativos a otras implementaciones paralelas de algoritmos de procesamiento de contenido visual, por ejemplo, de la librería *OpenCV*, que no estén recogidos para su uso a través del *framework* *OpenCL*.

REFERENCIAS

- [1] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. Elsevier Science, 2011.
- [2] M. Sasikumar, D. Shikhare, and R. P. Prakash, *INTRODUCTION TO PARALLEL PROCESSING*. PHI Learning, 2014.
- [3] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU Computing," *Proc. IEEE*, vol. 96, no. 5, pp. 879–899, May 2008.
- [4] R. M. Hord, *Parallel Supercomputing in MIMD Architectures*. Taylor & Francis, 1993.
- [5] Parallella, "The Parallella Board," 2014. [Online]. Available: <https://www.parallella.org/board/>. [Accessed: 03-Jun-2016].
- [6] S. N. Agathos, A. Papadogiannakis, and V. V Dimakopoulos, "Euro-Par 2015: Parallel Processing: 21st International Conference on Parallel and Distributed Computing, Vienna, Austria, August 24-28, 2015, Proceedings," L. J. Träff, S. Hunold, and F. Versaci, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 662–674.
- [7] Parallella, "Parallella-1.x Reference Manual." 2014.
- [8] Parallella, "Parallella Models." [Online]. Available: <http://www.parallella.org/parallella-models/>. [Accessed: 25-May-2016].
- [9] Xilinx Inc., "Zynq-7000 All Programmable SoC - Technical Reference Manual." 2015.
- [10] Xilinx Inc., "Zynq-7000 All Programmable SoC." [Online]. Available: <http://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html>. [Accessed: 03-Jun-2016].
- [11] Adapteva Inc., "Epiphany Architecture Reference." 2011.
- [12] Adapteva Inc., "Epiphany SDK Reference." 2010.
- [13] Y. Sapir, "Epiphany SDK Driver and Library Sources Published on GitHub," 2013. [Online]. Available: <https://www.parallella.org/2013/02/22/esdk-source-tree-published-today/>. [Accessed: 25-May-2016].

- [14] Adapteva Inc., “Epiphany Host Library (eHAL),” in *Epiphany SDK Reference*, 2010, pp. 117–150.
- [15] Adapteva Inc., “Epiphany Hardware Utility Library (eLib),” in *Epiphany SDK Reference*, 2010, pp. 74–116.
- [16] A. Olofsson, “Meet ‘Porcupine’, the Parallella breakout board,” 2014. [Online]. Available: <https://www.parallella.org/2014/12/15/meet-porcupine-the-parallella-breakout-board/>. [Accessed: 24-May-2016].
- [17] Raspberry Pi Foundation, “Camera Module.” [Online]. Available: <https://www.raspberrypi.org/products/camera-module/>. [Accessed: 20-May-2016].
- [18] J. Hughes, “The Raspberry Pi camera,” *The MagPi Magazine*, no. 14, pp. 4–8, 2013.
- [19] Parallella, “Create SD card.” [Online]. Available: <http://www.parallella.org/create-sdcard/>. [Accessed: 24-May-2016].
- [20] O. Jeppsson, “Pubuntu ESDK releases.” [Online]. Available: <https://github.com/parallella/pubuntu/releases>. [Accessed: 24-May-2016].
- [21] MiniTool Solution Ltd., “MiniTool Partition Wizard 7.6.” [Online]. Available: <https://www.partitionwizard.com/>. [Accessed: 24-May-2016].
- [22] T. Davis and J. Davis, “Win32 Disk Imager.” [Online]. Available: <https://sourceforge.net/projects/win32diskimager/>. [Accessed: 24-May-2016].
- [23] S. Munaut, “‘proto’ directory,” 2015. [Online]. Available: <http://people.osmocom.org/~tnt/parallella/proto.tar.bz2>. [Accessed: 25-May-2016].
- [24] Xilinx Inc., “Getting started with Xilinx SDK.” [Online]. Available: http://www.xilinx.com/html_docs/xilinx2016_1/SDK_Doc/index.html. [Accessed: 20-May-2016].
- [25] Xilinx Inc., “Platform Cable USB II.” [Online]. Available: <http://www.xilinx.com/products/boards-and-kits/hw-usb-ii-g.html>. [Accessed: 24-May-2016].
- [26] A. Olofsson, “Parallella Firmware,” 2016. [Online]. Available:

