

ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA



TRABAJO FIN DE GRADO

Diseño de un *Acelerador Hardware* FPGA para Aplicaciones de *Machine Learning* usando Plataforma Virtual

Titulación: Grado en Ingeniería en Tecnologías de la
Telecomunicación

Autor: Mario Daniel Guanche Hernández

Tutores: Dr. Pedro Pérez Carballo
Sonia Raquel León Martín

Fecha: Diciembre de 2020

ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA



TRABAJO FIN DE GRADO

Diseño de un Acelerador *Hardware* FPGA para Aplicaciones de
Machine Learning usando Plataforma Virtual

HOJA DE EVALUACIÓN

Calificación: _____

Presidente

Fdo.:

Vocal

Fdo.:

Secretario

Fdo.:

Fecha: Diciembre 2020

AGRADECIMIENTOS

Quisiera agradecer a mis tutores, Pedro Pérez Carballo y Sonia Raquel León Martín, quienes con su asesoramiento, tiempo y esfuerzo han facilitado la finalización de este trabajo. También, quiero agradecer a los encargados de soporte del Instituto Electrónico de Microelectrónica Aplicada la continua ayuda proporcionada para solucionar las diferentes incidencias que han surgido con las herramientas *software* empleadas en el transcurso de este proyecto y en definitiva a todas las personas que me han ayudado a conseguir este importante objetivo personal.

Resumen

En este Trabajo Fin de Grado (TFG) se pretende implementar y evaluar una aplicación de *machine learning* o aprendizaje automático [1]. Este concepto hace referencia a la rama dentro de la Inteligencia Artificial consistente en otorgar a una aplicación la capacidad de efectuar el análisis de los datos con el objetivo de identificar patrones. De esta manera, puede intuirse aspectos futuros relativos a una nueva entrada.

Para ello, se escoge el algoritmo *k-means*, el cual se define como un algoritmo de *machine learning* no supervisado. Los parámetros de entrada de este algoritmo consisten en una serie de elementos, los cuales son definidos por un mismo conjunto de atributos. La diferenciación de cada elemento se establece por medio de los valores que tomen sus atributos. El propósito del algoritmo será subdividir el conjunto de estos elementos en una cantidad fija de agrupaciones o clústeres especificada por el usuario. Estas agrupaciones se deberán realizar de acuerdo a las similitudes relativas de los elementos debido a sus valores de atributos, consiguiendo de manera simultánea la máxima diferenciación posible entre elementos que se encuentren en clústeres distintos.

La arquitectura de la aplicación estará conformada por un sistema híbrido que combinará funcionalidad *software* y aceleración *hardware* implementada sobre una placa de prototipado MPSoC Zynq UltraScale+, siendo ZCU102 el modelo concreto. La aceleración *hardware* será llevada a cabo por una serie de unidades de cómputo denominadas *kernels*. La comunicación entre la funcionalidad *software* y el acelerador *hardware* se efectuará por medio de interfaces AXI4 y AXI4-Lite.

Los recursos presentes en la placa de prototipado utilizada se subdividen en el sistema de procesamiento (PS) y la lógica programable (PL). Respecto al PS, este contendrá los microprocesadores que ejecutarán la funcionalidad *software*, así como las interfaces de la placa de prototipado. Por otra parte, la PL contiene los recursos relativos al *hardware* programable, los cuales implementarán la arquitectura *hardware* de los *kernels* del proyecto.

La tarea a realizar consistirá en verificar el funcionamiento del proyecto *k-means* utilizado. Para ello, se dispondrá de varios ficheros, cada uno de los cuales con distintos elementos de entrada al proceso de *clustering*, obteniéndose una serie de resultados de salida que serán posteriormente comparados con los resultados esperados y definidos en otro archivo.

Para la programación del sistema se utilizará el lenguaje C/C++. No obstante, la programación *hardware* se efectuará mediante una descripción de alto nivel realizada de manera indirecta en OpenCL. Para ello, se utilizarán una serie de *wrappers*, los cuales traducirán la

descripción *hardware* especificada mediante una serie de objetos y funciones desarrolladas para C++, en invocaciones de recursos de OpenCL.

La compilación de la aplicación se realizará mediante la ejecución de un archivo *makefile*. Esta compilación vendrá acompañada de un proceso de análisis de las distintas prestaciones temporales y de la ocupación de recursos del diseño, verificando además que se satisfagan las restricciones exigidas. Estos informes serán visualizados por medio de la herramienta Vitis Analyzer.

Abstract

This work is intended to implement and assess a machine learning application [1]. The machine learning concept refers to the branch within Artificial Intelligence consisting of enabling an application to perform data analysis in a search for patterns. As a result, the application can predict aspects about new entries.

To make this work, the *k-means* algorithm is chosen, which is regarded as an unsupervised machine learning algorithm. Its input parameters are a series of elements defined by the same set of attributes, but whose values differ from one element to the other. This algorithm aims to subdivide the set of elements into a number of clusters specified by the user. The elements in a same cluster must have attribute values as similar as possible. At the same time, the algorithm must achieve the maximum difference between elements belonging to different clusters.

The physical means where application is executed will be the MPSoC Zynq UltraScale+ prototyping board, with ZCU102 being the specific model. The application's architecture consists of a hybrid system which combines software and hardware acceleration functionalities. Hardware acceleration will be carried out by several computing units called kernels. Data transfers between software programming and hardware accelerator will be managed by the AXI4 and AXI4-Lite interfaces.

The board to use contains resources that are subdivided into processing system (PS) and user-programmable logic (PL). Regarding the PS, it contains the microprocessors where software functionality is executed, as well as the interfaces of the prototyping board. On the other hand, the PL contains resources related to programmable hardware, which will implement the hardware architecture of the project's kernels.

The task to be carried out will consist of verifying the operation of the *k-means* project. As input data, there will be several input files available. These files will contain input elements for the clustering process, obtaining a series of output results that will be later compared to the expected results defined in another file.

C/C++ will be used as programming language. However, the architecture for hardware accelerator will be programmed through a high-level description made indirectly in OpenCL. For this purpose, a series of wrappers will be used to translate the hardware description specified by objects and functions developed for C++ in calls to OpenCL resources.

The application's compilation will be done by running a makefile. This compilation comes together with an analysis of the design about its time performance and its use of PL resources. In this process, the hardware accelerator will be verified to comply with the constraints. These reports will be accessed through the Vitis Analyzer tool.

ÍNDICE

Memoria	1
Capítulo 1. Introducción	1
1.1. Antecedentes	1
1.2. Objetivos	2
1.3. Peticionario	3
1.4. Estructura del documento.....	3
Capítulo 2. Machine Learning	5
2.1. Introducción	5
2.2. Consideraciones preliminares	6
2.2.1. Información de entrenamiento.....	6
2.2.2. Búsqueda automatizada (<i>Hill Climbing</i>)	7
2.3. Clasificación mediante análisis probabilístico (Bayes)	8
2.4. Clasificación por vecino más cercano.....	12
2.5. Clasificadores lineales y polinómicos	14
2.5.1. Clasificadores binarios de ecuación lineal.....	15
2.5.2. Clasificación en más de dos grupos mediante múltiples clasificadores binarios.....	18
2.5.3. Determinación de ecuaciones polinómicas no lineales	18
2.6. Redes neuronales	21
2.6.1. <i>Multilayer perceptron</i>	22
2.6.2. <i>Radial-basis function network</i>	27
2.7. Aprendizaje no supervisado	30
2.7.1. <i>Clustering</i>	30
2.8. Conclusiones.....	35
Capítulo 3. Zynq UltraScale+ MPSoC	37
3.1. Introducción	37
3.2. Sistema de procesamiento.....	40
3.2.1. APU basada en ARM Cortex-A53.....	40

3.2.2.	Real-Time Unit ARM Cortex R5	42
3.2.3.	GPU ARM Mali-400.....	43
3.2.4.	Memoria PS	44
3.2.5.	Periféricos del PS	45
3.3.	PMU y CSU	45
3.4.	Operación de arranque	46
3.5.	Lógica programable	48
3.5.1.	CLBs	49
3.5.2.	DSP.....	50
3.5.3.	Memorias RAM.....	51
3.5.4.	Intercomunicación PS – PL	52
3.5.5.	Interfaz AXI	53
3.5.6.	EMIO	55
3.6.	Placa de prototipado	57
3.7.	Conclusiones.....	58
Capítulo 4.	Metodología de diseño.....	61
4.1.	Introducción	61
4.2.	Emulación software	63
4.3.	Emulación hardware.....	63
4.4.	Compilación para ejecución hardware.....	63
4.5.	Empaquetado del sistema	65
4.6.	Conclusiones.....	65
Capítulo 5.	Proyecto kmeans	67
5.1.	Introducción	67
5.2.	Arquitectura de la aplicación <i>kmeans</i>	67
5.3.	Función main	68
5.4.	fpga_kmeans	75
5.4.1.	fpga_kmeans_init.....	75
5.4.2.	fpga_kmeans_allocate.....	77
5.4.3.	fpga_kmeans_clustering	79

5.4.4.	fpga_kmeans_compute.....	80
5.4.5.	fpga_kmeans_deallocateMemory	86
5.4.6.	fpga_kmeans_print_report	87
5.5.	kmeans_clustering_cmodel.c.....	87
5.6.	krnl kmeans	91
5.6.1.	proc_memberships	93
5.6.2.	proc_new_centers.....	98
5.7.	timer.h.....	101
5.8.	krnl_kmeans.ini	104
5.9.	Conclusiones.....	104
Capítulo 6.	Implementación del diseño de la plataforma en Vitis	105
6.1.	Introducción	105
6.2.	Compilación del proyecto	106
6.3.	Adaptación a la placa ZCU102.....	107
6.4.	Construcción del diseño	109
6.4.1.	Análisis de la emulación software “sw_emu”	111
6.4.2.	Análisis de la emulación hardware “hw_emu”	113
6.4.3.	Arquitectura de la plataforma.....	119
6.4.4.	Análisis de la implementación “hw”	125
6.5.	Conclusiones.....	138
Capítulo 7.	Resultados	139
7.1.	Introducción	139
7.2.	Emulaciones software y hardware	139
7.3.	Ejecución sobre acelerador hardware Xilinx ZCU102	141
7.4.	Clasificación de imágenes hiperespectrales.....	144
7.5.	Comparativa ejecución software y con acelerador hardware	153
7.6.	Conclusiones.....	154
Capítulo 8.	Conclusiones y líneas futuras	155
8.1.	Conclusiones.....	155
8.2.	Líneas futuras	156

Bibliografía	159
Presupuesto	165
1.1. Costes recursos hardware	165
1.2. Costes recursos Software	166
1.3. Costes recursos humanos.....	166
1.4. Coste total	166
Pliego de Condiciones	171
1.1. Introducción	171
1.2. Recursos <i>Hardware</i>	171
1.3. Recursos <i>Software</i>	171

ÍNDICE DE FIGURAS

Figura 1: Arquitectura del dispositivo MPSoC Zynq UltraScale+ [4]	2
Figura 2: Ejemplo estados en el algoritmos de hill-climbing [7]	8
Figura 3: Clasificación mediante ecuación lineal (izquierda) y otra de grado superior (derecha) [8]	20
Figura 4: Estructuración de una red neuronal perceptron multicapa de 2 capas [8]	21
Figura 5: Red neuronal basada en funciones radiales [8]	28
Figura 6: Distribución de muestras en un espacio bidimensional [8]	32
Figura 7: Ejemplo de distribución de muestras donde k-means no sería eficaz [8]	34
Figura 8: Diagrama de bloques de la arquitectura Zynq UltraScale+ MPSoC [14]	39
Figura 9: Diagrama de bloques de la APU [15]	40
Figura 10: Vista general de la RPU [15]	42
Figura 11: Diagrama de Bloques de la GPU[20]	44
Figura 12: Conexionado entre las LUTs y los multiplexores de un CLB [25]	49
Figura 13: Diagrama de bloques del DSP [3]	50
Figura 14: Conexiones entre PS y PL [16]	52
Figura 15: Conexiones AXI PS – PL [16]	54
Figura 16: Placa de prototipado ZCU102	57
Figura 17: Configuración del switch SW6 para arranque desde la tarjeta SD	58
Figura 18: Flujo de diseño de la aplicación	62
Figura 19: Opciones de compilación de hardware programable [27]	64
Figura 20: Dependencias de ficheros de la aplicación	68
Figura 21: Grafo de llamadas de la función main	68
Figura 22: Grafo de llamadas de la función cluster().	72
Figura 23: Grafo de llamadas de la función fpga_kmeans_init	75
Figura 24: Grafo de llamadas de la función fpga_kmeans_allocate	77
Figura 25: Grafo de llamadas de la función fpga_kmeans_clustering	79
Figura 26: Grafo de llamadas de la función fpga_kmeans_compute	81
Figura 27: Grafo de llamadas de la función fpga_kmeans_deallocateMemory	86
Figura 28: Grafo de llamadas de la función fpga_kmeans_print_report	87
Figura 29: Grafo de llamadas de la función kmeans_clustering_cmodel	88
Figura 30: Grafo de llamadas de la función find_nearest_point	89
Figura 31: Grafo de llamadas de la función euclid_dist_2	90
Figura 32: Grafo de llamadas de la función kmeans_kernel_wrapper	91
Figura 33: Grafo de llamadas de la función load_clusters	94
Figura 34: Grafo de llamadas de la función load_features.	95
Figura 35: Grafo de llamadas de la función proc_new_centers	98
Figura 36: Grafo de llamadas de la función compute_new_centers	100
Figura 37: Grafo de llamadas de la función store_memberships	100
Figura 38: Grafo de llamadas de la función store_centers	101
Figura 39: Dependencias para time.h	102
Figura 40: Archivos que directa o indirectamente incluyen time.h	102
Figura 41: Informe “Summary” del apartado “krnl_kmeans (Software Emulation)”	111

Figura 42: Informe “Summary” del apartado “krnl_kmeans (Software Emulation)” (II)	112
Figura 43: Informe “Summary” del apartado “kmeans (Software Emulation)”	112
Figura 44: Informe Summary del apartado kmeans (Software Emulation) (II)	112
Figura 45: Confirmación de emulación software realizada sin incidencias.	113
Figura 46: Finalización de archivo Logs del subapartado kmeans (Software Emulation)	113
Figura 47: Informe “Summary” del apartado “krnl_kmeans (Hardware Emulation)” (I).....	114
Figura 48: Informe “Summary” del apartado “krnl_kmeans (Hardware Emulation)” (II).....	114
Figura 49: Diagrama del acelerador hardware elaborado por la opción “hw_emu”	115
Figura 50: Frecuencia de reloj y latencia de los módulos según “hw_emu”	115
Figura 51: Recursos PL requeridos según “hw_emu”, expuesto en informe “System Estimate” ..	116
Figura 52: Informe “System Guidance” obtenido tras la ejecución “hw_emu”	116
Figura 53: “Summary” del subapartado “kmeans (Hardware Emulation)”	117
Figura 54: “Summary” del subapartado “kmeans (Hardware Emulation)” (II)	117
Figura 55: Frecuencia de reloj y latencia del kernel según la emulación hardware	118
Figura 56: Recursos PL utilizados por un kernel según la estimación de la emulación hardware .	118
Figura 57. Arquitectura de la Plataforma	120
Figura 58: Incorporación del host en el diagrama de bloques	121
Figura 59: Diagrama de bloques dentro de ps_e	121
Figura 60: Introducción de los kernels en la arquitectura de la aplicación.....	122
Figura 61: Gestión de interrupciones de los kernels por el bloque AXI Interrupt Controller	123
Figura 62: Implementación de los bloques AXI Verification IP	124
Figura 63: Clocking Wizard	125
Figura 64: Bloque Processor System Reset	125
Figura 65: Informes sobre la ejecución de la opción hw	126
Figura 66: Parámetros de la compilación de la opción “hw”	126
Figura 67: Diagrama principal del acelerador hardware	127
Figura 68: Frecuencia de funcionamiento estimada de los distintos módulos.....	127
Figura 69: Latencia de los módulos (I).....	128
Figura 70: Latencia de los módulos (II).....	128
Figura 71: Recursos FPGA requeridos por cada módulo	128
Figura 72: Errores o incidencias en la implementación de los kernels	129
Figura 73: Contenido del apartado Timing Settings	129
Figura 74: Aspectos analizados en el apartado Check Timing.....	130
Figura 75: “Design Timing Summary” (I)	130
Figura 76: “Design Timing Summary” (II)	130
Figura 77: Clock Summary	131
Figura 78: Pulse Width Checks	131
Figura 79: Ejemplo de parte inicial de apartado “Max Delay Paths”	132
Figura 80: Informe “Summary” del kernel “kmeans”	134
Figura 81: Kernel Estimate	135
Figura 82: Informe de potencia generado en la compilación “hw”	136
Figura 83: Distribución de recursos del MPSoC en la implementación de kmeans.	137
Figura 84. Determinación de la ruta crítica según Vivado	138
Figura 85: Configuración del switch SW6 para arranque desde la tarjeta SD.....	142
Figura 86: Señalización de la placa de finalización del proceso de arranque	142

Figura 87: Habilitación de conexión USB para utilización OTG	143
Figura 88: Sistema implementado	143
Figura 89: Representación de imagen hiperespectral (izquierda) y de imagen RGB (derecha) [36]	145
Figura 90: Imágenes RGB de cáncer de piel obtenidas con un dermoscopio digital.	145

ÍNDICE DE TABLAS

Tabla 1: Características de los recursos hardware según la subfamilia (adaptada de [10])	38
Tabla 2: Componentes hardware en función de la subfamilia (adaptado de [10])	38
Tabla 3: Modos de arranque para la configuración del sistema [10]	47
Tabla 4: Cantidad de recursos de la parte PL [14][24]	48
Tabla 5: Periféricos I/O disponibles para MIO y EMIO [20]	56
Tabla 6: Informe HLS Synthesis generado al ejecutar la opción hw_emu	118
Tabla 7: Informe “Utilization”	133
Tabla 8: Informe HLS Synthesis: Latencias	135
Tabla 9. Informe HLS Synthesis: Recursos.....	135
Tabla 10: Características comunes a todos los casos.....	147
Tabla 11: Mapas de segmentación obtenidos en cada ejecución de kmeans (I). Los colores del mapa son aleatorios.....	149
Tabla 12: Mapas de segmentación obtenidos en cada ejecución de kmeans (II). Los colores del mapa son aleatorios.....	149
Tabla 13: Ejecución de la clasificación para 4 clusters en placa y en Matlab para la imagen P113_C1. Los colores del mapa son aleatorios.....	151
Tabla 14: Ejecución de la clasificación para 2 y 3 clusters en placa y en Matlab para las imágenes P15_C1 y P15_C2. Los colores del mapa son aleatorios.	151
Tabla 15: Ejecuciones de la clasificación para 2 clusters en placa y en Matlab. Los colores del mapa son aleatorios.....	152
Tabla 16: Comparativa entre la ejecución software y la ejecución con aceleradores hardware ..	154
Tabla 17: Presupuesto de los recursos hardware	165
Tabla 18: Presupuesto de los recursos software	166
Tabla 19: Presupuesto de los recursos humanos.....	166
Tabla 20: Presupuesto total	167

ÍNDICE DE ECUACIONES

Ecuación 1: Relación entre la probabilidad condicional y la probabilidad conjunta	9
Ecuación 2: Fórmula de Bayes.....	9
Ecuación 3: Relación entre la probabilidad del elemento y la de cada atributo si estos últimos son independientes	10
Ecuación 4: Evaluación de la probabilidad de pertenencia del elemento “x” a la clase “c”	10
Ecuación 5: Función de probabilidad influenciada por la frecuencia relativa y la presuposición del programador	11
Ecuación 6: Ecuación lineal de la frontera entre 2 clases	14
Ecuación 7: Ecuación del algoritmo Perceptron para la actualización de los parámetros w_i	15
Ecuación 8: Ecuación del algoritmo WINNOW para la actualización de los parámetros w_i	17
Ecuación 9: Ecuación polinómica de la frontera entre 2 clases	19
Ecuación 10: Relación entre la cantidad de parámetros w y las otras características de la ecuación polinómica.....	19
Ecuación 11: Error cuadrático medio	22
Ecuación 12: Función sigmoidea	23
Ecuación 13: Influencia de una neurona de salida en la equivocación del clasificador.....	25
Ecuación 14: Influencia en la equivocación del clasificador de una neurona que precede a las de salida	25
Ecuación 15: Ecuación de modificación de los parámetros “w”	25
Ecuación 16: Función de distribución gaussiana de dimensión “n” [8]	28

ÍNDICE DE CÓDIGOS

Código 1: Variables locales de la función main	69
Código 2: Establecimiento de los argumentos a especificar mediante el terminal de comandos ..	69
Código 3: Extracción de los parámetros indicados mediante la ejecución por línea de comandos	69
Código 4: Notificación de los clusters obtenidos y liberación de recursos	71
Código 5: Inicio de la función cluster	71
Código 6: Ejecución del proceso de clustering mediante funciones de fpga_kmeans	72
Código 7: Verificación mediante modelo software desarrollado en C	72
Código 8: Evaluación de la discordancia entre los resultados del modelo hardware-software y del modelo software del k-means.....	73
Código 9: Almacenamiento de los resultados de las agrupaciones de los elementos determinadas por el modelo en C de k-means	73
Código 10: Almacenamiento de los resultados de las agrupaciones de los elementos determinadas por el modelo hardware-software de k-means	74
Código 11: Comparación entre los resultados de la ejecución de k-means y los resultados esperados	74
Código 12: Liberación final de los recursos empleados en la función cluster	75
Código 13: Búsqueda de los dispositivos y extracción de archivo binario para la programación ...	76
Código 14: Programación de los dispositivos.....	76
Código 15: Implementación de los kernels mediante OpenCL	77
Código 16: Reserva de espacio de memoria para los registros de los kernels	78
Código 17: Creación de los objetos cl::Buffer	79
Código 18: Parámetros y variables creadas por la función “fpga_kmeans_clustering”	80
Código 19: Introducción del primer elemento de cada cluster en “fpga_kmeans_clustering”	80
Código 20: Escalamiento y reorganización de la información de los elementos a evaluar	81
Código 21: Función para el cálculo del factor de escalamiento.....	81
Código 22: Función para el escalamiento y reubicación del contenido de “feature”	82
Código 23: Programación de las señales de entrada a los kernels	82
Código 24: Transmisión host-kernel de la información de los elementos.....	83
Código 25: Función “scale_cluster”	83
Código 26: Intercambio de información entre host y kernels mediante OpenCL	84
Código 27: Ejecución de la funcionalidad del hardware programable mediante “kmeans_kernel_wrapper”	85
Código 28: Evaluación de los cambios en la membresía de los elementos	85
Código 29: Actualización de “new_centers_len” y “new_centers”	85
Código 30: Actualización de los centroides de los clusters y terminación de “fpga_kmeans_compute”	86
Código 31: Función “fpga_kmeans_deallocateMemory”	86
Código 32: Función “fpga_kmeans_print_report”	87
Código 33: Inicialización de “clusters” y “membership”	88
Código 34: Proceso de clustering en la función kmeans_clustering_cmodel (I).....	88
Código 35: Proceso de clustering en la función kmeans_clustering_cmodel (II).....	89

Código 36: Función “find_nearest_point”	89
Código 37: Función euclid_dist_2	90
Código 38: Finalización de la función kmeans_clustering_cmodel	90
Código 39: Función kmeans_kernel_wrapper.....	91
Código 40: Definición de la interfaz de los kernels	92
Código 41: División de la estructura del kernel en los submódulos “proc_memberships” y “proc_new_centers”	93
Código 42: Ejecución de “load_cluster” en “proc_memberships”	93
Código 43: Función “load_clusters”	94
Código 44: Carga y determinación de las agrupaciones de los elementos en el módulo “proc_memberships”	95
Código 45: Función “load_feature”	95
Código 46: Descripción del tipo de variable “point_dist_t”	96
Código 47: Inicialización de los elementos del array “pt”	97
Código 48: Etapa de clasificación de los elementos.....	97
Código 49: Transmisión de la clasificación de los elementos a “proc_new_centers”	98
Código 50: Creación e inicialización de las variables locales de “proc_new_centers”	99
Código 51: Distribución de la carga computacional del módulo “proc_new_centers”	99
Código 52: Programación de la función “compute_new_centers”	100
Código 53: Función “store_membership”	101
Código 54: Función “store_centers”	101
Código 55: Funciones para medir tiempos de ejecución	103
Código 56: Definición del tipo de variable “cPerfTimer”	103
Código 57: Contenido completo del archivo krnl_kmeans.ini	104
Código 58: Tareas principales del Makefile.....	106
Código 59: Sección del Makefile que restringe cualquier placa cuyo nombre contenga “ZC”	108
Código 60: Anulación de la restricción del Makefile sobre las placas cuyo nombre contenga “ZC”	108
Código 61: Sección del Makefile con parametrización del run_emulation.pl.....	108
Código 62: Modificación de la parametrización de run_emulation.pl.....	109
Código 63: Comando para la invocación de la compilación del proyecto	109
Código 64: Invocación de la aplicación “Vitis Analyzer”	110
Código 65: Ejecución de la emulación software.....	139
Código 66: Ejecución de la emulación hardware	139
Código 67: Resultados de ejecución de la emulación software.....	140
Código 68: Resultados de ejecución de la emulación hardware.....	141
Código 69: Comando para transferencia de los datos para la ejecución del proyecto a la tarjeta SD	141
Código 70: Resultado ejecución con los archivos de entrada “100” y “100.gold_c10”	144
Código 71: Escalado espacial de las imágenes en Matlab.....	145
Código 72: Selección de bandas espectrales de la imagen	146
Código 73: Redimensionamiento de las imágenes	146
Código 74: Generación de GoldenFiles en Matlab.....	146
Código 75: Generación de los mapas de segmentación	148

ÍNDICE DE GRÁFICAS

Gráfica 1: Representación de la ecuación sigmoidea en función del sumatorio [8].....	23
Gráfica 2: Función con más de un mínimo relativo [8]	26
Gráfica 3: Evolución de la función de Gauss con respecto a “ μ ” y “ σ ” en una dimensión [9]	28
Gráfica 4: Ejemplo de detección de clusters [8].....	30
Gráfica 5: Tiempos para el cálculo de las agrupaciones de las muestras y para la ejecución de la aplicación en placa	147

Memoria

Capítulo 1. Introducción

1.1. Antecedentes

Machine learning o aprendizaje automático [1] hace referencia a la rama dentro de la Inteligencia Artificial consistente en otorgar a una aplicación la capacidad de efectuar el análisis de los datos con el objetivo de identificar patrones. De esta manera, puede intuirse aspectos futuros relativos a una nueva entrada.

La idea de *machine learning* y de la AI se emplea en la tecnología actual. Ejemplos de ello son las múltiples aplicaciones de asistentes de voz tales como Siri en los terminales de Apple o Alexa desarrollada por Amazon. Por otra parte, el *machine learning* se encuentra en las recomendaciones personalizadas de contenido audiovisual realizadas por plataformas como *Netflix* o *Spotify*, ya que para ello es necesario analizar el historial de reproducciones del usuario [2].

El principal desafío que enfrenta la implementación de funcionalidad de *machine learning* en un sistema electrónico radica en que la fiabilidad de las configuraciones definidas depender de la cantidad de datos extraídos de las situaciones evaluadas y de la variedad de análisis estadísticos efectuados. Esto conlleva una capacidad de computación elevada, pero presenta la ventaja que puede ser paralelizable. Debido a ello, en este trabajo se plantea la implementación de esta funcionalidad utilizando la plataforma MPSoC Zynq UltraScale+ de Xilinx (Figura 1) como acelerador *hardware*. De esta manera, la paralelización puede lograrse utilizando tanto una estrategia *software* mediante procesamiento multiprocesador, como una estrategia basada en el paralelismo implícito de la implementación de una arquitectura *hardware* usando recursos de lógica programable [3].

Al igual que en el diseño de cualquier sistema computacional, una etapa fundamental en el proceso de diseño de una aplicación de *machine learning* es su verificación. Asimismo, una manera de reducir el costo tanto económico como temporal de dicho proceso es el uso de plataformas virtuales. Estos recursos consisten en sistemas que son capaces de desarrollar la funcionalidad de un sistema *hardware* sin necesidad de que este sea físicamente implementado, partiendo de las especificaciones de diseño indicadas por el usuario en un lenguaje de programación de alto nivel.

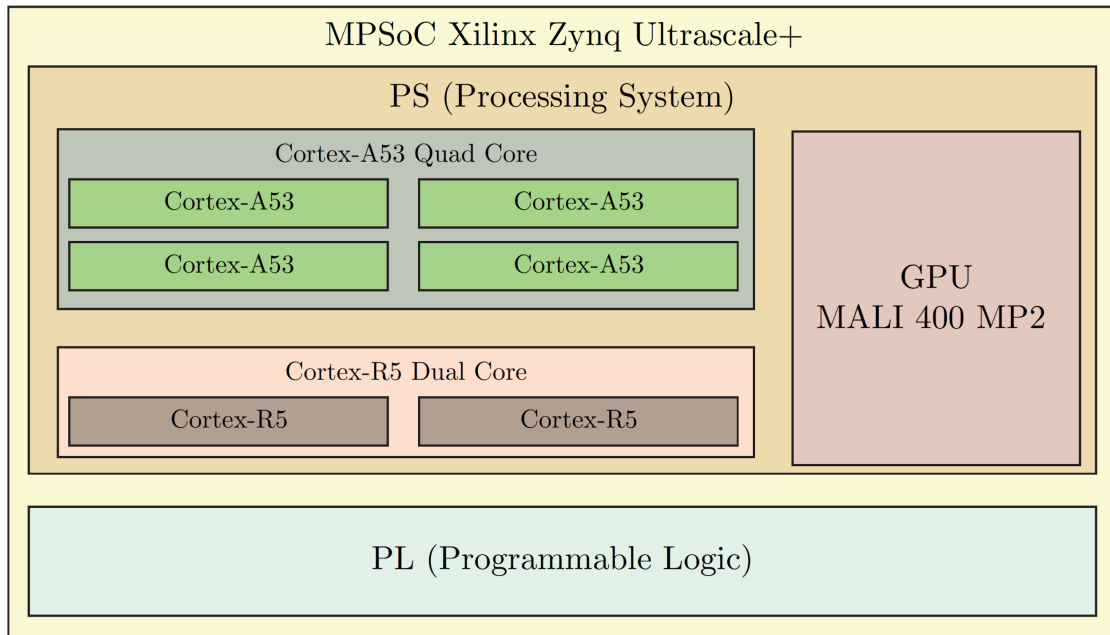


Figura 1: Arquitectura del dispositivo MPSoC Zynq UltraScale+ [4]

De esta manera, los diseñadores *hardware* podrán evaluar diferentes opciones de configuración del sistema, mientras que los programadores de *software* podrán ejecutar y depurar el código de ejecución antes de disponer de sistema físico [5].

Uno de los problemas que requieren un alto coste computacional es el procesamiento de imágenes hiperespectrales. En este trabajo se evalúa el diseño del acelerador *hardware* procesando un conjunto de 10 imágenes de cáncer de piel con una cantidad de 28 bandas espectrales que cubren el espectro visible e infrarrojo cercano (VNIR) desde 450 a 950 nm, con una resolución espacial de 10 x 10 píxeles.

1.2. Objetivos

El principal objetivo de este Trabajo Fin de Grado es el diseño de un sistema que combina funcionalidad *software* y arquitectura *hardware* destinado a la clasificación usando técnicas de *machine learning*. Para ello, se utiliza una plataforma virtual del dispositivo FPGA Zynq UltraScale+ de Xilinx basada en QEMU. Esta plataforma virtual permite verificar el funcionamiento del sistema formado por el sistema de procesamiento (PS) de Zynq y un modelo funcional de la aplicación de clasificación por *machine learning*. Finalmente, la implementación del algoritmo se ejecuta sobre la plataforma real para obtener resultados de su ejecución en un sistema hardware/software. Para lograr tal fin, es necesario definir los siguientes objetivos operativos o parciales:

- O1. Estudiar la arquitectura del dispositivo MPSoC FPGA Zynq UltraScale+ de Xilinx y sus principales componentes.
- O2. Estudiar diferentes alternativas de clasificación de *machine learning*.
- O3. Implementar una plataforma virtual para Zynq UltraScale+.
- O4. Simular el modelado del sistema de clasificación mediante *machine learning* en la plataforma virtual y estudiar las prestaciones de la arquitectura.
- O5. Documentar el trabajo realizado.

1.3. Peticionario

El Trabajo Fin de Grado ha sido ofertada por el Instituto Universitario de Microelectrónica Aplicada (IUMA), concretamente por la división de Sistemas Industriales y CAD (SICAD). Dicho trabajo resulta de gran interés, ya que abarca el concepto de *machine learning*, una temática en auge debido a su relevancia como vía para el desarrollo y perfeccionamiento de la Inteligencia Artificial (AI, *Artificial Intelligence*). Dicho campo, es uno de los grandes retos de la ingeniería en estos momentos por su capacidad de ser aplicada en distintos sectores de actividad.

Igualmente, actúa como petionario de este Trabajo Fin de Grado la Escuela de Ingeniería de Telecomunicación y Electrónica (EITE) de la Universidad de Las Palmas de Gran Canaria (ULPGC), siendo requisito para la obtención del título de Graduado en Ingeniería en Tecnologías de la Telecomunicación.

1.4. Estructura del documento

La estructura del presente documento es la siguiente. En este primer capítulo se presentan los objetivos de trabajo una vez contextualizado el problema.

En el capítulo 2 se explican los conceptos necesarios sobre *machine learning*, detallando las principales características que definen los algoritmos de *machine learning*. Se introducen algunas de las estrategias de *machine learning* aplicadas en la ingeniería actual, incluyendo los algoritmos de aprendizaje supervisado. Finalmente se introducen los conceptos de algoritmos de *clustering* que se utilizarán en el sistema para la clasificación.

En el capítulo 3 se explica en detalle la arquitectura de la placa de prototipado, sus recursos y las opciones de configuración disponibles. Los principales recursos incluyen el sistema de procesamiento compuesto con la unidad de procesamiento de aplicaciones (APU) y de tiempo real (RPU), el bloque PMU (*Platform Management Unit*) y CSU (*Configuration and Security Unit*),

destinados a la gestión del conjunto de recursos del MPSoC. Además, se explicarán los diferentes tipos de recursos de la parte de lógica programable (PL) del dispositivo MPSoC.

En el capítulo 4 se explica el flujo de diseño de referencia utilizado, poniendo énfasis en las tareas de diseño en alto nivel sobre Vitis.

En el capítulo 5 se analiza el código de referencia utilizado para la aplicación de *machine learning*. Como resultado de este análisis, se visualiza la algorítmica de la aplicación, así como algunas características especificadas con respecto a la arquitectura *hardware* diseñada. Este último aspecto se logra mediante directivas que controlan la generación de la arquitectura *hardware* utilizando herramientas de síntesis de alto nivel. Dichas directivas se utilizarán de forma exhaustiva para la compilación del proyecto.

Los aspectos principales de la compilación del diseño se explican en el capítulo 6. Se analiza la parametrización realizada, en función de la implementación objetivo utilizada: *software*, emulación *hardware* sobre la plataforma virtualizada o para la plataforma de prototipado *hardware* final. Todo ello se completa en el capítulo 7 con la ejecución de la aplicación final indicada, analizando los resultados obtenidos en términos de prestaciones, recursos y potencia utilizada. Finalmente se recogen las principales conclusiones del trabajo realizado y se plantean las posibles líneas de desarrollo futuro del trabajo.

Capítulo 2. Machine Learning

2.1. Introducción

Según Arthur Samuel, podemos definir una aplicación de *machine learning* como aquella que es capaz de aprender y mejorar su efectividad a partir de los datos recibidos. Para ello, se basa en el análisis estadístico para la realización de evaluaciones predictivas. Estas aplicaciones pueden ser clasificadas en tres categorías:

- a. **Aprendizaje supervisado.** Consiste en introducir una serie de muestras de datos que tienen un valor de salida preestablecido, algunos de los cuales se utilizan para entrenar al algoritmo, mientras que otros se emplean para verificar su funcionamiento. El entrenamiento permite al sistema determinar la salida que le correspondería a las entradas posteriores, aunque no se haya especificado dicha correspondencia. Un ejemplo de uso de este servicio se encuentra en la detección de *spam* en los mensajes de correo electrónico. El valor de salida puede ser la pertenencia a una determinada categoría, en el caso de que sea un sistema de clasificación, o un valor cuantitativo, en el caso de un sistema de regresión.
- b. **Aprendizaje sin supervisión.** Consiste en introducir datos que se encuentran sin clasificar en una aplicación de *machine learning* no entrenada. Asimismo, el algoritmo tiene como objetivo establecer posibles agrupaciones en función de características comunes en diversos elementos de entrada (*clustering*), o bien detectar posibles correlaciones entre las características de dichos elementos (*association*).
- c. **Aprendizaje mediante refuerzo.** Consiste en que el sistema aprende mediante la interacción con el entorno considerado. La manera de proceder de este algoritmo se puede enfocar desde una dinámica de recompensa y castigo. Cada acción tomada por el sistema origina, dependiendo de sus consecuencias, una valoración de su adecuación que puede ser negativa (castigo) o positiva (recompensa). De esta manera, el objetivo del algoritmo es maximizar siempre la recompensa, por lo que repetirá aquellas acciones que descubrió que reciben una mayor valoración positiva y evitará aquellas que presentaban una valoración negativa [6].

2.2. Consideraciones preliminares

A continuación, se van a estudiar distintos aspectos a considerar en función del tipo de entrenamiento y de la búsqueda de información con objeto de aumentar la precisión de las predicciones realizadas.

2.2.1. Información de entrenamiento

En el caso de los sistemas de aprendizaje supervisado, un aspecto decisivo para asegurar la mayor precisión en sus predicciones es la excelencia de los datos de entrenamiento. Sin embargo, pueden existir algunas imperfecciones en la información contenida en ellos. Asimismo, el sistema debe tener cierto grado de tolerancia ante estos errores que, en muchas ocasiones, pueden ser inherentes al proceso de medida.

- a. **Atributos irrelevantes.** Puede ocurrir que algunas de las características disponibles para los elementos de entrenamiento carezcan de importancia para la clasificación que se pretende establecer. Dicha situación supone un costo de computación adicional que resulta innecesario, pero normalmente ineludible en aquellos algoritmos en los que el proceso de determinación de los atributos se encuentre automatizado. La ausencia de estos atributos se establecerá en aquellos datos de entrenamiento que se elaboren manualmente.
- b. **Ausencia de atributos representativos.** Este tipo de circunstancia supone el problema inverso al de los atributos irrelevantes. Asimismo, la omisión de un atributo que se sospecha como no decisivo puede causar incongruencias en el criterio de clasificación desarrollado por el sistema si dispone de cierto grado de relevancia oculta.
- c. **Atributos redundantes.** Ocurre cuando se dispone de varios atributos que son deducibles entre sí. Un claro ejemplo de esta situación se observa cuando, considerando como datos de entrenamiento la información de un grupo de personas, 2 de los atributos considerados son la fecha de nacimiento y la edad. Sin embargo, su importancia con respecto a la funcionalidad del algoritmo es reducida en comparación con los dos casos anteriores.
- d. **Valor desconocido.** Puede ocurrir que el valor de alguno de los elementos para algunos de sus atributos se desconozca.
- e. **Valores de atributos erróneos.** Existen distintos factores tales como la poca fiabilidad de las fuentes de información, la imprecisión de los instrumentos de medida, la equivocación del usuario o erratas que originan que el valor de algunos de estos atributos pueda presentar cierto grado de error.
- f. **Categorías mal definidas.** Esto puede originar que las características de algunos de los elementos categorizados no se terminen de adecuar correctamente a las consideradas para

el grupo en el que se incluyen. En consecuencia, cualquier leve modificación en los atributos de dicho elemento puede ocasionar que este se clasifique en el grupo incorrecto.

2.2.2. Búsqueda automatizada (*Hill Climbing*)

Las técnicas *Hill Climbing* conforman una estrategia de búsqueda en la que se basan muchos sistemas de *machine learning* para realizar procesos de indagación de la solución buscada de manera automatizada. Para ello, la operación de búsqueda se define como un proceso de transición entre estados, partiendo de un estado inicial. Asimismo, el objetivo del algoritmo consiste en lograr que el aspecto tratado evolucione por medio de una serie de estados intermedios hacia un estado final que satisfaga los requisitos solicitados, determinando la secuencia de cambios que permiten esta transformación.

Durante el trascurso de este algoritmo se dispone de dos registros. En uno de ellos se almacenan todos los estados que no han sido evaluados pero que son accesibles desde un estado ya alcanzado con anterioridad. En el otro se almacenan todos los estados que el sistema ha evaluado, a los cuales no se debe regresar como consecuencia de una transición desde un estado futuro.

Las posibles transiciones de un estado actual al siguiente son establecidas por el conjunto de acciones entre las que se puede elegir. En el campo de la inteligencia artificial, dichas acciones se denominan operadores de búsqueda (*search operators*). Estos estados descubiertos como accesibles se introducen en el registro de estados en los que es posible posicionar el sistema, a excepción de aquellos que el algoritmo haya analizado previamente durante su ejecución. Posteriormente, se utiliza una función de evaluación, cuyo objetivo es determinar el siguiente estado a analizar entre los incluidos en el registro de estados accesibles. Este próximo estado será aquel que, según la función de evaluación, se aproxima en mejor medida al estado final deseado. Asimismo, se repetirá el mismo proceso sobre este nuevo estado actual que se efectuó sobre el estado predecesor, el cual será introducido en el conjunto de estados ya evaluados.

Desde el punto de vista ideal, el proceso de búsqueda automatizada terminaría encontrando un estado final que cumpla con todos los requisitos preestablecidos como criterios de búsqueda. Sin embargo, esto no siempre es posible. Debido a ello, es importante que el algoritmo de *Hill Climbing* implementado pueda sobrellevar la circunstancia de que el estado final ideal resulte inalcanzable. Para ello, se pueden aplicar distintos criterios de terminación del proceso de búsqueda. Por un lado, se tienen criterios relativos al gasto computacional implicado durante la ejecución del algoritmo como, por ejemplo, haberse superado el tiempo de ejecución máximo

otorgado al proceso o el número máximo de transiciones permitidas. Por otra parte, existen otros criterios que tienen en cuenta los resultados obtenidos en la ejecución del algoritmo. Asimismo, entre estos criterios se encuentra la consideración de haber llegado a un estado final que se asemeja suficientemente al deseado, o el hecho de que ya no queden próximos estados accesibles que evaluar.

En términos generales, se observa que, a medida que se suceden las transiciones de estado, el sistema se aproxima a las características deseadas. No obstante, habrá situaciones en las que el nuevo estado establecido se encuentre más alejado de los requisitos a cumplir que el estado predecesor, aunque sea la opción más cercana al estado final deseado desde la perspectiva de la función de evaluación considerando todos los estados accesibles no analizados. Pese a ello, este nuevo estado puede posibilitar que se encuentre un camino a otros estados cuya cercanía al resultado buscado sea mayor a la observada en cualquier estado anterior. Esta situación presenta cierta analogía con el proceso de escalar una montaña, donde a veces es necesario atravesar un valle descendiendo antes de retomar el ascenso a la cima. Este símil es el que le otorga el nombre *Hill Climbing* al algoritmo aquí estudiado.