- <https://github.com/parallella/parallella-hw/tree/master/archive/firmware>. [Accessed: 24-May-2016].
- [27] Antmicro Ltd, "Parallella Linux - quickstart guide." pp. 10–12, 2015.
- [28] A. Olofsson, "How to turn on the serial console on Parallella," 2015. [Online]. Available: <https://parallella.org/forums/viewtopic.php?f=49&t=3289>. [Accessed: 24-May-2016].
- [29] OmniVision Technologies Inc., "OV5647 datasheet." pp. 6–1, 2009.
- [30] Raspberry Pi Foundation, "Camera Module Usage." [Online]. Available: <https://www.raspberrypi.org/documentation/usage/camera/>. [Accessed: 24-May-2016].
- [31] F. Looijaard, M. D. Studebaker, and J. Delvare, "i2cdetect - Linux man page." [Online]. Available: <http://linux.die.net/man/8/i2cdetect>. [Accessed: 24-May-2016].
- [32] Adapteva Inc., "Introduction," in *Epiphany SDK Reference*, 2010, pp. 10–15.
- [33] The Code::Blocks Team, "Code::Blocks." [Online]. Available: <http://www.codeblocks.org/>. [Accessed: 23-May-2016].
- [34] B. K. Modak, *C++ Application Development with Code::Blocks*. Packt Publishing, 2013.
- [35] R. Tsuchiyama, T. Nakamura, T. Iizuka, A. Asahara, and S. Miki, *The OpenCL Programming Book*. Fixstars Corporation, 2010.
- [36] Brown Deer Technology, "STDCL API Reference v1.6." 2014.
- [37] N. Oppen, "Parallella Ideas, Desigs and Executions." [Online]. Available: <http://nicksparallellaideas.blogspot.com.es/>. [Accessed: 31-May-2016].
- [38] Brown Deer Technology, "Release Notes for the COPRTHR SDK version 1.6." 2014.
- [39] G. Bradski and A. Kaehler, *Learning OpenCV: Computer Vision with the OpenCV Library*. O'Reilly Media, 2008.
- [40] S. Brahmabhatt, *Practical OpenCV*. Apress, 2013.
- [41] Itseez Inc., "OpenCV modules," *OpenCV 3.1.0*, 2015. [Online]. Available: <http://docs.opencv.org/3.1.0/#gsc.tab=0>. [Accessed: 28-May-2016].

- [42] Itseez Inc., "OpenCL," *OpenCV Platforms*, 2016. [Online]. Available: <http://opencv.org/platforms/opencvcl.html>. [Accessed: 29-May-2016].
- [43] Itseez Inc., "OpenCV Downloads." [Online]. Available: <http://opencv.org/downloads.html>. [Accessed: 28-May-2016].
- [44] Itseez Inc., "Installation in Linux," *OpenCV 3.1.0*, 2015. [Online]. Available: http://docs.opencv.org/3.1.0/d7/d9f/tutorial_linux_install.html#gsc.tab=0. [Accessed: 28-May-2016].
- [45] Itseez Inc., "Using OpenCV with gcc and CMake," *OpenCV 3.0.0-dev documentation*, 2015. [Online]. Available: http://docs.opencv.org/3.0-last-rst/doc/tutorials/introduction/linux_gcc_cmake/linux_gcc_cmake.html#linux-gcc-usage. [Accessed: 28-May-2016].
- [46] S. Behera, M. N. Mohanty, and S. Patnaik, "A Comparative Analysis on Edge Detection of Colloid Cyst: A Medical Imaging Approach," in *Soft Computing Techniques in Vision Science*, S. Patnaik and Y.-M. Yang, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 63–85.
- [47] S. Clukey, "GitHub parallela-opencv-coprthr/opencv/," 2014. [Online]. Available: <https://github.com/sclukey/parallela-opencv-coprthr/tree/master/opencv>. [Accessed: 31-May-2016].
- [48] Coduin, "Memory speed," *Parallella Memory Details*, 2016. [Online]. Available: http://codu.in/ebps/docs/memory_details.html#memory-speed. [Accessed: 01-Jun-2016].