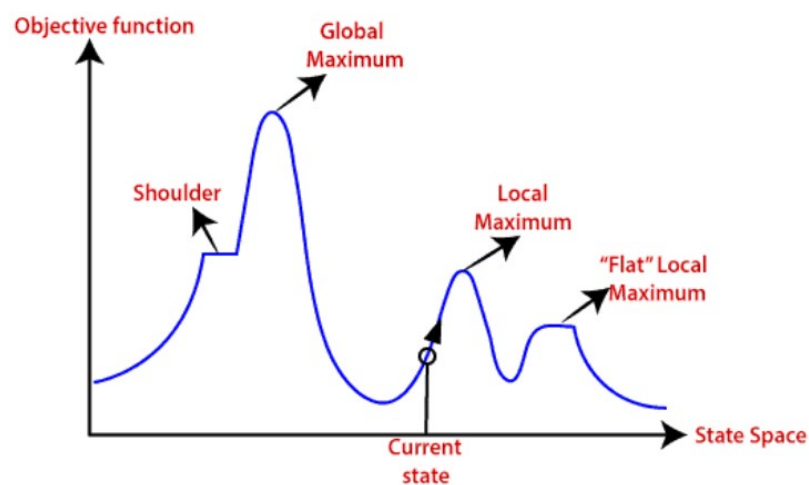


Figura 2: Ejemplo estados en el algoritmos de hill-climbing [7]

2.3. Clasificación mediante análisis probabilístico (Bayes)

El fundamento en el que se basa este sistema de clasificación consiste en estudiar qué clases contienen una cantidad más elevada de muestras con una serie de atributos de un determinado valor. De esta manera, aquellas muestras que cumplan con el patrón observado tendrán más posibilidades de pertenecer a esa clase en particular que a cualquier otra, siendo catalogados como tal.

Para entender este procedimiento, es necesario introducir los conceptos probabilidad condicional y la probabilidad conjunta. Ambos términos implican considerar una cantidad mínima de 2 características cualesquiera de los elementos estudiados como, por ejemplo, la pertenencia a una clase concreta o el valor de alguno de sus atributos. Asimismo, la probabilidad de que un elemento tenga una característica “x” condicionada con que posea la característica “x_{cond}”, sería equivalente a, dentro del conjunto elementos posibles, considerar que los elementos que cumplen con la característica “x_{cond}” son los únicos existentes, y dentro de este subconjunto evaluar la probabilidad de que un elemento aleatorio se adecúe a la característica “x”.

Por otra parte, la probabilidad conjunta de “x” y “x_{cond}” se refiere a la posibilidad de que ambas características se cumplan en un elemento aleatorio, considerando el total de elementos posibles. De esta manera, la probabilidad condicional “P(x|x_{cond})” y la probabilidad conjunta “P(x,x_{cond})”, se encuentran interrelacionadas, como se expone en la ecuación (1). A partir de dicha fórmula, se deduce la fórmula de Bayes que se expone en la ecuación (2).

$$P(x, x_{cond}) = P(x | x_{cond}) * P(x_{cond}) \quad (1)$$

$$P(x_{cond} | x) = \frac{P(x | x_{cond}) * P(x_{cond})}{P(x)} \quad (2)$$

Aplicando esta fórmula al problema de clasificación concreto, “x_{cond}” representaría la pertenencia a una determinada clase, mientras que “x” constituiría una o varias características de los elementos evaluados definidas como valores de sus atributos. De esta manera, si solo se tuviese en consideración un único atributo, se podría emplear la ecuación (2) para determinar, para cada clase, la probabilidad de que el elemento estudiado pertenezca a la misma. Con ello, se establece que la clasificación correcta de la muestra sería aquella en la que, dado el valor del atributo considerado, se consigue un valor mayor en el cálculo de la probabilidad.

Aunque en el caso de un único atributo, la ecuación (2) resulta insuficiente, y no es demasiado útil si se considera varios atributos. Esto se debe a que, al exigir un mayor número de coincidencias para el subconjunto de las muestras consideradas en el cálculo de la probabilidad condicional, la cantidad de muestras evaluables se reduce. En consecuencia, se carece de muestras suficientes para establecer correctamente la categoría del elemento introducido. Por lo tanto, sería necesaria una fórmula en la que la influencia ponderativa ejercida por cada característica del elemento de entrada fuese independiente de las demás. Para ello, es posible aplicar la transformación indicada en la ecuación (3), siempre que los atributos sean independientes entre sí.

En dicha ecuación, “x” simboliza el valor del conjunto de atributos que representa al elemento de entrada, mientras que “ x_i ” sería el valor de uno de sus atributos de manera específica. Asimismo, “c” simbolizaría la pertenencia a una clase concreta.

$$P(x | c) = \prod_{i=1}^n P(x_i | c) \quad (3)$$

Esto permite que se pueda medir, de manera comparativa, la probabilidad de pertenencia de un elemento de entrada “x” a cualquier clase presente en la clasificación, utilizando para ello la fórmula de la ecuación (4). De esta manera, la clase establecida para el elemento introducido será aquella en la que la ecuación ofrezca un valor más elevado.

$$P(x, c) = P(c) * \prod_{i=1}^n P(x_i | c) \quad (4)$$

El aspecto clave del funcionamiento de este método de clasificación recae en la fiabilidad de las funciones de determinación de la probabilidad. La extracción de los resultados probabilísticos se establece principalmente mediante la frecuencia relativa de las características buscadas dentro del conjunto de muestras de entrenamiento. De esta manera, en las versiones más simples del proceso de deducción de la probabilidad, su valor y el de la frecuencia relativa se consideran idénticos. Esto quiere decir que, si dentro del conjunto o subconjunto de elementos que representan el total para la función de probabilidad existen 100 elementos y solo 20 de ellos cumplen con las características evaluadas, el resultado devuelto en este caso por la función de probabilidad es de 0.2. Sin embargo, existen circunstancias con respecto al banco de entrenamiento del clasificador que afectan a la utilidad de las funciones de probabilidad de manera particular en comparación con otros sistemas de clasificación. Estas son las siguientes:

- a. **Conjunto de muestras de entrenamiento demasiado escaso.** Esta situación ocasiona que exista cierta facilidad en que la frecuencia relativa no plasme el valor de la probabilidad para todas las características a evaluar. Como posible ejemplo de ello se puede citar las probabilidades de cara o cruz en el lanzamiento de una moneda al aire, que pueden considerarse iguales para ambas opciones. Asimismo, si solo se lanza la moneda 4 veces, resulta fácilmente factible que en lugar de obtenerse 2 veces cada una de las opciones, de manera que la frecuencia relativa y la probabilidad coincidiesen, una de las caras aparezca en 3 ocasiones. Este suceso originar, si solo se tuviese en cuenta la frecuencia relativa, que

una cara tuviera una probabilidad, de acuerdo con el análisis estadístico, de 0.75, y la restante un 0.25.

Si no es posible disponer de una mayor cantidad de muestras de entrenamiento y el programador no está de acuerdo con la probabilidad evidenciada por la frecuencia relativa, se le puede dar cierta influencia a las probabilidades presupuestas por este individuo. Además, esta influencia es parametrizable mediante una variable denominada “m” que, a efectos prácticos, simbolizaría a medida que su valor es más elevado, un mayor grado de certeza que el programador confía tener en sus afirmaciones. La combinación del valor de la frecuencia relativa y de la probabilidad supuesta se efectuaría por medio de la ecuación mostrada en la ecuación (5), donde “N_{all}” es la cantidad de elementos considerados, “N_x” la cantidad de elementos dentro del conjunto que cumplen con el patrón “x” y “π_x” el valor de probabilidad estimado por el programador. Como resultado se obtiene la probabilidad P_x.

$$P_x = \frac{N_x + m * \pi_x}{N_{all} + m} \quad (5)$$

- b. **Interdependencia de los atributos.** Como se comentó con anterioridad, este método de clasificación basado en la ecuación de Bayes requiere que los atributos considerados sean independientes. Esta condición suele no cumplirse con cierta frecuencia en los atributos involucrados en un sistema de clasificación cualquiera. En el caso de que la correlación entre 2 o más atributos sea leve, esta sería fácilmente aproximable como nula. En dicha situación, la precisión del cálculo probabilístico quedaría comprometida. Sin embargo, ello no implicaría que el sistema siga resolviendo correctamente el problema de clasificación.

Para el caso de que la correlación sea muy significativa y, por lo tanto, su relevancia no pueda ser ignorada, se pueden transformar los elementos a clasificar. Esto supondría una serie de cambios como la eliminación de los atributos redundantes o la combinación de atributos. Asimismo, un ejemplo de redundancia sería la obtenida si, entre los atributos del elemento, se considera la fecha de nacimiento y la edad de una misma persona, ya que ofrecen la misma información desde perspectivas diferentes. Debido a ello, sería necesario excluir a uno de estos dos atributos con el objetivo de que el sistema opere correctamente. Respecto a la combinación de atributos, esta consiste en que, si se tienen una serie de atributos que aportan diferente información, por lo que no son perfectamente deducibles una con respecto a la otra, pero que aun así presentan cierta correlación, entonces pueden

sustituirse estos atributos por uno nuevo en el que se contenga la información total aportada por ambos.

- c. **Atributos de naturaleza continua.** El problema de esta situación radica en que las funciones de probabilidad vistas hasta ahora se basan fundamentalmente en dividir la cantidad de elementos encontrados que presentan el patrón buscado entre el total de elementos evaluados. Para ello, se requiere que los atributos dispongan de un conjunto finito de valores, aspecto que no caracteriza a una variable continua, ya que siempre dispondrá de infinitud de valores intermedios.

Para lidiar con esta situación, una alternativa consiste en discretizar los valores de los atributos continuos. Este hecho se traduce en segmentar el rango de valores del atributo continuo en una serie de intervalos. De esta manera, si dos elementos ubican el valor del atributo en un mismo intervalo, se considerará, a nivel de cálculo, que sus valores son coincidentes.

Otra estrategia consiste en reemplazar la función de probabilidad por la función de densidad de probabilidad. Para ello, se supone la disponibilidad de una cantidad infinitamente grande de muestras de entrenamiento. A diferencia de las funciones de probabilidad anteriores, los resultados de la función de densidad de probabilidad son de carácter continuo. Además, el resultado otorgado será más elevado cuanto mayor sea el grado de similitud con una mayor cantidad de muestras para el patrón observado.

2.4. Clasificación por vecino más cercano

El principio en el que se apoya, en su versión más simple, consiste en la categorización del elemento de entrada en el mismo grupo que el elemento que presente un mayor grado de similitud con él. Dicha similitud quedará establecida en función de los valores de sus atributos, cuyas semejanzas entre sí se podrán valorar con distintas estrategias, dependiendo de si estos son de naturaleza discreta o continua.

Una posible estrategia en el caso de atributos de valor discreto, que consiste en contar la cantidad de atributos entre dos elementos cuyos valores coinciden, aumentando la semejanza considerada entre estos a medida que se encuentran más coincidencias. Por otra parte, para el caso de atributos de valores continuos, la estrategia más acertada consiste en medir la cercanía geométrica, considerando cada atributo como una coordenada distinta dentro de un espacio vectorial en el que se posicionan los elementos. Esta estrategia puede utilizarse también cuando en

el conjunto de atributos a considerar, existen tanto atributos de valor continuo como atributos de valor discreto. Para el caso de los atributos discretos, la pauta a seguir es añadir un sumando de valor más elevado cuanto más significativa sea la falta de similitud entre los valores comparados, lo cual será establecido a criterio del programador. Cabe destacar que el adecuado escalamiento de los atributos es un aspecto muy importante para asegurar que unos pocos atributos no eclipsen la relevancia de otros en el cálculo de la distancia geométrica por cuestiones de magnitud.

La ecuación más comúnmente conocida que puede aplicarse al cálculo del distanciamiento entre dos elementos y, por ende, de su grado de similitud suele ser la distancia de Euclides, es decir, el cálculo del módulo que une los dos puntos en los que se encuentran los elementos a comparar. Sin embargo, existen otras fórmulas que pueden adecuarse mejor al problema de clasificación concreto, ya que representan de manera más precisa la notoriedad que se le quiera dar a la evolución del distanciamiento entre dos elementos en función de las posibles divergencias en valor de sus atributos. Independientemente de la fórmula aplicada, esta deberá cumplir con las siguientes condiciones.

1. La distancia nunca puede ser negativa.
2. La distancia entre dos vectores idénticos debe valer 0.
3. Debe cumplir la propiedad conmutativa, es decir, que la distancia desde un elemento de referencia al otro será equivalente a la calculada si la referencia se considera en sentido inverso.
4. Debe cumplir la desigualdad triangular. Esto quiere decir que, sea " x_1 " y " x_2 " los elementos a comparar e " y_1 " un punto de referencia cualquiera, la suma de la distancia de " x_1 " y " x_2 " con respecto a y_1 adquirirá un valor igual o mayor a la distancia existente entre " x_1 " y " x_2 ".

Cabe destacar que, debido al cierto margen de error que puede presentarse tanto en el valor de los atributos como en la clasificación de los elementos estudiados, confiar únicamente en el elemento más cercano para categorizar el elemento entrante puede dar resultados relativamente poco fidedignos. Por este motivo, es aconsejable valorar un mayor número de elementos cercanos a la hora de clasificar el objeto de entrada si el margen de error así lo justifica. Sin embargo, la cantidad de elementos vecinos a establecer no debe incrementarse de manera indefinida, ya que, en función de lo significativa que sea con respecto al total de los elementos de referencia tomados, puede resultar contraproducente.

En el caso de la clasificación con una cantidad plural de elementos cercanos, la categoría del elemento de entrada se establece, siguiendo la versión más simple del algoritmo, en aquel grupo en el que se haya detectado una mayor cantidad de elementos identificados como cercanos.

Sin embargo, se puede optimizar la eficacia del algoritmo de *sorting* si la relevancia de los elementos cercanos se parametriza en función del distanciamiento que presente con respecto al elemento entrante, siendo inversamente proporcionales.

La utilidad del sistema de clasificación depende del banco de elementos de entrenamiento usado como referencia. Para ello, es necesario filtrar aquellos elementos considerados perjudiciales, ya que pueden inducir a una clasificación errónea, así como aquellos que sean redundantes, pues simplemente añaden carga computacional de manera bastante innecesaria.

Para el caso de muestras que generan clasificación incorrecta, estas se limitan a aquellas que se encuentran en una región fronteriza y a aquellas que se encuentran rodeadas de muestras de una categoría distinta a la propia. Para la detección de estas, un procedimiento muy utilizado es el de *Tomek Links* [8], el cual las identifica como parejas de muestras que son las más cercanas entre sí y pertenecen a distintas clases. Por otra parte, para eliminar muestras redundantes, la estrategia consiste en desarrollar un subconjunto del banco de muestras de entrenamiento y comprobar si las muestras excluidas son clasificadas correctamente basándose en dicho subconjunto. En los casos afirmativos, se constata que la muestra es redundante. En los casos contrarios, constituye una muestra a considerar dentro del subconjunto, por lo que se incorpora y se repite el proceso para el resto de las muestras descartadas.

2.5. Clasificadores lineales y polinómicos

Al igual que el caso previo, se parte de una serie de muestras de referencia ubicadas dentro de un espacio vectorial. Dichas muestras se utilizan como referencia para delimitar zonas de pertenencia a un tipo concreto de clase, de manera que las muestras entrantes son clasificadas en función de en qué zona del espacio vectorial se encuentran localizadas. Para ello, el aspecto fundamental consiste en establecer correctamente las fronteras entre las regiones, las cuales se definen matemáticamente mediante ecuaciones lineales o polinómicas, cuyo grado será más elevado cuanto más sofisticadas sean las fronteras trazadas. En el caso de la ecuación lineal, esta quedará definida de acuerdo con la ecuación (6), en la que “ x_i ” hace referencia a los atributos que definen a las muestras clasificadas y “ w_i ” representa la ponderación otorgada a cada una, la cual puede adquirir valores tanto positivos como negativos. Por otra parte, “ $-w_0$ ” constituiría el valor umbral [8].

$$w_0 + \sum_{i=1}^n w_i x_i = 0 \quad (6)$$

Asimismo, el objetivo de la aplicación de clasificación será determinar el valor de los parámetros w . De esta manera, dependiendo del valor del conjunto de atributos de la muestra, la ecuación tomará un valor positivo o negativo, perteneciendo a una clase distinta en cada caso. Una situación peculiar que puede ocurrir es la de la muestra cuyos valores de atributos le otorgan un valor nulo, lo que representaría que la muestra se encuentra justo en la frontera entre las dos clases. En dicha circunstancia, la determinación de a cuál de las dos clases pertenece suele establecerse bien aleatoriamente o bien al grupo mayoritario del conjunto de entrenamiento.

2.5.1. Clasificadores binarios de ecuación lineal

En el caso de que la frontera a establecer sea únicamente entre dos clases y pueda expresarse mediante una ecuación lineal, la determinación de la misma resulta relativamente simple. Asimismo, existen principalmente dos procedimientos para ello, los cuales se basan en la modificación recurrente de los parámetros w . Uno de ellos es el procedimiento de *Perceptron*, consistente en sumar o restar una cantidad determinada. Por otra parte, se encuentra el procedimiento *WINNOWER* que emplea la multiplicación y la división.

2.5.1.1. Método *Perceptron*

Consiste en que, partiendo de unos valores iniciales de los parámetros de ponderación y del valor umbral relativamente pequeños, se evalúa si la ecuación categoriza correctamente a las muestras en sus respectivas clases y, en caso contrario, se modifican los factores de ponderación de los atributos involucrados y el valor umbral. De esta manera, si el valor obtenido con la ponderación actual de los atributos es negativo para una muestra de entrenamiento, pero se le presupone positivo, entonces se efectuará el incremento de los parámetros w que participasen en el cálculo. En el caso opuesto, si se obtiene un resultado positivo, pero debió ser negativo, se decrementarán dichos parámetros. En ambos casos, el incremento o decremento de cada parámetro " w_i " será directamente proporcional al valor que posea el atributo " x_i " al que se le asocia, como se expone en la ecuación (7). Esta ecuación es capaz de operar tanto con atributos que toman valores discretos como continuos. Sin embargo, es conveniente que dichos valores se encuentren normalizados, es decir, que varíen entre 0 y 1.

$$w_i = w_i + \eta[c - h]x_i \quad (7)$$

Como se observa en la ecuación anterior, la actualización de " w_i " dependerá, aparte del valor del atributo que corresponda, de la coincidencia entre la clase a la que según la ecuación pertenece la muestra (" c ") y la clase que se le presupone en el banco de entrenamiento (" h "). Para

el correcto funcionamiento de la ecuación es necesario que, tanto “c” como “h”, adquieran el valor 1 para indicar que la muestra pertenece a la clase positiva, es decir que el resultado obtenido o a obtener mediante la ecuación (6) debe ser superior a 0. Por otro lado, ambos parámetros adquirirán el valor cero en el caso contrario de que consideren que la muestra pertenece o debería pertenecer a la clase negativa. Asimismo, no se efectuará ninguna modificación de “ w_i ” si su atributo correspondiente tiene valor nulo, o si los valores de “c” y “h” son coincidentes en el caso de una muestra determinada.

Por último, el parámetro “ η ” de la ecuación (7) se denomina índice de aprendizaje (*learning rate*). Su valor permanecerá constante durante todo el proceso del algoritmo, siendo este un valor positivo, no nulo e inferior a la unidad que deberá ser escogido por el diseñador del algoritmo. A nivel gráfico, un mayor índice de aprendizaje supondrá una evolución más brusca de la ecuación lineal.

El proceso se reitera en cada muestra de entrenamiento hasta que se logre que el resultado devuelto por la aplicación y el especificado para una misma muestra coincidan, momento en el cual se procede a la siguiente. Durante este proceso puede ocurrir que el ajuste de los parámetros de ponderación y del valor umbral para una muestra específica origine incongruencias en los resultados para las muestras anteriormente analizadas. Por este motivo, es necesario que, una vez se han verificado todas las muestras, se repita el proceso para corregir desajustes, comenzando así una nueva etapa. Estas etapas se sucederán hasta conseguir que la categorización establecida por la ecuación lineal y la predispuesta coincidan para todas las muestras.

El número de etapas que conlleve el proceso es indefinido, ya que, por un lado, dicho valor suele ser linealmente proporcional al número de atributos que definen las muestras. Por otra parte, también dependerá de manera aleatoria de los valores de inicialización de los parámetros w_i , así como del valor del índice de aprendizaje establecido. No obstante, independientemente del valor escogido para dichas variables, el procedimiento de Perceptron logrará establecer la frontera entre 2 clases siempre que estas sean separables mediante una función lineal [8].

2.5.1.2. Método WINNOW

Consiste en ajustar los valores de ponderación de los atributos, que inicialmente tendrán valor unitario, por medio de la multiplicación o división de un factor constante “ α ” de valor mayor a la unidad. Asimismo, si el resultado devuelto por el clasificador para una muestra dada es negativo, pero esta se encuentra preclasificada como positiva, se aumentará el valor de los parámetros de ponderación de los atributos que participen en el cálculo mediante la multiplicación.

En el caso opuesto, es decir, si el clasificador ofrece un resultado positivo para una muestra determinada, pero se la considera con resultado negativo, se aplicará la división por el mismo factor. El valor de este factor queda a criterio del programador, y no es posible determinar un valor óptimo para el problema de clasificación concreto más allá del proceso experimental. Sin embargo, suelen conseguirse buenos resultados cuando este es dos. La fórmula utilizada corresponde con la ecuación (8), la cual se describe utilizando, a excepción del factor “ α ”, la misma nomenclatura empleada para la ecuación del algoritmo de *Perceptron* previamente comentada [8].

$$w_i = w_i \alpha^{c-h} \quad (8)$$

Un aspecto importante de la ecuación (8) en el que difiere de la ecuación (7) es la ausencia del atributo “ x_i ” en la descripción de la primera. Sin embargo, es indispensable en el algoritmo de *WINNOW* que la modificación de cualquier parámetro de ponderación no se realice de manera independiente del valor del atributo al que corresponda en la muestra. Esta situación resulta fácil de lidiar en el caso de atributos que adquieran valores binarios, considerando que, si el valor de “ x_i ” es 0 para la muestra dada, no deberá modificarse el valor de su parámetro “ w_i ”. En el caso contrario, sí podrá efectuarse dicha modificación siempre que la ecuación no haya determinado una correcta clasificación de la muestra. Para el caso de atributos que adquieran valores no binarios, la adecuación de la modificación o no de sus parámetros “ w_i ” se establecerá dependiendo de si superan o no un determinado valor umbral. Para el caso de atributos de valor normalizado entre 0 y 1, este valor umbral podría ser el valor intermedio 0.5, de manera que, si el atributo no supera en el caso de la muestra estudiada dicho valor, su parámetro “ w_i ” quedará inalterado.

Otra característica peculiar que distingue a los dos procedimientos estudiados para la obtención del clasificador lineal es que en el algoritmo *WINNOW* el valor umbral se mantiene constante, siendo este igual al número de atributos menos 0.1. Por otra parte, se encuentra el hecho de que, dado que el valor inicial de los parámetros de ponderación de los atributos es 1, y el factor “ α ” es positivo, nunca será posible obtener valores negativos en algunos de estos parámetros. Para sobrellevar esta circunstancia, se debe duplicar el número de atributos utilizados, de manera que, por cada atributo inicial se tendrá un atributo que tomará siempre los mismos valores que el primero, y otro que tomará el valor binario opuesto. Esto último quiere decir que, si el primer atributo toma el valor 1, el otro tomará el valor 0 y viceversa. Una de las principales utilidades características que presenta este algoritmo de clasificación es la rapidez con la que es capaz de reducir el valor de los parámetros de ponderación de aquellos atributos que sean irrelevantes.

2.5.2. Clasificación en más de dos grupos mediante múltiples clasificadores binarios

Para poder abordar la categorización en varios grupos, una forma de proceder sería utilizar una agrupación de varias ecuaciones lineales, encargándose cada una de ellas de delimitar la zona del espacio vectorial perteneciente a una clase concreta. Cada una de estas ecuaciones es extraída utilizando cualquier algoritmo de clasificación binario, como el de *Perceptron* o *WINNOWER* vistos anteriormente, por lo que heredan el requisito de que todas las clases sean separables. Como es evidente, dado la naturaleza binaria de cada uno de los clasificadores individuales, es necesario realizar una categorización del valor esperado para las muestras de entrenamiento específica para cada clasificador. De esta manera, a cada clasificador se le indica que el valor esperado para una muestra concreta es positivo si se encuentra dentro de la clase delimitada por su ecuación y negativo en el caso contrario.

Si se considera que cada posible caso de entrenamiento pertenecerá siempre a una única clase, entonces debe ocurrir, desde el punto de vista matemático, que cada muestra introducida sea clasificada como positivo según una de las ecuaciones lineales y como negativo para el resto de los clasificadores. Sin embargo, puede ocurrir que los casos de entrenamiento sean insuficientes y poco adecuados para determinar con suficiente exactitud la región que pertenece a una categoría concreta. Además, puede suceder que las regiones de algunas clases no puedan ser separables del resto del espacio vectorial mediante una ecuación lineal. Debido a estas incidencias, es posible que regiones de distintas clases se intercepten entre sí, lo que se traduce en que algunas muestras de entrada puedan ser catalogadas en más de una clase. Para tratar con esta situación, se utiliza un clasificador maestro que simplemente establece la clase a la que pertenecerá la muestra estudiada como aquella para la cual la ecuación lineal aplicada otorga el valor positivo más elevado.

Aunque este procedimiento de categorización en varias clases presenta una relativa sencillez, solo es fiable cuando la cantidad de clases existentes es reducida, aproximadamente entre 3 y 5. Esto se debe a que para una cantidad mayor de clases habría demasiada desproporción entre las muestras catalogadas como positivas y las clasificadas como negativas dentro de una clase concreta, ya que las últimas serían mucho más abundantes. Esto origina que la correcta clasificación de las muestras obtenida mediante este procedimiento sea difícil en aquellas situaciones en las que los valores de los atributos presenten cierto grado de error.

2.5.3. Determinación de ecuaciones polinómicas no lineales

Con cierta frecuencia, la separación del área perteneciente a una clase específica dentro del espacio vectorial no es expresable de manera precisa con una ecuación lineal. Debido a ello, es

necesario definir la frontera de la región en la que se ubica la clase mediante ecuaciones de grados más elevados. Sin embargo, es posible establecer cierta analogía entre una ecuación de un grado cualquiera y una ecuación lineal, ya que la primera puede expresarse como se indica en la ecuación (9)

$$\sum_{i=0}^n w_i z_i = 0 \quad (9)$$

Como puede observarse la ecuación (9) es idéntica a la expuesta en la ecuación (6) si se considera que “ z_0 ” será igual a 1, condición cumplida también en las ecuaciones no lineales para definir un término “ w_0 ”, independiente de cualquiera de los atributos, para establecer el valor umbral. La diferencia fundamental entre ambas ecuaciones radica en los términos z de la ecuación (9), los cuales no hacen referencia a un solo atributo. En su lugar, dichos términos se definen como productos de una cantidad de atributos que se encuentra entre 0 y el grado considerado para la ecuación polinómica, pudiendo existir tanto atributos que se repitan como atributos que no existan en la ecuación concreta de cada uno de los términos z .

Asimismo, el procedimiento para obtener la ecuación polinómica partiendo de los procedimientos utilizados para ecuaciones lineales, consistiría en utilizar un multiplicador para extraer el valor de los términos z que se obtiene para una muestra determinada debido al valor de los atributos de este último. Posteriormente, se pueden aplicar los mismos procedimientos de obtención de clasificadores lineales estudiados anteriormente, considerando los términos z como los “nuevos atributos” de cada muestra. Finalmente, se obtendrá los parámetros w de la ecuación polinómica que delimita el área de una clase, siempre que el polinomio escogido y definido a partir del grado mismo sea capaz de describir matemáticamente dicha región.

Aunque las ecuaciones polinómicas ofrezcan mayores posibilidades de delimitar la frontera de una clase cuanto más elevado sea el grado considerado en la misma para el proceso computacional, también presentan algunas desventajas. Asimismo, la desventaja más evidente es el aumento significativo del número de parámetros w (“ N_w ”) a calcular cuanto mayor es el grado de la ecuación (“ r ”) y el número de atributos considerados (“ n ”). Esta relación puede representarse matemáticamente mediante la ecuación (10)

$$N_w = \frac{(n+r)!}{n! r!} \quad (10)$$

Por otra parte, un incremento del grado de la ecuación puede resultar innecesario o incluso perjudicial para determinar correctamente la región de una clase. Respecto al tema de su posible inutilidad, esta dependerá de la cantidad de muestras consideradas y el número de atributos presentes en las mismas. Asimismo, el análisis estadístico desarrollado sobre la materia ha permitido esclarecer una regla matemática que, en términos generales, suele cumplirse para una categorización aleatoria de las muestras de entrenamiento. De esta manera, si cada muestra cuenta con una cantidad relevante de atributos (superior a 10), y esta cantidad es superior a la mitad de la cantidad de muestras utilizadas, entonces las posibilidades de encontrar una situación en la que no se puedan clasificar correctamente las muestras mediante una ecuación lineal son muy reducidas [8]. De forma análoga, esta norma también se cumple para aquellos clasificadores de grado más elevado, considerando los términos z en lugar de los atributos.

Aunque las ecuaciones de mayor grado son las que más posibilidades tienen de hallar las fronteras de la región de una clase atendiendo a la preclasificación de las muestras, este aspecto puede tornarse negativo ante la existencia de muestras incorrectamente categorizadas. Un ejemplo de ello es el que se observa en la Figura 3, en el cual para el caso de la ecuación lineal existe una muestra mal clasificada.

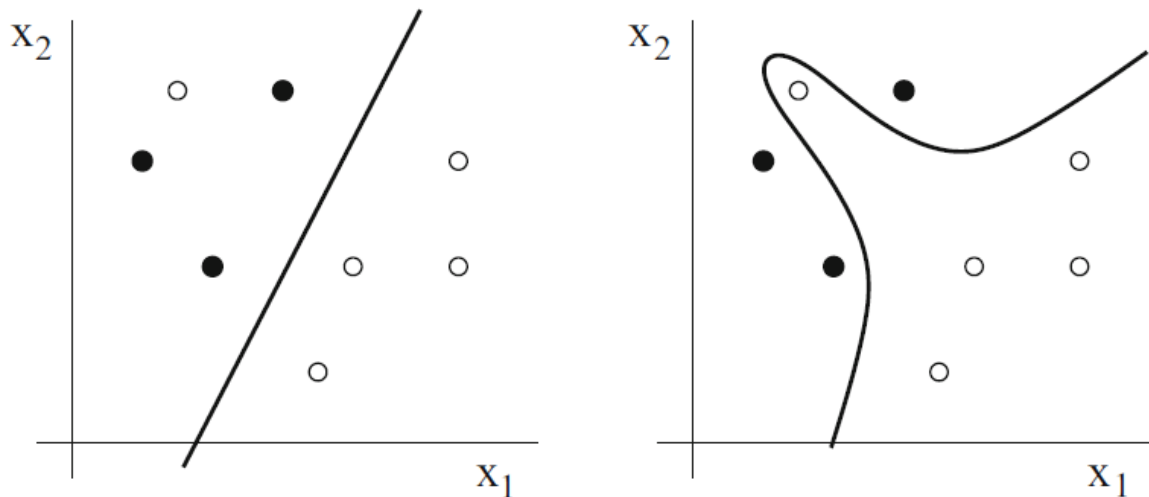


Figura 3: Clasificación mediante ecuación lineal (izquierda) y otra de grado superior (derecha) [8]

Aunque es posible que la muestra que la ecuación lineal es incapaz de categorizar correctamente de acuerdo con su preclasificación pertenezca a la clase correcta, también existe la posibilidad de que ocurra lo contrario. Este último caso suele ser el más probable para aquellas muestras que se encuentran relativamente distanciadas de otras muestras de su misma clase. Dicha preclasificación errónea no supondría ningún problema en la determinación de la ecuación lineal. Sin embargo, no sucedería lo mismo en la otra ecuación polinómica, la cual lograría

equivocadamente introducir esta muestra en la zona de la clase presupuesta como correcta. En consecuencia, una región que realmente pertenecería a otra clase sería incorrectamente otorgada, originando un mal funcionamiento del sistema de clasificación.

2.6. Redes neuronales

Una opción intensamente explorada para la implementación de la funcionalidad de *machine learning* son las denominadas redes neuronales. La estructuración de estas se basa en una serie de unidades de procesamiento denominadas neuronas, las cuales se organizan en varias de capas como se ilustra en la Figura 4.

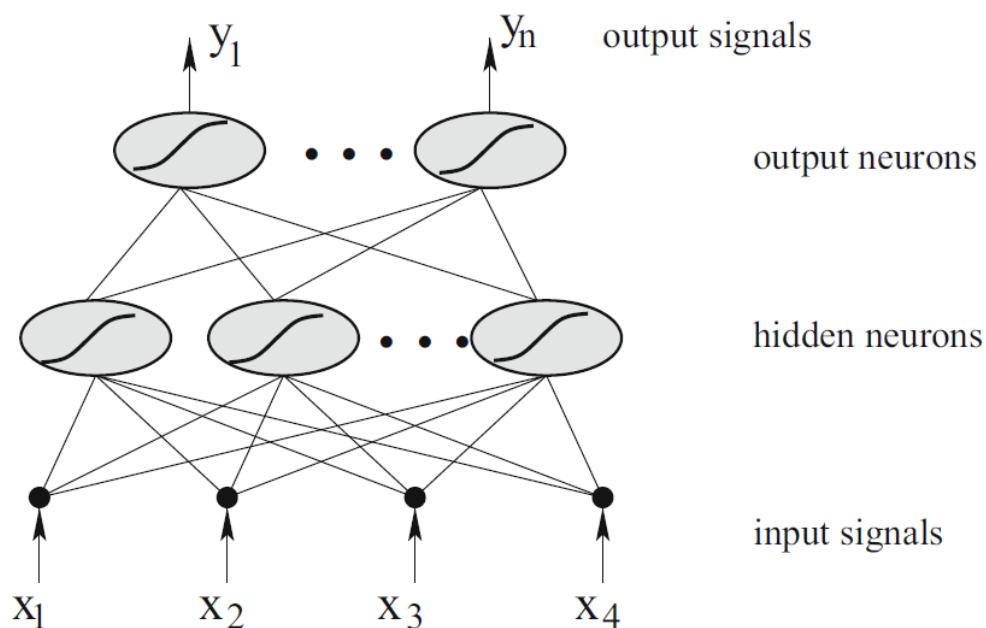


Figura 4: Estructuración de una red neuronal perceptron multicapa de 2 capas [8]

Como puede contemplarse en la Figura 4, cada neurona dispone de múltiples señales de entrada, pero tiene una única salida. Asimismo, la señal saliente puede ser o bien una de las salidas finales de la red neuronal, suponiendo que la neurona se encuentre en la capa llamada *output neurons* en la Figura 4, o bien una señal de entrada a las neuronas de la capa adyacente superior, como es el caso de las neuronas de la capa *hidden neurons* en la Figura 4. Las entradas de la red neuronal corresponden con los atributos de la muestra a clasificar, mientras que cada salida expresa de manera ponderativa cuán posible es la pertenencia de la muestra de entrada a una clase determinada. De esta manera, si se tiene una red neuronal de 3 salidas “ y_1 ”, “ y_2 ” e “ y_3 ” y, para una muestra concreta, se obtienen respectivamente los valores 0.7, 0.3 y 0.9, entonces la clase a la que pertenecería la muestra según la red neuronal sería la asociada a la salida “ y_3 ”.

La característica de utilizar múltiples salidas, una por clase, permite a la red neuronal cierto grado de funcionalidad añadida. Esto se debe a que no solo establece la categoría de la muestra en función de cuál sea la salida con valor más elevado, sino que intuye el grado de convicción con el que realiza esa afirmación. Asimismo, dicha convicción será mayor cuanto más significativa sea la diferencia del valor más alto que se obtenga en las salidas de la red con respecto al segundo más elevado. Este aspecto ocasiona que un cierto cambio en la estrategia para analizar el grado de error de una red neuronal pueda ser conveniente, de manera que se considere no solo qué cantidad con respecto al total de muestras utilizadas se encuentran mal clasificadas, sino también cuán convencido estaba el sistema de clasificación en cada una de las clasificaciones equívocas. Para ello, y considerando que las salidas de la red neuronal presentan sus valores en los intervalos entre 0 y 1, puede utilizarse la fórmula del error cuadrático medio (MSE, *Mean Square Error*) que se describe en la ecuación (11). En dicha ecuación, “m” representa la cantidad de clases distintas del sistema de clasificación e “ y_i ” representa el valor obtenido en una de las salidas de la red neuronal. Por otra parte, “ t_i ” simboliza el valor esperado para una salida concreta de la red neuronal, suponiendo que esta hubiese otorgado la clasificación correcta a la muestra evaluada con absoluta convicción. Esto significaría que “ t_i ” tendría el valor máximo (1) para el caso de la salida que corresponda con la clase correcta y el valor mínimo (0) para el resto de las salidas.

$$MSE = \frac{1}{m} \sum_{i=1}^m (t_i - y_i)^2 \quad (11)$$

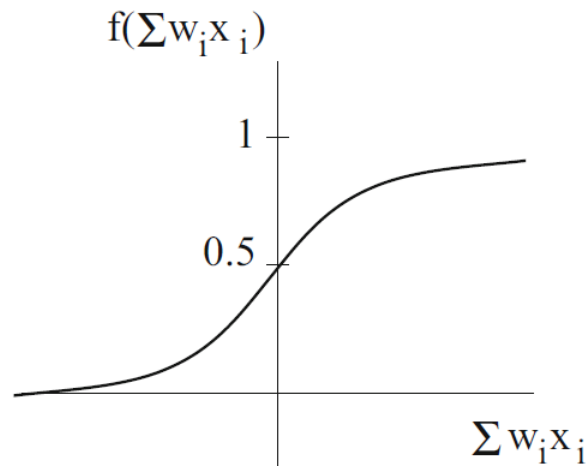
El objetivo de las neuronas consiste en realizar una determinada operación matemática con los valores de entrada para generar como resultado un valor de salida. Cabe destacar que los valores de entrada a cada neurona podrán tener ponderaciones distintas entre sí, lo que origina que sus señales difieran en el grado de relevancia en el cálculo. Respecto a las clases de redes neuronales existentes, estas se subdividen en varios tipos, siendo dos de ellos las redes *perceptron* multicapa (*multilayer perceptrons*) y las redes basadas en funciones radiales (*radial-basis function network*).

2.6.1. *Multilayer perceptron*

En este tipo de redes neuronales, se utilizan los valores de entrada a la neurona como elementos de una suma ponderada que a su vez constituye la variable independiente en una función de transferencia, que será la misma para todas las neuronas, independientemente de la capa en la que se encuentren. Asimismo, existen múltiples opciones de función de transferencia

válidas como, por ejemplo, es el caso de la función sigmoidea detallada en la ecuación (12), donde Σ representa el sumatorio de las entradas ponderadas y cuya gráfica se detalla en la Gráfica 1.

$$f(\Sigma) = \frac{1}{1 + e^{-\Sigma}} \quad (12)$$



Gráfica 1: Representación de la ecuación sigmoidea en función del sumatorio [8]

De manera semejante a la estructuración indicada en la Figura 4, la red neuronal *perceptron* multicapa se compone de una serie de capas cuya cantidad es variable, pero que típicamente suelen ser 3 o 4. Un aspecto ventajoso de las redes neuronales es que, con la cantidad suficiente de neuronas pertenecientes a capas ocultas y ajustando correctamente los factores de ponderación a la entrada de cada neurona, toda función de clasificación posible es implementable. De esta manera, la creación de un correcto sistema de clasificación que categorice adecuadamente las muestras de un conjunto de pruebas consiste en la determinación de estas características de la red y no en lo compleja que resulte la ecuación de clasificación a intuir, como ocurría en los clasificadores polinómicos. Este principio se conoce como el teorema de la universalidad [8].

Al igual que ocurre con el procedimiento de *Perceptron* para la deducción de clasificadores polinómicos, el proceso de aprendizaje de la red neuronal consiste en ajustar mediante el incremento o decremento cada uno de los parámetros de ponderación presentes en la red. Dichos parámetros serán inicializados con un valor aleatorio muy reducido, típicamente entre -0.1 y 0.1. Sin embargo, estos parámetros se encuentran asociados a las señales de entrada de cada neurona de la red y no únicamente a los atributos, que solamente sería el caso de las neuronas de la capa inicial vista desde la entrada a la red neuronal.

El proceso de ajuste se realiza en cada caso en función del error cuadrático medio. Esto quiere decir que en el proceso de modificación de los parámetros de ponderación no solo se busca que la red neuronal confirme la pertenencia del caso a la clase que se le asignó, sino que muestre una incertidumbre nula al afirmar dicha pertenencia. Desgraciadamente, este último aspecto no siempre es posible y el clasificador solo puede aproximarse hasta cierto nivel.

Otra semejanza que este proceso guarda con respecto al algoritmo de *Perceptron* para el establecimiento de clasificadores polinómicos es que, para asegurar la correcta categorización de cada muestra, se reitera su verificación en varias etapas. De esta manera, las etapas se sucederán hasta lograr que en una de ellas no se precise la modificación de los parámetros de ponderación por parte de ninguna muestra. Esto se debe a que las modificaciones de los parámetros de clasificación efectuadas a partir de las conclusiones extraídas de una muestra dada pueden ocasionar una peor clasificación por parte de la red neuronal en muestras anteriormente evaluadas. Asimismo, el número de etapas requeridas suele ser muy elevado, alcanzando fácilmente una magnitud de varios miles.

La modificación de los parámetros w sobre todo desde el punto de vista de la proporción relativa de los cambios entre ellos, así como su sentido, pueden establecerse mediante el cálculo del gradiente de la función que expresa el error cuadrático medio en función de todos los parámetros de ponderación presentes en la red neuronal. Por otra parte, la determinación de las modificaciones de los parámetros de ponderación también puede extraerse mediante el algoritmo de *Hill Climbing*. Estos dos procedimientos se basan en que antes de efectuar cualquier cambio en los parámetros w , se observa la función del error cuadrático medio en función de los parámetros de ponderación. Luego, se analiza qué dirección debe tomar, desde punto de vista de la representación gráfica, la modificación de los parámetros w , para que partiendo del punto actual se consiga una caída más pronunciada del error cuadrático medio.

Un aspecto destacable del proceso de conformación de la red neuronal basada en el gradiente es la posibilidad de establecer, mediante una ecuación matemática, una métrica de la influencia de cada neurona en la obtención de resultados insatisfactorios cuando se evalúa una muestra de entrada. Con ello se consigue localizar en qué partes de la red es necesaria una modificación más notoria de los parámetros de ponderación, correspondiendo dichas partes a las neuronas con una influencia en el grado de error más significativa. Este hecho posibilita una estrategia de modificación de los parámetros más semejante a la aplicada en el caso de los clasificadores polinómicos conformados mediante *Perceptron*. Asimismo, este proceso requiere empezar los cálculos con las neuronas asociadas a las capas que generan las señales de salida. En

dicho caso, la influencia en el error de clasificación “ δ_i ” se determina mediante la ecuación (13) si la función de transferencia de las neuronas es una sigmoide. En la ecuación, “ y ” representa el valor de salida obtenido y “ t ” el valor de salida preestablecido.

$$\delta_i = y(1 - y)(t - y) \quad (13)$$

Una vez obtenido el valor de la influencia en el error de clasificación de las neuronas de la capa de salida, se puede determinar la influencia “ δ_j ” presente en las neuronas de la capa predecesora. Para ello, se deberá utilizar la fórmula descrita en la ecuación (14), en la que “ h ” hace referencia a la salida de la neurona evaluada y “ w_{ji} ” simboliza la ponderación que cada neurona de la capa superior otorga a la salida “ h ”.

$$\delta_j = h(1 - h) \sum_i (\delta_i w_{ji}) \quad (14)$$

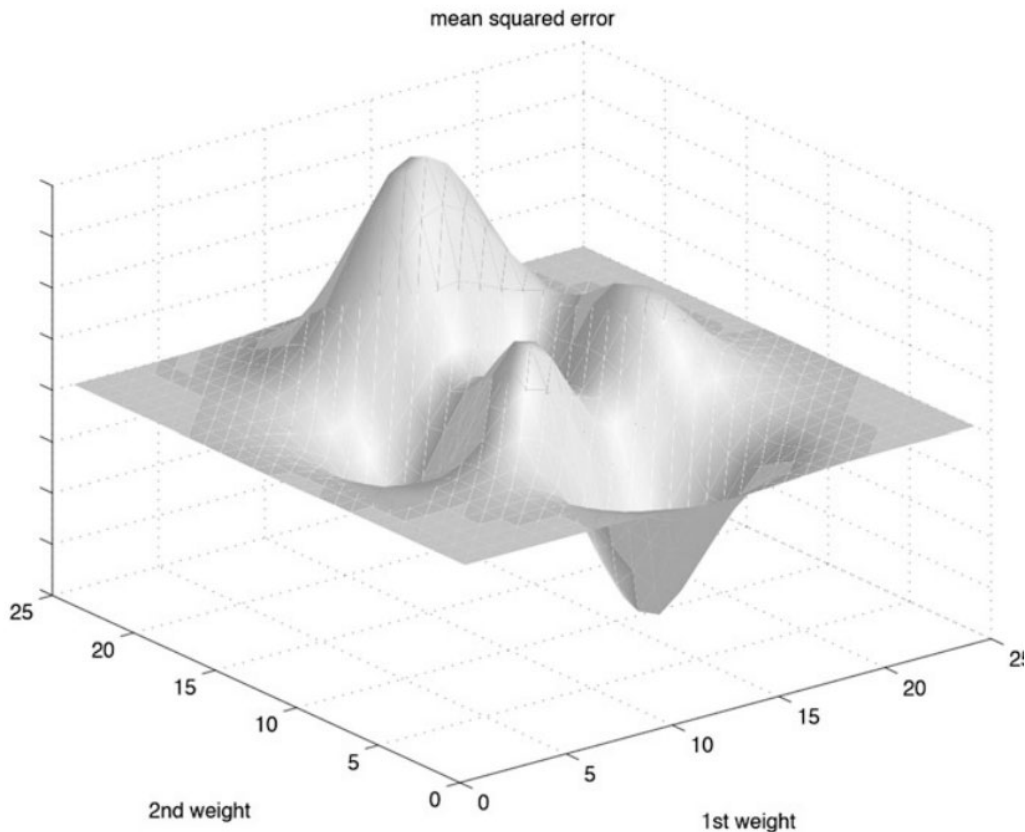
Como puede observarse, la ecuación (13) es muy similar a la ecuación (14), a excepción del sumatorio de la segunda que sustituye al factor de la diferencia entre la salida esperada y la obtenida que se introduce en la primera ecuación. La ecuación (14) es válida para el cálculo de la influencia de cualquier neurona salvo las de salida, siempre que como variables “ δ_i ” se consideren las influencias de las neuronas de la capa inmediatamente superior. El valor obtenido es utilizado para la modificación del parámetro de ponderación de la entrada “ x ” a la neurona considerada. No obstante, dicha modificación dependerá también del valor de la entrada “ x ” para la muestra analizada, así como del índice de aprendizaje “ η ”. Esta última variable tomará un valor entre 0 y la unidad. En la ecuación (15) se especifica la operación matemática ejercida en la modificación de cualquier parámetro w .

$$w = w + \eta \delta x \quad (15)$$

Pese a sus ventajas, las redes neuronales del tipo *perceptron* multicapa presentan algunos inconvenientes. Entre ellos puede destacarse su elevada carga computacional. La causa de ello se encuentra en la relativamente significativa necesidad de cómputo que requiere cada parámetro w , sobre todo en las capas inferiores. Además, debe tenerse en cuenta que la cantidad de parámetros w se eleva con gran facilidad cuando se desea implementar un clasificador que requiera cierta sofisticación.

Otro problema relativo a la red neuronal ocurre cuando se aplica el gradiente para determinar la relación de proporción en las modificaciones de los parámetros w . Esto se debe a que

el gradiente solo permite determinar la dirección dentro del espacio vectorial en la que se consigue un descenso más rápido. En consecuencia, el sistema puede dirigirse y establecerse en un mínimo relativo de la función del error cuadrático medio, el cual se observaría como un socavón en la gráfica de la función. Sin embargo, es fácilmente factible que la función posea más de un mínimo relativo, como es el caso de la función de 2 variables expresada gráficamente en la Gráfica 2.



Gráfica 2: Función con más de un mínimo relativo [8]

Por este motivo, puede suceder que el mínimo relativo tomado no corresponda con el mínimo absoluto. Para gestionar esta situación, es necesario añadir cierta funcionalidad adicional que permita, por un lado, identificar cuándo la configuración del sistema se encuentra en un mínimo relativo y, por otro lado, efectuar acciones correctoras. El primer aspecto es fácilmente alcanzable manteniendo registro de los valores de error cuadrático medio conseguidos durante el transcurso de varias etapas del proceso de entrenamiento de la red neuronal. De esta manera, si se observa que la reducción del error no evoluciona, significaría que el sistema presenta la configuración de un mínimo relativo.

Una manera de proceder ante la cuestión de los mínimos relativos se desarrolla por medio del índice de aprendizaje. Ello consiste en que dicho índice, en vez de permanecer constante durante todo el proceso, se modificará en los cambios de etapas del entrenamiento, siguiendo en todo momento una tendencia decreciente. La utilidad de esta variabilidad se debe a que, en las etapas iniciales, el sistema suele encontrarse más alejado de la configuración deseada. Por este motivo la efectuación de modificaciones más notorias ayudaría a acelerar el proceso de acercamiento, así como el número de etapas requeridas.

Además, podría ser útil para saltar los mínimos relativos. No obstante, también sería necesario que, en el caso de ser muy elevado, el índice de aprendizaje se redujese para evitar que se sobrepase el mínimo absoluto cuando la configuración del sistema se encuentra relativamente próxima. Asimismo, la reducción del valor del índice de aprendizaje se efectuará o bien de manera sistemática a medida que transcurren las etapas del entrenamiento o bien evaluando el sentido del avance seguido por la configuración del sistema desde el punto de vista del espacio vectorial. Esta última estrategia consistiría en observar el sentido del avance en las dos últimas modificaciones de los parámetros de ponderación, de manera que, si se encuentran en sentidos opuestos, significaría que se habría sobrepasado el mínimo y que sería necesario una reducción del índice de aprendizaje para que no volviese a ocurrir.

Otra forma de resolver el problema del mínimo relativo es aumentar el número de neuronas utilizadas en la red, lo cual daría además más flexibilidad a la capacidad de clasificar adecuadamente las muestras. Sin embargo, esta mayor flexibilidad puede ser en algunos casos excesiva, facilitando que, debido a la presencia de muestras de entrenamiento inicialmente mal catalogadas, ocurra de manera inconsciente desde la perspectiva de la aplicación de *machine learning* que el sistema de clasificación resulte menos satisfactorio.

2.6.2. Radial-basis function network

Las funciones de transferencia utilizadas por las neuronas de una red neuronal *perceptron* multicapa presentan cierta ineficacia a la hora de tratar con muestrass no linealmente separables. No obstante, eso no significa que una red neuronal de esta naturaleza sea incapaz de ello, pero sí que requeriría de un mayor grado de complejidad en su constitución para competir con otras redes neuronales de una tipología diferente. Para ello, una buena alternativa son las redes basadas en funciones radiales.

La diferencia operativa de este tipo de redes con respecto al anteriormente analizado radica únicamente en todas las capas que anteceden a la capa de salida desde la perspectiva de entrada

al clasificador. Dicha diferencia consiste en que la función de transferencia de las neuronas en esas capas es una función de distribución gaussiana, configurando una constitución de la red como la que se ejemplifica en la Figura 5.

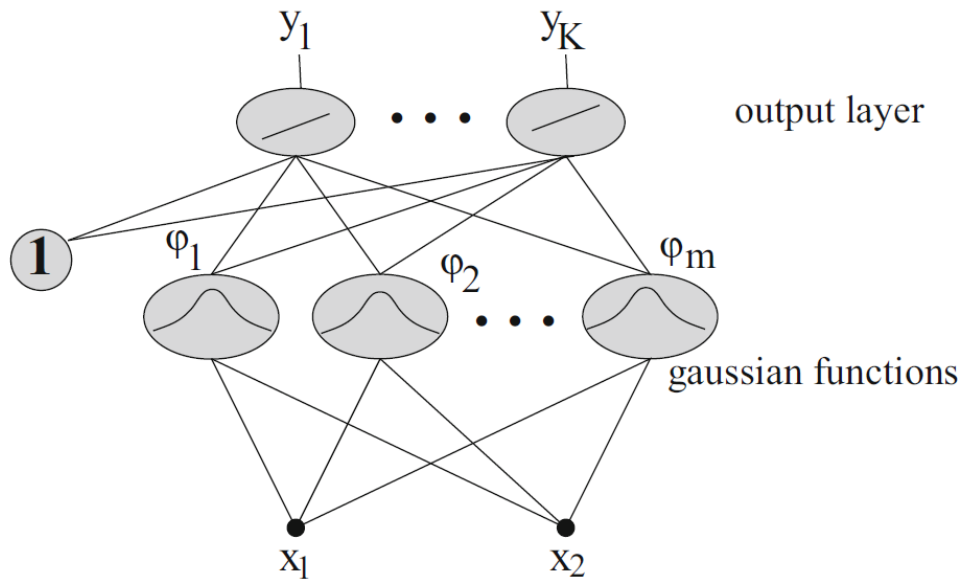
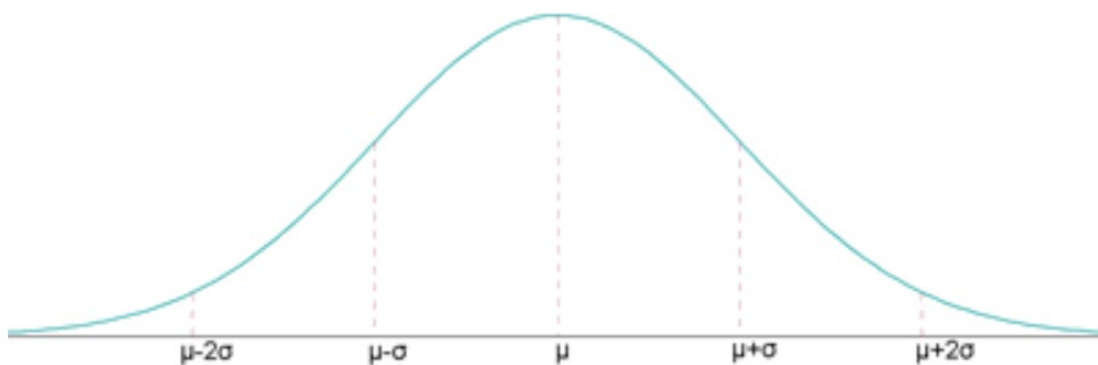


Figura 5: Red neuronal basada en funciones radiales [8]

La función de transferencia gaussiana poseerá tantas dimensiones como entradas " x_i " tenga, y generará una respuesta " φ " obedeciendo a lo descrito en la ecuación (16). En dicha función, " σ^2 " constituye la varianza del sistema. Por otra parte, " μ_{ji} " representa, en cada iteración del sumatorio, a una de las coordenadas que describe el centro de la distribución gaussiana. Cabe recordar que dicho centro coincidirá con el valor máximo de la función. Además, la función de distribución evolucionará desde la perspectiva unidimensional como se representa en la Gráfica 3.

$$\varphi = e^{-\frac{\sum_{i=1}^n (x_i - \mu_{ji})^2}{2\sigma^2}} \quad (16)$$



Gráfica 3: Evolución de la función de Gauss con respecto a " μ " y " σ " en una dimensión [9]

La utilización de la función de distribución gaussiana en todas las neuronas no presentes en la capa de salida ocasiona que, desde el punto de vista de la última capa del proceso de clasificación, las muestras, que inicialmente no son linealmente separables, sean representados mediante un sistema de coordenadas distinto. En consecuencia, estas muestras suelen volverse linealmente separables y con ello ser perfectamente catalogados por la capa de salida.

Suponiendo un caso de diseño de 2 capas, la configuración de la red recaería, por un lado, en la ponderación de cada entrada de las neuronas de la capa de salida. Estas ponderaciones serían fácilmente establecidas mediante el algoritmo de *Perceptron* visto para los clasificadores polinómicos. Por otra parte, se encuentra la configuración del punto de referencia de la función de distribución gaussiana de las neuronas pertinentes mediante los parámetros " μ ", así como de su varianza mediante " σ ".

Para entender cómo configurar las neuronas que implementan distribución gaussiana, hay que entender su funcionalidad desde el punto de vista de la clasificación. La utilidad de las mismas radica en que presentan una función que pondera con un valor entre cero y la unidad la cercanía de la muestra al punto de referencia establecido dentro del espacio vectorial, obteniéndose el valor unitario en el caso de que sean coincidentes. Asimismo, el valor de la función se decrementa de manera no lineal con la distancia de separación, estableciéndose la rapidez de esta disminución con el parámetro " σ ". Cabe destacar que la función posee simetría radial respecto al punto central de la distribución, lo que implica que el resultado depende de la distancia de la muestra a dicho centro, pero no de la dirección en la que se considere dicho distanciamiento.

Una estrategia habitual a la hora de establecer el centro de la función de distribución de cada neurona es que este coincida con la descripción de coordenadas de una de las muestras de entrenamiento. De esta manera, las muestras a evaluar por parte de la red neuronal serían representadas, para la capa final de esta, en función de su semejanza relativa a las muestras de referencia. Para dicha configuración, y dada la cantidad de muestras de entrenamiento típicamente requeridos en la mayoría de las aplicaciones reales de clasificadores (varios miles), se suele escoger solo algunos de ellos. Los resultados suelen ser lo suficientemente satisfactorios cuando la elección se realiza aleatoriamente. No obstante, la funcionalidad de las capas anteriores a la de salida será más efectiva si se procuran introducir como referencia a las muestras ubicadas en el centro de una región perteneciente a una clase concreta. Para su identificación, se efectuarían procedimientos de *clustering* [8].

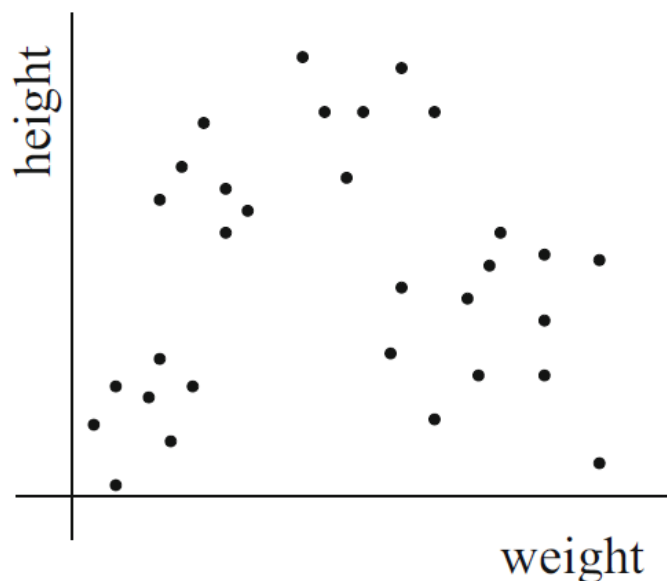
2.7. Aprendizaje no supervisado

A lo largo de este capítulo relativo al *machine learning*, se han visto una serie de clasificadores cuya principal característica común es que en todos ellos el usuario especifica las categorías que rigen la clasificación de manera explícita. Además, también comparten la necesidad de requerir de la existencia de muestras de entrenamiento con su categoría ya establecida, siendo esta información utilizada como referencia para establecer los criterios de pertenencia a cada categoría.

Sin embargo, como se mencionó al inicio de este apartado, las funcionalidades de *machine learning* no siempre se restringen a estos requisitos, existiendo aplicaciones de aprendizaje no supervisado. En dicho caso, la ocupación del algoritmo es la búsqueda de características comunes en las muestras observadas. Dentro de esta estrategia de *machine learning*, las aplicaciones de *clustering* presentan una relevancia significativa.

2.7.1. Clustering

Su objetivo consiste en subdividir el conjunto de las muestras evaluadas en una serie de agrupaciones denominadas *clusters*. Asimismo, cada una de estas agrupaciones contendrán varias muestras, los cuales poseerán un grado de semejanza relativa entre sí significativo, presentando mayor diferenciación con respecto al resto de muestras pertenecientes a otros *clusters*. En la Gráfica 4 puede verse un ejemplo de detección de *clusters* en un conjunto de muestras distribuidas en un espacio vectorial bidimensional.



Gráfica 4: Ejemplo de detección de clusters [8]

Cada uno de los *clusters* requiere, desde el punto de vista del algoritmo, de una forma de identificarse basada en las características del mismo tales como su localización, tamaño o distribución. Para ello, puede confiarse en los centroides, los cuales se definen como una posición en el espacio vectorial cuyas coordenadas se obtienen mediante la media aritmética de las muestras incluidas en la agrupación a la que representa. Asimismo, el procedimiento de clasificación en *clusters* más habitual consiste en medir la distancia que separa a la muestra evaluada del centroide de cada una de las agrupaciones. De esta manera se establecerá que la muestra pertenece al *cluster* cuyo centroide se encuentre más cercano.

Cabe destacar que las agrupaciones no deben interceptarse entre sí, lo que significa que una misma muestra no puede categorizarse en varios *clusters*. El número de *clusters* a considerar es variable, oscilando entre las opciones extremas en las que un único *cluster* agrupa a todas las muestras introducidas y en la que existen múltiples *clusters*, cada uno con una única muestra. Asimismo, la cantidad de *clusters* puede ser tanto un parámetro especificado por el usuario como un aspecto que el algoritmo debe intuir.

La búsqueda de *clusters* presentan una serie de aplicaciones muy útiles que, en muchas ocasiones, permiten complementar la funcionalidad de los algoritmos de aprendizaje supervisado. Entre dichas utilidades se encuentra la posibilidad de estimar el valor desconocido de algunos de sus atributos. El valor para dicho atributo será aquel que representa la media o el valor más frecuente dentro del *cluster* en el que la muestra se encuentre incluido, estableciéndose dicha inclusión a partir del grado de semejanza evaluado con los valores de los atributos conocidos de la muestra. De esta manera, se consigue otorgar un valor a estos atributos que, si bien no coincidirá exactamente con el valor real para la muestra dado, se aproximará mejor que si se realizase esta estimación a partir del total de muestras introducidos en los cuales el valor del atributo es conocido, no teniendo en cuenta la cercanía relativa entre las muestras.

Otras utilidades son la vista en el apartado 2.6.2 y atajar el proceso de desarrollo de clasificadores en el aprendizaje supervisado. Este último aspecto se debe a que muy frecuentemente los atributos que se encuentran en un mismo *cluster* y, por lo tanto, tienen atributos con valores bastante similares suelen considerarse como pertenecientes a una misma clase. Debido a ello, el programador puede decidir aplicar la estrategia de agrupar las muestras de entrada en *clusters* y luego efectuar la clasificación de estas agrupaciones.

2.7.1.1. *K-means*

k-means constituye uno de los algoritmos de *clustering* existentes más simples. El algoritmo se inicializa con una cantidad fija de “*k*” *clusters*, que se crean agrupando en cada uno de estos una misma cantidad de muestras. Una vez creadas las agrupaciones iniciales, se calculan sus centroides. La elección de cómo realizar el reparto inicial de estas muestras puede efectuarse o bien partiendo de una primera muestra por *cluster* que es elegida aleatoriamente, o bien partiendo de unas nociones básicas sobre el resultado buscado. Esta última opción ayudaría a que la distribución inicial se asemeje en mayor medida al resultado final de la ejecución, ocasionando que la resolución del algoritmo termine con mayor inmediatez. La primera opción requeriría que el centroide de los *clusters* se fuera actualizando a medida que se introduce una nueva muestra, ya que sus coordenadas coinciden inicialmente con las de la primera muestra, y estas pueden no ser la referencia más idónea para la indicar el posicionamiento del *cluster*.

Una vez establecida la agrupación inicial, el algoritmo analiza cada uno de las muestras. En dicho análisis, mide la distancia dentro del espacio vectorial que separa a la muestra de los centroides de los *clusters* establecidos. De esta manera, si el *cluster* más cercano y aquel que contiene a la muestra estudiada no coinciden, entonces se reubicará la muestra dentro del primer *cluster* mencionado. Dicho cambio altera el contenido de ambos *clusters*, por lo que es necesario recalcular sus respectivos centroides. Este proceso se reitera para cada muestra hasta conseguir que todas ellas pertenezcan al *cluster* que se encuentre más próximo.

Este procedimiento de *clustering* presenta un inconveniente muy significativo, debido a que la inicialización de los *clusters* influye en la agrupación final de las muestras. Como ejemplo para explicar este hecho se puede tomar la distribución en un espacio bidimensional con varias muestras que se expone en la Figura 6.

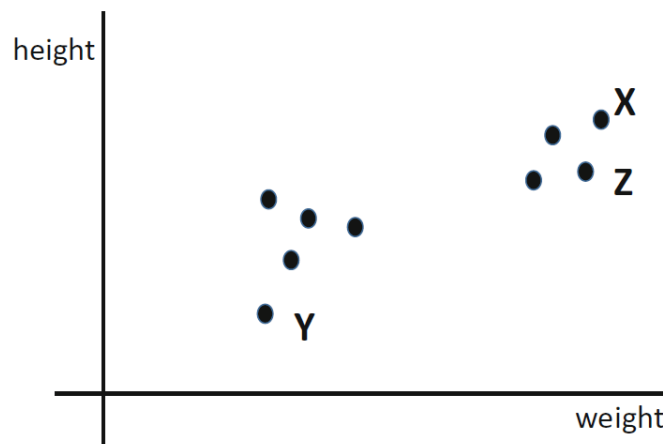


Figura 6: Distribución de muestras en un espacio bidimensional [8]

Sobre el ejemplo anterior, supóngase que el programador especifica al algoritmo que distribuya las muestras en 2 *clusters*. Si el programa escoge a las muestras “X” e “Y” como primer elemento de cada agrupación, el algoritmo conseguirá determinar adecuadamente las agrupaciones más visualmente evidentes, lo cual sería el resultado más deseable. No obstante, si fueran “X” y “Z” los elementos que constituyeran la versión inicial de los *cluster*, podría ocurrir que las agrupaciones establecidas, aun cumpliendo con los requisitos de terminación especificados al algoritmo, no fueran las adecuadas. Esto se debe a que son agrupaciones mucho más dispersas de lo estrictamente necesario y que además contendrían muchas muestras que estarían más cerca de muestras del otro *cluster* que de los presentes en el suyo propio.

Para solucionar este problema se necesita, en primer lugar, una manera de ponderar la excelencia de la asociación de *clusters* establecida en función de la distancia de cada muestra evaluada con respecto al centroide de la agrupación a la que pertenece. Por este motivo, se crea la variable SD (*Summed Distance*, Distancias Sumadas) cuyo cálculo consiste en la suma de dichas distancias. De esta manera, el algoritmo podrá determinar la agrupación óptima de las muestras en “k” *clusters* como aquella en la que el valor de SD se minimice. Asimismo, dado que la variación en los resultados finales de la agrupación depende de las primeras muestras escogidas para cada *cluster*, el algoritmo probará distintas alternativas, obteniéndose así distintos resultados finales sobre los cuales se elegirá atendiendo al criterio del SD mínimo.

Respecto a la cuestión del número de *clusters* a considerar, cabe destacar que, al ser un parámetro prefijado por el programador, es muy probable que la cantidad establecida no sea la más adecuada según las relaciones de proximidad relativas entre las muestras distribuidas en el espacio vectorial. Al igual que para la cuestión de la inicialización, la opción más factible sería probar el algoritmo con diferentes opciones de número de *clusters*, eligiéndose aquella que ofrezca unos mejores resultados. La elección de la cantidad adecuada de *clusters* se determina como aquella en la que el valor de SD, normalizado por el número de *clusters*, sea más reducido. El requisito de la normalización se debe a la necesidad de anular la influencia positiva que tiene la utilización de un número más elevado de *clusters* en la reducción del valor de SD obtenido.

Aunque se preestablezca la cantidad de *clusters* desde el inicio del algoritmo, es posible, mediante un proceso de análisis adicional, aumentar o disminuir dicha cantidad por medio de la partición o unión de *clusters* respectivamente. Para que dichos ajustes propicien que la subdivisión en *clusters* del conjunto de muestras sea más acertada, es necesario evaluar la conveniencia de las acciones de unión y partición. Para la unión se deberá medir la distancia que separa los centroides de dos *clusters*, de manera que, si esta es reducida con respecto a la distancia media medida entre

clusters, significará que estos dos *clusters* deben ser unificados. Respecto a la separación, esta se realizará si la distancia media entre las muestras de un *cluster* es elevada. En dicha situación, todas las muestras se considerarán separados del *cluster* original y se reagruparán en dos nuevos *clusters*. Para esta reorganización, suele ser suficiente con tomar a las dos muestras más distantes entre sí como primeras muestras de los *clusters*, y efectuar la reasignación del resto de las muestras al *cluster* conveniente según el procedimiento *k-means* clásico.

Una variante bastante útil del algoritmo *k-means* se basa en la aplicación jerárquica. Asimismo, en esta versión del procedimiento el conjunto de muestras consideradas es separado en dos *clusters*. Posteriormente, el algoritmo se vuelve a ejecutar para cada *cluster* de manera individual, repartiendo el contenido de cada uno entre dos nuevos *clusters*. Este procedimiento de partición reiterada de los *clusters* proseguirá hasta que se satisfaga el criterio de terminación del algoritmo, el cual puede ser tanto haber alcanzado la cantidad de *clusters* máxima especificada por el usuario, o la distancia mínima de separación entre *clusters* cercanos.

2.7.1.2. Optimizaciones

Si bien el algoritmo *k-means* como estrategia de *clustering* es relativamente útil, su capacidad para la detección de *clusters* se restringe de manera bastante significativa a agrupaciones cuya distribución presenta una morfología sin concavidades. Este requisito no suele suponer ningún problema en la mayor parte de las aplicaciones como, por ejemplo, la configuración de una red neuronal basada en funciones radiales. Sin embargo, puede ocurrir que el conjunto y la distribución de las muestras no cumpla con estos requisitos, ocasionando que el algoritmo realice agrupaciones inadecuadas. Esta situación podría ocurrir, por ejemplo, con la distribución de muestras que se expone en la Figura 7.

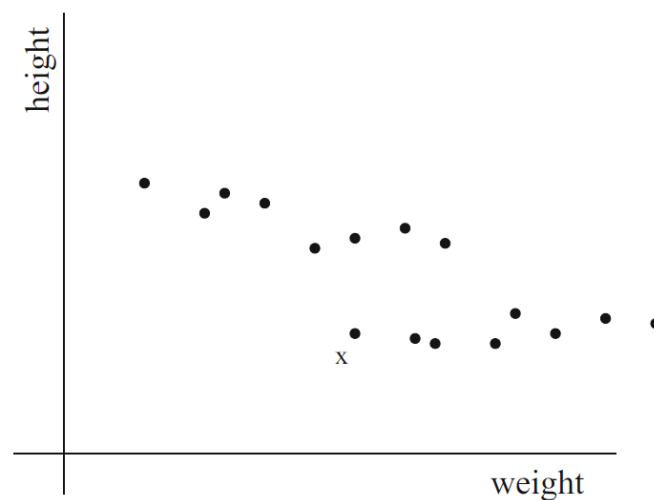


Figura 7: Ejemplo de distribución de muestras donde *k-means* no sería eficaz [8]

A simple vista, se puede deducir que la distribución de las muestras de la Figura 7 se puede agrupar en dos *clusters* con una forma alargada en el plano horizontal. Sin embargo, en el caso de la muestra "x" es factible que la misma sea agrupado en el *cluster* equivocado ya que, de acuerdo con los cálculos, es el que se encuentra más próximo utilizando como posición de referencia su centroide. Para conseguir que el sistema extraiga la agrupación adecuada, la modificación a efectuar consistiría en definir la distancia entre 2 *clusters* como la mínima distancia existente entre 2 muestras que pertenecen a *clusters* opuestos. Sin embargo, este cambio de estrategia supondría cierto aumento de la carga computacional del sistema. Esto se debe a que, si " N_A " es la cantidad de muestras de un *cluster*, y " N_B " es la del otro, entonces sería necesario medir un total de " $N_A \times N_B$ " distancias para establecer la distancia de separación entre las 2 agrupaciones. No obstante, dado que la cantidad de muestras por *cluster* suele ser lo suficientemente reducida, esta mayor necesidad de cómputo no suele suscitar ningún inconveniente.

Por otra parte, existe la estrategia de *clustering* de la agregación jerárquica. Este algoritmo comienza con tantos *clusters* como muestras haya, existiendo una única muestra por cada *cluster*. Asimismo, durante la ejecución del procedimiento, se reiterará un proceso consistente en localizar aquella pareja de *clusters* que se encuentran menos distanciados y unificarla. Como condición para terminar el proceso de agregación jerárquica, puede establecerse que la ejecución finalice cuando la cantidad de *clusters* es inferior a un determinado umbral mínimo o cuando la distancia mínima dentro del conjunto de *clusters* exceda un valor específico.

2.8. Conclusiones

En este capítulo se ha explicado el concepto de *machine learning* y su relevancia, así como la clasificación principal de las aplicaciones existentes para su implementación según sean de aprendizaje supervisado, no supervisado o mediante refuerzo. Además, se han descrito una serie de aspectos a tener en cuenta con respecto a las muestras de entrada, y se ha realizado un estudio de distintas alternativas de *machine learning*, definiendo su funcionamiento, sus virtudes, pero también sus limitaciones.

Dentro de las opciones evaluadas, se decide optar por una aplicación de *k-means* como opción a implementar en la placa de prototipado para este proyecto. Los detalles de este proceso, así como las características específicas de esta aplicación se describen en capítulos posteriores.

Capítulo 3. Zynq UltraScale+ MPSoC

3.1. Introducción

Para el desarrollo de este trabajo se utiliza el MPSoC (*Multi-processor System- On-Chip*) Zynq UltraScale+. La elección de esta plataforma *hardware* se debe a que constituye un sistema heterogéneo que permite la implementación de sistemas electrónicos que disponen de un sistema de multiprocesamiento con la posibilidad de incluir aceleradores *hardware* a medida en la FPGA, por tanto, combinan tanto la funcionalidad *hardware* como funcionalidad *software*. La funcionalidad *software* se logra mediante una serie de unidades de procesamiento incorporadas dentro de los recursos PS (*Processing System*) que se incluyen dentro de la arquitectura. Por otra parte, la funcionalidad *hardware* se obtendría por medio de los recursos de lógica programable (PL, *Programmable Logic*) [3].

Zynq UltraScale+ es una familia de dispositivos MPSoC FPGA desarrollada por Xilinx [10]. Se trata de la segunda generación de dispositivos SoC de la serie Zynq. El dispositivo Zynq es un SoC (*System on Chip*) que integra una APU (*Application Processing Unit*) dual, una lógica programable basado en la Serie 7 de Xilinx y un conjunto de periféricos [11].

La familia Zynq UltraScale+ supone la evolución de los sistemas SoC a los sistemas MPSoC, cuya principal novedad fue la incorporación de bloques de procesadores de distintas categorías dentro del mismo chip. En consecuencia, esta plataforma posibilita el multiprocesamiento de aplicaciones *software* [12]. Los dispositivos están fabricados en el proceso CMOS FinFET de 16 nm de TSMC [13]. Este reducido tamaño por transistor permite una mayor rapidez, así como un menor consumo energético con respecto a las tecnologías de implementación anteriores (FinFET de 20 nm). La reducción en el tamaño de los transistores supone un incremento en un 50 % en la velocidad de funcionamiento del sistema, así como una disminución en un 60 % de la potencia consumida [12], [13].

Con respecto al conjunto de los recursos *hardware* que se incorpora en esta arquitectura, este variará en función de la subfamilia a la que pertenezca. Dichas subfamilias se designan como EG, CG y EV, siendo el significado de cada una de las letras el evidenciado en la Tabla 1 [10]. Las

Tipo de procesamiento diferencias más notorias con respecto a los recursos *hardware* que implementan se encuentran indicadas en la Tabla 2.

Tabla 1: Características de los recursos hardware según la subfamilia (adaptada de [10])

Identificador según el tipo/número de Procesadores		Tipo de procesamiento	
C: Dual APU, Dual RPU	E: Quad APU, Dual RPU, Single GPU	G: General Purpose	V: Vídeo

Tabla 2: Componentes hardware en función de la subfamilia (adaptado de [10])

Funcionalidad	Descripción	Sub-Familia		
		CG	EG	EV
Application Processor Unit (APU)	Basado en la arquitectura ARM Cortex-A53, con soporte de SO y ejecución de aplicaciones (2 a 4 cores)	2 cores	4 cores	4 cores
Real-Time Processing Unit (RPU)	Arm Cortex-R5 cores para tareas en tiempo real	2 cores	2 cores	2 cores
Graphics Processing Unit (GPU)	Soportes Gráfico dedicado	-	✓	✓
Video Codec Unit (VCU)	Implementado como un <i>hard IP</i> en el PL, proporciona soporte para los formatos de compresión de vídeo estandarizados H.265 y H.264	-	-	✓
Configuration and Security Unit (CSU)	Funcionalidad de arranque seguro, Soporte para Arm TrustZone®, y monitorización de tensiones y temperatura	✓	✓	✓
Platform Management Unit (PMU)	Para la gestión de potencia, seguridad e integridad funcional del sistema	✓	✓	✓
Memoria	256KB OCM, interfaces para memorias externas de varios tipos	✓	✓	✓
Controladores para Direct Memory Access (DMA)	Dos controladores DMA de 8 canales, uno en cada dominio de potencia	✓	✓	✓
High performance interfaces (PS)	PCIe Gen2, USB3.0, SATA 3.1, DisplayPort, Gigabit Ethernet	✓	✓	✓
Bloques IPs integrados en el PL	PCI Express	✓ ^a	✓ ^a	✓ ^b
	150G	-	✓ ^a	-
	100G Ethernet MAC	-	✓	-

a. Incluidos en un subconjunto de dispositivos. b. Incluidos en todos los dispositivos.

Para el caso de la placa de prototipado utilizada, ZCU102 [14], el dispositivo integrado es el Zynq UltraScale+ XCZU9EG-2FFVB1156 MPSoC, por tanto, pertenece a la subfamilia EG. La unidad de procesamiento o APU (*Application Processing Unit*) disponible está formada por 4 núcleos ARM Cortex-A43, una unidad de tiempo real o RPU (*Real-Time Processing Unit*) de 2 núcleos basado en ARM Cortex-R5 y una GPU (*Graphical Processing Unit*) ARM Mali-400.

Por otra parte, esta plataforma dispone, al igual que el resto de las subfamilias de Zynq UltraScale+, de recursos de memoria *On-Chip* y múltiples interfaces de entrada y salida, entre otros recursos pertenecientes al PS. Respecto a la PL, esta se ajusta a la arquitectura Xilinx UltraScale [3]. Asimismo, la organización de los recursos *hardware* sigue el diagrama de bloques representado en la Figura 8.

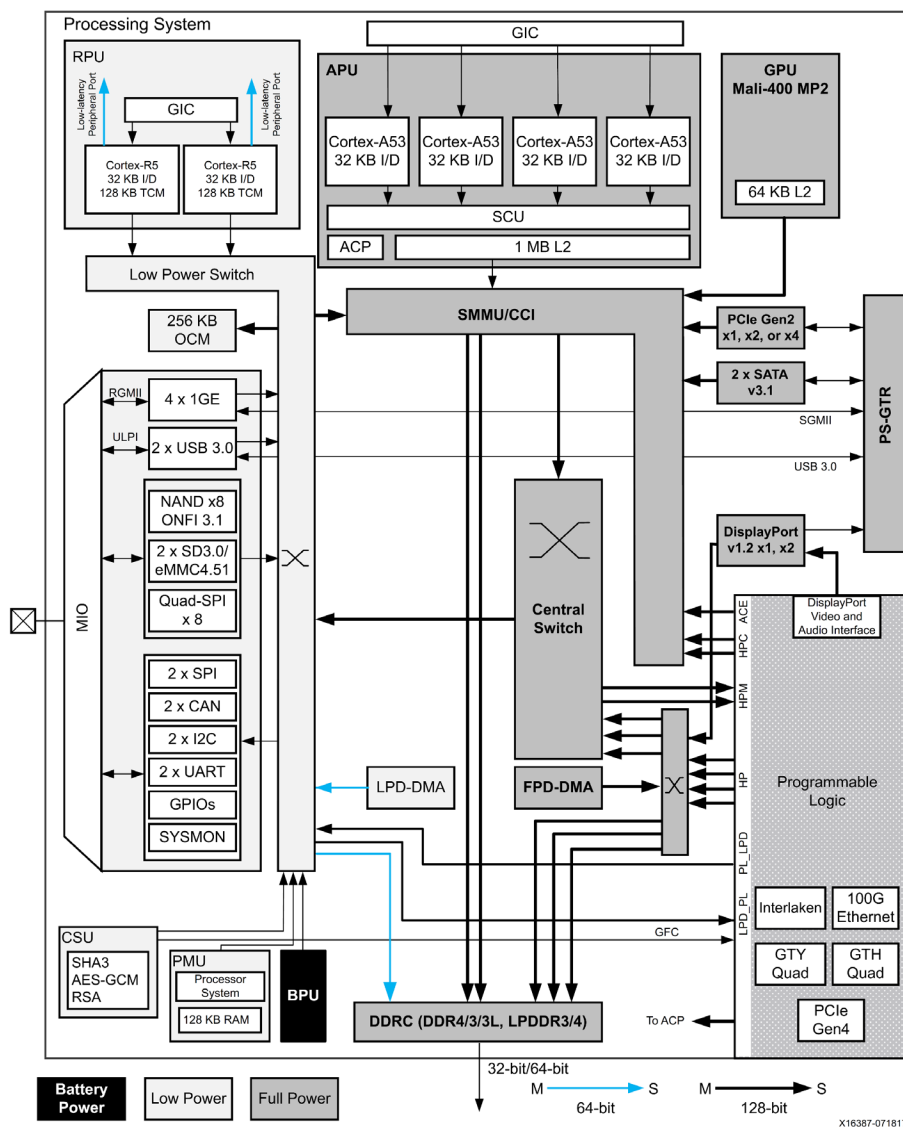


Figura 8: Diagrama de bloques de la arquitectura Zynq UltraScale+ MPSoC [14]

3.2. Sistema de procesamiento

El sistema de procesamiento o PS esta formado por un conjunto de recursos heterogéneos que soportan el multiprocesamiento en diferentes dominios: propósito general (APU basado en ARM Cortex A53), tiempo real (RTU basado en ARM Cortex R5) y GPU (usando ARM Mali-400). A continuación, se describe en detalle los recursos mencionados.

3.2.1. APU basada en ARM Cortex-A53

En la Figura 9 se muestra el diagram de bloques de la APU. Como se puede apreciar, dispone de 4 núcleos ARM Cortex A53, que implementa una arquitectura ARMv8-A, capaz de funcionar en 64 y 32 bits, pudiendo ejecutar los conjuntos de instrucciones A64, A32 y T32.

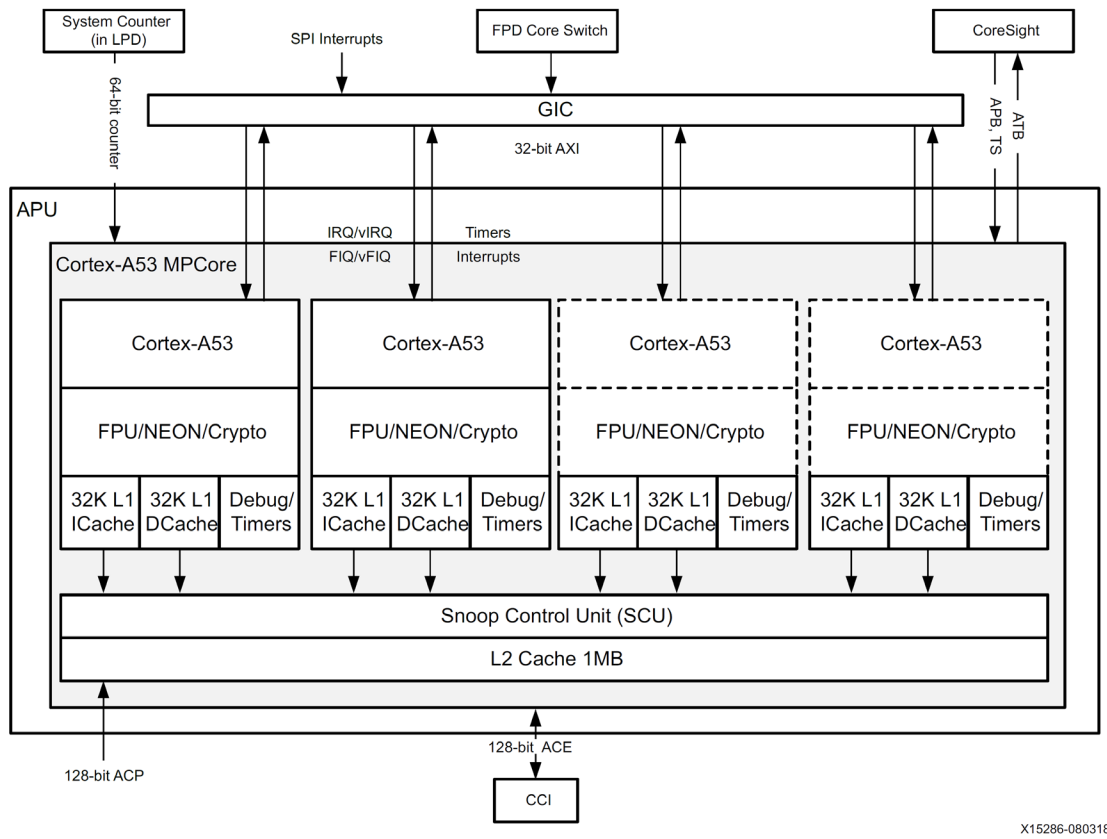


Figura 9: Diagrama de bloques de la APU [15]

Por otra parte, la APU implementa la funcionalidad ARM TrustZone, destinada a dotar de mayor seguridad al proceso de ejecución del *software* [3][10]. Cada uno de los núcleos puede funcionar con o sin sistema operativo. Asimismo, en función de cómo se coordine el funcionamiento conjunto de los núcleos de la APU, se pueden obtener las siguientes configuraciones.

- a. **Configuración simétrica.** Todos los procesadores de la APU ejecutan un mismo sistema operativo, siendo Linux el más usual. De esta manera, todos los segmentos del código *software* desarrollado podrán ejecutarse indistintamente en cualquiera de los núcleos.
- b. **Configuración asimétrica supervisada.** Los núcleos funcionan de manera independiente entre sí, pudiendo utilizar configuraciones distintas respecto al sistema operativo que implementan. Sin embargo, en esta configuración se dispone de un hipervisor, el cual se encarga del arbitraje de la ejecución paralela de las aplicaciones que se ejecutan en cada procesador [16].