PLIEGO DE CONDICIONES

PLIEGO DE CONDICIONES

El procedimiento desarrollado, y los resultados obtenidos a lo largo del presente TFG son válidos para los modelos del *hardware* y las versiones del *software* que se indican en la Tabla PL.1 y en la Tabla PL.2, respectivamente.

RECURSOS *HARDWARE*

Tabla PL.1. Condiciones *hardware*

Equipo	Modelo
Pc o laptop	<ul style="list-style-type: none">• Tarjeta con interfaz USB• Lector de tarjetas SD
Placa Parallella	Versión <i>Kickstarter</i> del modelo <i>Embedded Platform</i> (Zynq Z-7020)
Placa Porcupine	Versión 2
Raspberry Pi Camera	Rev 1.3

RECURSOS *SOFTWARE*

Tabla PL.2. Condiciones *software*

Aplicación	Versión
Sistemas Operativos en PC o laptop	<ul style="list-style-type: none">• Ubuntu 14.04 LTE• Windows 10
Herramientas para la gestión de la tarjeta micro SD	<ul style="list-style-type: none">• MiniTool Partition Wizard Home Edition Free v7.6• Win32 Disk Imager v0.9.5
Herramientas de diseño y síntesis <i>hardware</i>	<ul style="list-style-type: none">• Xilinx SDK 2015.1• Xilinx ISE Design 14.4
Entorno de desarrollo	Code::Blocks v13.12
Librería OpenCV	OpenCV v3.1.0
Librerías OpenCL	STDCL y OCL (SDK COPRTHR v1.6)
Distribuciones de la imagen de la tarjeta micro SD para la placa Parallella	<ul style="list-style-type: none">• Imagen <i>headless</i> 2015.1 SO: Ubuntu 14.04• Imagen <i>hdmi</i> 2014.6 SO: Ubuntu 14.04

PRESUPUESTO

PRESUPUESTO

Durante la realización del presente TFG ha sido necesario utilizar diferentes recursos, tanto materiales como humanos, con el fin de desarrollar el procedimiento de integración de la placa *Raspberry Pi Camera* en la placa Parallella, así como la generación, validación y análisis de las aplicaciones creadas. En este apartado se recogen todos los costes asociados a estos recursos materiales y humanos que se han visto implicados en el proyecto.

La cuantía del trabajo elaborado se ha fijado según las indicaciones de contrataciones de la Universidad de Las Palmas de Gran Canaria. De este modo, el total del presupuesto se ha desglosado en las siguientes secciones, en las que se representan los distintos costes según su naturaleza.

- Coste de Recursos Humanos
- Coste de Recursos Hardware
- Coste de Recursos Software
- Coste Total

RECURSOS HUMANOS

Para determinar el coste asociado a los recursos humanos, se hace uso de las últimas tablas de honorarios publicadas el 4 de noviembre de 2010 por la Universidad de Las Palmas de Gran Canaria. Según éstas, el coste total de un Técnico superior, Diplomado o Ingeniero Técnico, con una dedicación de 20 horas semanales, asciende a 901,06€. Por lo tanto, se tiene que el coste aproximado asociado a una hora de trabajo será de 11,26€.

Para la realización del presente TFG, el número de horas invertidas se corresponde con el tiempo asociado a 12 créditos ECTS, es decir, 300 horas. Por lo tanto, el coste de recursos humanos se corresponde con:

$$\text{Honorarios (coste RRHH)} = 11,26\text{€/h} \cdot 300\text{h} = \mathbf{3.378,00\text{€}}$$

RECURSOS *HARDWARE*

La Tabla P.1 especifica los recursos *hardware* amortizables que han sido utilizados a lo largo del proyecto y sus costes asociados en función de su tiempo de uso con respecto al periodo de amortización, que se ha establecido en 3 años.