Entre sus características de funcionamiento principales se encuentra una frecuencia de reloj máxima de hasta 1.5 GHz y que cada uno de los núcleos dispone de alimentación independiente, lo que permite escoger la activación individualizada de cada uno de ellos independientemente del estado de activación del resto de las APUs. Además, es capaz de operar con valores representados en punto flotantes de precisión simple y doble gracias a su FPU (*Floating Point Unit*). Para posibilitar un modo de ejecución SIMD (*Single Instruction Multiple Data*) la APU incorpora una extensión *hardware* denominada ARM NEON *Advanced*.

Para la interconexión entre las APUs y la PL, se dispone de las interfaces ACP (*Accelerator Coherency Port*) y ACE (*AXI Coherency Extension*). También dispone de una extensión de encriptación, la cual soporta los métodos de encriptación y desencriptación AES (*Advanced Encryption Standard*), SHA1/256 y RSA (*Rivest Shamir and Adleman*).

Respecto a los recursos de temporización, la APU ofrece el soporte de ARM para *timers* genéricos [3], 2 bloques IP (*Intellectual Property*) que disponen de 3 *timers* contadores cada uno [17], [18], un *timer watchdog* y un *timer* global del sistema. Además, dispone del módulo GIC 400 para gestionar las interrupciones siguiendo una jerarquía de prioridades [3].

Además, puede utilizar distintos formatos de memoria caché, ya sea con soporte para corrección de errores (ECC, *Error Correcting Code*) o con un mecanismo de detección mediante paridad. Asimismo, la memoria interna a la APU contiene una memoria caché L1 de un total de 64 KB de capacidad de almacenamiento asignada a cada CPU, y que se distribuye equitativamente en memoria para el almacenamiento de datos y en memoria para el almacenamiento de instrucciones.

Por otra parte, también se dispone de una memoria caché L2 de 1 MB que es compartida por todas las CPUs de la APU y que es utilizada como recurso compartido con el resto de componentes de la plataforma para permitir la comunicación de estos con la APU [10], [19]. Para la gestión de esta memoria es de especial relevancia el bloque SCU (*Snoop Control Unit*), que se

encarga de actualizar el contenido de los registros de las caché L1 que representan variables compartidas entre los procesadores de la APU, mediante transacciones realizadas siguiendo el protocolo MOESI (*Modified Owned Exclusive Shared Invalid*) [20]. Asimismo, cada núcleo de la APU dispone de su propia MMU (*Memory Management Unit*) que se encarga del mapeo de direcciones virtuales a direcciones físicas [20].

3.2.2. Real-Time Unit ARM Cortex R5

La RTU o Real-Time Unit es una unidad de procesamiento en tiempo real compuesta de dos núcleos ARM Cortex-R5 y con una frecuencia de funcionamiento de la CPU máxima de 600 MHz (Figura 10). Su diseño incorpora la arquitectura ARM7-R, la cual solo permite modos de operación de 32 bits, pudiendo ejecutar solamente los conjuntos de instrucciones A32 y T32 [3]. Al igual que la APU, la RPU es capaz de trabajar con aritmética de punto flotante de precisión simple y doble utilizando una FPU.

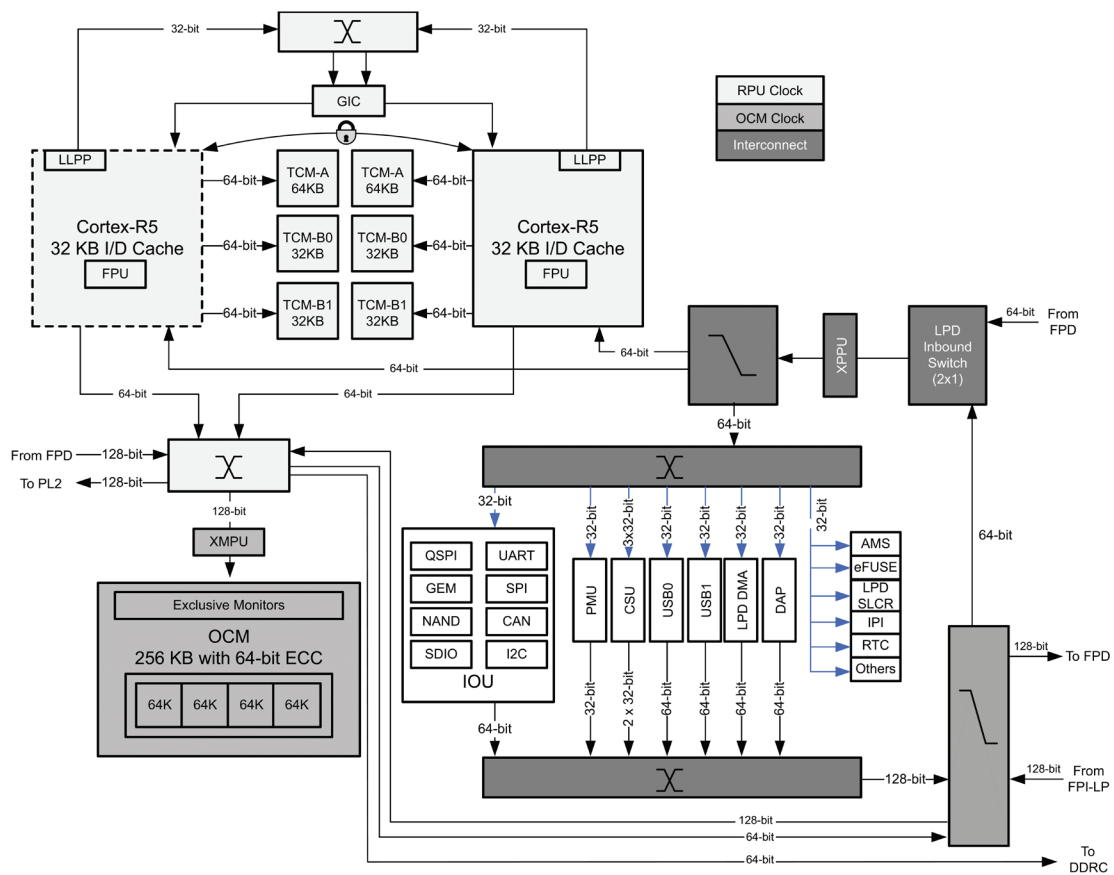


Figura 10: Vista general de la RPU [15]

El modo de operación conjunta entre los dos núcleos puede establecerse únicamente durante la inicialización de la RPU, pudiéndose escoger entre el modo *split* y el modo *lock-step*. El primero de ellos consiste en que cada una de las CPUs opera de manera independiente, mientras

que en el segundo modo ambas CPUs ejecutan la misma secuencia de instrucciones con una reducida separación temporal de un ciclo de reloj y medio entre ellos. Cabe destacar que la utilidad del modo de operación *lock-step* está orientada a detectar errores de funcionamiento en la lógica interna de los procesadores, lo que resultará evidente si ambos procesadores no devuelven los mismos resultados de ejecución [10][16].

Los recursos de *timers* de los que dispone son un tanto más reducidos que en el caso de la APU, limitándose únicamente a un *timer watchdog* y a 2 bloques IP que contienen 3 *timers* cada uno [3]. Al igual que la APU, la RPU dispone de un controlador de interrupciones propio [20].

Dentro de los recursos de memoria caché de los que dispone cada núcleo de manera individual, se encuentran 32KB de memoria L1. Además, se dispone de una memoria caché TCM (*Tightly Coupled Memory*) de 128 KB por cada CPU, la cual puede combinarse para formar una única memoria de 256 KB en el modo de ejecución *lock-step* [3]. La memoria TCM puede clasificarse, en términos de latencia, como una memoria L1. Sin embargo, se diferencia de una memoria caché normal en que permite ofrecer respuestas con una latencia determinista, un aspecto muy relevante en la ejecución de tareas críticas [21]. Además, esta memoria puede ser accedida como esclava mediante una interfaz AXI.

Como medidas de seguridad, todas las memorias locales a la RPU implementan un ECC capaz de corregir errores de 1 *bit* y de detectar errores de 2 *bits* en el contenido de un registro de memoria. Adicionalmente, se incorpora una MPU (*Memory Protection Unit*) por cada procesador con el propósito de restringir el acceso a su memoria caché [3][10].

3.2.3. GPU ARM Mali-400

Respecto a la unidad de procesamiento gráfica (GPU) dispuesta como recurso PS de la plataforma, algunos aspectos destacables son una frecuencia de funcionamiento máxima de 667 MHz y una tasa de relleno de píxel de 2 Mpixels/s/MHz. La estructuración interna de la GPU se expone en el diagrama de bloques de la Figura 11.

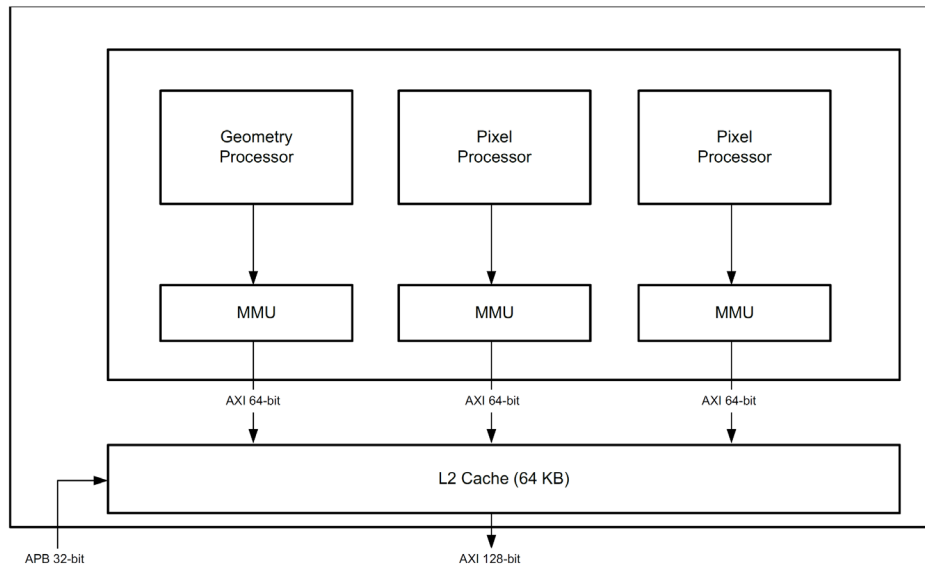


Figura 11: Diagrama de Bloques de la GPU[20]

Respecto a su programación, la GPU da soporte a las APIs OpenVG 1.1, OpenGL ES 1.1 y OpenGL ES 2.0 [3]. Además, permite la ejecución SIMD de 4 operaciones simultáneas con datos de 32 bits para los que se admite aritmética de punto flotante [20].

Entre los recursos *hardware* que componen la GPU se encuentran 3 procesadores, siendo uno de estos un procesador de geometría y los otros 2 procesadores de píxeles. La alimentación de cada uno de los procesadores se puede controlar de manera individualizada, maximizando así el ahorro energético. Por otra parte, la GPU dispone de 64 KB de memoria caché L2 de solo lectura, [3] la cual es accedida también por el resto de dispositivos de la plataforma por medio de una interfaz esclava APB (*Advance Peripheral Bus*), posibilitando así la interacción de la GPU [10].

3.2.4. Memoria PS

Como recursos de almacenamiento de datos presentes en la parte PS y que no se integran en ninguno de los módulos de procesamiento, se encuentran las memorias OCM (*On-Chip Memory*) y el controlador DDR (*Double Data Rate*).

Respecto a la memoria OCM, esta consiste en una memoria RAM (*Random Access Memory*) de 256 KB capaz de operar a una frecuencia de reloj máxima de 600 MHz. Además, utiliza ECC e interactúa con el resto del sistema mediante una interfaz AXI esclava de 128 bits [20], [10].

Por otra parte, para la gestión de la memoria DDR se incluye un subsistema controlador de memoria. Entre sus características destacan que es capaz de soportar los estándares DDR3, DDR3L, LPDDR3 y DDR4 con un ancho del bus de 64 o 32 *bits* y también el LPDDR4 de 32 bit. Adicionalmente,

incorpora ECC para cualquiera de los módulos de funcionamiento anteriores. La máxima velocidad de transmisión de datos de este módulo se consigue en el estándar DDR4, siendo esta de 2400 Mb/s. Asimismo, la máxima capacidad de almacenamiento SRAM (*Static RAM*) que es capaz de gestionar es de 32 GB [3]. Para comunicarse con el resto de módulos de la placa de prototipado, el subsistema DDR dispone de 6 interfaces AXI que les permite realizar operaciones de lectura y escritura en memoria [20], [10].

3.2.5. Periféricos del PS

La plataforma Zynq UltraScale+ dispone de una serie de interfaces de entrada y salida, gran parte de las cuales son multiplexadas en el bloque MIO (*Multiplexed Input/Output*) en su acceso al MPSoC. Asimismo, dentro de las conexiones seriales gestionadas por el bloque MIO se encuentran 2 UART (*Universal Asynchronous Receiver Transmitter*), 2 SPI (*Serial Peripheral Interface*), 2 USB 3.0 (*Universal Serial Bus*), 2 CAN (*Controller Area Network*), 2 I2C (*Inter-Integrated Circuit*) y 4 *Gigabit Ethernet*. Por otra parte, dispone de 2 interfaces SD/eMMC para tarjetas de memoria, las cuales soportan los estándares SD (*Secure Digital*) 3.0 y eMMC (*embedded MultiMediaCard*) 4.51 [10]. Además, se disponen de 3 bancos de 26 pines GPIO (*General Purpose Input Output*).

Otro recurso proporcionado en la placa de prototipado para la gestión de interfaces de entrada y salida es el SIOU (*Serial Input Output Unit*). Este módulo contiene 4 controladores de periféricos de alta velocidad consistentes en transceptores PS-GTR. Estos transceptores poseen una velocidad de flujo de datos de hasta 6 Gb/s, por lo que pueden operar como interfaz de acceso del MPSoC a cualquiera de los bloques periféricos de alta velocidad. De esta manera, este bloque incorpora 2 interfaces de comunicación SATA (*Serial Advanced Technology Attachment*), una interfaz PCIe (*Peripheral Component Interconnect Express*) y una interfaz DisplayPort al conjunto de periféricos albergados por la placa [10].

3.3. PMU y CSU

El PMU (*Platform Management Unit*) es el módulo *hardware* encargado de la inicialización del sistema y de la gestión del estado de activación de cada uno de los componentes que lo conforma. Además, se ocupa de la gestión de errores en el funcionamiento del PS de la plataforma y de la ejecución de librerías de test *software*. Para comunicarse con el resto de módulos, la PMU utiliza una interfaz AXI de 32 bits [3], [20].

El CSU (*Configuration and Security Unit*) es el módulo central de la gestión de la seguridad del sistema. Asimismo, entre sus tareas se encuentran monitorizar los sensores de temperatura,

ejecutar el modo seguro de arranque del sistema, proveer recursos de aceleración *hardware* criptográfica y almacenar y gestionar claves de seguridad. Estructuralmente, se subdivide en un bloque SPB (*Secure Processor Block*) y un bloque CIB (*Crypto Interface Block*).

El módulo SPB contiene un procesador Microblaze de triple redundancia que se encarga de ejecutar las operaciones de arranque. Para ello, también dispone de memoria ROM, una pequeña memoria RAM privada y los registros de control y status para la ejecución de operaciones en modo seguro. Otro elemento muy característico del módulo SPB es el bloque PUF (*Physically Unclonable Function*), el cual genera las claves de encriptación concretas del dispositivo físico. Por otra parte, el módulo CIB (*Crypto Interface Block*) que contiene las interfaces AES-GCM, DMA, SHA-3, RSA y PCAP (*Processor Configuration Access Port*) [20].

3.4. Operación de arranque

Para la configuración del sistema, dispone de 2 procedimientos de arranque atendiendo al grado de seguridad que se requiera para asegurar su integridad. La PMU y la CSU operan conjuntamente para la inicialización del sistema. Este procedimiento se subdivide en 3 etapas cuando no se exige ningún requisito de seguridad [10].

- **Preconfiguración.** La activación del *reset* reinicia la CSU y la PMU. A su vez, estos módulos resetearán los distintos componentes del sistema, el código *software* a ejecutar y el sistema de depurado. Posteriormente, la PMU ejecuta el programa de arranque del sistema que se encuentra almacenado en una memoria ROM propia.
- **Configuración.** Se ejecuta el código BootROM almacenado en la memoria ROM destinada a la CSU. Este código interpreta la cabecera de arranque para configurar el sistema y cargar el código FSBL (*First-Stage Boot Loader*) del PS en la memoria RAM *on-chip* (OCM). Esta cabecera define una serie de parámetros de arranque como el modo de seguridad de la inicialización o la elección del procesador que ejecutará el código FSBL. Durante el arranque, la CSU también carga en la RAM de la PMU el código *firmware* desarrollado por el usuario que establece la manera de operar del módulo PMU.
- **Posconfiguración.** Después de iniciar la ejecución del código FSBL, la CSU se encarga de desarrollar las respuestas oportunas frente a fallas de seguridad del sistema. Asimismo, la CSU se encargará de ofrecer servicios relativos a la encriptación, así como la configuración de la PL especificada en el *bitstream* generado por el usuario mediante PCAP [20].

Para establecer un modo de inicialización seguro se incorpora una etapa adicional de autenticación. Dicho proceso permite asegurar que el archivo de configuración no ha sido desarrollado o modificado por terceros, impidiendo su ejecución en el caso contrario. Esta etapa se basa fundamentalmente en el algoritmo de cifrado asimétrico RSA.

Como etapa opcional, el modo de inicialización segura ofrece un procedimiento que permite asegurar la confidencialidad, es decir, que los archivos de configuración sean ininteligibles para las personas no autorizadas. Este recurso es de gran utilidad para proteger la propiedad intelectual frente al proceso de ingeniería inversa y se fundamenta en el algoritmo de encriptación AES [10].

El fichero *bitstream* destinado a establecer la implementación del sistema en la plataforma *hardware* se puede generar utilizando la aplicación de diseño Xilinx Vivado. Dicho fichero se transfiere a la placa por medio de una interfaz de conexión disponible, ya sea QSPI, eMMC18, NAND, tarjetas SD, JTAG (*Joint Test Action Group*) y USB [22]. La elección de la interfaz de programación, se establece mediante los interruptores del *switch* SW6 presente en la placa de prototipado [23]. En la Tabla 3 se indican las opciones disponibles.

Tabla 3: Modos de arranque para la configuración del sistema [10]

Boot Mode		Mode Pins [3:0]	Description
JTAG	PS JTAG	0000	JTAG with dedicated pins, reaching all controllers on the chain.
	PJTAG (MIO #0)	1000	JTAG through MIO, reaching Arm DAP only.
	PJTAG (MIO #1)	1001	JTAG through MIO, reaching Arm DAP only.
SPI	QSPI (24b)	0001	Quad-SPI using 24 bit addressing
	QSPI (32b)	0010	Quad-SPI using 32 bit addressing
SD	SD0 (2.0)	0011	SD card controller for version 2.0
	SD1 (2.0)	0101	SD card controller for version 2.0
	SD1 LS (3.0)	1110	SD card controller for version 3.0, including voltage level-shifters
NAND		0100	NAND flash memory, with 8-bit data bus
eMMC		0110	eMMC version 4.5, using 1.8V
USB0 (2.0)		0111	USB version 2.0

Dentro de los diferentes modos de arranque disponibles, tanto el PS JTAG como el arranque por SD Card son los utilizados en este. En el caso de PS JTAG permite programar el dispositivo y el

depurado del sistema implementado. En el caso de la SD Card, facilita la integración del sistema operativo Linux para ejecutar la aplicación del sistema, así como almacenar datos de entrada y resultados.

3.5. Lógica programable

La lógica programable (PL) incorporada en el dispositivo consiste en un conjunto de elementos que operan juntamente con el objetivo de implementar la arquitectura *hardware* que requiere el sistema diseñado. En este caso podemos considerar que la lógica opera sobre un conjunto de recursos de memoria que facilitan su programación. Se trata de un conjunto de estructuras lógicas que se repiten de forma regular y que se conectan entre sí por medio de unas líneas de conexión conmutadas mediante transistores que se agrupan en matrices de conmutación.

El PL dispone de los siguientes recursos para la generación de funciones, procesamiento y almacenamiento:

- Configurable Logic Block o CLB.
- Bloques DSP.
- Almacenamiento: Ultra RAM, Block RAM, y RAM Distribuida.
- Bloques especiales: PCIe, Controladores de alta velocidad.

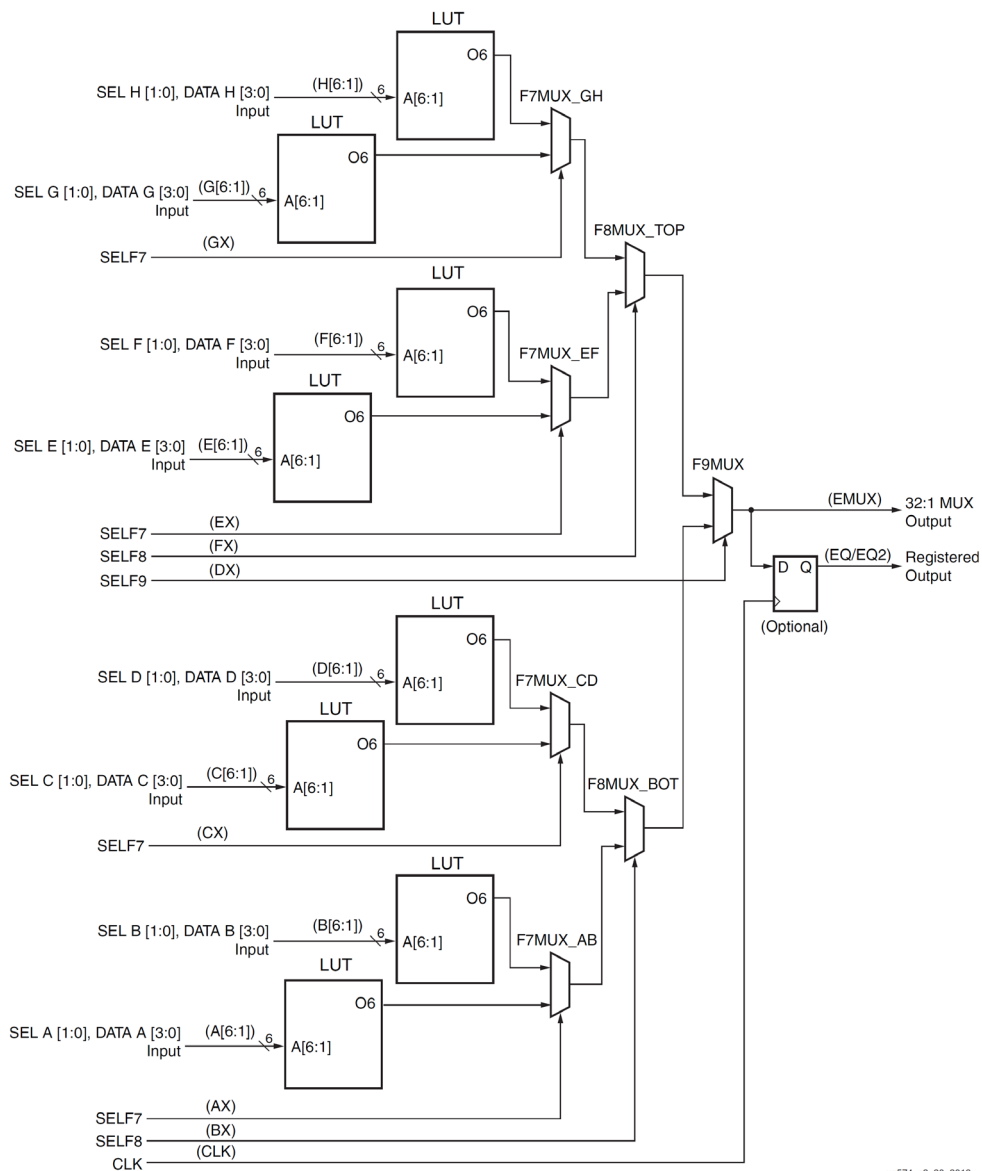
En la Tabla 4 se muestra la cantidad de los recursos más representativos que se encuentran en la parte PL del dispositivo XCZU9EG-2FFVB1156E MPSoC disponible en la placa ZCU102.

Tabla 4: Cantidad de recursos de la parte PL [14][24]

Recurso	Cantidad
CLB	274 K
DRAM (mediante LUTs)	8.8 Mb
BRAM	32.1 Mb
DSP	2520
Transceptores GTH 16.3Gb/s	24

3.5.1. CLBs

Cada CLB contiene 8 LUTs (*Look Up Tables*), 16 flip/flops y multiplexores. Además, incorporan lógica de acarreo *lookahead* de 8 bits para las operaciones aritméticas. Dicho componente dispone de una línea de entrada y otra de salida que puede conectarse a elementos externos al CLB, pudiéndose así implementar operaciones aritméticas con operandos que utilicen un mayor número de bits. Cada LUT puede configurarse como una LUT de 6 entradas o como 2 LUTs de 5 entradas, que poseen salidas independientes pero que comparten sus entradas. No obstante, la distribución de los multiplexores en la arquitectura de la CLB, la cual corresponde con la mostrada en la Figura 12, permite combinar y escoger entre las salidas de las LUTs contenidas en una CLB, pudiendo crear cualquier función lógica que no exceda de 9 entradas.



ug574_c2_20_2013

Figura 12: Conexión entre las LUTs y los multiplexores de un CLB [25]

Cada CLB conforma un *slice*, los cuales consisten en configuraciones programadas en la CLB para que esta desarrolle una funcionalidad específica. Asimismo, existen 2 posibles opciones de *slices*, las cuales son *SLICEL* o *SLICEM*. La denominación *SLICEL* hace referencia a una configuración de la CLB con el propósito de que esta implemente lógica. Por otra parte, *SLICEM* consiste en una configuración de la CLB que además de las funciones del *SLICEL* permite ser utilizada como de memoria RAM distribuida (512 bits) o como un registro de desplazamiento (256 bits).

Otro aspecto clave para lograr la versatilidad funcional de los CLB, es la posibilidad de escoger entre varias salidas. Asimismo, es posible escoger entre 4 salidas por cada LUT, siendo 2 de ellas salidas asíncronas, útiles para implementar lógica combinacional, y las otras 2 salidas síncronas útiles para lógica secuencial. Las salidas asíncronas serían la salida directa de la LUT, y la salida de alguno de los multiplexores, mientras que las síncronas corresponderían a 2 salidas de 2 flip/flops. Estos bloques pueden ser configuradas para registrar las 2 salidas de cada LUT, el valor de acarreo asociado a cada LUT, la suma de la salida de la LUT con el valor del bit de acarreo que le corresponde, la salida de los multiplexores conectados directamente a las LUTs y las entradas directas de la CLB que no son entradas de las LUTs [25].

3.5.2. DSP

Su objetivo principal es aportar la funcionalidad aritmética a la arquitectura *hardware* para aplicaciones complejas evitando el consumo de CLBs [25]. En la Figura 13 se muestra una representación de su arquitectura interna. Este recurso se compone de un sumador de 48 *bits*, un *pre-adder* de 27 *bits* y un multiplicador de 27x18 *bits*, además de una ALU y de un detector de patrones.

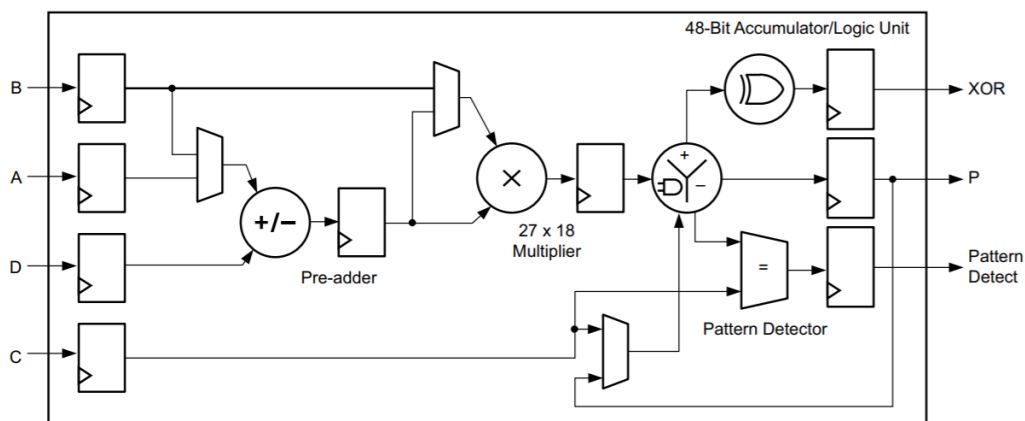


Figura 13: Diagrama de bloques del DSP [3]

En la Figura 13, se observa que el *pre-adder* opera con datos de entrada de 27 *bits* y es usado para implementar filtros simétricos. El multiplicador de 27 x 18 *bits* soporta operaciones con

signo siguiendo una representación en complemento a 2. La unidad aritmética SIMD adquieren los datos de 2 buses de entrada de 48 *bits* cada una y generan una respuesta doble de 24 *bits* o una respuesta cuádruple de 12 *bits*. Asimismo, la unidad aritmética puede configurarse para ejecutar distintas operaciones, que pueden ser suma, resta y suma acumulativa. Opcionalmente puede ejecutar hasta 10 operaciones lógicas con datos de 48 *bits*.

El resultado de la unidad aritmética o lógica puede utilizarse directamente como salida del DSP, como entrada de un módulo detector de patrón o como entrada de una XOR. Respecto a las aplicaciones del detector de patrón, este es de gran utilidad para el redondeo convergente o simétrico. Además, permite implementar funciones lógicas con datos de 96 *bits* cuando se usa conjuntamente con la unidad lógica. Por otra parte, el módulo para la operación XOR facilita la implementación de las estrategias de FEC (*Forward Error Correction*) y CRC (*Cyclic Redundancy Checking*), que permiten la detección y la corrección de errores mediante la aplicación de redundancia.

Los bloques DSP ofrecen amplias capacidades de extensión de su funcionalidad mediante componentes *hardware* adicionales, aumentando así su utilidad. Además, también presenta una configuración de *pipelining*, lo que aumenta su rendimiento [3]. Funcionalmente este bloque puede ser utilizado para el desplazamiento y multiplexación dinámicos de buses, la generación de direcciones de memoria y el mapeo de las interfaces de entrada y salida [3].

3.5.3. Memorias RAM

Aparte de la posibilidad de utilizar las CLBs como memoria distribuida, la lógica programable de los dispositivos UltraScale incorporan bloques específicos que permiten desarrollar dicha funcionalidad. Estos bloques se clasifican en bloques RAM (BRAM) y bloques UltraRAM.

Los BRAM son bloques de memoria síncronos de 36 Kb, que pueden configurarse como un único bloque de memoria o como 2 bloques de memoria independientes de 18 Kb cada uno, poseyendo además 2 puertos independientes. Cabe destacar que estos bloques de memoria permiten distintas configuraciones (32K x 1, 16K x 2, 8K x 4, 4K x 9, 2K x 18, 1K x 36 o 512 x 72), pudiéndose establecer de manera independiente en cada uno de los puertos de una BRAM. Por otra parte, también posibilita la corrección de errores de 1 *bit* y la detección de aquellos de 2 *bits* mediante la incorporación en el código de 8 *bits* adicionales. Además, los BRAM pueden operar como FIFOs.

Por otro lado, los bloques UltraRAM, son memorias de alta densidad de 288 Kb con puerto dual síncrono incorporadas en el desarrollo de las familias *UltraScale+*. Cada uno de sus puertos

pueden ser de lectura como de escritura, operando con un ancho de registros de 72 bits. Al igual que los bloques BRAM, este tipo de memorias introduce 8 bits de redundancia por medio de una codificación *Hamming*, posibilitando además la corrección de errores de 1 bit y la detección de errores de 2 bits [3]. A diferencia de los BRAM no pueden ser inicializados por el diseñador y por tanto no se pueden configurar como memorias ROM.

3.5.4. Intercomunicación PS – PL

La coordinación entre la ejecución del *software* de aplicación y la operación del acelerador *hardware* es un aspecto clave a la hora de asegurar un buen rendimiento y rapidez en el funcionamiento de sistemas que requieran funcionalidad en ambos dominios. Dicha coordinación recae en las interfaces que permiten intercomunicar las partes PS y PL del MPSoC (Figura 14). Las interfaces de comunicación en chip más relevantes son las múltiples interfaces AXI y la interfaz EMIO, las cuales se explicarán a continuación.

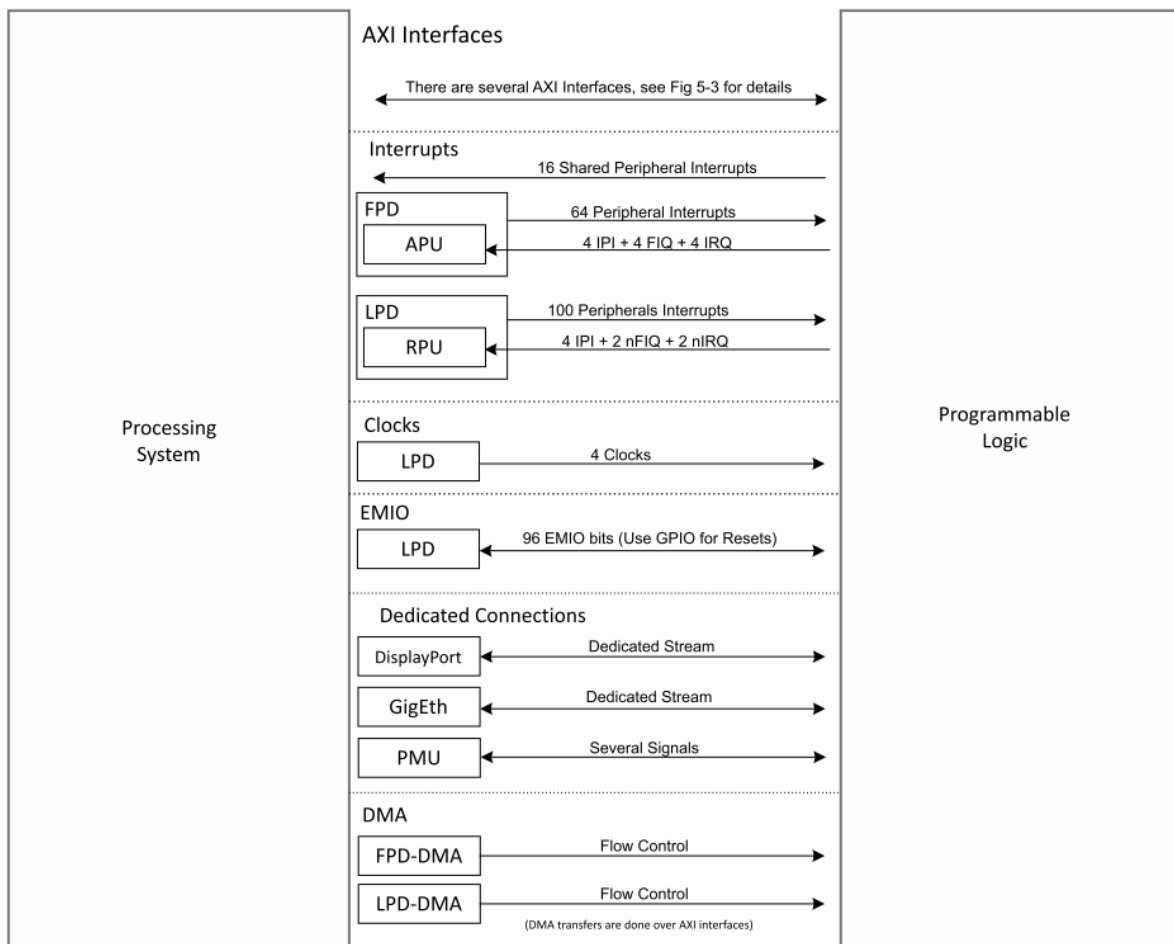


Figura 14: Conexiones entre PS y PL [16]

3.5.5. Interfaz AXI

Consiste en un conjunto de recursos de interconexión que permiten la comunicación entre distintos componentes del sistema. Su desarrollo fue realizado por la empresa ARM como parte de su arquitectura AMBA (*Advanced Microcontroller Bus Architecture*) [16]. El protocolo de comunicación sigue un modo de funcionamiento maestro-esclavo, siendo capaz de soportar tanto transmisiones de un único dato como a ráfagas. Cabe destacar que cada interfaz contiene 5 canales cuya funcionalidad es la siguiente.

- Canal de dirección de lectura: Este canal permite al módulo maestro de la comunicación AXI especificar la dirección, así como los *bits* de control de una operación de lectura de datos procedentes de la memoria de un módulo esclavo.
- Canal de dirección de escritura: Este canal permite al módulo maestro de la comunicación AXI especificar la dirección, así como los *bits* de control de una operación de escritura en la memoria de un módulo esclavo.
- Canal de lectura de datos: Se utiliza por parte de un módulo esclavo de la comunicación para especificar el dato de lectura requerido por el módulo maestro.
- Canal de escritura de datos: Su funcionalidad es utilizada por el módulo maestro para indicar el dato a escribir en la dirección indicada en el canal de dirección de escritura.
- Canal de escritura de la respuesta: Permite que el módulo esclavo notifique la correcta realización de la operación de escritura.

Para gestionar el acceso simultáneo de varios módulos a una misma memoria, se establece una política de tráfico basada en diferentes niveles de prioridad. Asimismo, el tráfico al que se le dota de mayor grado de prioridad suele ser aquel que requiera de baja latencia, siendo un posible caso el de la transferencia de datos entre procesadores APU y RPU.

Por otra parte, también existe tráfico con una mayor tolerancia frente a la latencia, pero que necesita de un *throughput* elevado. Este es el caso, por ejemplo, de la comunicación de datos entre los procesadores GPU y la lógica programable.

En una tercera categoría se encuentra el tráfico isócrono, el cual se caracteriza por requerir para su validez que la transferencia de datos se realice a una velocidad constante. Este tipo de tráfico suele ser muy tolerante a las latencias, excepto cuando está cerca de finalizar el *timeout*, adquiriendo en dichas circunstancias el mayor nivel de prioridad. Como ejemplo de este tráfico se pueden citar las transferencias de contenido de imagen y vídeo [16].

Además de para efectuar comunicaciones entre componentes *hardware* dentro de las partes PS o PL de la placa de prototipado, se disponen de enlaces AXI entre varios componentes que pertenecen al PS y la lógica programable para posibilitar la capacidad de estos dominios de operar conjuntamente. Dichos enlaces se representan conceptualmente en la Figura 15.

Cabe destacar que el módulo CCI se refiere al módulo *Cache-Coherent Interconnect* empleado como elemento de conmutación en las transacciones de memoria [16]. Su función es asegurar la coherencia del sistema, lo cual consiste en que los datos compartidos entre diversos bloques de memoria se encuentren en la versión actualizada dentro de la memoria caché de cada componente que los utilice. Por otra parte, el módulo SMMU (*System Memory Management Unit*) se encarga de traducir direcciones de memoria entre el PS y el PL, lo que permite que el PL pueda utilizar direcciones virtuales [10].

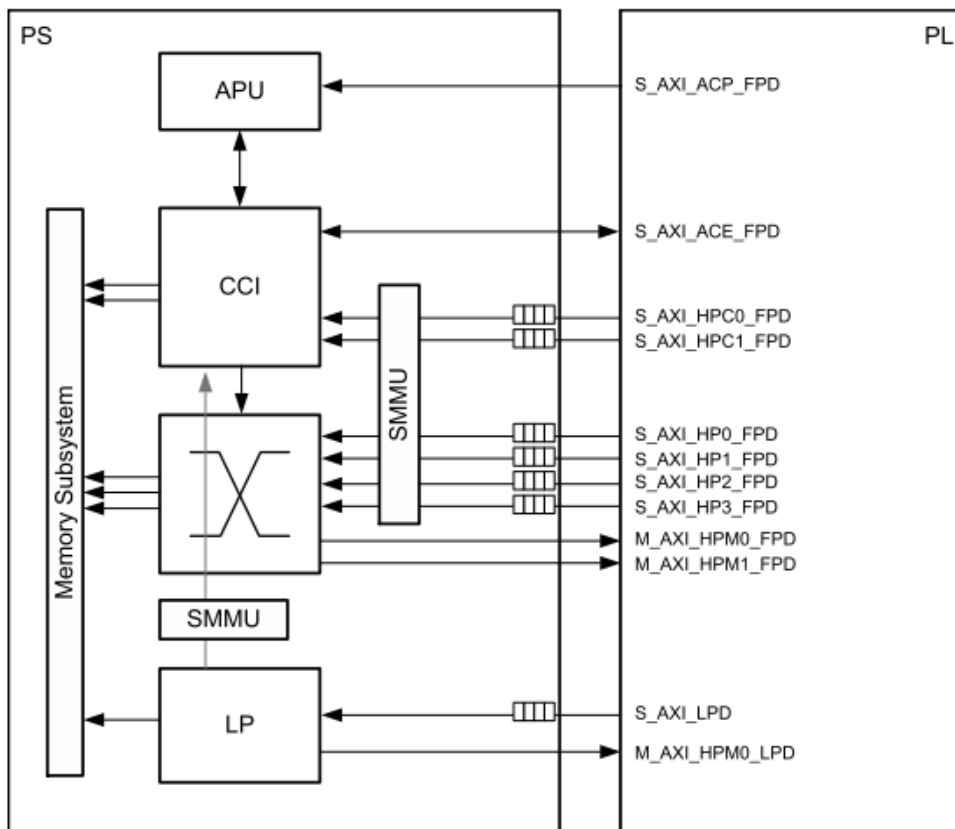


Figura 15: Conexiones AXI PS – PL [16]

Respecto a los nombres de los enlaces AXI mostrados en la Figura 15, estos se ajustan al estándar AXI: “S” para interfaces esclavas y “M” para interfaces maestras. Por otra parte, las 3 letras finales indican el dominio de potencia en el que se habilita dicho enlace, pudiendo ser dominio de

baja potencia, si el nombre termina en LPD (*low-power domain*). Asimismo, si el nombre acaba en FPD (*full-power domain*) significa que está disponible cuando el sistema opera a pleno rendimiento. Los enlaces AXI que intercomunican la PS y la PL se pueden categorizar en 4 tipos de interfaces.

- ACE (*AXI Coherency Extension*): Permite una interfaz de 128 *bits* de ancho. Su diferencia principal con respecto al AXI estándar es que dispone de 5 canales adicionales, 3 de los cuales se utilizan para establecer la coherencia del sistema y los otros 2 para verificación [10]. De esta manera, permite asegurar la coherencia de la memoria caché sin requerir la ejecución de código *software* y que las transacciones se realicen de forma ordenada [20].
- HPCP (*High Performance Coherent Port*): Es una interfaz con un ancho configurable de 128, 64 o 32 *bits* [20]. Cabe destacar que esta interfaz posibilita una transmisión rápida de los datos debido a que dispone de colas FIFO en cada puerto, facilitando la transferencia a ráfagas [10].
- ACP (*Accelerator Coherency Port*): Consiste en una interfaz esclava de 128 *bits* que posibilita un punto de acceso asíncrono y coherente a memoria caché que, en este caso, es desde la lógica programable hacia la APU. Esto permite simplificar el *software* ya que la parte PL puede disponer de varios módulos maestros desde la perspectiva de la comunicación AXI con acceso a memorias caché y a subsistemas de memoria de la misma manera que los procesadores de la APU. La coherencia es asegurada por medio del módulo SCU (*Snoop Control Unit*) [20].
- Interfaces AXI para LPD (*Low Power Domain*): Dispone de opciones de configuración de 128, 64 y 32 *bits*. Un aspecto destacable de estas interfaces es que proveen la opción de acceso por parte del PL a las memorias OCM y TCM que presenta menor latencia [20].

3.5.6. EMIO

Para lograr el acceso directo a los periféricos del sistema, la PL dispone de una interfaz denominada EMIO (*Extended MIO*). Esta interfaz permite el acceso a una parte de los periféricos disponibles para la parte PS. Además, algunos de estos periféricos presentan una manejabilidad y funcionalidad reducida cuando son utilizados por medio de dicha interfaz.

Cabe destacar que EMIO proporciona acceso a una serie de pines GPIO (*General Purpose Input/Output*) de manera exclusiva, ocurriendo esto mismo en el caso del bloque MIO. De esta manera, se logra que las partes PS y PL dispongan de pines GPIOs propios [10], [20]. En el caso de la PL, cada *pad* GPIO se encuentra encapsulado en un bloque IOB (*Input and Output Block*) que se organizan en bancos de diferentes longitudes. Asimismo, estos bancos se clasifican en 3 tipos dependiendo del aspecto funcional de una interfaz de conexión física en el que se especializan.

- *High Performance* (HP): Son capaces de operar en un rango de tensión entre 1 y 1.8 V. Se organizan en bancos de 52 pines cada uno y son especialmente útiles para comunicaciones que requieran de alta velocidad.
- *High Range* (HR): Al igual que en el caso anterior, se organizan en bancos de 52 pines. Su principal interés radica en que es capaz de soportar un amplio rango de variabilidad de la tensión, ya que opera con niveles entre 1 y 3.3 V.
- *High Density* (HD): Los pines de este tipo se organizan en bancos de 24 y funcionan con tensiones entre 1.2 y 3.3 V. Se especializa en la implementación de interfaces de baja velocidad [10]. Su diseño está optimizado para adaptarse a estándares de transmisión y recepción no diferencial, permitiendo alcanzar una velocidad máxima de 250 Mb/s [26].

En la Tabla 5 se muestran las interfaces de entrada y salida de las que dispone la placa de prototipado, se indican cuáles son accesibles desde la PS por medio de MIO, y cuáles pueden ser utilizadas en la PL mediante EMIO.

Tabla 5: Periféricos I/O disponibles para MIO y EMIO [20]

Interfaz	MIO Access	EMIO Access
GEM{0:3}	Sí (RGMII)	Sí (GMII)
SDIO{0, 1}	Sí	Sí (con prestaciones reducidas)
USB{0, 1}	Sí	No
I2C{0, 1}	Sí	Sí
SPI{0, 1}	Sí	Sí (con prestaciones reducidas)
UART{0, 1}	Sí	Sí
CAN{0, 1}	Sí	Sí
Bancos GPIO{0:2}	Sí	No
Bancos GPIO{3:5}	No	Sí
Quad-SPI	Sí	No
NAND	Sí	No
LPD_SWDT, FPD_SWDT	Sí	Sí
CSU_SWDT	No	No
TPIU Trace	Sí (hasta 16 bits)	Sí (hasta 32 bits)

3.6. Placa de prototipado

La utilización del MPSoC analizado en apartados anteriores se realiza gracias a la inserción de este integrado en una plataforma, la cual se muestra en la Figura 16, donde se señalan los elementos de utilidad.

La funcionalidad de cada uno de los elementos señalados es la siguiente:

- Lector SD: Se utiliza en la introducción mediante tarjeta SD de la aplicación ya compilada, los distintos archivos para la configuración de los recursos implicados. Esto incluye el archivo “.xclbin” para la programación del PL del MPSoC y el archivo “BOOT.BIN” para la inicialización del sistema.

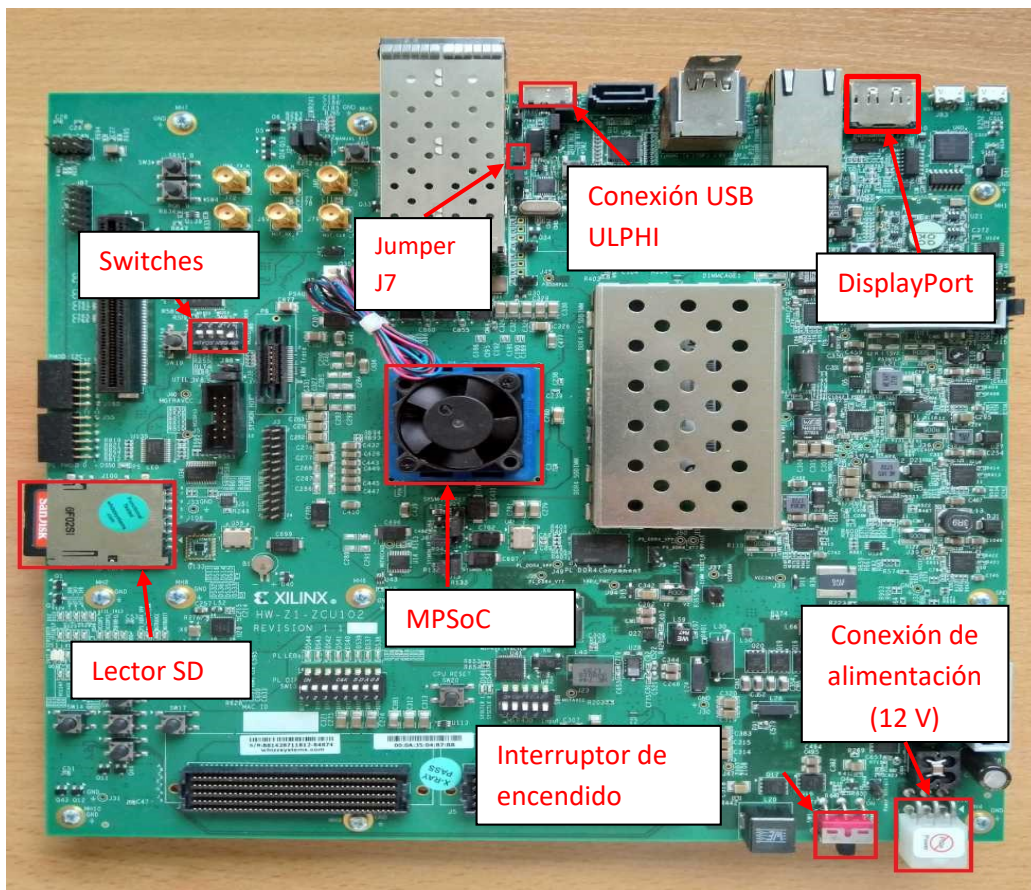


Figura 16: Placa de prototipado ZCU102

La utilidad de cada uno de los elementos señalados es la siguiente:

- *Switch SW6*: Este *switch* establece qué medio utilizar para la programación de la placa según la posición de los microinterruptores. En este caso, se ha utilizado la

opción de programación mediante tarjeta SD, estableciéndose la posición de los microinterruptores que se indica en la Figura 17.



Figura 17: Configuración del switch SW6 para arranque desde la tarjeta SD

- MPSoC: Contiene los componentes PS y PL descritos en apartados anteriores de este capítulo. Constituye el elemento principal de la placa, ya que contiene el microprocesador utilizado en la parte *software* de la aplicación, y los recursos de lógica programable utilizados en la implementación del acelerador *hardware*.
- Jumper J7: Se establece para la configuración del puerto “USB ULPI”, de manera que este pueda aceptar la recepción de teclado.
- USB ULPI: Se utiliza para la inserción de teclado con *touchpad*, el cual permitirá interactuar con la interfaz gráfica de Linux ejecutada en la propia placa.
- DisplayPort: Se utiliza para conectar la pantalla utilizada en la visualización de la interfaz gráfica.

3.7. Conclusiones

En este capítulo se han analizado los distintos elementos que componen la placa de prototipado ZCU102. En este estudio se detallaron los diferentes recursos disponibles en la lógica programable del MPSoC, así como la cantidad de estos. De esta información se concluye que la lógica programable resulta de gran idoneidad para la implementación física del acelerador *hardware*. Esto se debe a que dispone de una enorme cantidad de recursos distintos con funcionalidades simples pero versátiles que se adaptarán adecuadamente a los requisitos de la arquitectura del acelerador *hardware* ideado. Además, también se disponen en esta lógica programable de recursos de funcionalidad específica para la implementación de interfaces AXI, las cuales gestionarán eficientemente las comunicaciones entre la parte *software* y de aceleración *hardware* del proyecto.

También se analizaron los recursos de los que dispone la parte PS del MPSoC. Entre ellos se encuentran varias APUs que serán perfectamente capaces de cumplir con las exigencias de la

aplicación de *machine learning* escogida con respecto a la parte *software*. Con respecto a las capacidades de las APUs, resulta de especial interés la posibilidad que ofrecen de soportar la ejecución de sistema operativo Linux. Este hecho permite la incorporación de una interfaz gráfica desarrollada por la propia placa, lo que facilita la interacción con esta.

Capítulo 4. Metodología de diseño

4.1. Introducción

En este apartado se define la metodología empleada para transformar una descripción funcional en un lenguaje de alto nivel como C++ en un archivo ejecutable para su posterior ejecución en la placa de prototipado. Asimismo, se realiza un diagrama de flujo del procedimiento, además de una descripción del funcionamiento de cada etapa del mismo. El procedimiento se inicia con la compilación, pudiéndose diferenciar tres casos. En primer lugar, se tienen las emulaciones *hardware* y *software*, que se utilizan con fines de validación y depuración. Por otra parte, se encuentra la compilación *hardware* utilizado para generar el binario de extensión “.xclbin” cargado en el dispositivo Xilinx.

En la Figura 18 se realiza una representación de la metodología aplicada en la implementación y posterior ejecución del proyecto.

Como se observa en la Figura 18, las tres opciones de compilación fueron utilizadas. Cada una de ellas ofrece ventajas y limitaciones, siendo recomendable la utilización de todas estas en el desarrollo de una aplicación *hardware* / *software*. Las características de estas opciones son las siguientes:

- a. **Emulación *software*.** Esta opción permite compilar y enlazar rápidamente los distintos ficheros que constituyen la descripción de la aplicación. La ejecución de esta emulación se realiza en un microprocesador x86 que emula el microprocesador ARM que se utilizará realmente en la placa de prototipado como *host* de la aplicación. Para ello, se utiliza el emulador QEMU (*Xilinx Quick Emulator*). Este tipo de emulación permite verificar rápidamente la descripción algorítmica tanto en el código del *host* como en la lógica de los *kernels*.
- b. **Emulación *hardware*.** Su compilación es análoga a la de la emulación *software* para el código correspondiente al *host*. Sin embargo, en esta opción el código de los *kernels* se compila en un modelo de comportamiento RTL que se ejecuta en el simulador de Vivado. Esta compilación requiere de un tiempo mayor en su desarrollo que en el caso

de la emulación *software*, pero proporciona una visión “*cycle-accurate*” de la lógica del *kernel*.

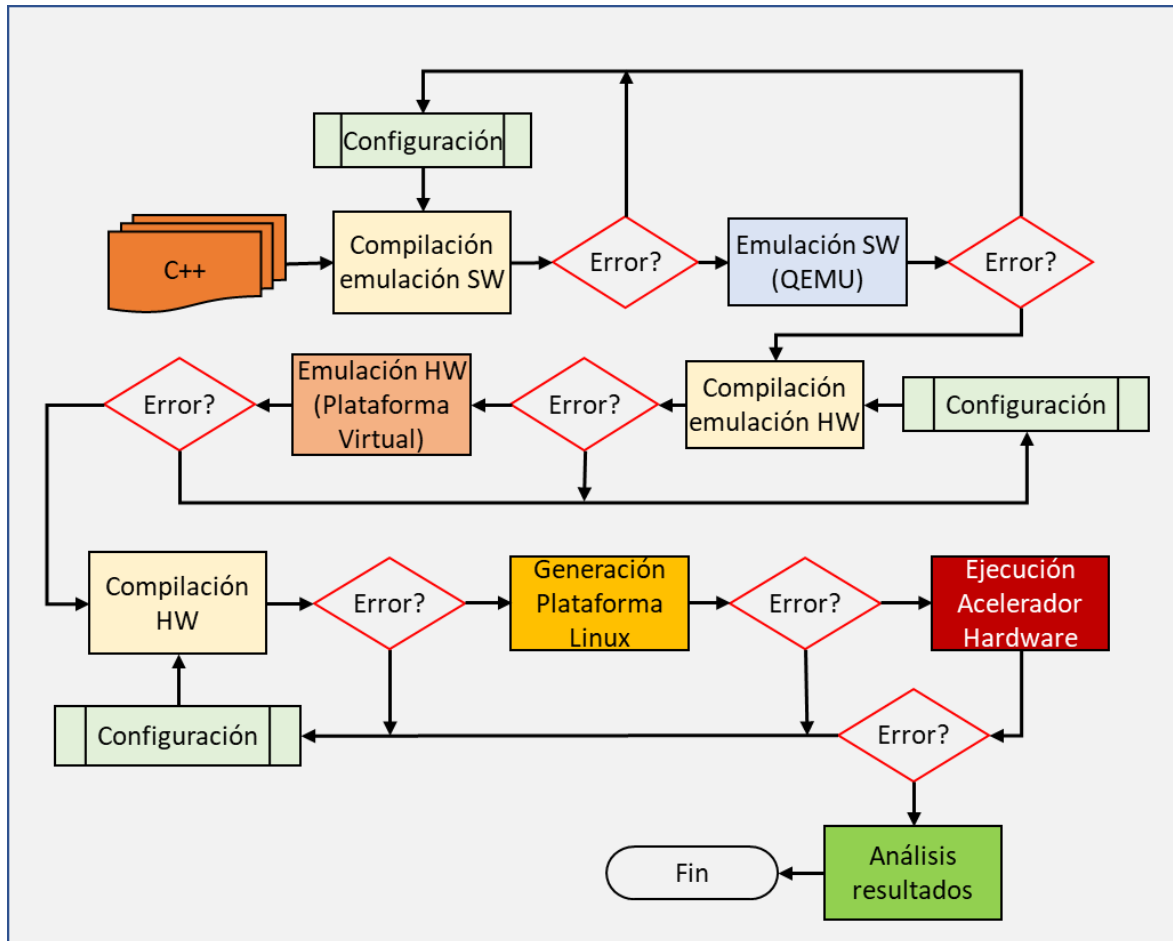


Figura 18: Flujo de diseño de la aplicación

- c. **Hardware del sistema.** La compilación del código del *host* se desarrolla de la misma manera que la observada en las dos opciones anteriores, pero esta vez su ejecución se realiza en una APU ARM de 64 bits presente en el MPSoC. Por otra parte, los *kernels* son implementados en la lógica programable del mismo MPSoC, existiendo aceleración *hardware* real, mediante la ejecución del archivo “.xclbin” producido en la compilación. Los datos de rendimiento y los resultados que captura aquí son el rendimiento real de la aplicación acelerada.

En los siguientes apartados se abordan los tipos de emulación *software/hardware* utilizados en este proyecto, así como la compilación *hardware* real.

4.2. Emulación software

El objetivo principal de la emulación *software* consiste en verificar la funcionalidad de la ejecución *software* y los *kernels*. Para la emulación *software*, tanto el código del *host* como el código del *kernel* se compilan para ejecutarse en el procesador x86. Para ello, se utiliza el compilador v++ que realiza la transformación mínima del código del *kernel* para crear el binario para la FPGA, con el fin de ejecutar el *host* y el código del *kernel* juntos.

En el contexto de la plataforma *software* de Vitis, la emulación *software* en una CPU es equivalente al proceso de desarrollo iterativo típico de la programación de CPU / GPU. En este tipo de estilo de desarrollo, un programador compila y ejecuta continuamente una aplicación a medida que se desarrolla.

4.3. Emulación hardware.

El flujo de emulación *hardware* permite verificar la corrección funcional de la descripción RTL del binario FPGA sintetizado a partir del código del *kernel* C++.

Cada *kernel* se compila en un modelo de hardware (RTL). Durante la emulación de hardware, los *kernels* se ejecutan en el simulador lógico Vivado. La emulación *hardware* proporciona estimaciones de rendimiento y recursos para la implementación del *hardware*. En la emulación *hardware*, los tiempos de compilación y ejecución son más largos que en la emulación *software*, pero proporciona una vista detallada a nivel de ciclo de la actividad del *kernel*. Para su validación, Xilinx recomienda que se utilicen pequeños *datasets* durante la emulación *hardware* a fin de mantener los tiempos de ejecución manejables.

4.4. Compilación para ejecución hardware

Dada la naturaleza híbrida del sistema, la compilación del proyecto partiendo de la descripción funcional en C++ sigue dos flujos de diseño. Asimismo, uno de ellos correspondería a la compilación para la ejecución en PL del acelerador hardware, la cual precisaría de los archivos "fpga_kmeans.cpp", "fpga_kmeans.h", "krnl_kmeans.cpp", "krnl_kmeans.h", "kmeans_config.h" y "krnl_kmeans.ini" como código fuente. Por otra parte, se encuentra la compilación para ejecución en PS, que utilizará como código fuente el descrito en el resto del conjunto de archivos en C/C++ del proyecto.

Con respecto a la compilación del acelerador hardware las opciones establecidas por Xilinx con respecto al flujo de diseño son las presentadas en la Figura 19.

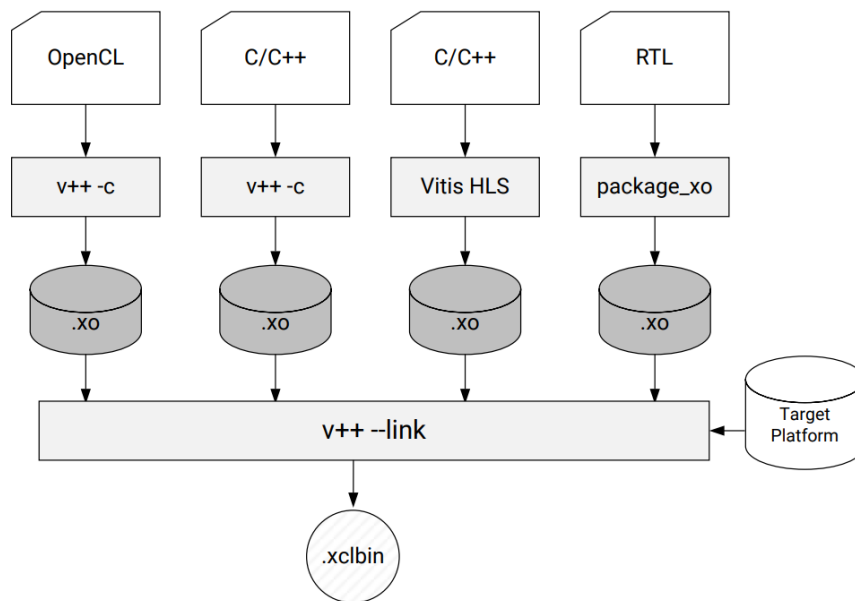


Figura 19: Opciones de compilación de hardware programable [27]

En el caso de esta aplicación, el compilador utilizado se corresponde a “v++”. De esta manera, cada *kernel* descrito en el código fuente es compilado por este compilador, generando como resultado un archivo de extensión “.xo” (*Xilinx object*). Posteriormente, el compilador “v++” enlaza los distintos *kernels* para conformar el acelerador *hardware*. Dicho proceso implicará que el compilador evalúe las características de la placa de prototipado, las cuales serán directamente extraídas del archivo especificado como parámetro “DEVICE” en la invocación de la función “build” descrita en el *makefile*. Como resultado de ello, se genera el archivo “.xclbin”, el cual constituye un *bitstream* que programa el PL de la placa para virtualizar el acelerador *hardware*.

Respecto al *software*, se utiliza el compilador “aarch64-linux-gnu-g++”, el cual constituye una variante de “*cross-compiler*” orientada a microprocesadores ARM de arquitectura de 64 *bits*, como es el caso de las APUs de la placa de prototipado empleada. Cabe destacar que dicho compilador se encuentra incluido como uno de los recursos *software* que conforma la plataforma *software* de la aplicación Vitis. Como resultado de esta compilación, cada archivo que conforma el código fuente de la ejecución es transformado en un archivo objeto, es decir, de extensión “.o”. Estos nuevos archivos son enlazados entre sí utilizando la librería compartida XRT (*Xilinx Runtime*). Como resultado de esta última operación, se obtiene el ejecutable a implementar en la parte PS de la placa de prototipado.

4.5. Empaquetado del sistema

Después de compilar y vincular el código del *kernel*, se empaqueta el binario del dispositivo, junto con los archivos de soporte necesarios en un solo archivo, para compilar un paquete que se pueda ejecutar para emulación *software* o *hardware*, o para ejecución en la placa de prototipado. El empaquetado se realiza mediante la opción “package” o “-p” del compilador “v++”. Este comando le permite empaquetar su diseño y definir varios archivos necesarios para arrancar y configurar el dispositivo Xilinx para su uso durante la emulación o en sistemas de producción. El empaquetamiento resulta un paso obligatorio para todas las plataformas de procesadores integrados, como es el caso de este proyecto. Las acciones que efectúa esta opción son crear una tarjeta SD u otros medios para la programación de la placa, definir el sistema operativo, cargar la aplicación y el código del *kernel* en formato .xclbin [27].

4.6. Conclusiones

En este capítulo se ha abordado la visión del procedimiento incurrido en la emulación y la ejecución del proyecto desde la perspectiva de las herramientas *software* utilizadas durante la metodología de diseño aplicada. De esta manera, se obtiene una visión de los procesos en bajo nivel que dan soporte a diversas acciones y decisiones tomadas en el desarrollo del proyecto.

Capítulo 5. Proyecto *kmeans*

5.1. Introducción

Para abordar el desarrollo del trabajo se parte de un *software* de referencia proporcionado por Xilinx que implementa el *core* del algoritmo de *kmeans* para la realización del *clustering* básico de datos con distintas características (*features*) [28]. El proyecto incluye un conjunto de funciones desarrolladas en C++, un conjunto de utilidades para su compilación y posterior presentación de resultados. En este capítulo se explica la arquitectura de la aplicación, los detalles de la implementación, poniendo especial énfasis en distintos aspectos relacionados con su implementación *hardware* y las transformaciones realizadas para poder realizar la implementación con distintos *datasets* orientados al procesamiento de imágenes hiperespectrales.

Para la implementación de este trabajo se usa una compilación basada en Makefiles, invocando el compilador de Vitis (V++) y las herramientas necesarias para verificar el diseño y generar sobre la tarjeta SD los ficheros necesarios para el arranque del sistema sobre la placa ZCU102 de Xilinx. La compilación está soportada en Linux, creando una solución empotrada en la que el software se ejecuta en la APU del dispositivo Zynq UltraScale+ y el acelerador se implementa en el PL.

5.2. Arquitectura de la aplicación *kmeans*

La aplicación *kmeans* se invoca desde la función *main* que está incluida en el fichero *host.cpp*. En la Figura 20 se muestran los ficheros incluidos en la función *main*. Como se puede apreciar, la función *cluster* se invoca desde la función *main* siendo el núcleo principal de computación. Además, dicha función *cluster* invoca diversas funciones pertenecientes a la clase *FPGA_KMEANS*, la cual engloba las funciones necesarias para la ejecución de la parte del algoritmo correspondiente a la FPGA. Además, hay un conjunto de funciones adicionales que facilitan el análisis de las opciones de línea de comandos de la aplicación, la lectura y escritura de *buffers* y el movimiento de datos entre los distintos *buffers* del sistema. El gráfico de llamadas se muestra en la Figura 21.

A continuación, se describen los principales ficheros fuente de la aplicación.

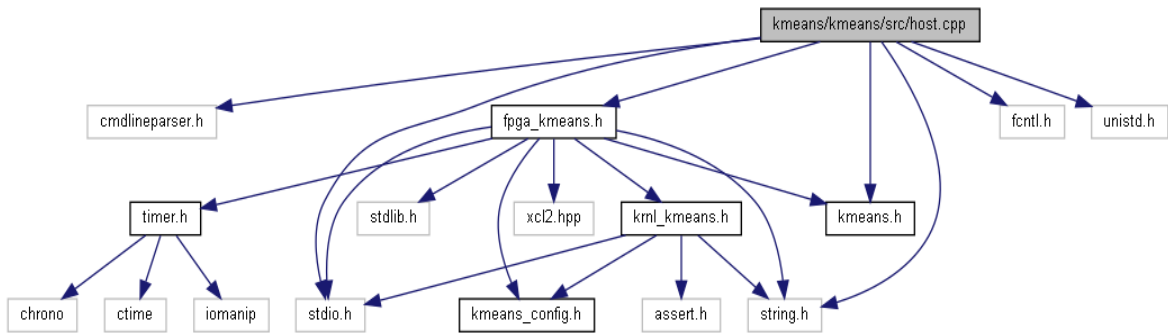


Figura 20: Dependencias de ficheros de la aplicación

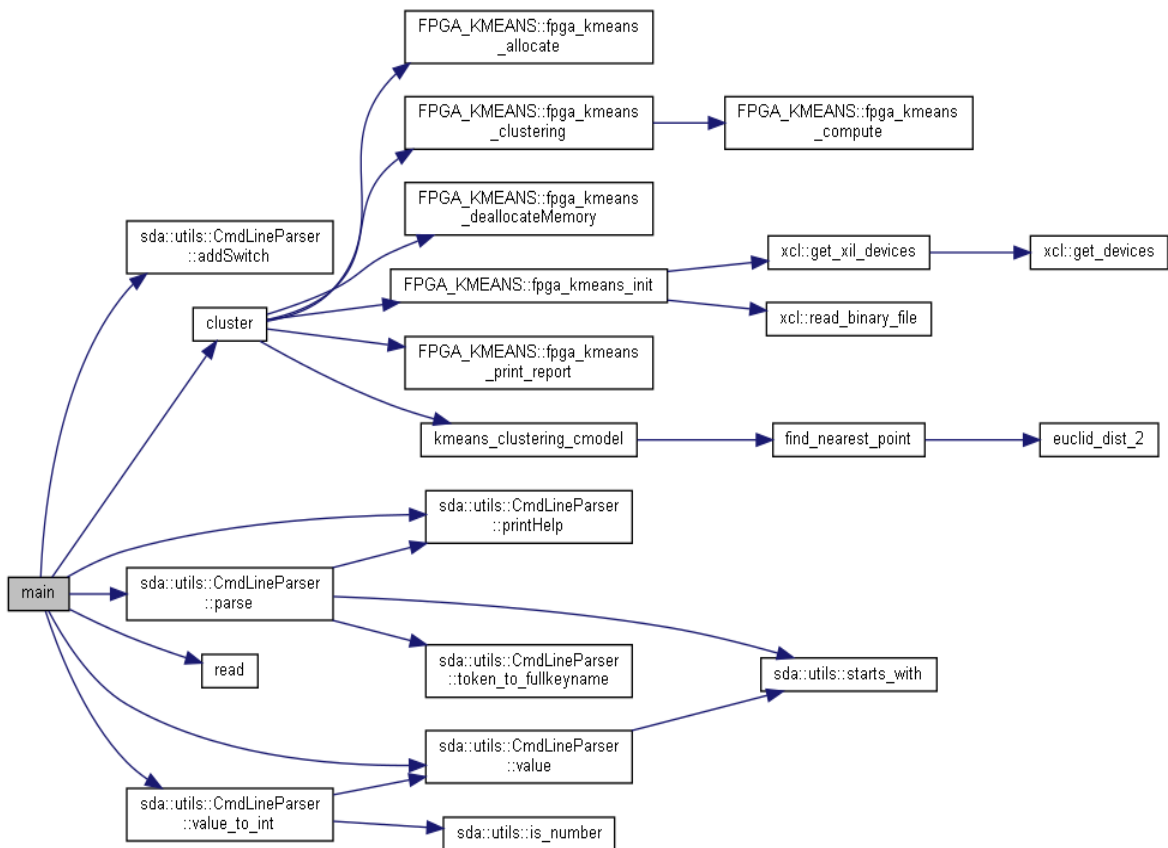


Figura 21: Grafo de llamadas de la función main

5.3. Función main

Como se indicó, el fichero host.cpp contiene la función principal de la ejecución del sistema, (función *main*), que se ha presentado en la Figura 21. Esta función requiere de una serie de variables globales, las cuales se detallan a su inicio como se muestra en el Código 1.

```

int main(int argc, char **argv) {
    float *buf;
    char line[1024];
    int isBinaryFile = 0;
    float threshold = 0.001;
    int nclusters = 5;
    int nfeatures = 0;
    int npoints = 0;
    float **features;
    float **cluster_centres = NULL;
    int i, j;
    int nloops = 1; // Default value
    int isOutput = 0;

```

Código 1: Variables locales de la función main

Cabe destacar que algunos de los parámetros de entrada del sistema deben ser especificados por el usuario mediante línea de comandos. Debido a ello, será necesario un objeto de la clase `sda::utils::CmdLineParser`. cuyo objetivo es realizar el paso de información desde el terminal de líneas de comando hacia la aplicación. Dicha clase se implementa en el fichero `<common/includes/cmdparser/cmdlineparser.h>` disponible en la jerarquía del proyecto. De esta manera, se utilizará el comando `addSwitch` presente en esta clase para establecer cuáles serán los argumentos especificables mediante la interfaz de usuario CLI (Command Line Interface).

```

parser.addSwitch("--xclbin_file", "-x", "input binary file string", "");
parser.addSwitch("--input_file", "-i", "input test data flie", "");
parser.addSwitch("--compare_file", "-c", "Compare File to compare result",
    "");

parser.addSwitch("--nclusters", "-n", "number of clusters", "5");
parser.addSwitch("--threshold", "-t", "thresold value", "0.001");
parser.addSwitch("--output", "-o", "output cluster center coordinates", "0");
parser.parse(argc, argv);

```

Código 2: Establecimiento de los argumentos a especificar mediante el terminal de comandos

A continuación, la función realiza la extracción del contenido indicado por el usuario mediante un terminal de comandos. Para ello, se empleará funciones “value” como se especifica en Código 3.

```

// Read settings
std::string binaryFile = parser.value("xclbin_file");
std::string filename = parser.value("input_file");
std::string goldenfile = parser.value("compare_file");

nclusters = parser.value_to_int("nclusters");
threshold = atof((parser.value("threshold")).c_str());
isOutput = parser.value_to_int("output");

```

Código 3: Extracción de los parámetros indicados mediante la ejecución por línea de comandos

Las distintas opciones de *parser* permiten obtener un *string* o un entero o un flotante a partir de un valor clave.

Como se puede observar, el resultado de la ejecución extrae los nombres de referencia de los archivos y parámetros a especificar:

- “*xclbin_file*”, archivo de programación del *hardware* programable para la implementación de los *kernels*.
- “*input_file*”, describe, a partir de una serie de atributos, cada uno de los elementos a clasificar mediante el proceso de *clustering*.
- “*compare_file*”, se utiliza en la verificación del sistema, comparándolo con los resultados de la ejecución de la aplicación.
- “*ncluster*”, la cantidad de *cluster* a establecer, siendo 5 su valor por defecto si el usuario no lo especifica.
- “*threshold*”, especifica el número de cambios máximos permitidos en la última etapa de la ejecución de *k-means* con respecto a la clasificación de los elementos. Asimismo, si se establece el valor por defecto 0.001, significaría que el algoritmo deberá verificar en su última evaluación de cada elemento que todos estos están correctamente clasificados en los *clusters*.
- “*output*”, salida del algoritmo

Posteriormente, se carga el contenido del archivo “*input_file*”, el cual es almacenado en variables locales al *main*, y se ejecuta el proceso de *clustering* especificado en la función *cluster* de este archivo. La aplicación notificará al usuario los *clusters* resultantes mediante la indicación de las coordenadas de los centroides de los mismos. Finalmente, se liberarán mediante las funciones *free()* y *delete()* los recursos ocupados mediante asignación dinámica de memoria. Estas dos últimas acciones se desarrollarán conforme al Código 4.

```

/*****
                                COMMAND LINE OUTPUT
*****/
// Cluster center coordinates: displayed only for when k=1
if (isOutput == 1) {
    printf("\n===== Centroid Coordinates =====\n");
    for (i = 0; i < nclusters; i++) {
        printf("\n\n%d:", i);
        for (j = 0; j < nfeatures; j++) {
            printf(" %.2f", cluster_centres[i][j]);
        }
        printf("\n\n");
    }
}
// Free up memory
delete fpga;
free(features[0]);
free(features);

```

Código 4: Notificación de los clusters obtenidos y liberación de recursos

Como se comentó con anterioridad, este archivo incluye la función *cluster*, la cual engloba el desarrollo del proceso de *clustering* una vez que los datos de entrada al sistema han sido correctamente preprocesados. Esta función comienza con la creación de sus variables locales y la reserva dinámica de espacio de memoria para los *arrays* “membership” y “cmodel_membership”, tal como se muestra en el Código 5.

```

void cluster(FPGA_KMEANS *fpga,
            int npoints, /* number of data points */
            int nfeatures, /* number of attributes for each point */
            float **features, /* array: [npoints][nfeatures] */
            int nclusters, /* range of min to max number of clusters */
            // int max_nclusters,
            float threshold, /* loop terminating factor */
            float ***cluster_centres, /* out: [best_nclusters][nfeatures] */
            int nloops, /* number of iteration for each number of clusters */
            std::string &binaryFile, /* Binary file string */
            const char *goldenFile) {

    int *membership; /* which cluster a data point belongs to */
    int *cmodel_membership; /* which cluster a data point belongs to */
    float **tmp_cluster_centres; /* hold coordinates of cluster centers */
    int i;

    // Allocate memory for membership
    membership = (int *)malloc(npoints * sizeof(int));
    cmodel_membership = (int *)malloc(npoints * sizeof(int));
    if ((membership == NULL) | (cmodel_membership == NULL)) {
        fprintf(stderr, "Error: Failed to run malloc\n");
        exit(1);
    }
}

```

Código 5: Inicio de la función cluster

El grafo de llamadas de la función “cluster” se muestra en la Figura 22. Como se puede apreciar, se invocan distintas funciones de la clase *FPGA_KMEANS* y a la función “kmeans_clustering_cmodel” (Código 6).

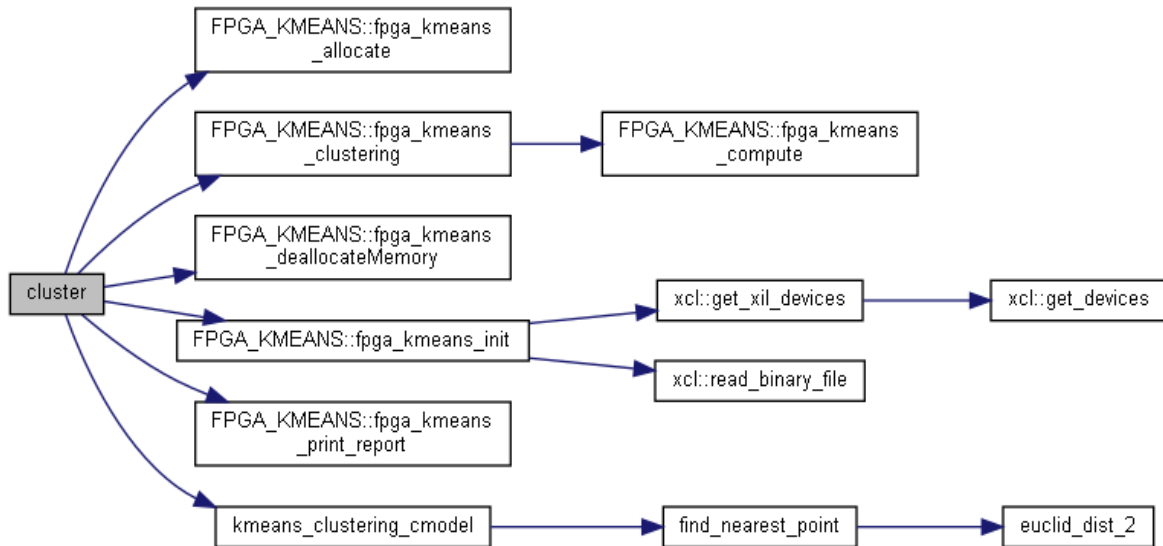


Figura 22: Grafo de llamadas de la función cluster().

```

fpga->fpga_kmeans_init(binaryFile);

// Allocate device memory
fpga->fpga_kmeans_allocate(npoints, nfeatures, nclusters);

// Iterate nloops times for each number of clusters
for (i = 0; i < nloops; i++) {
    printf("Running device execution \n");
    // Initialize initial cluster centers
    tmp_cluster_centres =
        fpga->fpga_kmeans_clustering(features, threshold, membership);
}
    
```

Código 6: Ejecución del proceso de clustering mediante funciones de fpga_kmeans

Una vez ejecutado el proceso de *clustering*, se efectuará la verificación del correcto funcionamiento del sistema *hardware-software*. Para ello, podrá utilizarse, siempre que se encuentre habilitado, la ejecución de un modelo *software* del proceso *k-means* enteramente escrito en C, cuyo desarrollo se implementa en la función “*kmeans_clustering_cmodel*” del archivo C homónimo. Asimismo, la utilización de esta función se muestra en el Código 7.

```

#ifdef VERIFY_USING_CMODEL
    printf("Running host execution \n");
    int iteration;
    float **tmp_cmodel_cluster_centres; // Hold coordinates of cluster centers
                                        // of Cmodel

    auto h_start = std::chrono::high_resolution_clock::now();
    tmp_cmodel_cluster_centres =
        kmeans_clustering_cmodel(features, nfeatures, npoints, nclusters,
                                threshold, &iteration, cmodel_membership);
    auto h_end = std::chrono::high_resolution_clock::now();
    auto h_time = std::chrono::duration<double, std::nano>(h_end - h_start);
    printf("Host iteration %d \n", iteration);
    printf("Host execution time %f ms\n", h_time.count() / 1E6);
#endif
    
```

Código 7: Verificación mediante modelo software desarrollado en C

Como puede observarse en el código anterior, el sistema medirá el tiempo de ejecución del modelo C y guardará las coordenadas de los centroides de los *clusters* resultantes en la variable “*tmp_cmodel_cluster_centres*”. Al terminar la ejecución se realizará una comparación entre los resultados obtenidos en cada una de las alternativas del algoritmo *k-means* como se muestra en el Código 8. Dicha comparación se realizará evaluando la membresía de los elementos agrupados en *clusters*.

```
int mismatch = 0;
int j;
for (j = 0; j < npoints; j++) {
    if (cmodel_membership[j] != membership[j]) {
        mismatch++;
    }
}
float mismatch_rate = float(100 * mismatch) / npoints;
if (mismatch_rate > 10) {
    printf("FAILED:Based on C-Model: Points Membership Mismatch %d "
           "Mismatch Rate %.3f \n",
           mismatch, mismatch_rate);
} else {
    printf("PASSED:Based on C-Model: Points membership with Match "
           "Rate %.3f and mismatch %d with Cmodel. \n",
           100.0 - mismatch_rate, mismatch);
}
if (tmp_cmodel_cluster_centres) {
    free((tmp_cmodel_cluster_centres)[0]);
    free(tmp_cmodel_cluster_centres);
}
}
```

Código 8: Evaluación de la discordancia entre los resultados del modelo hardware-software y del modelo software del *k-means*

Como último paso tras la ejecución del modelo en C de *k-means*, se almacenarán los resultados de la misma en un archivo con el nombre “*membership_cmodel.out*” como se muestra en el Código 9.

```
FILE *outFileC;
outFileC = fopen("membership_cmodel.out", "w");
for (int j = 0; j < npoints; j++) {
    fprintf(outFileC, "%d\n", cmodel_membership[j]);
}
fclose(outFileC);
#endif
```

Código 9: Almacenamiento de los resultados de las agrupaciones de los elementos determinadas por el modelo en C de *k-means*

Después de la opción de verificación mediante modelado en C de *k-means*, el algoritmo prosigue guardando los resultados de la ejecución del sistema *hardware-software* en un archivo denominado “*membership.out*”. Para ello, desarrollará un procedimiento expuesto en el Código 10 que, como puede observarse, es análogo al del Código 9.

```
FILE *outFile;
outFile = fopen("membership.out", "w");
for (int j = 0; j < npoints; j++) {
    fprintf(outFile, "%d\n", membership[j]);
}
fclose(outFile);
```

Código 10: Almacenamiento de los resultados de las agrupaciones de los elementos determinadas por el modelo hardware-software de k-means

Posteriormente, se verificará que la clasificación de los elementos mediante la versión *hardware-software* del algoritmo *k-means* del proyecto ha sido correcta evaluando su concordancia con los resultados esperados y que se especifican en el “compare_file” previamente comentado. Como resultado de esta comparación, la aplicación notificará al usuario del grado de similitud observado. Este proceso se ejecuta de acuerdo a lo mostrado en el Código 11.

```
if (goldenFile && strcmp(goldenFile, "")) {
    int mismatch = 0;
    // Compare the result with expected golden file
    FILE *inFile;
    if ((inFile = fopen(goldenFile, "r")) == NULL) {
        fprintf(stderr, "Error: no such file (%s)\n", goldenFile);
        exit(1);
    }
    for (int j = 0; j < npoints; j++) {
        int value;
        fscanf(inFile, "%d", &value);
        if (value != membership[j])
            mismatch++;
    }
    float mismatch_rate = float(100 * mismatch) / npoints;
    if (mismatch_rate > 10) {
        printf("FAILED:Based on Golden File: Points Membership "
            "Mismatch %d Mismatch Rate %.4f \n",
            mismatch, mismatch_rate);
    } else {
        printf("PASSED:Based on Golden File: Points membership "
            "with Match Rate %.4f and mismatches only %d "
            "compare to GoldenFile. \n",
            100.0 - mismatch_rate, mismatch);
    }
}
```

Código 11: Comparación entre los resultados de la ejecución de k-means y los resultados esperados

Finalmente la función de *cluster* termina con la liberación de los recursos utilizados durante la función, como se indica en el Código 12.


```

if (*cluster_centres) {
    free((*cluster_centres)[0]);
    free(*cluster_centres);
}
*cluster_centres = tmp_cluster_centres;
}
fpga->fpga_kmeans_deallocateMemory();
fpga->fpga_kmeans_print_report();

free(membership);
free(cmodel_membership);

```

Código 12: Liberación final de los recursos empleados en la función *cluster*

5.4. fpga_kmeans

En este fichero se describen una serie de funciones requeridas durante el proceso de *clustering* que se encargan de la gestión de los recursos *hardware*. Asimismo, el código dispone de 2 alternativas para la gestión de dichos recursos, que son mediante objetos OpenCL, los cuales se utilizarán indirectamente por medio de *wrappers* de C++, o utilizando sentencias `#pragma HLS`. Las funciones requeridas se componen en las siguientes [29].

5.4.1. fpga_kmeans_init

Se encarga de la inicialización de los *kernels* en el caso de utilizarse OpenCL. La función comienza con la búsqueda de los dispositivos conectados a la plataforma de Xilinx y la extracción del contenido del fichero para la programación del *hardware* programable. El contenido del archivo binario será gestionado por medio del objeto de la clase `cl::Program`, como se expone en el Código 13. El grafo de llamadas de la función “*fpga_kmeans_init*” se muestra en la Figura 23.