Tabla P.1. Coste de recursos *hardware* (I)

Recurso	Coste	Tiempo de uso	Amortización
Ordenador portátil Medion Akoya P66 MD99173	770,00€	4 meses	85,56€
TOTAL			85,56€

Por otro lado, en la Tabla P.2 se muestra el resto del material *hardware* utilizado y en el que, debido a su bajo precio, su amortización coincide con su valor de adquisición.

Tabla P.2. Coste de recursos *hardware* (II)

Recurso	Coste	Amortización
Placa Parallella Kickstarter Edition	87,09€	87,09€
Placa Porcupine 2	39,58€	39,58€
Raspberry Pi Camera Rev 1.3	30,74€	30,74€
Cableado y equipo auxiliar (monitor, ratón y teclado, router)	170,00€	170,00€
TOTAL		327,41€

Con ello, se realiza la suma de ambos y se obtiene el coste total de los recursos *hardware*, tal y como se recoge en la Tabla P.3.

Tabla P.3. Coste total de recursos *hardware*

Concepto	Coste
Coste de recursos <i>hardware</i> (I)	85,56€
Coste de recursos <i>hardware</i> (II)	327,41€
TOTAL	412,97€

RECURSOS SOFTWARE

De igual modo, los recursos *software* necesarios para puesta a punto del proyecto y sus costes derivados están reflejados en la Tabla P.4.

Tabla P.4. Coste de recursos *software*

Recurso	Tipo de licencia	Coste
MiniTool Partition Wizard Home Edition Free	Free	-
Win32 Disk Imager	Pública	-
Xilinx SDK	Free	-
Xilinx ISE Design	Free	-
Entorno Code::Blocks para programación C/C++	Pública	-
Mendeley Desktop	Pública	-
Microsoft Office 2016	Universitaria	79,00€
Minicom	Pública	-
OpenCV	Pública	-
SDK COPRTHR	Pública	-
TOTAL		79,00€

COSTE TOTAL DEL PROYECTO

Así, en la Tabla P.5 se refleja la suma de los costes originados de los distintos recursos utilizados, además de la aplicación del Impuesto General Indirecto Canario (IGIC), que es del 7%, dando lugar al coste total del proyecto.

Tabla P.5. Coste total

Recursos	Coste
Recursos humanos	3.378,00€
Recursos <i>hardware</i>	412,97€
Recursos <i>software</i>	79,00€
SUBTOTAL	3.869,97€
IGIC (7%)	270,90€
TOTAL	4.140,87€

El presupuesto total del Trabajo Fin de Grado “*Procesamiento de contenido visual utilizando la placa Parallella*” asciende a la cantidad de *cuatro mil ciento cuarenta euros con ochenta y siete céntimos* (4.140,87 euros).

Las Palmas de Gran Canaria, a 9 de Junio de 2016

Fdo.: D. Samuel Rodríguez Rodríguez

ANEXOS

ANEXO – CONTENIDO DEL CD-ROM

En este Anexo se presenta el contenido del CD-ROM adjunto a este documento, que se corresponde con la siguiente estructura:

- Memoria del TFG “*Procesamiento de contenido visual utilizando la placa Parallella*” en formato PDF.
- Directorio “*Proyectos*” en el que se encuentran los diferentes proyectos en *Code::Blocks* con el código fuente utilizado para la implementación de las aplicaciones de procesamiento de imágenes creadas. Este directorio comprende:
 - Subdirectorio “*Imágenes*” con las imágenes utilizadas en los distintos procedimientos descritos a lo largo de la memoria. Además, también se incluyen las imágenes resultantes de la aplicación de los diferentes algoritmos en la placa Parallella.
 - Subdirectorio “*RGB2Gray_CPU*” con el proyecto relativo a la aplicación que convierte una imagen del dominio RGB a escala de grises de forma secuencial en la CPU.
 - Subdirectorio “*RGB2Gray_Epiphany*” con el proyecto referente a la aplicación que convierte una imagen del dominio RGB a escala de grises de forma paralelizada en el chip Epiphany (bajo la implementación interna del *framework* OpenCL que proporciona OpenCV).
 - Subdirectorio “*sobel_parallel*” con el proyecto correspondiente a la implementación del filtro de detección de bordes Sobel de forma paralela en la placa Parallella.