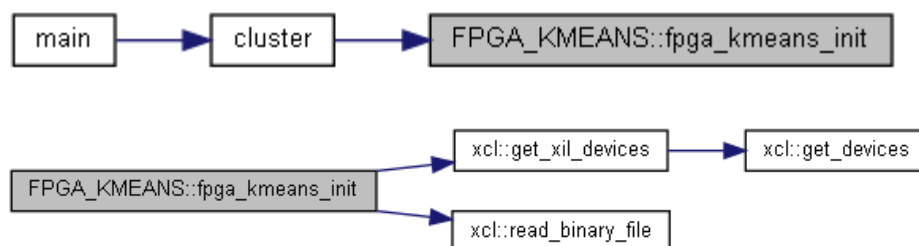


Figura 23: Grafo de llamadas de la función *fpga_kmeans_init*

```

int FPGA_KMEANS::fpga_kmeans_init(std::string &binaryFile) {
    TIMER_START(5);

#ifdef __USE_OPENCL__
    cl_int err;

    // get_xil_devices() is a utility API which will find the xilinx
    // platforms and will return list of devices connected to Xilinx platform
    auto devices = xcl::get_xil_devices();
    // read_binary_file() is a utility API which will load the binaryFile
    // and will return the pointer to file buffer.
    auto fileBuf = xcl::read_binary_file(binaryFile);
    cl::Program::Binaries bins{{fileBuf.data(), fileBuf.size()}};

```

Código 13: Búsqueda de los dispositivos y extracción de archivo binario para la programación

Una vez descargado el contenido del fichero binario, se procede a la programación de los dispositivos. Al desconocerse originariamente cuáles son los dispositivos correctos, la aplicación probará con cada uno de los que fueron detectados por la función “xcl::get_xil_devices()” del Código 13. La programación implicada en este aspecto se muestra en el Código 14.

```

int valid_device = 0;
for (unsigned int i = 0; i < devices.size(); i++) {
    auto device = devices[i];
    // Creating Context and Command Queue for selected Device
    OCL_CHECK(err, m_context = cl::Context(device, NULL, NULL, NULL, &err));
    OCL_CHECK(err,
        m_q = cl::CommandQueue(m_context, device,
            CL_QUEUE_PROFILING_ENABLE |
            CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE, &err));

    std::cout << "Trying to program device[" << i
        << "]: " << device.getInfo<CL_DEVICE_NAME>() << std::endl;
    m_prog = cl::Program(m_context, {device}, bins, NULL, &err);
    if (err != CL_SUCCESS) {
        std::cout << "Failed to program device[" << i
            << "]" with xclbin file!\n";
    } else {
        std::cout << "Device[" << i << "]: program successful!\n";
        valid_device++;
        break; // we break because we found a valid device
    }
}
if (valid_device == 0) {
    std::cout << "Failed to program any device found, exit!\n";
    exit(EXIT_FAILURE);
}

```

Código 14: Programación de los dispositivos

La programación de cada dispositivo implicado comienza, con la creación de un objeto “m_context” de la clase “cl::Context” y un objeto “m_q” de la clase “cl::CommandQueue”. La clase “cl::Context” permite crear un contexto, mientras que la clase “cl::CommandQueue” permite gestionar la comunicación entre la programación *software* y el acelerador *hardware*. Posteriormente, se crea la programación OpenCL del *hardware* programable por medio de un

objeto de la clase “cl::Program” denominado “m_prog”. Luego, la aplicación notificará cuáles de las programaciones de los dispositivos se ha efectuado correctamente y cuáles no.

Para implementación de *kernels* cuya constitución obedezca a la programación definida en “m_prog”, se utilizarán objetos de la clase “cl::Kernel”, generándose tantos de estos como instancias se desean implementar dentro de la FPGA. Asimismo, esta parte del código se evidencia en el Código 15.

```

for (int i = 0; i < NUM_CU; i++) {
    std::string cuname = "kmeans:{kmeans_" + std::to_string(i + 1) + "}";
    OCL_CHECK(err, m_kernel_kmeans[i] = cl::Kernel(m_prog, cuname.c_str(),
        &err));
}
#endif

```

Código 15: Implementación de los kernels mediante OpenCL

5.4.2. fpga_kmeans_allocate

El objetivo de esta función consiste en la creación de los registros accedidos por los *kernels* tanto para recibir información procedente del *host* como para depositar los resultados de su ejecución. Para ello, se reservará espacio de memoria de manera dinámica para cada registro, utilizando la función “malloc” como se detalla en el Código 16. El grafo de llamadas de la función “fpga_kmeans_allocate” se muestra en la Figura 24.



Figura 24: Grafo de llamadas de la función fpga_kmeans_allocate

```
m_new_memberships = (int *)malloc(m_buf_members_sz);
if (m_new_memberships == NULL) {
    fprintf(stderr, "Error: Failed to allocate memory for
                m_new_memberships\n");
    exit(EXIT_FAILURE);
}

for (int i = 0; i < NUM_CU; i++) {
    m_new_centers[i] = (unsigned int *)malloc(m_buf_centers_sz);
    if (m_new_centers[i] == NULL) {
        fprintf(stderr, "Error: Failed to allocate memory for m_new_centers\n");
        exit(EXIT_FAILURE);
    }
}

m_scaled_clusters = (unsigned int *)malloc(m_buf_cluster_sz);
if (m_scaled_clusters == NULL) {
    fprintf(stderr, "Error: Failed to allocate memory for
                m_scaled_clusters\n");
    exit(EXIT_FAILURE);
}

m_scaled_feature = (unsigned int *)malloc(m_buf_feature_sz);
if (m_scaled_feature == NULL) {
    fprintf(stderr, "Error: Failed to allocate memory for
                m_scaled_feature\n");
    exit(EXIT_FAILURE);
}
```

Código 16: Reserva de espacio de memoria para los registros de los kernels

Si la aplicación tiene habilitada la opción de emplear recursos OpenCL, entonces se definirá cada uno de los registros como un objeto de la clase `cl::Buffer`, como se indica en el Código 17. En dicho código, la accesibilidad de los *buffers* por parte de los *kernels* se establece al encontrarse ambos elementos en un mismo contexto, es decir, que tienen acceso a un mismo objeto `cl::Context`. Por otra parte, el acceso del *host* a estos *buffers* se implementa al especificar el *flag* “CL_MEM_ALLOC_HOST_PTR”. Además, dicho *flag* vendrá acompañado del de “CL_MEM_READ_ONLY” o “CL_MEM_WRITE_ONLY” según sea un *buffer* de solo lectura o solo escritura desde la perspectiva de los *kernels*.

```

#ifdef __USE_OPENCL__
    cl_int err;
    OCL_CHECK(err, m_buf_feature = cl::Buffer(m_context, CL_MEM_ALLOC_HOST_PTR |
                                                    CL_MEM_READ_ONLY,
                                                    m_buf_feature_sz, NULL, &err));
    OCL_CHECK(err, m_buf_cluster = cl::Buffer(m_context, CL_MEM_ALLOC_HOST_PTR |
                                                CL_MEM_READ_ONLY,
                                                m_buf_cluster_sz, NULL, &err));
    OCL_CHECK(err, m_buf_members = cl::Buffer(m_context, CL_MEM_ALLOC_HOST_PTR |
                                                CL_MEM_WRITE_ONLY,
                                                m_buf_members_sz, NULL, &err));

    for (int i = 0; i < NUM_CU; i++) {
        OCL_CHECK(err, m_buf_centers[i] = cl::Buffer(
            m_context, CL_MEM_ALLOC_HOST_PTR | CL_MEM_WRITE_ONLY,
            m_buf_centers_sz, NULL, &err));
    }
#endif

    TIMER_STOP(6);

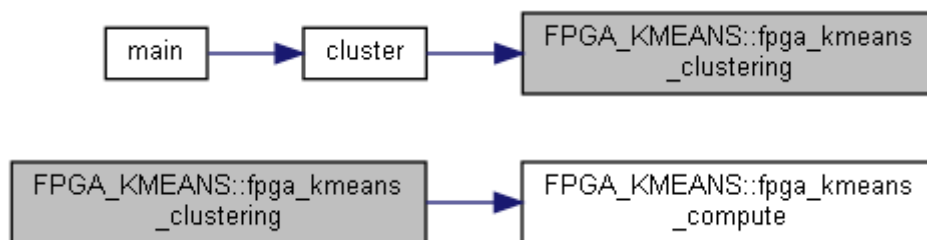
    return 0;
}

```

Código 17: Creación de los objetos `cl::Buffer`

5.4.3. fpga_kmeans_clustering

Se encarga de la inicialización de los recursos implicados en la función “`fpga_kmeans_compute`”, así como de la posterior liberación de los recursos de memoria ocupados mediante asignación dinámica. De esta manera, el código crea los registros “`new_centers_len`”, “`clusters`” y “`new_centers`”, como se muestra en el Código 18. El grafo de llamadas de la función “`fpga_kmeans_clustering`” se muestra en la Figura 25.

Figura 25: Grafo de llamadas de la función `fpga_kmeans_clustering`

```

float **
FPGA_KMEANS::fpga_kmeans_clustering(float **feature, // in: [npoints][nfeatures]
                                     float threshold, int *membership) // out: [npoints]
{
    int nfeatures = m_nfeatures;
    int npoints = m_npoints;
    int nclusters = m_nclusters;

    int temp, i, j, n = 0;
    int *new_centers_len; /* [nclusters]: no. of points in each cluster */
    float **clusters; /* out: [nclusters][nfeatures] */
    unsigned int **new_centers; /* [nclusters][nfeatures] */

    int *initial; /* used to hold the index of points not yet selected
                  prevents the "birthday problem" of dual
                  selection (?) considered holding initial cluster indices,
                  but changed due to possible, though unlikely, infinite
                  loops */
    int initial_points;

    /* nclusters should never be > npoints
       that would guarantee a cluster without points */
    if (nclusters > npoints)
        nclusters = npoints;

```

Código 18: Parámetros y variables creadas por la función “fpga_kmeans_clustering”

La inicialización de las variables se limita principalmente a la asignación dinámica de espacio de memoria para los *arrays*. No obstante, la función también se encarga de inicializar las variables del *array* “membership” a -1, además de introducir un primer elemento en cada *cluster* sin que estos se repitan. Esta última acción se ejecutará de acuerdo al código expuesto en el Código 19.

```

for (i = 0; i < npoints; i++) {
    initial[i] = i;
}
initial_points = npoints;

/* randomly pick cluster centers */
for (i = 0; i < nclusters && initial_points >= 0; i++) {
    for (j = 0; j < nfeatures; j++)
        clusters[i][j] = feature[initial[n]][j]; // remapped

    /* swap the selected index to the end (not really necessary,
       could just move the end up) */
    temp = initial[n];
    initial[n] = initial[initial_points - 1];
    initial[initial_points - 1] = temp;
    initial_points--;
    n++;
}

```

Código 19: Introducción del primer elemento de cada cluster en “fpga_kmeans_clustering”

5.4.4. fpga_kmeans_compute

Su objetivo es el desarrollo del algoritmo *k-means* dentro del sistema *hardware-software*. En una etapa inicial, además de crearse las variables locales, también se introduce el contenido del *array* “m_scaled_feature”. Dicho *array* contendrá la información del *array* “feature”, el cual

contiene la información de todos los elementos evaluados. Sin embargo, la información no será simplemente traspasada de un *array* a otro, sino que el valor de las variables será escalado y reorganizado, como se expone en el Código 20. Para el cálculo del factor de escalamiento, se empleará la función “*calculate_scale_factor*”, cuyo contenido se detalla en el Código 21. Para escalar y reubicar las variables del registro “*feature*” dentro de “*m_scaled_feature*”, se utiliza la función “*scale_and_remap_features*”, que contiene el código evidenciado en el Código 22. El grafo de llamadas de la función “*fpga_kmeans_compute*” se muestra en la Figura 26.

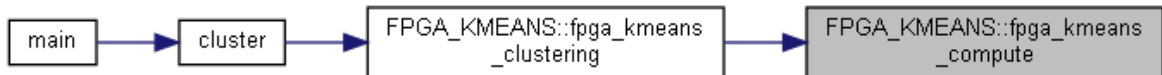


Figura 26: Grafo de llamadas de la función *fpga_kmeans_compute*

```

void FPGA_KMEANS::fpga_kmeans_compute(
    float **feature, // in: [npoints][nfeatures]
    float threshold, int *membership, float **clusters, int *new_centers_len,
    unsigned int **new_centers) {
    int loop = 0;
    int delta = 0;

    int n_features = m_nfeatures;
    int n_points = m_npoints;
    int n_clusters = m_nclusters;

    float scale_factor = calculate_scale_factor(feature[0],
                                                n_points * n_features);

    scale_and_remap_features(m_scaled_feature, feature, n_points, n_features,
                            scale_factor);
  }
  
```

Código 20: Escalamiento y reorganización de la información de los elementos a evaluar

```

static float calculate_scale_factor(float *mem, int size) {
    float min = mem[0];
    float max = mem[0];
    for (int i = 0; i < size; i++) {
        float value = mem[i];
        if (value < min)
            min = value;
        if (value > max)
            max = value;
    }
    float diff = max - min;
    float scale_factor = diff / 0x00FFFFFF;

    printf("Float to integer scale factor = %f MaxFloat=%f and MinFloat=%f \n",
          scale_factor, max, min);

    return scale_factor;
}
  
```

Código 21: Función para el cálculo del factor de escalamiento

```

void scale_and_remap_features(unsigned int *remapped, float **features,
                             int npoints, int nfeatures, float scale_factor){
    unsigned ptr = 0;
    for (int p = 0; p < npoints; p += 16) {
        for (int f = 0; f < nfeatures; f++) {
            for (int i = 0; i < 16; i++) {
                if ((p + i) < npoints) {
                    remapped[ptr++] = features[p + i][f] / scale_factor;
                } else {
                    remapped[ptr++] = 0;
                }
            }
        }
    }
}

```

Código 22: Función para el escalamiento y reubicación del contenido de “feature”

La ejecución de la función proseguirá de manera distinta dependiendo de si se establece la opción de ejecución mediante recursos de OpenCL o mediante programación con #pragma HLS. En el caso de utilizarse recursos OpenCL, el primer paso es programar las entradas de los *kernels* por medio de la función “setArg” presente para los objetos de la clase “cl::Kernel”. Este paso se realizará en sucesivas iteraciones de un bucle *for*, como se expone en el Código 23.

```

// Set kernel arguments up front - they do need to change during the rest of
// the computation
for (unsigned i = 0; i < m_num_cu_calls; i++) {
    unsigned start_point = i * m_num_points_per_cu;
    unsigned npoints = std::min(m_num_points_per_cu, n_points - start_point);
    unsigned nclusters = n_clusters;
    unsigned nfeatures = n_features;
    unsigned feature_offset = nfeatures * (start_point / 16);
    unsigned members_offset = start_point / 16;

    int nargs = 0;
    OCL_CHECK(err, err = m_kernel_kmeans[i].setArg(nargs++, m_buf_feature));
    OCL_CHECK(err, err = m_kernel_kmeans[i].setArg(nargs++, m_buf_cluster));
    OCL_CHECK(err, err = m_kernel_kmeans[i].setArg(nargs++, m_buf_members));
    OCL_CHECK(err, err = m_kernel_kmeans[i].setArg(nargs++, m_buf_centers[i]));
    OCL_CHECK(err, err = m_kernel_kmeans[i].setArg(nargs++, sizeof(cl_int),
                                                    (void *)&npoints));
    OCL_CHECK(err, err = m_kernel_kmeans[i].setArg(nargs++, sizeof(cl_int),
                                                    (void *)&nclusters));
    OCL_CHECK(err, err = m_kernel_kmeans[i].setArg(nargs++, sizeof(cl_int),
                                                    (void *)&nfeatures));
    OCL_CHECK(err, err = m_kernel_kmeans[i].setArg(nargs++, sizeof(cl_int),
                                                    (void *)&feature_offset));
    OCL_CHECK(err, err = m_kernel_kmeans[i].setArg(nargs++, sizeof(cl_int),
                                                    (void *)&members_offset));
}

```

Código 23: Programación de las señales de entrada a los kernels

Una vez inicializadas las entradas de los *kernels* se realizarán las transferencias entre el *host* y la placa de prototipado, las cuales serán gestionadas por medio de un objeto de la clase “cl::CommandQueue”. Asimismo, la transferencia de datos comenzará con el *host* transmitiendo la información de los elementos contenida en “m_scaled_feature” al buffer “m_buf_feature”

asociado a los *kernels*. Este envío de información se gestionará por medio de la función “enqueueWriteBuffer” y será bloqueante debido al valor “CL_TRUE” de su segundo parámetro. Este aspecto implica que la ejecución del *host* se paralizará hasta que la transferencia se complete. Por otra parte, la lectura por parte de los *kernels* del contenido descargado en “m_buf_feature” será efectuada según lo especificado en la función “enqueueMigrateMemObjects”. En el Código 24 se observa la utilización de las 2 funciones comentadas.

```
// Write features to the device
OCL_CHECK(err, err = m_q.enqueueWriteBuffer(m_buf_feature, CL_TRUE, 0,
                                             m_buf_feature_sz,
                                             m_scaled_feature, NULL, NULL));

OCL_CHECK(err, err = m_q.enqueueMigrateMemObjects({m_buf_feature}, 0, NULL,
                                                  NULL));
m_q.enqueueBarrier();
```

Código 24: Transmisión host-kernel de la información de los elementos

Tras el envío de la información relativa a los elementos, se comienza un proceso que es reiterado en sucesivas etapas hasta que se deje de producir una cantidad de cambios considerada relevante en los resultados obtenidos. Si esta situación no llega a ocurrir, la ejecución de estas etapas finalizará cuando se ejecuten 1000. Estas etapas comienzan con el escalamiento de la información acerca de los *clusters*, utilizando para ello la función “scale_clusters” detallada en el Código 25.

```
void scale_clusters(unsigned int *scaled, float **clusters, int n,
                  float scale_factor) {
    for (int i = 0; i < n; i++) {
        scaled[i] = (unsigned int)((float)clusters[0][i] / (float)scale_factor);
    }
}
```

Código 25: Función “scale_cluster”

Posteriormente, se transfiere la información escalada de los *clusters* hacia los *kernels*, iniciando un proceso de intercambio de información entre el *host* y los *kernels*, así como de coordinación de la funcionalidad, entre ambas. En este aspecto, la aplicación podrá utilizar las 2 alternativas de gestión del *hardware* programable comentadas con anterioridad. Asimismo, si escoge la opción de gestión de los *kernels* mediante recursos de OpenCL, se ejecutará el código descrito en el Código 26.

```

// Schedule the writing of updated clusters values to the device
OCL_CHECK(err, err = m_q.enqueueWriteBuffer(m_buf_cluster, CL_TRUE, 0,
                                             m_buf_cluster_sz,
                                             m_scaled_clusters,
                                             NULL, NULL));

// Schedule kernel execution
for (unsigned i = 0; i < m_num_cu_calls; i++) {
    OCL_CHECK(err, err = m_q.enqueueTask(m_kernel_kmeans[i], NULL, NULL));
}

//Ensure enqueueReadBuffer happens after all the enqueueTask have
//completed
m_q.enqueueBarrier();

// Schedule the reading of new memberships values back to the host
OCL_CHECK(err, err = m_q.enqueueReadBuffer(m_buf_members, CL_TRUE,
                                           0, m_buf_members_sz,
                                           m_new_memberships,
                                           NULL, NULL));

for (unsigned i = 0; i < m_num_cu_calls; i++) {
    OCL_CHECK(err, err = m_q.enqueueReadBuffer(m_buf_centers[i], CL_TRUE, 0,
                                             m_buf_centers_sz,
                                             m_new_centers[i],
                                             NULL, NULL));
}

```

Código 26: Intercambio de información entre host y kernels mediante OpenCL

De acuerdo con lo expuesto en el Código 26, el intercambio de información entre *host* y *kernels* empieza con la transferencia de la información relativa a los *clusters*, y que es contenida por el *host* en el registro “m_scaled_clusters”. Esta información se almacenará en el registro “m_buf_clusters” perteneciente al subsistema *hardware* que componen los *kernels*. Una vez termine esta transferencia, se utilizará la función “enqueueTask” para ordenar la ejecución de los *kernels* implementados. La ejecución de la aplicación en el *host* se parará hasta que los *kernels* finalicen su operación, estableciéndose este hecho mediante la llamada a la función “enqueueBarrier”. Por último, se utilizará la función “enqueueReadBuffer” para descargar en el *host* los resultados obtenidos por los *kernels* acerca de la pertenencia de los elementos a los *clusters* y de los nuevos centroides de estos últimos. Estos resultados se encuentran en el subsistema *hardware* en los registros “m_buf_members” y “m_buf_centers” respectivamente. De esta manera, la información de la membresía de los elementos se transferirá al registro “m_new_memberships” presente en el *host*, mientras que la información de los centroides se guardará en el registro “m_new_centers” [29].

En el caso de no emplearse recursos OpenCL en la implementación y utilización de los *kernels*, la aplicación deberá usar programación en C++ acompañada de instrucciones #pragma HLS para abordarlo. Para ello, ejecutará la función “kmeans_kernel_wrapper” descrita en el archivo “krnl_kmeans.cpp” y cuyo uso se evidencia en el Código 27.

```

#else
  for (unsigned i = 0; i < m_num_cu_calls; i++) {
    unsigned start_point = i * m_num_points_per_cu;
    unsigned npoints = std::min(m_num_points_per_cu, n_points - start_point);
    unsigned nclusters = n_clusters;
    unsigned nfeatures = n_features;
    unsigned feature_offset = nfeatures * (start_point / 16);
    unsigned members_offset = start_point / 16;

    kmeans_kernel_wrapper(m_scaled_feature, m_scaled_clusters,
                          m_new_memberships, m_new_centers[i], npoints,
                          nclusters, nfeatures, feature_offset,
                          members_offset);
  }
#endif

```

Código 27: Ejecución de la funcionalidad del hardware programable mediante “kmeans_kernel_wrapper”

Tras extraer los resultados determinados por los *kernels* se comprobará cuántos elementos han cambiado de *cluster*, y se guardará la cantidad de estos cambios en una variable denominada “delta”. Este hecho se expone en el Código 28.

```

delta = 0;
for (int p = 0; p < n_points; p++) {
  if (m_new_memberships[p] != membership[p]) {
    delta++;
    membership[p] = m_new_memberships[p];
  }
}

```

Código 28: Evaluación de los cambios en la membresía de los elementos

Luego, se actualizará los contenidos de “new_centers” y “new_centers_len”, los cuales se utilizarán con posterioridad en el cálculo de los centroides de los *clusters*. Dichas actualizaciones serán las detalladas en el Código 29.

```

for (int p = 0; p < n_points; p++) {
  int index = m_new_memberships[p];
  new_centers_len[index]++;
}
// Sum the partial centers computed by each CU to form the actual new
// centers
for (int c = 0, i = 0; c < n_clusters; c++) {
  for (int f = 0; f < n_features; f++, i++) {
    for (unsigned cu = 0; cu < m_num_cu_calls; cu++) {
      new_centers[c][f] += m_new_centers[cu][i];
    }
  }
}

```

Código 29: Actualización de “new_centers_len” y “new_centers”

Finalmente, se actualizará el resultado de los centroides de los *clusters*, el cual será almacenado en el registro “clusters” del *host*. Luego se borrará el contenido de “new_centers” y “new_centers_len” con el objetivo de evitar resultados erróneos en la ejecución de las próximas

etapas. Si se incumple la condición de inicio de una nueva etapa, la función “fpga_kmeans_compute” terminará su ejecución notificando la cantidad de etapas que fueron requeridas. Todas estas acciones finales de la función se muestran en el Código 30.

```

for (int c = 0; c < n_clusters; c++) {
    for (int f = 0; f < n_features; f++) {
        if (new_centers_len[c] > 0)
            clusters[c][f] = (new_centers[c][f] * scale_factor) /
                new_centers_len[c]; /* take average i.e. sum/n */
        new_centers[c][f] = 0.0; /* set back to 0 */
    }
    new_centers_len[c] = 0; /* set back to 0 */
}
TIMER_STOP;

} while ((delta > threshold) && (loop < 1000)); /* makes sure loop
                                                terminates */

printf("\nNumber of iterations : %d \n", loop);
}

```

Código 30: Actualización de los centroides de los clusters y terminación de “fpga_kmeans_compute”

5.4.5. fpga_kmeans_deallocateMemory

Se encarga de liberar por medio de la función “free” los recursos de memoria asignados a los registros “m_scaled_feature”, “m_scaled_clusters”, “m_new_memberships” y “m_new_centers”, como se especifica en el Código 31. El Grafo de llamadas de la función “fpga_kmeans_deallocateMemory” se muestra Figura 27.



Figura 27: Grafo de llamadas de la función fpga_kmeans_deallocateMemory

```

int FPGA_KMEANS::fpga_kmeans_deallocateMemory() {
    free(m_scaled_feature);
    free(m_scaled_clusters);
    free(m_new_memberships);

    for (int i = 0; i < NUM_CU; i++) {
        free(m_new_centers[i]);
    }

    return 0;
}

```

Código 31: Función “fpga_kmeans_deallocateMemory”

5.4.6. fpga_kmeans_print_report

El objetivo de esta función es notificar al usuario el tiempo incurrido en la ejecución los distintos aspectos del algoritmo *k-means* desarrollado, así como de la totalidad del mismo, suponiendo que se hayan utilizado recursos OpenCL para ello. Entre dichos aspectos se encuentra la inicialización de los kernels (Device Initialization) la asignación de recursos de memoria para los buffers (Buffer Allocation) y la actualización de del contenido de distintos registros a lo largo de la ejecución del proceso de *k-means*. Las acciones definidas por esta función se especifican en el Código 32. El grafo de llamadas de la función “fpga_kmeans_print_report” se muestra en la Figura 28.

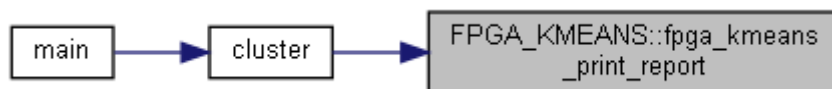


Figura 28: Grafo de llamadas de la función *fpga_kmeans_print_report*

```

/*****
                                     FPGA KMEANS PRINT REPORT()
*****/
int FPGA_KMEANS::fpga_kmeans_print_report() {
#ifdef __USE_OPENCL__
printf("-----\n");
printf(" Performance Summary                \n");
printf("-----\n");
printf(" Device Initialization      : %12.4f ms\n", TIMER_REPORT_MS(5));
printf(" Buffer Allocation          : %12.4f ms\n", TIMER_REPORT_MS(6));
printf("-----\n");
printf(" Compute Memberships       : %12.4f ms\n", TIMER_REPORT_MS(1));
printf(" Update Delta              : %12.4f ms\n", TIMER_REPORT_MS(2));
printf(" Update Centers            : %12.4f ms\n", TIMER_REPORT_MS(3));
printf(" Update Clusters           : %12.4f ms\n", TIMER_REPORT_MS(4));
printf(" Total K-Means Compute Time : %12.4f ms\n", TIMER_REPORT_MS(0));
printf("-----\n");
#endif
return 0;
}
  
```

Código 32: Función “fpga_kmeans_print_report”

5.5. kmeans_clustering_cmodel.c

Este fichero contiene el código en C que implementa la versión de la aplicación *k-means* desarrollada para operar utilizando únicamente recursos *software*. Para ello dispone, como función principal, una que posee el mismo nombre que posee el fichero. La función comienza con la generación de una serie de variables, así como la reserva dinámica de espacio de memoria de una

serie de registros. Además, también se inicializa cada *cluster* con un elemento cualquiera que no se incluirá inicialmente en los demás, y el registro de la pertenencia de cada elemento a un *cluster* tendrá valor -1 al inicio. Este aspecto del programa de *k-means* se muestra en el Código 33.

```

/* randomly pick cluster centers */
for (i = 0; i < nclusters; i++) {
  // n = (int)rand() % npoints;
  for (j = 0; j < nfeatures; j++)
    clusters[i][j] = feature[n][j];
  n++;
}

for (i = 0; i < npoints; i++)
  membership[i] = -1;

```

Código 33: Inicialización de “clusters” y “membership”

Después de la inicialización de las variables y los registros requeridos, se procede a la ejecución del algoritmo de *k-means* propiamente dicho. Para ello, la función prosigue con el código descrito en Código 34 y en Código 35. El grafo de llamadas de la función “kmeans_clustering_cmodel” se muestra en la Figura 29.

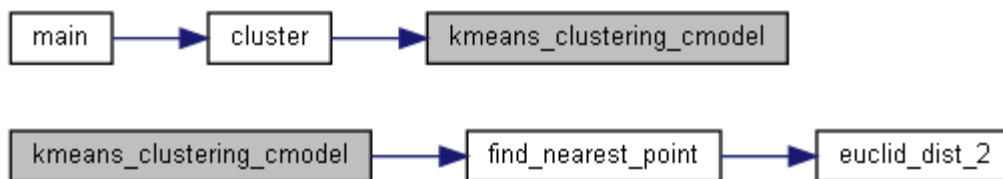


Figura 29: Grafo de llamadas de la función *kmeans_clustering_cmodel*

```

do {
  delta = 0.0;

  for (i = 0; i < npoints; i++) {
    /* find the index of nearest cluster centers */
    index = find_nearest_point(feature[i], nfeatures, clusters, nclusters);
    /* if membership changes, increase delta by 1 */
    if (membership[i] != index)
      delta += 1.0;

    /* assign the membership to object i */
    membership[i] = index;

    /* update new cluster centers : sum of objects located within */
    new_centers_len[index]++;
    for (j = 0; j < nfeatures; j++)
      new_centers[index][j] += feature[i][j];
  }
}

```

Código 34: Proceso de clustering en la función *kmeans_clustering_cmodel (I)*

```

/* replace old cluster centers with new_centers */
for (i = 0; i < nclusters; i++) {
  for (j = 0; j < nfeatures; j++) {
    if (new_centers_len[i] > 0)
      clusters[i][j] = new_centers[i][j] / new_centers_len[i];
    new_centers[i][j] = 0.0; /* set back to 0 */
  }
  new_centers_len[i] = 0; /* set back to 0 */
}

(*iteration)++;
} while (delta > threshold);

```

Código 35: Proceso de clustering en la función `kmeans_clustering_cmodel` (II)

Como puede observarse, cada etapa de algoritmo (iteración del bucle *do-while*) comenzará evaluando, para cada elemento, cuál es el *cluster* que se encuentra más cerca. Para ello utilizará la función “`find_nearest_point`”, cuyo código se especifica en el Código 36. El grafo de llamadas de la función “`find_nearest_point`” se muestra en la Figura 30.

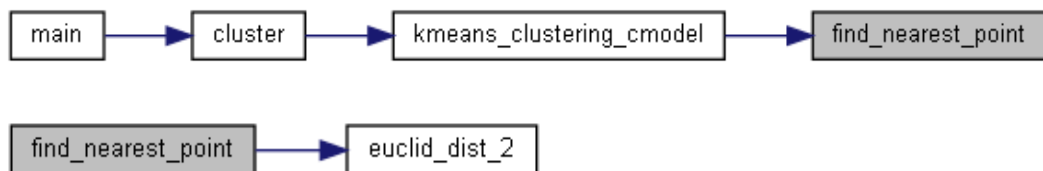


Figura 30: Grafo de llamadas de la función `find_nearest_point`.

```

int find_nearest_point(float *pt, /* [nfeatures] */
                      int nfeatures, float **pts, /* [npts][nfeatures] */
                      int npts) {
  int index, i;
  float max_dist = FLT_MAX;

  /* find the cluster center id with min distance to pt */
  for (i = 0; i < npts; i++) {
    float dist;
    dist = euclid_dist_2(pt, pts[i], nfeatures); /* no need square root */
    if (dist < max_dist) {
      max_dist = dist;
      index = i;
    }
  }
  return (index);
}

```

Código 36: Función “`find_nearest_point`”

Dicha función determinará la distancia que separa al elemento estudiado “`pt`” de cada punto “`pts[i]`” por medio de la suma cuadrática de las diferencias entre sus valores de atributos. Para este cálculo se empleará la función “`euclid_dist_2`”, cuya programación se presenta en el Código 37. El grafo de llamadas de la función “`euclid_dist_2`” se muestra en la Figura 31.

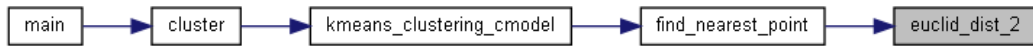


Figura 31: Grafo de llamadas de la función euclid_dist_2

```

/*----< euclid_dist_2() >-----*/
/* multi-dimensional spatial Euclid distance square */
__inline float euclid_dist_2(float *pt1, float *pt2, int numdims) {
    int i;
    float ans = 0.0;

    for (i = 0; i < numdims; i++)
        ans += (pt1[i] - pt2[i]) * (pt1[i] - pt2[i]);

    return (ans);
}
  
```

Código 37: Función euclid_dist_2

Una vez que se ha evaluado cuál es el *cluster* más cercano al elemento, se procederá a ver si dicho *cluster* ha cambiado de una etapa del algoritmo a otra. En caso afirmativo, se registrará este suceso mediante el incremento en una unidad de la variable denominada “delta”, cuyo valor indicará la cantidad de elementos que han cambiado de *cluster* durante el transcurso de la etapa. Luego, se actualizará la pertenencia del elemento estudiado, incrementándose en 1 la cantidad de elementos del *cluster* escogido. Esta última variable se encuentra identificada como “new_centers_len[index]” en el Código 35. Tras finalizar el primer bucle *for*, se utilizarán los registros “new_centers_len” y “new_centers” para establecer el valor del centroide de cada *cluster*. El código medirá también el número de etapas transcurridas en la ejecución del algoritmo, cuyas reiteraciones finalizarán cuando se cumpla la condición de salida, que será que la cantidad de elementos que han cambiado de agrupación sea inferior al valor umbral “threshold”.

Al finalizar la ejecución de la versión *software* en C del algoritmo de *k-means*, la función liberará los recursos de memoria ocupados por los registros “new_centers” y “new_centers_len” y devolverá los *clusters* evaluados, como se detalla en el Código 38.

```

free(new_centers[0]);
free(new_centers);
free(new_centers_len);

return clusters;
}
  
```

Código 38: Finalización de la función kmeans_clustering_cmodel

5.6. krnl kmeans

Se encarga de abordar la implementación y el uso de los *kernels* dentro del conjunto del sistema, cuando no se encuentra habilitada la opción de utilizar recursos OpenCL. Se subdivide en los ficheros “krnl_kmeans.cpp” y “krnl_kmeans.h”. Asimismo, la utilización de las funciones que describen se efectúa únicamente por medio de la función “kmeans_kernel_wrapper”. El grafo de llamadas de dicha función se muestra en la Figura 32.

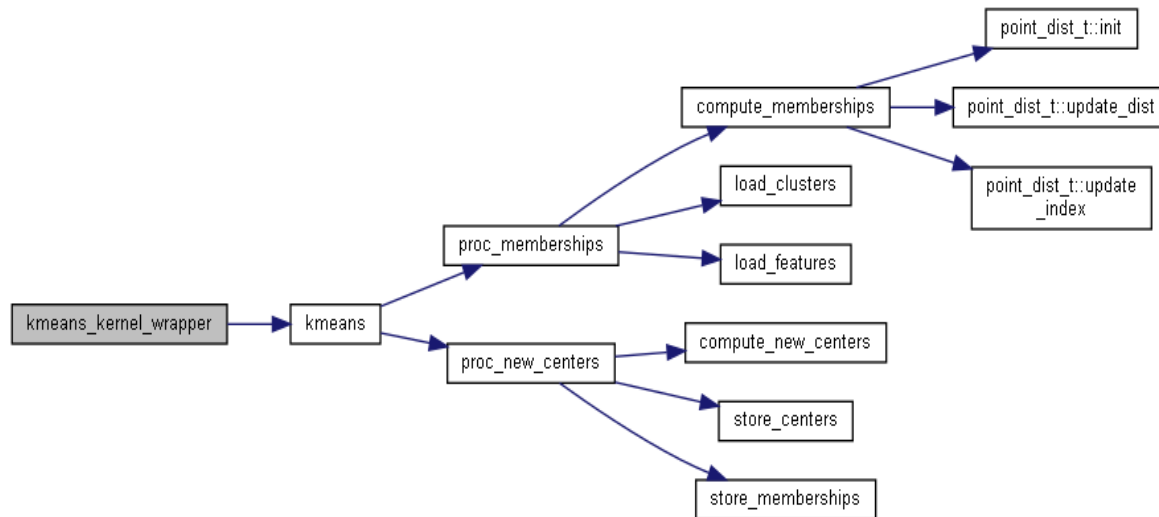


Figura 32: Grafo de llamadas de la función kmeans_kernel_wrapper

Dicha función consiste solamente en la invocación con adaptación de algunos tipos de variables de la función “kmeans”, como se muestra en el Código 39.

```

void kmeans_kernel_wrapper(unsigned int *features, unsigned int *clusters,
                          int *membership, unsigned int *new_centers,
                          int npoints, int nclusters, int nfeatures,
                          unsigned feature_offset, unsigned members_offset) {
    kmeans(reinterpret_cast<ap_int<512> *>(features),
           reinterpret_cast<ap_int<512> *>(clusters),
           reinterpret_cast<ap_int<512> *>(membership),
           reinterpret_cast<ap_int<512> *>(new_centers), npoints, nclusters,
           nfeatures, feature_offset, members_offset);
}
  
```

Código 39: Función kmeans_kernel_wrapper

Dentro de la descripción de la función “kmeans”, el primer paso consiste en establecer el conjunto de puertos que conformarán la interfaz de los *kernels*. Para ello, se utilizarán las instrucciones #pragma HLS INTERFACE, como se indica en el Código 40. Además, también se utilizarán instrucciones #pragma HLS stream para definir interfaces de comunicación entre los distintos submódulos que componen un *kernel*.

```

void kmeans(ap_int<512> *features, ap_int<512> *clusters,
            ap_int<512> *membership, ap_int<512> *new_centers, int npoints,
            int nclusters, int nfeatures, unsigned feature_offset,
            unsigned members_offset) {
#pragma HLS INTERFACE m_axi port = features bundle = gmem0 offset = slave
#pragma HLS INTERFACE m_axi port = clusters bundle = gmem0 offset = slave
#pragma HLS INTERFACE m_axi port = membership bundle = gmem0 offset = slave
#pragma HLS INTERFACE m_axi port = new_centers bundle = gmem0 offset = slave
#pragma HLS INTERFACE s_axilite port = features bundle = control
#pragma HLS INTERFACE s_axilite port = clusters bundle = control
#pragma HLS INTERFACE s_axilite port = membership bundle = control
#pragma HLS INTERFACE s_axilite port = new_centers bundle = control
#pragma HLS INTERFACE s_axilite port = npoints bundle = control
#pragma HLS INTERFACE s_axilite port = nclusters bundle = control
#pragma HLS INTERFACE s_axilite port = nfeatures bundle = control
#pragma HLS INTERFACE s_axilite port = feature_offset bundle = control
#pragma HLS INTERFACE s_axilite port = members_offset bundle = control
#pragma HLS INTERFACE s_axilite port = return bundle = control

    static hls::stream<unsigned int> index_str[PARALLEL_POINTS];
    static hls::stream<unsigned int> feature_str[PARALLEL_POINTS];

#pragma HLS stream variable = index_str depth = 1
#pragma HLS stream variable = feature_str depth = c nfeatures

```

Código 40: Definición de la interfaz de los kernels

Cada una de las directivas INTERFACE asocia un puerto físico presente en el acelerador *hardware* a una de las variables descritas en el código *software*. Esta asociación se establece especificando el nombre de la variable como valor del parámetro “port”. Asimismo, todos los puertos definidos mediante #pragma HLS INTERFACE que tengan un mismo valor de “bundle”, compartirán el mismo puerto de transmisión físico. De esta manera, la implementación *hardware* de los *kernels* deberá disponer de 2 puertos de conexión de estos con el *host*. Uno de estos puertos operará de acuerdo al protocolo AXI4-Lite, debido a que todas las variables asociadas a él mediante #pragma HLS especifican el modo “s_axilite”. Por otra parte, el otro puerto seguirá el protocolo AXI4 con un puerto de direccionamiento de 32 *bits*, al establecerse para sus variables asociadas el modo “m_axi”. Cabe destacar que la interfaz que opere según el protocolo AXI4 será una interfaz esclava para esta aplicación, como queda establecido mediante la especificación “offset=slave” [30].

Finalmente, la función “kernel” establecerá la subdivisión de la estructura del *kernel* en 2 módulos, cuyas respectivas descripciones en alto nivel se corresponde a las especificadas en las funciones “proc_memberships” y “proc_new_centers”. Dichos módulos se ejecutarán de manera concurrente, como establece la instrucción #pragma HLS DATAFLOW, aunque manteniendo cierto grado de *pipelining* debido a la dependencia existente entre las funcionalidades de ambos módulos. En el Código 41 se evidencia la parte final de la función “kernel” [30].

```
#pragma HLS DATAFLOW

proc_memberships(index_str, feature_str, features, clusters, npoints,
                 nclusters, nfeatures, feature_offset);

proc_new_centers(membership, new_centers, index_str, feature_str, npoints,
                 nclusters, nfeatures, members_offset);
}

} // end extern "C"
```

Código 41: División de la estructura del kernel en los submódulos “proc_memberships” y “proc_new_centers”

5.6.1. proc_memberships

El objetivo del módulo “proc_memberships” consistirá en evaluar la distancia de cada elemento al *cluster* más cercano, con el objetivo de realizar los cambios oportunos en las agrupaciones previamente establecidas. Para ello, el primer paso, después de generarse las variables locales que correspondan, será efectuar la descarga por parte del submódulo “proc_memberships” de la información relativa a los *clusters*. Con este propósito, se utilizará la función “load_clusters”, la cual no se considerará desde la perspectiva del diseño *hardware* como un submódulo dentro de “proc_memberships”, al igual que ocurre con el resto de funciones cuya invocación es subyacente a la de “proc_memberships”. Esto se debe a la inclusión de la especificación #pragma HLS INLINE RECURSIVE dentro de la descripción de “proc_membership”. En el Código 42 se muestra la utilización de la función “load_clusters”. Asimismo, la descripción de la función “load_clusters” se especifica en el Código 43 [30].

```
void proc_memberships(hls::stream<unsigned int> index_str[PARALLEL_POINTS],
                    hls::stream<unsigned int> feature_str[PARALLEL_POINTS],
                    ap_int<512> *features, ap_int<512> *clusters,
                    int npoints, int nclusters, int nfeatures,
                    unsigned feature_offset) {
#pragma HLS INLINE RECURSIVE

    unsigned max_nvalid_points = PARALLEL_POINTS;
    unsigned min_nvalid_points =
        npoints - (npoints / PARALLEL_POINTS) * PARALLEL_POINTS;
    unsigned max_rd_feature_count =
        nfeatures * ((max_nvalid_points - 1) / 16 + 1);
    unsigned min_rd_feature_count =
        nfeatures * ((min_nvalid_points - 1) / 16 + 1);
    unsigned rd_feature_offset = feature_offset;

    unsigned num_iterations = (npoints + PARALLEL_POINTS - 1) / PARALLEL_POINTS;

    unsigned int l_clusters[L_CLUSTERS_SZ];
#pragma HLS ARRAY_PARTITION variable = l_clusters cyclic factor = 16

    load_clusters(l_clusters, clusters, nclusters, nfeatures);
```

Código 42: Ejecución de “load_cluster” en “proc_memberships”



Figura 33: Grafo de llamadas de la función `load_clusters`

```

void load_clusters(unsigned int l_clusters[L_CLUSTERS_SZ],
                  ap_int<512> *clusters, int nclusters, int nfeatures) {
  unsigned nreads = (nclusters * nfeatures + 15) / 16;

  ld_clusters:
  for (int i = 0; i < nreads; i++) {
    #pragma HLS LOOP_TRIPCOUNT min = c_ld_clusters max = c_ld_clusters
    #pragma HLS PIPELINE

    ap_int<512> tmp = clusters[i];
    for (int j = 0; j < 16; j++) {
      l_clusters[i * 16 + j] = tmp.range(j * 32 + 31, j * 32);
    }
  }
}
  
```

Código 43: Función “`load_clusters`”

Como puede observarse en el Código 42, la invocación de la función “`load_clusters`” viene acompañada de la instrucción `#pragma HLS ARRAY_PARTITION`. La utilidad de la misma es especificar la fragmentación del *array* indicado como variable en una cantidad máxima de *subarrays* indicada mediante el parámetro “factor”. Cada uno de estos *subarrays* se guardarán en elementos de memoria con acceso independiente, lo que permitirá paralelizar la ejecución de operaciones sobre el *array* al completo. El grafo de llamadas de dicha función se muestra en la Figura 33.

Posteriormente, el módulo “`proc_memberships`” ejecutará un proceso reiterativo en el cual descargará, por medio de la función “`load_features`”, la información acerca de los elementos a clasificar y determinará la clasificación correcta de estos utilizando la función “`compute_memberships`”. Cabe destacar que todos los elementos almacenados en el registro “`l_features`” podrán ser accedidos de manera simultánea, especificándose este detalle mediante una instrucción `#pragma HLS ARRAY_PARTITION`. En el Código 44 se muestra esta parte final de la ejecución del módulo “`proc_memberships`”. Por otra parte, en el Código 45 se observa la descripción de la función “`load_cluster`”, y en la Figura 34 se muestra la localización de esta función dentro de la jerarquía de llamadas con respecto a la función principal del acelerador *hardware* (`kmeans_kernel_wrapper`) [30].

```

for (unsigned int i = 0; i < num_iterations; i++) {
#pragma HLS LOOP_TRIPCOUNT min = c_num_iter max = c_num_iter

    unsigned int l_features[PARALLEL_POINTS][CONST_NFEATURES];
#pragma HLS ARRAY_PARTITION variable = l_features complete dim = 1

    unsigned rd_feature_count = (i == (num_iterations - 1))
                                ? min_rd_feature_count
                                : max_rd_feature_count;

    load_features(l_features, features, nfeatures, rd_feature_count,
                 rd_feature_offset);

    compute_memberships(index_str, feature_str, l_features, l_clusters,
                       nclusters, nfeatures);
}
}

```

Código 44: Carga y determinación de las agrupaciones de los elementos en el módulo “proc_memberships”



Figura 34: Grafo de llamadas de la función load_features.

```

void load_features(unsigned int l_features[PARALLEL_POINTS][CONST_NFEATURES],
                  ap_int<512> *feature, int nfeatures, unsigned nmem_reads,
                  unsigned &offset) {
    unsigned f = 0;
    unsigned p = 0;

ld_features:
    for (int i = 0; i < nmem_reads; i++, f++) {
#pragma HLS LOOP_TRIPCOUNT min = c_ld_features max = c_ld_features
#pragma HLS PIPELINE

        if (f == nfeatures) {
            f = 0;
            p++;
        }

        ap_int<512> tmp = feature[offset + i];
        for (int j = 0; j < 16; j++) {
            l_features[p * 16 + j][f] = tmp.range(j * 32 + 31, j * 32);
        }
    }

    offset += PARALLEL_POINTS * nfeatures / 16;
}

```

Código 45: Función “load_feature”

De esta manera, la carga computacional principal del módulo “proc_memberships” queda descrita en la función “compute_memberships”. Dado que durante este proceso se requiere ir evaluando la distancia que separa al elemento de cada *cluster* e ir almacenando la distancia más cercana obtenida, se define un tipo de estructura de datos denominada “point_dist_t”. Asimismo,

se dispondrá de un objeto “point_dist_t” por cada elemento evaluado, el cual dispondrá de 3 atributos. El primero, llamado “index”, se utilizará para especificar el índice del *cluster* más próximo al elemento, siendo los otros atributos “min_dist”, para indicar la mínima distancia de separación encontrada entre el elemento considerado y un *cluster*, y “dist”, para guardar los resultados obtenidos durante la evaluación del último *cluster* con respecto a su distanciamiento del elemento. Además, los objetos de esta estructura poseerán una serie de funciones que operarán con estos atributos. En el Código 46 se especifican los atributos y funciones del tipo de variable “point_dist_t”.

```
typedef ap_uint<7> index_t;

struct point_dist_t {
    index_t index;
    unsigned long min_dist;
    unsigned long dist;

    void init() {
        index = 0;
        min_dist = MAX_VALUE;
        dist = 0;
    }

    void update_dist(unsigned long point_value, unsigned int cluster_value) {
        long diff = (point_value - cluster_value);
        dist += diff * diff;
    }

    void update_index(index_t cluster_id) {
        if (dist < min_dist) {
            min_dist = dist;
            index = cluster_id;
        }
        dist = 0;
    }
};
```

Código 46: Descripción del tipo de variable “point_dist_t”

La ejecución del proceso “compute_memberships” comienza con la creación e inicialización de un *array* de elementos “point_dist_t”, existiendo uno por cada elemento a agrupar en un *cluster*. Esta parte inicial de la función se muestra en el Código 47. Desde el punto de vista del diseño *hardware*, este se orientará a maximizar la paralelización en esta etapa inicial, como establecen las instrucciones #pragma HLS ARRAY_PARTITION y #pragma HLS UNROLL. Esta última instrucción se utiliza dentro de un bucle para especificar que se deben implementar varias copias del diseño *hardware* requerido para ejecutar una de las iteraciones del bucle. En este caso, serán tantas copias como iteraciones del bucle existan [30].

```

void compute_memberships(
    hls::stream<unsigned int> index_str[PARALLEL_POINTS],
    hls::stream<unsigned int> feature_str[PARALLEL_POINTS],
    unsigned int features[PARALLEL_POINTS][CONST_NFEATURES],
    unsigned int clusters[L_CLUSTERS_SZ], int nclusters, int nfeatures) {
    point_dist_t pt[PARALLEL_POINTS];
    #pragma HLS ARRAY_PARTITION variable = pt complete

    for (int p = 0; p < PARALLEL_POINTS; p++) {
        #pragma HLS UNROLL
        pt[p].init();
    }
}

```

Código 47: Inicialización de los elementos del array “pt”

Posteriormente, se determinará, mediante la función “update_dist”, la distancia que separa a cada elemento de cada *cluster*, estableciéndose con “update_index” el índice del *cluster* más cercano contemplado hasta el momento. Cabe destacar que la información relativa a los elementos será transferida al *stream* “feature_str” para su envío al módulo “proc_new_centers”. En el Código 48 se expone el código de este proceso de clasificación de los elementos.

```

calc_indexes:
    for (int c = 0, offset = 0; c < nclusters; c++, offset += nfeatures) {
        #pragma HLS LOOP_TRIPCOUNT min = c_nclusters max = c_nclusters

        for (int f = 0; f < nfeatures; f++) {
            #pragma HLS LOOP_TRIPCOUNT min = c_nfeatures max = c_nfeatures
            #pragma HLS PIPELINE

            for (int p = 0; p < PARALLEL_POINTS; p++) {
                #pragma HLS UNROLL
                pt[p].update_dist(features[p][f], clusters[offset + f]);
                // Stream the features used in this iteration to the next process to
                // compute the new centers
                if (c == (nclusters - 1))
                    feature_str[p].write(features[p][f]);
            }
        }

        for (int p = 0; p < PARALLEL_POINTS; p++) {
            #pragma HLS UNROLL
            pt[p].update_index(c);
        }
    }
}

```

Código 48: Etapa de clasificación de los elementos

Por último, se transmiten al módulo “proc_new_centers” por medio del *stream* “index_str” la clasificación de los elementos evaluados, como queda especificado en el Código 49.

```

write_index_str:
  for (int p = 0; p < PARALLEL_POINTS; p++) {
#pragma HLS UNROLL
    // Instead of writing the indexes to the membership buffer in global
    // memory,
    // stream them to the next process to compute the new centers
    // The next process will take care of writing the index to global memory
    // instead.
    index_str[p].write(pt[p].index);
  }
}

```

Código 49: Transmisión de la clasificación de los elementos a “proc_new_centers”

5.6.2. proc_new_centers

El objetivo de “proc_new_centers” es recalculer los centroides de los *clusters*, de manera que permanezcan acordes a los cambios efectuados en las pertenencias de los elementos. Como es natural en una función cualquiera, esta comienza con la generación e inicialización de las variables locales. En este caso, puede destacarse el registro “l_new_centers”, el cual se utilizará durante la ejecución de la funcionalidad de este módulo para guardar los resultados que se vayan obteniendo acerca de los nuevos centroides. Esta parte inicial del proceso descrito por la función “proc_new_centers” se expone en el Código 50. El grafo de llamadas de la función “proc_new_centers” se muestra en la Figura 35.

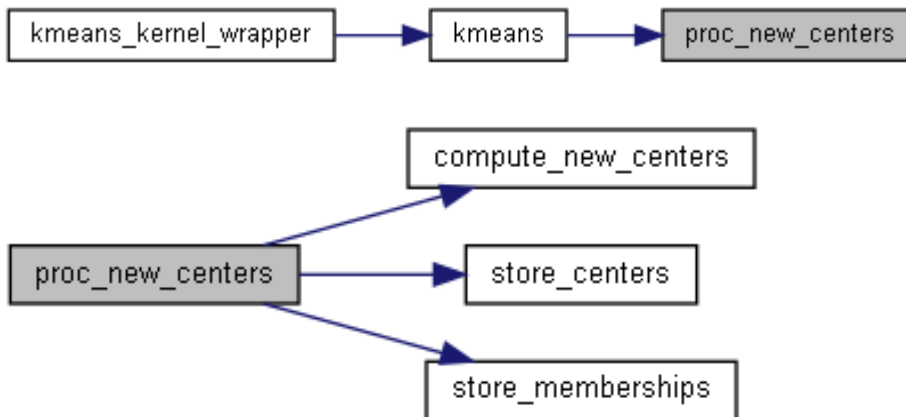


Figura 35: Grafo de llamadas de la función proc_new_centers


```

void proc_new_centers(ap_int<512> *membership, ap_int<512> *new_centers,
                    hls::stream<unsigned int> index_str[PARALLEL_POINTS],
                    hls::stream<unsigned int> feature_str[PARALLEL_POINTS],
                    int npoints, int nclusters, int nfeatures,
                    unsigned members_offset) {
#pragma HLS INLINE RECURSIVE

    unsigned max_nvalid_points = PARALLEL_POINTS;
    unsigned min_nvalid_points = npoints ( npoints / PARALLEL_POINTS ) *
        PARALLEL_POINTS;
    unsigned max_wr_members_count = (max_nvalid_points - 1) / 16 + 1;
    unsigned min_wr_members_count = (min_nvalid_points - 1) / 16 + 1;
    unsigned wr_members_offset = members_offset;
    unsigned wr_centers_count = 0;

    unsigned num_iterations = (npoints + PARALLEL_POINTS - 1) / PARALLEL_POINTS;

    unsigned int l_new_centers[L_CLUSTERS_SZ];
#pragma HLS ARRAY_PARTITION variable = l_new_centers cyclic factor = 16

    // Zero the new centers
    init_centers:
    for (int i = 0; i < L_CLUSTERS_SZ; i++) {
#pragma HLS PIPELINE      l_new_centers[i] = 0;
    }
}

```

Código 50: Creación e inicialización de las variables locales de “proc_new_centers”

Posteriormente, se subdivide la carga computacional de “proc_new_centers” en 3 funciones denominadas “compute_new_centers”, “store_memberships” y “store_centers” como se muestra en el Código 51. Dicha partición solo tendrá efecto en términos de la descripción funcional del módulo *hardware*, pero no en la organización jerárquica de su arquitectura, como se indica en el Código 50.

```

for (unsigned int i = 0; i < num_iterations; i++) {
#pragma HLS LOOP_TRIPCOUNT min = c_num_iter max = c_num_iter

    index_t l_index[PARALLEL_POINTS];
#pragma HLS ARRAY_PARTITION variable = l_index complete

    unsigned wr_members_count = (i == (num_iterations - 1))
        ? min_wr_members_count
        : max_wr_members_count;

    compute_new_centers(index_str, feature_str, l_new_centers, l_index,
                      npoints, nclusters, nfeatures, wr_centers_count);

    store_memberships(membership, l_index, wr_members_count,
                    wr_members_offset);
}

store_centers(new_centers, l_new_centers, nclusters, nfeatures);
}

```

Código 51: Distribución de la carga computacional del módulo “proc_new_centers”

El objetivo de la función “compute_new_centers” consistirá en introducir en el registro “l_new_centers” la información necesaria para calcular los nuevos centroides. Para ello, se ejecutará el código descrito en el Código 52. Cabe destacar que la instrucción #pragma HLS PIPELINE establecerá la concurrencia de la ejecución de las sentencias especificadas en el Código 52 dentro del bucle interno [30]. El grafo de llamadas de la función “compute_new_centers” se muestra en la Figura 36



Figura 36: Grafo de llamadas de la función compute_new_centers

```

index_t index;

calc_centers:
  for (int p = 0; p < PARALLEL_POINTS; p++, npoints_cnt++) {
    for (int f = 0; f < nfeatures; f++) {
      #pragma HLS LOOP_TRIPCOUNT min = c_nfeatures max = c_nfeatures
      #pragma HLS PIPELINE II = 1
      #pragma HLS dependence variable = l_new_centers inter false

      if (f == 0) {
        index = index_str[p].read();
        l_index[p] = index;
      }

      unsigned int feature = feature_str[p].read();

      // Since we've increased the buffer size to be multiple of 16
      // Make sure we don't include unwanted values in the calculation
      if (npoints_cnt < npoints)
        l_new_centers[index * nfeatures + f] += feature;
    }
  }
}

```

Código 52: Programación de la función “compute_new_centers”

Por otra parte, en el Código 53 se muestra la descripción de la función “store_memberships”. El propósito de la misma consiste en almacenar en el registro “membership” de diseño *hardware* especificado para el *hardware* programable, los resultados obtenidos relativos a la evaluación de la pertenencia de los elementos a los *clusters*. De manera similar, la función “store_center” expuesta en el Código 54 se utilizará para guardar en el registro “new_centers” los resultados de los centroides de los *clusters*. El grafo de llamadas de la función “store_memberships” se muestra en la Figura 37. Para la función “store_center” se representa el grafo de llamadas en la Figura 38.



Figura 37: Grafo de llamadas de la función store_memberships

```

void store_memberships(ap_int<512> *membership, index_t ndex[PARALLEL_POINTS],
                     unsigned nmem_writes, unsigned &offset) {
  st_members:
  for (int i = 0; i < nmem_writes; i++) {
    #pragma HLS LOOP_TRIPCOUNT min = c_st_members max = c_st_members
    #pragma HLS PIPELINE

    ap_int<512> tmp = 0;
    for (int j = 0; j < 16; j++) {
      tmp.range(j * 32 + 31, j * 32) = index[i * 16 + j];
    }

    membership[offset + i] = tmp;
  }

  offset += PARALLEL_POINTS / 16;
}

```

Código 53: Función “store_membership”



Figura 38: Grafo de llamadas de la función store_centers

```

void store_centers(ap_int<512> *new_centers,
                 unsigned int l_new_centers[L_CLUSTERS_SZ], int nclusters,
                 int nfeatures) {
  unsigned nwrites = (nclusters * nfeatures + 15) / 16;

  st_centers:
  for (int i = 0; i < nwrites; i++) {
    #pragma HLS LOOP_TRIPCOUNT min = c_st_centers max = c_st_centers
    #pragma HLS PIPELINE

    ap_int<512> tmp;
    for (int j = 0; j < 16; j++) {
      tmp.range(j * 32 + 31, j * 32) = l_new_centers[i * 16 + j];
    }
    new_centers[i] = tmp;
  }
}

```

Código 54: Función “store_centers”

5.7. timer.h

Especifica las instrucciones utilizadas para la medida de los tiempos de ejecución que son notificados al usuario por medio de la función “fpga_kmeans_print_report” explicada en la página 87 de esta memoria. Dichas funciones son “TIMER_INIT”, “TIMER_START”, “TIMER_STOP”, “TIMER_STOP_ID” y “TIMER_REPORT”, cuyas descripciones son las que se adjuntan en el Código 55. Cabe destacar que dichas funciones ejecutarán alguna funcionalidad siempre que no esté

definido el elemento “_DISABLE_TIMERS_”. En caso contrario, la invocación de estas funciones no tendrá efecto práctico. En la Figura 39 se indican las dependencias que requiere el fichero timer.h, mientras que en la Figura 40 se representa las dependencias de otros archivos de timer.h.

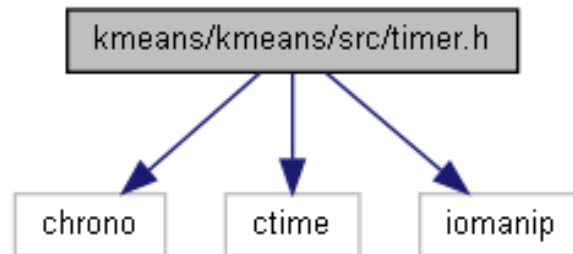


Figura 39: Dependencias para time.h

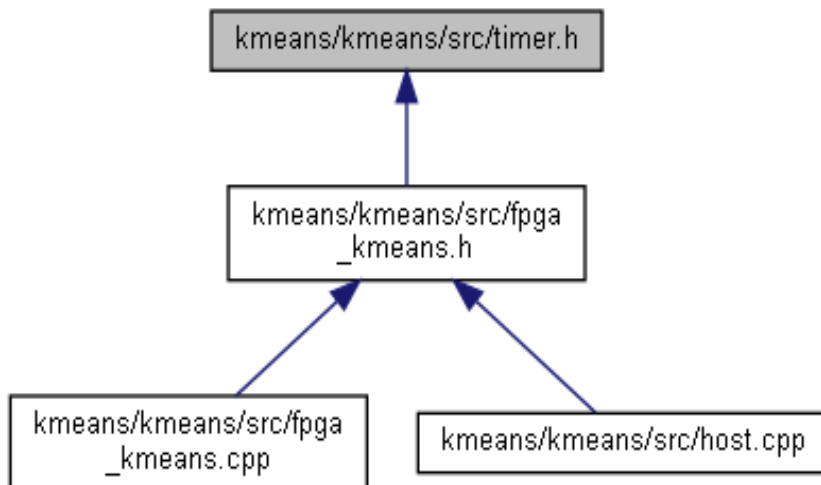


Figura 40: Archivos que directa o indirectamente incluyen time.h

```

extern cPerfTimer *_g_timer;
extern int _g_timer_last_id;

#ifndef __DISABLE_TIMERS__
#define TIMER_INIT(a)
\
  cPerfTimer *_g_timer = new cPerfTimer[a];
\
  int _g_timer_last_id = 0;
#define TIMER_START(a)
\
  _g_timer[a].start();
\
  _g_timer_last_id = (a);
#define TIMER_STOP _g_timer[_g_timer_last_id].stop();
#define TIMER_STOP_ID(a) _g_timer[a].stop();
#define TIMER_REPORT_MS(a) _g_timer[a].get_ms()
#else
#define TIMER_INIT(a)
#define TIMER_START(a)
#define TIMER_STOP
#define TIMER_STOP_ID(a)
#define TIMER_REPORT(a)
#endif

```

Código 55: Funciones para medir tiempos de ejecución

Para el desarrollo de estas funciones se utilizará un tipo de estructura denominada “cPerfTimer”, cuya descripción se evidencia en el Código 56 presente en el archivo “timer.h”.

```

struct cPerfTimer {
  std::chrono::high_resolution_clock::time_point m_start;
  std::chrono::high_resolution_clock::time_point m_end;
  std::chrono::duration<double> m_total;

  std::string m_name;

  cPerfTimer() { initialize(); }
  void initialize() { m_total = std::chrono::duration<double>(0.0); }
  void start() { m_start = std::chrono::high_resolution_clock::now(); }
  void stop() {
    m_end = std::chrono::high_resolution_clock::now();
    m_total += (m_end - m_start);
  }
  double get_ms() { return 1000 * m_total.count(); }
};

```

Código 56: Definición del tipo de variable “cPerfTimer”

De esta manera, se obtiene que la utilidad funcional de las funciones de “timer.h” que se utilizarán fuera de este fichero serán las siguientes.

- **TIMER_INIT(a)**: Creará una cantidad “a” de objetos “cPerfTimer”. Estos objetos serán referenciados mediante un puntero llamado “_g_timer”.
- **TIMER_START(a)**: Extraerá el tiempo de reloj marcado por el reloj del sistema. Además, indicará mediante la variable global “_g_timer_last_id” del fichero “timer.h” que el objeto

indicado por el índice numérico “a” ha sido el último para el que se ha invocado esta función.

- `TIMER_STOP`: Extraerá el tiempo de reloj marcado por el reloj del sistema y calculará la diferencia entre este resultado y el obtenido en la última invocación de “`TIMER_START(a)`”, obteniéndose así el tiempo de ejecución requerido en las instrucciones intermedias. El resultado de esta operación será guardado en una variable del tipo `std::chrono::duration<double>` denominada “`m_total`” [31].
- `TIMER_STOP_ID(a)`: Su ejecución es similar a la de la función “`TIMER_STOP`”. No obstante, su diferencia radica en que se especifica cuál es el objeto “`cPerfTimer`” del que se extraerá la referencia temporal adecuada.
- `TIMER_REPORT_MS(a)`: Devolverá el tiempo medido por el objeto identificado con el índice “a”. Dicho tiempo se expresará en milisegundos.

5.8. `krnl_kmeans.ini`

Aparte de la información extraída de la descripción funcional de los *kernels* desarrollada mediante archivos `.cpp`, existen datos relativos al diseño de acelerador *hardware* que no son especificados por estos archivos, pero que resultan de gran importancia. Dichos datos son el número de *kernels* que conformarán el acelerador *hardware* y la frecuencia de funcionamiento de los mismos, los cuales se especifican en el archivo “`krnl_kmeans.ini`”. Asimismo, la frecuencia de funcionamiento especificada para los *kernels* es de 190 MHz, mientras que el número de *kernels* implicados en la arquitectura del acelerador *hardware* son 2. Estos detalles se observan en la Código 57, donde se expone el contenido del archivo “`krnl_kmeans.ini`”.

```
kernel_frequency=190
[connectivity]
nk=kmeans:2
```

Código 57: Contenido completo del archivo `krnl_kmeans.ini`

5.9. Conclusiones

En este capítulo se analizó el código fuente de la aplicación, el cual constituye una descripción HLS del mismo desarrollada en C++. Asimismo, se detalla la integración de cada archivo dentro de la funcionalidad de la arquitectura, así como las dependencias que presenta con respecto al código descrito en otros archivos. De esta manera, se obtiene una visión en alto nivel del funcionamiento de la aplicación.

Capítulo 6. Implementación del diseño de la plataforma en Vitis

6.1. Introducción

En este capítulo se describe la implementación realizada para el diseño de referencia, así como las distintas transformaciones efectuadas que permiten su aplicación al *dataset* definido de imágenes hiperespectrales. Para ello, se parte de un conjunto de funciones básicas presentes en el conjunto de reglas definido en el fichero Makefile para crear las aplicaciones correspondientes a tres tipos de implementaciones:

- Emulación *software* (sw_emu).
- Emulación *hardware* (hw_emu).
- Implementación *hardware/software* (hw).

Cada una de estas implementaciones está orientada a ser utilizada durante distintas etapas del flujo de diseño, ya sea el depurado del *software*, el depurado del *hardware* o la integración final del acelerador *hardware* en el dispositivo, respectivamente.

A lo largo de este capítulo se mostrarán distintos detalles de la metodología de diseño utilizada, tal como se comentó anteriormente, de las transformaciones realizadas y de los resultados obtenidos en términos de recursos utilizados, latencias y potencia consumida.

Como se especificó con anterioridad, para la implementación de la aplicación se emplea un flujo de diseño basado en la utilización de Makefiles. Por este motivo, se utilizará la opción de línea de comandos de las distintas herramientas y *scripts*, en especial del compilador Vitis, de las herramientas de síntesis de alto nivel (Vivado HLS) y del entorno de implementación de Vivado. En los casos de interés, se han mostrado los resultados en el correspondiente entorno gráfico, como por ejemplo en el analizador de Vitis.

6.2. Compilación del proyecto

Para la compilación del proyecto se utilizan un conjunto de reglas definidas en el Makefile principal. Cabe destacar que el sistema dispone de un menú de ayuda, accesible mediante la orden “make help”, donde se detalla un conjunto de tareas principales, el cual se expone en el Código 58.

```

make all TARGET=<sw_emu|hw_emu|hw>
    DEVICE=<FPGA platform>
    HOST_ARCH=<aarch32/aarch64/x86>
    EDGE_COMMON_SW=<rootfs and kernel image path>
    Command to generate the design for specified Target and Shell.
    By default, HOST_ARCH=x86. HOST_ARCH and EDGE_COMMON_SW is required for SoC shells

make clean
    Command to remove the generated non-hardware files.

make cleanall
    Command to remove all the generated files.

make test DEVICE=<FPGA platform>
    Command to run the application.
    This is same as 'check' target but does not have any makefile dependency.

make sd_card TARGET=<sw_emu|hw_emu|hw>
    DEVICE=<FPGA platform>
    HOST_ARCH=<aarch32/aarch64/x86>
    EDGE_COMMON_SW=<rootfs and kernel image path>
    Command to prepare sd card files.
    By default, HOST_ARCH=x86. HOST_ARCH and EDGE_COMMON_SW is required for SoC shells

make check TARGET=<sw_emu|hw_emu|hw>
    DEVICE=<FPGA platform>
    HOST_ARCH=<aarch32/aarch64/x86>
    EDGE_COMMON_SW=<rootfs and kernel image path>
    Command to run application in emulation.
    By default, HOST_ARCH=x86. HOST_ARCH and EDGE_COMMON_SW is required for SoC shells

make build TARGET=<sw_emu|hw_emu|hw>
    DEVICE=<FPGA platform>
    HOST_ARCH=<aarch32/aarch64/x86>
    EDGE_COMMON_SW=<rootfs and kernel image path>
    Command to build xclbin application.
    By default, HOST_ARCH=x86. HOST_ARCH and EDGE_COMMON_SW is required for SoC shells

```

Código 58: Tareas principales del Makefile

Como puede apreciarse, las tareas del Makefile que constituyen el conjunto especificado en el Código 58, son las siguientes:

- **all:** genera el diseño completo para un determinado dispositivo indicado por DEVICE y para un determinado tipo de implementación, ya sea empotrada o de tipo PCIe. En este proyecto se utiliza la placa ZCU102, por lo que se considera un sistema empotrado. Para esta tarea es preciso elegir el tipo de implementación, ya sea en emulación *software* o en emulación *hardware* usando la plataforma virtual basada en QEMU para el PS, o la implementación *hardware* final para el dispositivo seleccionado.
- **check, test:** Estas tareas permiten ejecutar la aplicación en modo emulación. Con ello es posible chequear la funcionalidad de la implementación sin disponer del *hardware* real.

La utilidad de “check” y “test” resultan equivalentes, salvo en el chequeo de dependencias que se realiza en el makefile.

- **build:** crea la aplicación .xclbin para poder ser ejecutada sobre las capas proporcionadas por Xilinx Run Time (XRT). Es preciso ejecutar esta tarea para poder generar la tarjeta SD necesaria para arrancar el sistema.
- **sd_card:** permite crear en la tarjeta SD los ficheros necesarios de la aplicación, del sistema operativo y otros ficheros de datos. Como consecuencia de ello vuelca a las particiones de la tarjeta los ficheros de arranque (boot) del sistema operativo Linux, así como una instalación completa del *root file system*. El diseñador puede agregar los ficheros de datos necesarios para su aplicación.
- **clean, cleanall:** se trata de tareas auxiliares que permiten limpiar el proyecto, ficheros objeto y ejecutables. En el caso de cleanall se borran las implementaciones hardware realizadas.

La utilización de estas funciones en los procesos citados se explica en detalle en los siguientes apartados.

6.3. Adaptación a la placa ZCU102

En la versión original del proyecto, la aplicación escogida estaba desarrollada para una serie de placas de prototipado entre las que se encuentran varios modelos de Xilinx, no estando disponible para la placa ZCU102 empleada para este proyecto. Por este motivo, fueron necesarias dos modificaciones en el Makefile, consistiendo la primera en eliminar la restricción que inhabilita cualquier opción especificada en DEVICE que presente en su denominación la inscripción “ZC”. Para evidenciar este cambio, en el Código 59 se muestra la sección del Makefile original donde se introduce la modificación, y en el Código 60 se expone el cambio efectuado. Los cambios introducidos permiten ejecutar el proyecto utilizando como *target* la placa ZCU102, con dispositivos MPSoC de la familia UltraScale, que inicialmente no estaba disponible.

```
ifeq ($(findstring zc, $(DEVICE)), zc)
$(error This example is not supported for $(DEVICE))
endif
ifeq ($(findstring vck, $(DEVICE)), vck)
$(error This example is not supported for $(DEVICE))
endif
ifeq ($(findstring samsung, $(DEVICE)), samsung)
$(error This example is not supported for $(DEVICE))
endif
```

Código 59: Sección del Makefile que restringe cualquier placa cuyo nombre contenga "ZC"

```
#ifeq ($(findstring zc, $(DEVICE)), zc)
#$(error This example is not supported for $(DEVICE))
#endif
ifeq ($(findstring vck, $(DEVICE)), vck)
$(error This example is not supported for $(DEVICE))
endif
ifeq ($(findstring samsung, $(DEVICE)), samsung)
$(error This example is not supported for $(DEVICE))
endif
```

Código 60: Anulación de la restricción del Makefile sobre las placas cuyo nombre contenga "ZC"

Otro ajuste necesario ha sido posibilitar la emulación para dispositivos de la familia UltraScale. Este aspecto se logra mediante el cambio en la parametrización de la invocación del archivo escrito en Perl denominado `run_emulation.pl`. La función de este archivo consiste en gestionar las acciones del emulador QEMU durante las emulaciones *hardware* y *software*. En el Código 61 puede observarse la sección del Makefile donde debe realizarse el cambio, y en el Código 62 se muestra el cambio efectuado. El cambio consiste en sustituir el valor "7" del último parámetro de la invocación del archivo por "U". De esta manera, se señala que la placa de prototipado emulada no se incluye en la familia de productos 7 Series de Xilinx, sino en la familia UltraScale. Además, se han resuelto un conjunto de dependencias de Perl presentes en `run_emulation.pl`.

```
ifeq ($(HOST_ARCH), x86)
$(CP) $(EMCONFIG_DIR)/emconfig.json .
XCL_EMULATION_MODE=$(TARGET) ./$(EXECUTABLE) -x
$(BUILD_DIR)/krnl_kmeans.xclbin -i ./data/100 -c ./data/100.gold_c10 -n 10
else
$(ABS_COMMON_REPO)/common/utility/run_emulation.pl "$LAUNCH_EMULATOR"
| tee run_app.log" "$RUN_APP_SCRIPT" $(TARGET) " "${RESULT_STRING}" "7"
endif
else
ifeq ($(HOST_ARCH), x86)
./$(EXECUTABLE) -x $(BUILD_DIR)/krnl_kmeans.xclbin -i ./data/100 -c
./data/100.gold_c10 -n 10
endif
endif
```

Código 61: Sección del Makefile con parametrización del `run_emulation.pl`

```

ifeq ($(HOST_ARCH), x86)
    $(CP) $(EMCONFIG_DIR)/emconfig.json .
    XCL_EMULATION_MODE=$(TARGET) ./$(EXECUTABLE) -x
$(BUILD_DIR)/krnl_kmeans.xclbin -i ./data/100 -c ./data/100.gold_c10 -n 10
else
    #$(ABS_COMMON_REPO)/common/utility/run_emulation.pl
    "./${LAUNCH_EMULATOR} | tee run_app.log" "./${RUN_APP_SCRIPT} ${TARGET}"
    "${RESULT_STRING}" "7"
    #$(ABS_COMMON_REPO)/common/utility/run_emulation.pl "./${LAUNCH_EMULATOR}
    | tee run_app.log" "./${RUN_APP_SCRIPT} ${TARGET}" "${RESULT_STRING}" "U"
endif
else
ifeq ($(HOST_ARCH), x86)
    ./$(EXECUTABLE) -x $(BUILD_DIR)/krnl_kmeans.xclbin -i ./data/100 -c
./data/100.gold_c10 -n 10
endif
endif
endif

```

Código 62: Modificación de la parametrización de *run_emulation.pl*

6.4. Construcción del diseño

Una vez realizado el análisis de la estructura del proyecto, se procede a la creación del diseño. Esto se logra invocando la función “build” descrita en el archivo Makefile asociado al proyecto. Dicha invocación se efectuará en una ventana de terminal, tal como se muestra en el Código 63.

```

host% make build TARGET=hw HOST_ARCH=aarch64
DEVICE=xilinx_zcu102_base_202010_1 EDGE_COMMON_SW=/opt/xilinx/xilinx-zynqmp-
common-v2020.1

```

Código 63: Comando para la invocación de la compilación del proyecto

Como puede observarse, el primer parámetro especificado se denomina “TARGET”, el cual puede adquirir los valores “hw”, “hw_emu” y “sw_emu”. En el Código 63 la opción especificada es la “hw”, no obstante, se utilizan las opciones disponibles en la verificación del proyecto. Respecto a las opciones “hw_emu” y “sw_emu”, estas se utilizan para la emulación del sistema descrito por el proyecto, con el propósito de verificar la funcionalidad de la aplicación sin necesidad de implementar los *kernels* en la placa de prototipado.

La diferencia entre la emulación “hw_emu” y la “sw_emu” se encuentra en el nivel de abstracción que implican. La opción “sw_emu” permite comprobar únicamente la funcionalidad algorítmica descrita por la aplicación, sin considerar los aspectos temporales del sistema propios de su implementación real. Por otra parte, la emulación “hw_emu” tiene en consideración la sincronización requerida entre la ejecución de la aplicación sobre la plataforma virtual. El *software* se ejecuta sobre el emulador QEMU y el *hardware* sobre el simulador digital. En consecuencia, la

verificación resulta más realista, pero precisa de un mayor tiempo para su realización: opción “hw_emu”=> 8:51 minutos y “sw_emu” => 54 segundos.

Como última opción de compilación explorada se encuentra la “hw”, mediante la cual se logra implementar el sistema diseñado y simultáneamente desarrollar el ejecutable de la aplicación. Cabe destacar que esta última opción de compilación es la que requiere de un mayor tiempo para su consecución: opción “hw” => 33:37 minutos.

El siguiente parámetro requerido por el comando “make build” se denomina “HOST_ARCH”. Su propósito es indicar el tipo de microprocesador establecido como *host* del sistema. Para ello, las opciones posibles son un microprocesador de la familia x86 de Intel, un ARM de 32 bits o un ARM de 64 bits, especificándose como “x86”, “aarch32” o “aarch64” respectivamente. En esta aplicación, se escoge la opción “aarch64”, ya que las APUs disponibles en la placa de prototipado pertenecen a la familia ARM de 64 bits. A continuación, se encuentra el parámetro “DEVICE” mediante el cual se especifica un archivo de la extensión .xpfm (Xilinx® platform definition) que contiene una descripción de alto nivel de la plataforma utilizada como acelerador *hardware*. Asimismo, el archivo indicado para esta opción es el correspondiente a la placa de prototipado ZCU102 [32]. Por último, se tiene el parámetro “EDGE_COMMON_SW”, en el cual se define el directorio de referencia utilizado para el acceso a los recursos “ZynqMP common image” en los que se contiene un *kernel* de Linux y un sistema de archivos raíz.

Al finalizar la ejecución de los comandos de compilación, se obtienen una serie de informes de análisis de la solución, incluyendo la arquitectura y las prestaciones del acelerador *hardware*. La organización de todos los informes producidos tras una invocación del comando “make build” se establecerá por medio de un archivo de extensión “.xclbin.link_summary”, el cual se generará automáticamente junto al resto de informes. Además, este último archivo permite acceder a los informes utilizando la herramienta Vitis Analyzer. Para ello, se deberá introducir en la ventana terminal el comando “vitis_analyzer <archivo .xclbin.link_summary>”, como se detalla en el Código 64.

```
host% vitis_analyzer $HOME/VAEm2/ demo/kmeans/build_dir.hw.xilinx_zcu102_base_202010_1/krn1_kmeans.xclbin.link_summary
```

Código 64: Invocación de la aplicación “Vitis Analyzer”

Otro aspecto común a todas las opciones de ejecución de “make build” se encuentra en la organización de los informes presentados por la aplicación Vitis Analyzer, ya que esta siempre constará de un apartado “krnl_kmeans” que los engloba. Este apartado incluye información de

“kmeans”, en el cual se incluye información de prestaciones del *kernel* “kmeans” como unidad de cómputo replicada en la arquitectura del acelerador *hardware*.

6.4.1. Análisis de la emulación software “sw_emu”

La opción “sw_emu” del comando “make build” solamente verifica la algorítmica del proyecto, sin considerar la coordinación temporal entre los elementos del sistema ni el consumo de recursos PL por parte del acelerador *hardware*. Debido a ello, la opción “sw_emu” no elaborará los informes generados por las opciones “hw” o “hw_emu” que conciernen a estos 2 últimos aspectos.

Los únicos informes producidos en esta opción de emulación *software* y que poseen contenido son informes del tipo “Summary” y “Logs”. En la Figura 41 y la Figura 42 se muestra el contenido del archivo “Summary” del apartado principal “krnl_kmeans” generado en la emulación “sw_emu”. Asimismo, el contenido de este informe señala que la verificación realizada en la compilación de la emulación *software* no ha detectado ningún error, ya que el campo “GUIDANCE” especifica el mensaje “No violations”.


Summary - [/home/users/PFC/divsicad/mgvanche/VAEm/demo/kmeans/build_dir.sw_emu.xilinx_zcu102_base_202010_1/krnl_kmeans.			
 krnl_kmeans	/home/users/PFC/divsicad/mgvanche/VAEm/demo/kmeans/build_dir.sw_emu.xilinx_zcu102_base_202010_1/krnl_kmeans....		Softw
STATUS		Completed	
		Logs	
GUIDANCE	No violations	System Guidance	
STARTED	September 23, 2020 13:17	COMPLETED	September 23, 2020 13:17
		ELAPSED	23s
PLATFORM	xilinx_zcu102_base_202010_1		

Figura 41: Informe “Summary” del apartado “krnl_kmeans (Software Emulation)”

```
KERNELS
kmeans 2 compute unit Compile Summary

COMMAND LINE
-DPARALLEL_POINTS=96
-t sw_emu
--platform xilinx_zcu102_base_202010_1
--save-temps
-g
--temp_dir ./build_dir.sw_emu.xilinx_zcu102_base_202010_1
-l
-O3
--config krnl_kmeans.ini
-obuild_dir.sw_emu.xilinx_zcu102_base_202010_1/krnl_kmeans.xclbin
x.sw_emu.xilinx_zcu102_base_202010_1/kmeans.xo
```

Figura 42: Informe “Summary” del apartado “krnl_kmeans (Software Emulation)” (II)

Por otra parte, en la Figura 43 y la Figura 44 puede observarse el contenido del “Summary” presente en el subapartado “kmeans”.

STATUS	
Completed	
Logs	
STARTED	September 23, 2020 13:16
COMPLETED	September 23, 2020 13:17
ELAPSED	46s
PLATFORM	xilinx_zcu102_base_202010_1

Figura 43: Informe “Summary” del apartado “kmeans (Software Emulation)”

```
COMMAND LINE
-DPARALLEL_POINTS=96
-t sw_emu
--platform xilinx_zcu102_base_202010_1
--save-temps
-g
--temp_dir ./_x.sw_emu.xilinx_zcu102_base_202010_1
-c
-k kmeans
-lsrc
-o_x.sw_emu.xilinx_zcu102_base_202010_1/kmeans.xo src/krnl_kmeans.cpp
```

Figura 44: Informe Summary del apartado kmeans (Software Emulation) (II)

En el informe “Logs” del apartado principal se detallan los resultados obtenidos en cada una de las sucesivas acciones efectuadas por el compilador. Asimismo, la compilación se ha desarrollado en cada etapa sin errores, lo cual se señala con el mensaje “V++ internal step status:

success” indicado en cada una de las acciones descritas. La correcta ejecución de la compilación de la emulación *software* se confirma también en la última etapa de la misma consistente en la generación de los archivos de estimación del sistema, como se evidencia en la Figura 45.

```
V++ internal step: generating system estimate report file

timestamp: 26 Oct 2020 09:24:07

output:
/home/users/PFC/divsicad/mgvanche/VAEm2/demo/kmeans/build_dir.sw_emu.xilinx_zc
u102_base_202010_1/reports/link/system_estimate_krnl_kmeans.txt

V++ internal step status: success

-----

system estimate report task completed
```

Figura 45: Confirmación de emulación *software* realizada sin incidencias.

Con respecto a los informes “*Logs*”, cabe destacar que el archivo “*Logs*” contenido en el subapartado “*kmeans*” es más escueto al obtenerse ejecutándose la opción “*sw_emu*”, que en el caso de las opciones “*hw*” y “*hw_emu*”. Este hecho se debe a que, como se detalla en la Figura 46 acerca de la parte final del “*Log*” generado en la emulación *software*, los procesos posteriores a la comprobación de la síntesis del código fuente del proyecto no son efectuados. Entre estos procesos se incluyen los relativos a la programación y coordinación de las acciones del acelerador *hardware* (*scheduling*), así como el desarrollo de los modelos RTL no son efectuados en la opción “*sw_emu*”, pero sí en las opciones “*hw*” y “*hw_emu*”. En la Figura 46, se indica que la verificación del código de los *kernels* ha concluido exitosamente, no habiendo ninguna incidencia reseñable.

```
INFO: [HLS 200-1493] Running only source code synthesis checks, skipping
scheduling and RTL generation.

HLS completed successfully

INFO: [Common 17-206] Exiting vitis_hls at Wed Sep 23 13:17:20 2020...
```

Figura 46: Finalización de archivo *Logs* del subapartado *kmeans* (*Software Emulation*)

6.4.2. Análisis de la emulación *hardware* “*hw_emu*”

A diferencia de lo ocurrido con la opción “*sw_emu*”, la emulación *hardware* genera una variedad de informes más semejante a la producida por la compilación mediante “*make build*”. Asimismo, en la Figura 47 y la Figura 48 se ilustra el contenido del archivo “*Summary*” del apartado “*krnl_kmeans*”, cuya información relativa a la aplicación evaluada coincide con la expuesta en otros informes “*Summary*” generados al ejecutar las opciones “*sw_emu*” o “*hw*”.

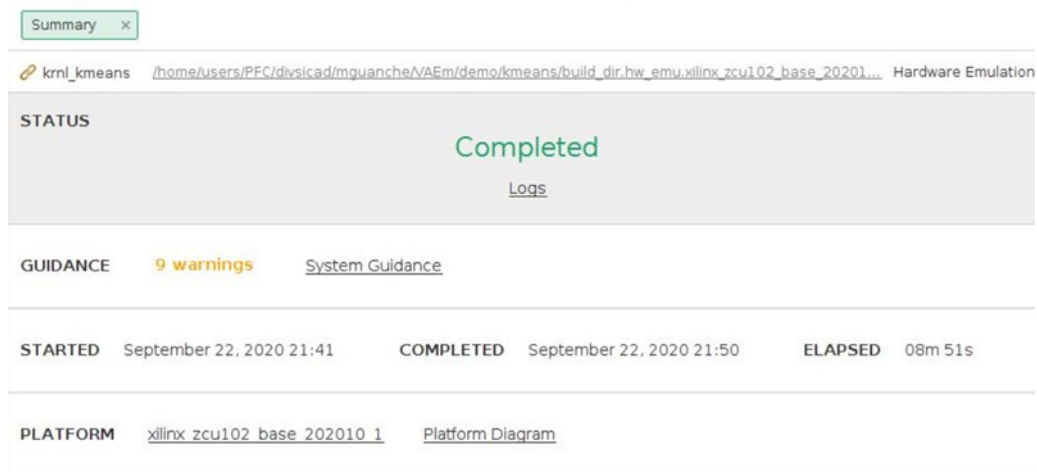


Figura 47: Informe “Summary” del apartado “krnl_kmeans (Hardware Emulation)” (I)

```
-DPARALLEL_POINTS=96
-t hw_emu
--platform xilinx_zcu102_base_202010_1
--save-temps
-g
--temp_dir ./build_dir.hw_emu.xilinx_zcu102_base_202010_1
-l
-O3
--config krnl_kmeans.ini
-obuild_dir.hw_emu.xilinx_zcu102_base_202010_1/krnl_kmeans.xclbin
_x.hw_emu.xilinx_zcu102_base_202010_1/kmeans.xo
```

Figura 48: Informe “Summary” del apartado “krnl_kmeans (Hardware Emulation)” (II)

La ejecución de la opción “hw_emu” produce un archivo denominado “System Diagram”, cuyo contenido se muestra en la Figura 49. En dicha imagen se observa que no existe estimación con respecto a la cantidad de módulos DSP por cada *kernel*. Además, la estimación de la cantidad de LUTs y BRAMs es más elevada en el caso de la ejecución de la opción “hw_emu” que en el caso de la compilación, ya que se trata de estimaciones en alto nivel sin realizar optimizaciones propias de la implementación física.

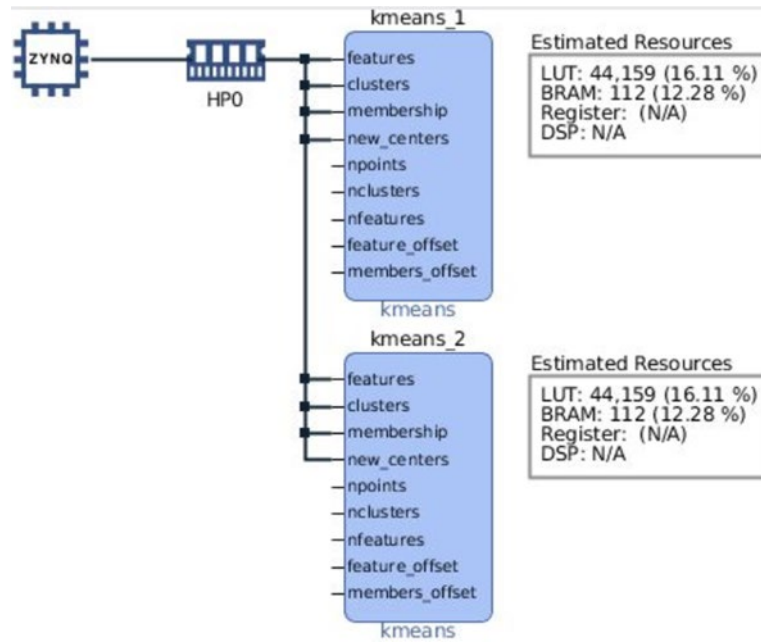


Figura 49: Diagrama del acelerador hardware elaborado por la opción "hw_emu"

Todo ello se recoge en el informe "System Estimate". Los primeros aspectos tratados serán la frecuencia de reloj y la latencia de los módulos que componen el acelerador hardware a cualquier nivel de jerarquía (Figura 50).

Timing Information (MHz)				
Compute Unit	Kernel Name	Module Name	Target Frequency	Estimated Frequency
kmeans_1	kmeans	proc_memberships5	149.925034	205.465378
kmeans_1	kmeans	proc_new_centers	149.925034	205.465378
kmeans_1	kmeans	kmeans	149.925034	205.465378
kmeans_2	kmeans	proc_memberships5	149.925034	205.465378
kmeans_2	kmeans	proc_new_centers	149.925034	205.465378
kmeans_2	kmeans	kmeans	149.925034	205.465378

Latency Information				
Compute Unit	Kernel Name	Module Name	Start Interval	Best (cycles)
kmeans_1	kmeans	proc_memberships5	8766248 ~ 9476679	8766248
kmeans_1	kmeans	proc_new_centers	8617852 ~ 8618134	8617852
kmeans_1	kmeans	kmeans	8766249 ~ 9476680	9479617
kmeans_2	kmeans	proc_memberships5	8766248 ~ 9476679	8766248
kmeans_2	kmeans	proc_new_centers	8617852 ~ 8618134	8617852
kmeans_2	kmeans	kmeans	8766249 ~ 9476680	9479617

Latency Information					
Compute Unit	Avg (cycles)	Worst (cycles)	Best (absolute)	Avg (absolute)	Worst (absolute)
kmeans_1	9121464	9476679	58.445 ms	60.813 ms	63.181 ms
kmeans_1	8617993	8618134	57.455 ms	57.456 ms	57.457 ms
kmeans_1	9479758	9479899	63.201 ms	63.202 ms	63.202 ms
kmeans_2	9121464	9476679	58.445 ms	60.813 ms	63.181 ms
kmeans_2	8617993	8618134	57.455 ms	57.456 ms	57.457 ms
kmeans_2	9479758	9479899	63.201 ms	63.202 ms	63.202 ms

Figura 50: Frecuencia de reloj y latencia de los módulos según "hw_emu"

Por otra parte, este informe “*System Estimate*” indicará el consumo de recursos PL que, según el análisis efectuado por la emulación *hardware*, implicará cada módulo. Los resultados de esta estimación se detallan en la Figura 51.

Area Information							
Compute Unit	Kernel Name	Module Name	FF	LUT	DSP	BRAM	URAM
kmeans_1	kmeans	proc_memberships5	19285	44159	0	112	0
kmeans_1	kmeans	proc_new_centers	3291	7013	0	16	0
kmeans_1	kmeans	kmeans	49481	65181	0	254	0
kmeans_2	kmeans	proc_memberships5	19285	44159	0	112	0
kmeans_2	kmeans	proc_new_centers	3291	7013	0	16	0
kmeans_2	kmeans	kmeans	49481	65181	0	254	0

Figura 51: Recursos PL requeridos según “hw_emu”, expuesto en informe “*System Estimate*”

En el informe “*System Guidance*” se detallan las distintas incidencias obtenidas al diseñar la arquitectura *hardware* ajustándose a la descripción HLS de los *kernels*. Cabe destacar que estas incidencias son idénticas a las detectadas en el proceso de compilación y que posteriormente se notifican en los archivos “*System Guidance*” y “*Kernel Guidance*”. Esto se debe a que tanto la opción la opción “hw_emu” como “hw” como llegan a desarrollar un diseño RTL a partir de la descripción en lenguaje de alto nivel del acelerador *hardware*. Debido a ello, ambos procesos requieren analizar la descripción HLS de los *kernels*, evaluando así su factibilidad y detectando las incongruencias del mismo. En la Figura 52 se expone una captura de pantalla de este informe. Al igual que ocurrió con la opción “hw” este informe coincide en contenido con lo descrito en el informe “*Kernel Guidance*”.

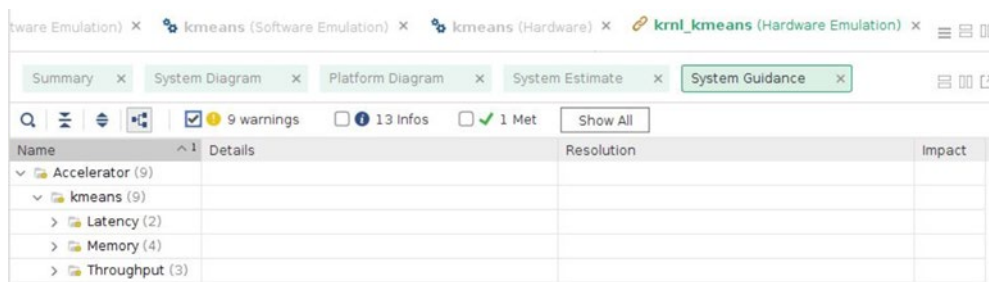


Figura 52: Informe “*System Guidance*” obtenido tras la ejecución “hw_emu”

Además de los informes resultantes de la evaluación del conjunto del acelerador *hardware*, esta emulación ha generado varios informes que evalúan el diseño del *kernel* “kmeans”. Entre estos informes se encuentra un “*Summary*” cuyo contenido se expone en la Figura 53 y la Figura 54.

Summary

STATUS

Completed

[Log](#)

STARTED	September 22, 2020 21:35	COMPLETED	September 22, 2020 21:41	ELAPSED	06m 01s
----------------	--------------------------	------------------	--------------------------	----------------	---------

PLATFORM [xilinx_zcu102_base_202010_1](#)

Figura 53: "Summary" del subpartado "kmeans (Hardware Emulation)"

```

COMMAND LINE
-DPARALLEL_POINTS=96
-t hw_emu
--platform xilinx_zcu102_base_202010_1
--save-temps
-g
--temp_dir ./_x.hw_emu.xilinx_zcu102_base_202010_1
-c
-k kmeans
-lsrc
-o _x.hw_emu.xilinx_zcu102_base_202010_1/kmeans.xo src/krn1_kmeans.cpp

```

Figura 54: "Summary" del subpartado "kmeans (Hardware Emulation)" (II)

Posteriormente, se encuentra el informe "Kernel Estimate". El contenido de este informe será similar al especificado para el informe "System Estimate", con la salvedad de que el diseño *hardware* analizado es de un *kernel* y no el del acelerador *hardware* al completo. En la Figura 55 se indica la frecuencia de funcionamiento y la frecuencia objetivo de los elementos del *kernel*, así como sus respectivas latencias. En la Figura 56 se indica la cantidad y tipo de recursos consumidos por un *kernel*.

Timing Information (MHz)				
Compute Unit	Kernel Name	Module Name	Target Frequency	Estimated Frequency
kmeans_1	kmeans	proc_memberships5	149.925034	205.465378
kmeans_1	kmeans	proc_new_centers	149.925034	205.465378
kmeans_1	kmeans	kmeans	149.925034	205.465378

Latency Information				
Compute Unit	Kernel Name	Module Name	Start Interval	Best (cycles)
kmeans_1	kmeans	proc_memberships5	8766248 ~ 9476679	8766248
kmeans_1	kmeans	proc_new_centers	8617852 ~ 8618134	8617852
kmeans_1	kmeans	kmeans	8766249 ~ 9476680	9479617

Latency Information					
Compute Unit	Avg (cycles)	Worst (cycles)	Best (absolute)	Avg (absolute)	Worst (absolute)
kmeans_1	9121464	9476679	58.445 ms	60.813 ms	63.181 ms
kmeans_1	8617993	8618134	57.455 ms	57.456 ms	57.457 ms
kmeans_1	9479758	9479899	63.201 ms	63.202 ms	63.202 ms

Figura 55: Frecuencia de reloj y latencia del kernel según la emulación hardware

Area Information								
Compute Unit	Kernel Name	Module Name	FF	LUT	DSP	BRAM	URAM	
kmeans_1	kmeans	proc_memberships5	19285	44159	0	112	0	
kmeans_1	kmeans	proc_new_centers	3291	7013	0	16	0	
kmeans_1	kmeans	kmeans	49481	65181	0	254	0	

Figura 56: Recursos PL utilizados por un kernel según la estimación de la emulación hardware

Otro informe característico en la descripción del modelo de *kernel* desarrollado en la emulación *hardware* es el informe “HLS Synthesis”, cuyo contenido se expone en la Tabla 6.

Tabla 6: Informe HLS Synthesis generado al ejecutar la opción *hw_emu*

Name	Latency (cycles)	Latency (ns)	Iteration Latency	Interval	Trip Count	Pipelined
kmeans	9479899	6,32E+10		9476680		dataflow
proc_memberships5	9476679	6,32E+10		9476679		no
ld_clusters	213	1,42E+06	3	1	212	yes
VITIS_LOOP_275_1	9476359	6,32E+10	3683		2573	no
ld_features	205	1,37E+06	3	1	204	yes
calc_indexes_VITIS_LOOP_117_2	3402	2,27E+07	3	1	3400	yes
proc_new_centers	8618134	5,75E+10		8618134		no
init_centers	3408	2,27E+07	1	1	3408	yes
VITIS_LOOP_320_1	8614404	5,74E+10	3348		2573	no
calc_centers_VITIS_LOOP_156_1	3269	2,18E+07	7	1	3264	yes
st_members	6	40.002	2	1	6	yes
st_centers	213	1,42E+06	3	1	212	yes

Name	BRAM	BRAM (%)	DSP	DSP (%)
kmeans	254	13	406	16
proc_memberships5	112	6	398	15
ld_clusters				
VITIS_LOOP_275_1				
ld_features				
calc_indexes_VITIS_LOOP_117_2				
proc_new_centers	16	~0	8	~0
init_centers				
VITIS_LOOP_320_1				
calc_centers_VITIS_LOOP_156_1				
st_members				
st_centers				

Name	FF	FF (%)	LUT	LUT (%)
kmeans	49481	9	65181	23
proc_memberships5	19285	3	44159	16
ld_clusters				
VITIS_LOOP_275_1				
ld_features				
calc_indexes_VITIS_LOOP_117_2				
proc_new_centers	3291	~0	7013	2
init_centers				
VITIS_LOOP_320_1				
calc_centers_VITIS_LOOP_156_1				
st_members				
st_centers				

6.4.3. Arquitectura de la plataforma

Durante el proceso de compilación del proyecto, a excepción del caso de la emulación *software*, se ha realizado un diseño de la arquitectura que permita ejecutar, por medio de la utilización de recursos del MPSoC, la funcionalidad descrita en C++. En la Figura 57 se expone el diagrama de bloques de Vivado que representa la organización de la arquitectura desarrollada.

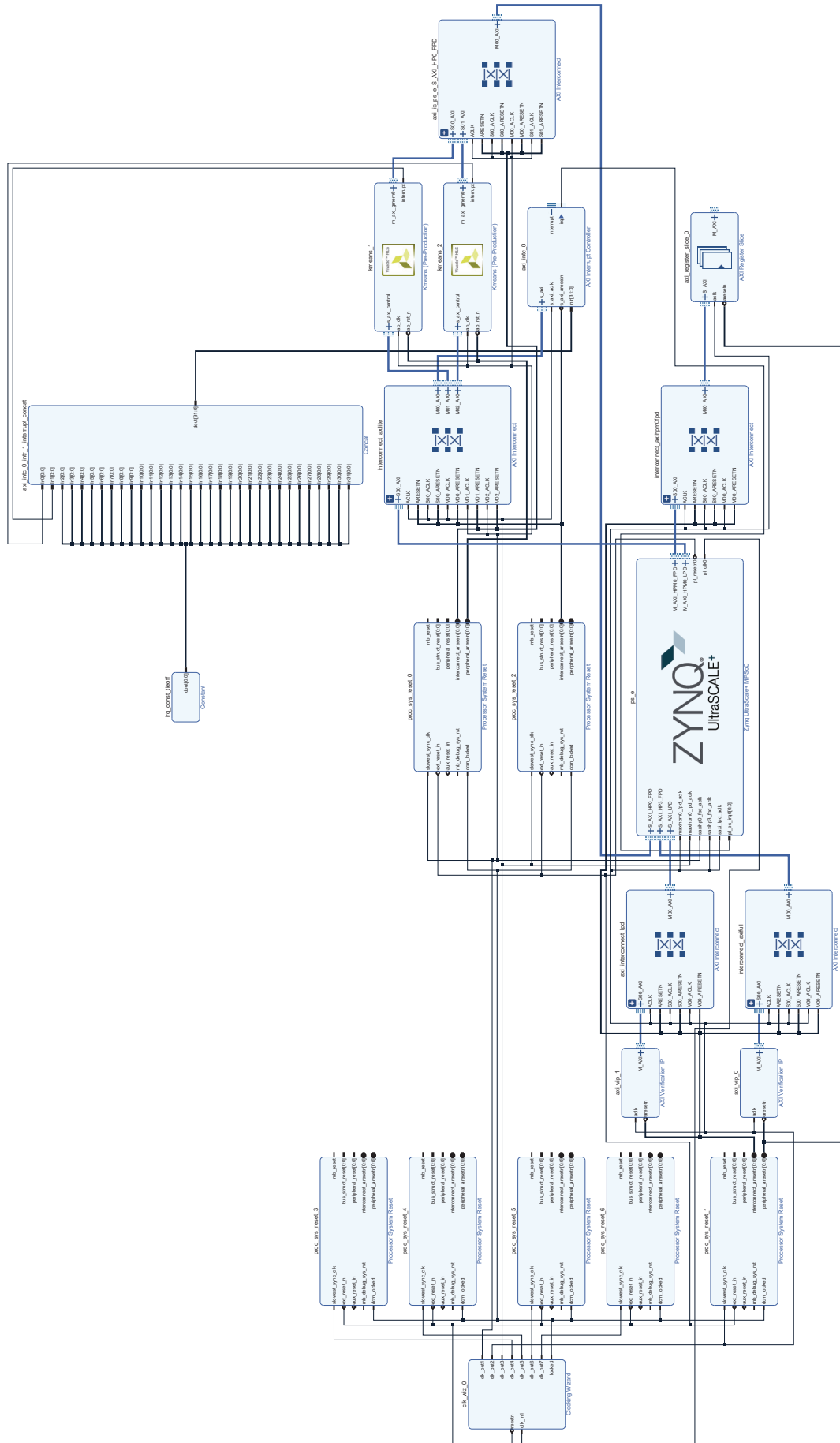


Figura 57. Arquitectura de la Plataforma

En la Figura 57 puede apreciarse la existencia de un bloque ps_e, el cual se correspondería con el *host* de la arquitectura. La comunicación del PS con el resto del sistema es gestionada en su mayor parte por varios bloques AXI Interconnect de manera directa. En la Figura 58 se muestra el bloque ps_e junto con algunos de los bloques AXI Interconnect que lo comunican.



Figura 58: Incorporación del host en el diagrama de bloques

Cabe destacar que cada bloque de interconexión AXI conectado al *host* gestiona únicamente flujos de entrada o salida, pero no bidireccionales. En la Figura 59 se detalla un diagrama acerca de los recursos internos al bloque ps_e, donde además se observa cada uno de los diferentes puertos AXI utilizados y la direccionalidad de los mismos.

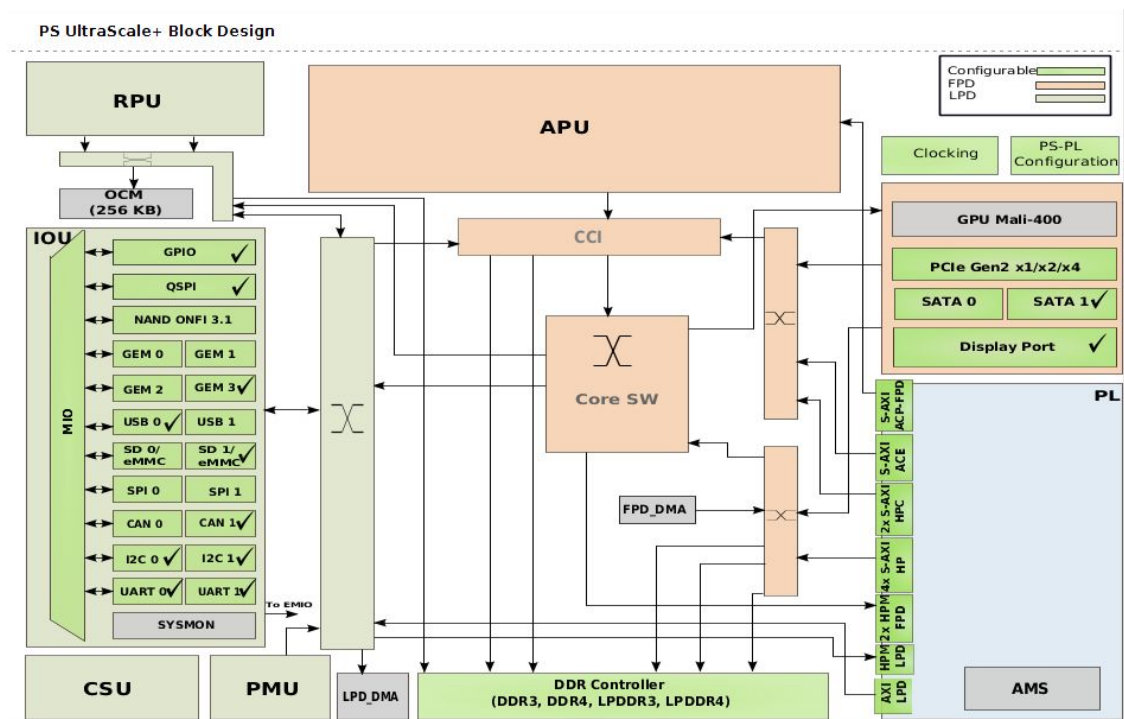


Figura 59: Diagrama de bloques dentro de ps_e

En el diagrama de bloques de la aplicación resulta de especial interés la comunicación entre el *host* y los *kernels*. Asimismo, para la transmisión de los datos desde el *host* hasta los *kernels* se utiliza el puerto M_AXI_HPMO_LPD presente en el bloque ps_e, el cual se conecta a un bloque AXI Interconnect que opera bajo el protocolo AXI-Lite. Este bloque multiplexa la entrada de datos de los dos *kernels* implementados en este proyecto, así como de un bloque AXI Interrupt Controller. En la Figura 60 se puede apreciar este hecho dentro del diagrama de bloques. Los *kernels* aparecen en este diagrama identificados como “kmeans_1” y “kmeans_2”. Con respecto a las señales de reloj y *reset* de los *kernels* referenciadas como ap_clk y ap_rst_n, estas son generadas y recibidas directamente desde la lógica interna del bloque ps_e. También se observa que las salidas de los *kernels* denominadas como “m_axi_gmem0” son multiplexadas por otro bloque AXI Interconnect, el cual conmutará estas salidas a la entrada S_AXI_HP0_FPD del bloque ps_e. De esta manera, se establece el medio para la transferencia de los resultados de ejecución de los *kernels* hacia la placa de prototipado.

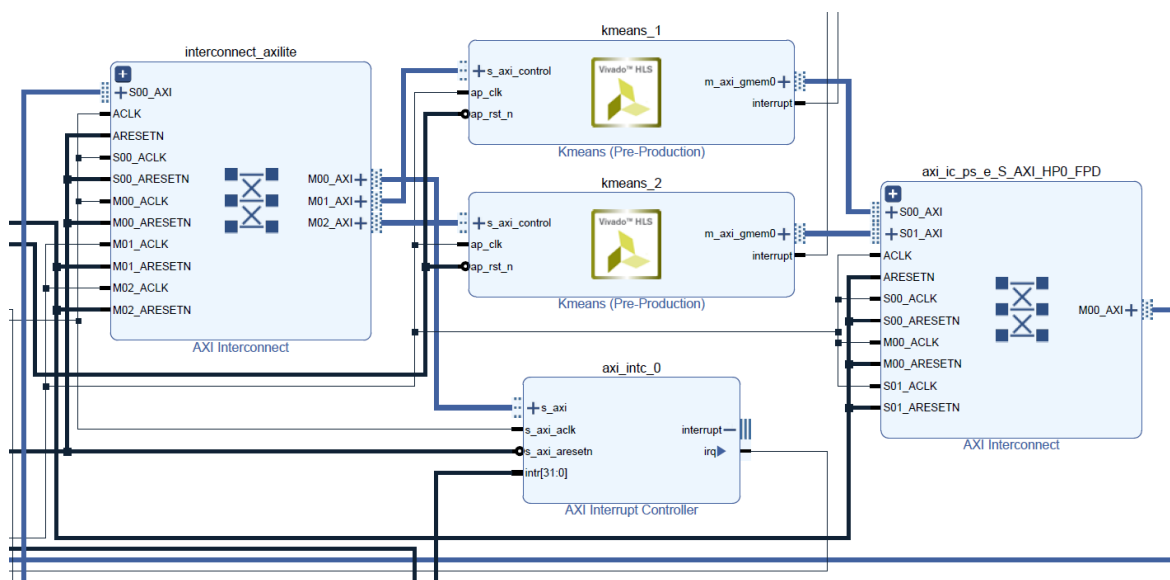


Figura 60: Introducción de los kernels en la arquitectura de la aplicación

Como mecanismo para la consulta entre el *host* y los *kernels* se utilizan interrupciones IRQ. Para ello, cada *kernel* dispone de una salida interrupt que utiliza para solicitar una interrupción al *host* cuando finaliza la ejecución solicitada. Las señalizaciones de interrupción de los *kernels* son concatenadas y conectadas al bloque AXI Interrupt Controller (Figura 61).

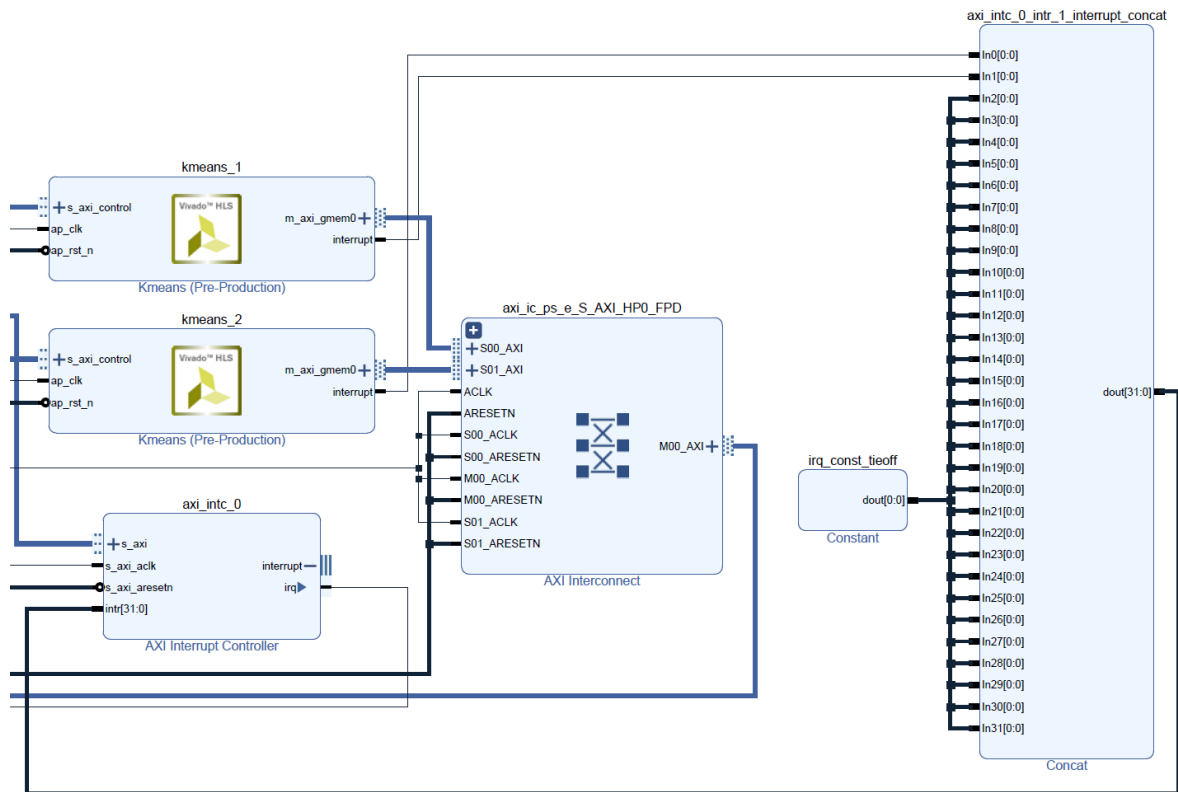


Figura 61: Gestión de interrupciones de los kernels por el bloque AXI Interrupt Controller

La transacción de la notificación de interrupción finaliza conectando la salida `irq` del bloque AXI Interrupt Controller a la entrada `ps_pl_irq` del bloque `ps_e`.

Otros bloques que forman parte del diagrama de Vivado son dos bloques AXI Verification IP. La utilidad de estos consiste en verificar la conectividad, así como la funcionalidad de los maestros y esclavos AXI mediante flujo de diseño RTL customizado. Asimismo, el usuario puede utilizar estos módulos para monitorizar las transacciones AXI además de testearlas [33]. En la Figura 62 se expone la presencia de estos bloques en la arquitectura del proyecto, además de su conexión al *host* por medio de dos bloques AXI Interconnect denominados como `axi_interconnect_lpd` e `interconnect_axifull`.

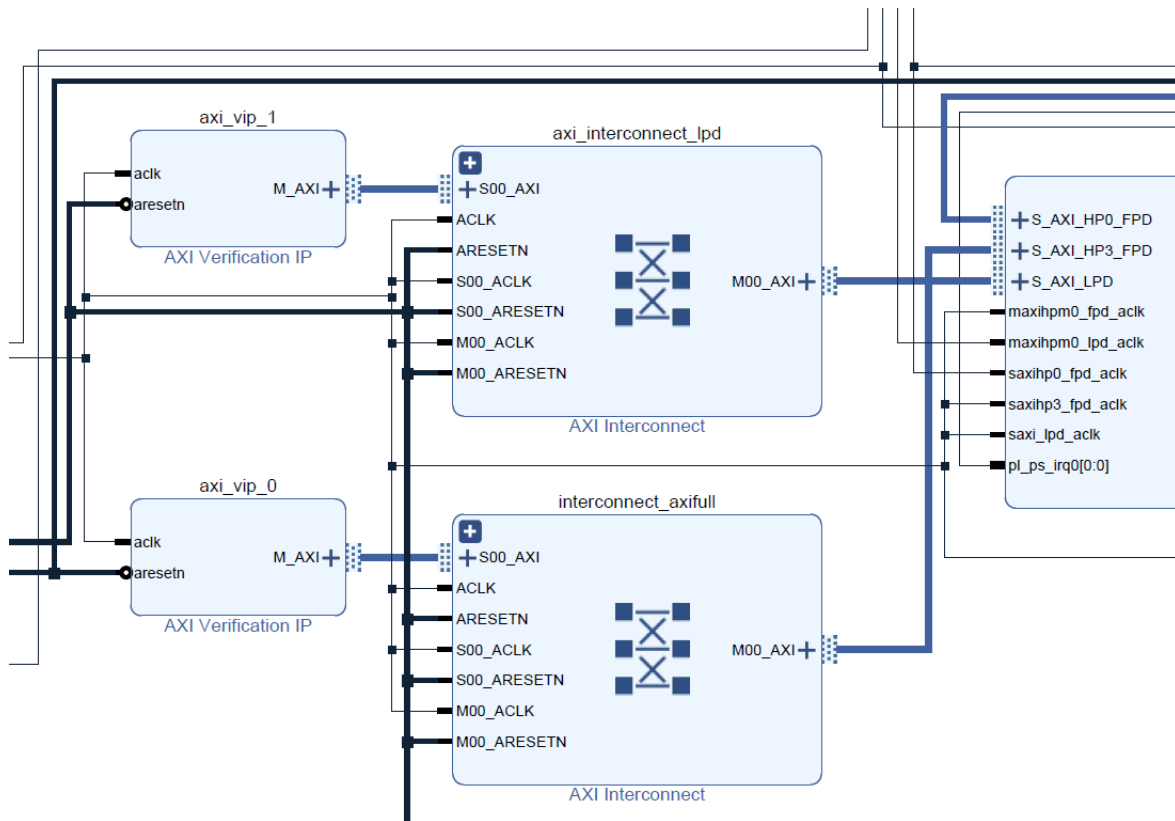


Figura 62: Implementación de los bloques AXI Verification IP

Otro bloque que se incluye en el diagrama de la arquitectura es un bloque Clocking Wizard. Su función consiste en generar varias señales de reloj a partir de la señal de reloj de la parte PS del sistema, la cual recibe en la entrada `clk_in1`. En la Figura 63 se muestra el bloque Clocking Wizard existente en la arquitectura.

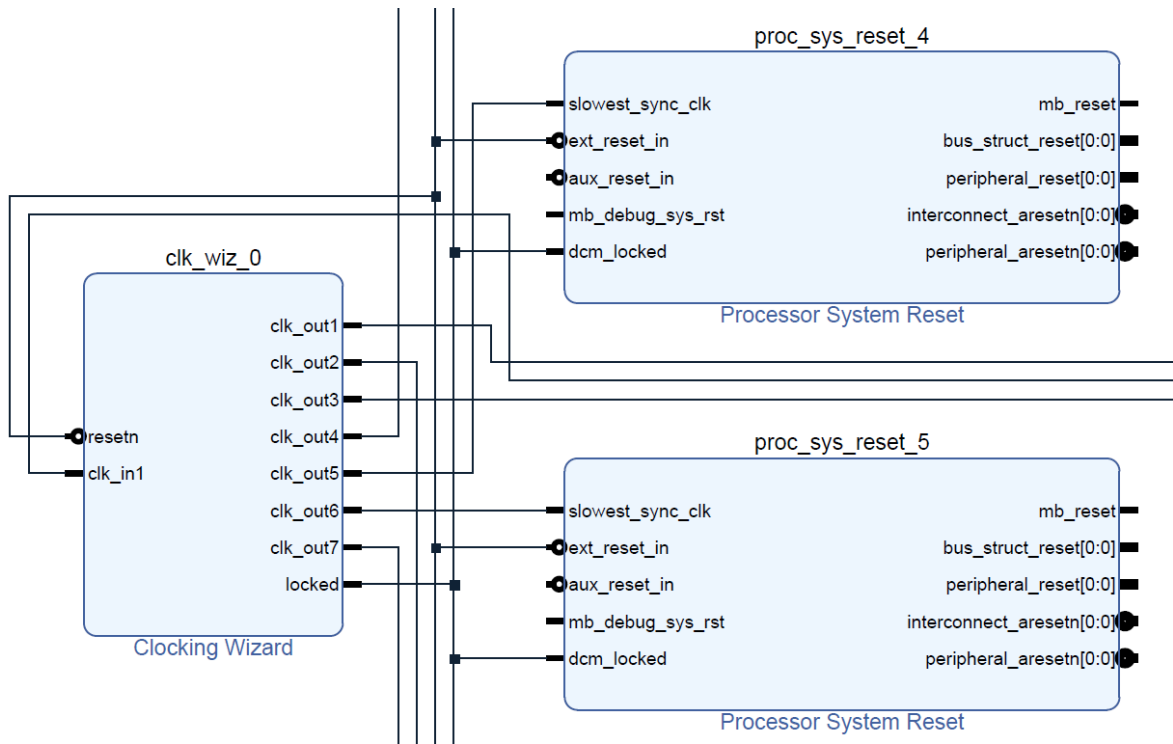


Figura 63: Clocking Wizard

Como último bloque reseñable de la arquitectura, se encuentran los bloques Processor System Reset. La función de estos bloques consiste en gestionar el reseteo de sectores de la arquitectura desarrollada. En el diagrama se encuentran siete bloques de este tipo, mostrándose dos ellos en la Figura 64 identificados como proc_sys_reset_0 y proc_sys_reset_2.

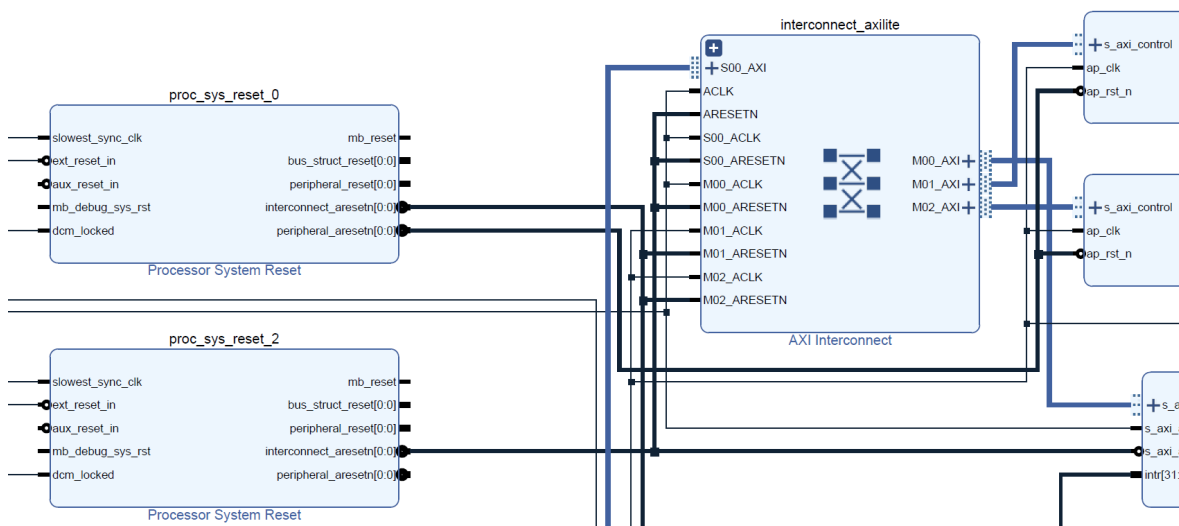


Figura 64: Bloque Processor System Reset

6.4.4. Análisis de la implementación "hw"

En la Figura 65 se muestra la ventana principal de Vitis Analyzer, donde se expone además parte del contenido del informe "Summary" del apartado "krnl_kmeans (Hardware)". Dicho

informe aporta datos genéricos acerca del proceso de compilación del acelerador *hardware*, tales como el tiempo incurrido en la compilación, o los parámetros de configuración establecidos para la compilación. Este último aspecto se muestra en la Figura 66, dentro del subpartado “*COMMAND LINE*”.

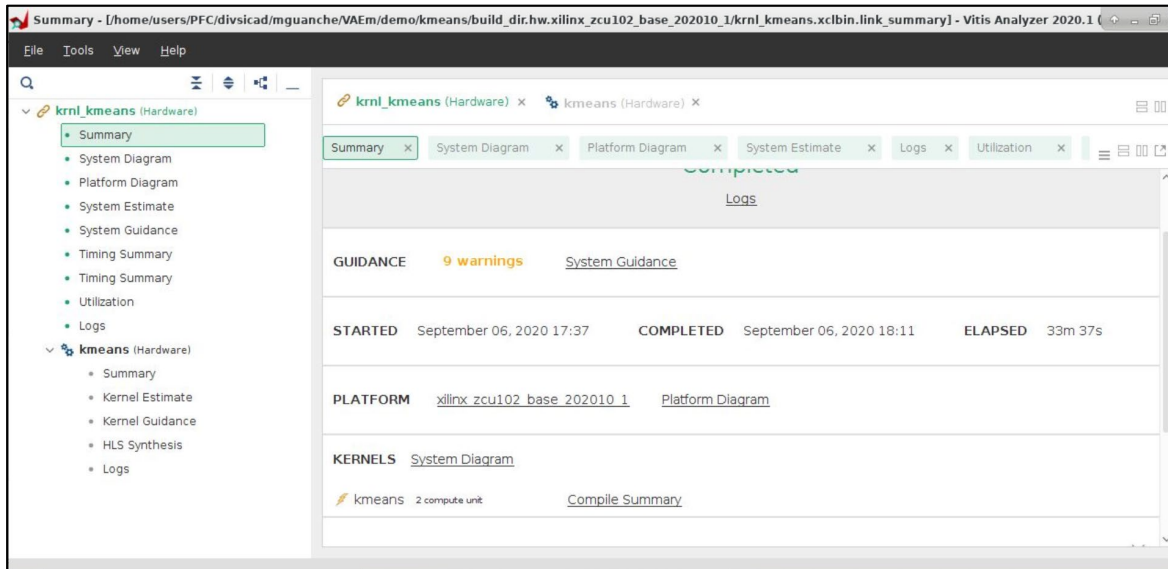


Figura 65: Informes sobre la ejecución de la opción *hw*

```
COMMAND LINE
-DPARALLEL_POINTS=96
-t hw
--platform xilinx_zcu102_base_202010_1
--save-temps
--temp_dir ./build_dir.hw.xilinx_zcu102_base_202010_1
-l
-O3
--config krnl_kmeans.ini
-obuild_dir.hw.xilinx_zcu102_base_202010_1/krnl_kmeans.xclbin_x.hw.xilinx_zcu102_base_202010_1/kmeans.xo
```

Figura 66: Parámetros de la compilación de la opción “*hw*”

El resultado de la implementación del acelerador *hardware* se resume en un diseño denominado “*krnl_kmeans*”, que consiste en 2 *kernels* idénticos llamados “*kmeans_1*” y “*kmeans_2*”, que se presenta en la pestaña “*System Diagram*” (Figura 67).

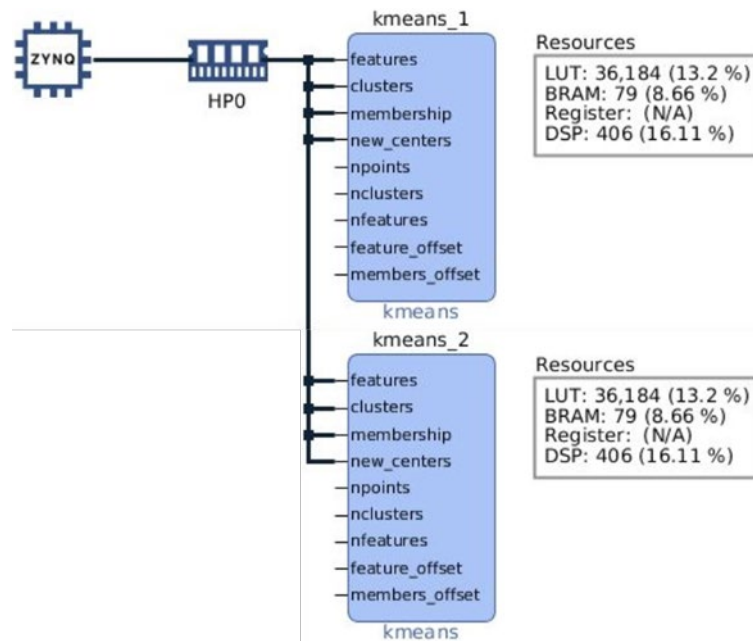


Figura 67: Diagrama principal del acelerador hardware

Cabe destacar que las entradas de los módulos “kmeans” coinciden en nombre y funcionalidad con los parámetros de entrada de la función “kmeans” del archivo `krnl_kmeans.cpp` visto con anterioridad. Los *kernels* se implementan en la parte PL del sistema, comunicándose a través de la interfaz HPO con el *host* implementado en el PS del MPSoC.

En la pestaña “*System Estimate*” puede leerse los resultados en términos de rendimiento obtenidos en la arquitectura virtualizada en la lógica programable. En dicho informe, los resultados se indican de manera individualizada tanto para los *kernels* “kmeans” como para los módulos fundamentales que los componen, siendo estos últimos los especificados en la descripción en código fuente de la aplicación como las funciones “`proc_new_centers`” y “`proc_new_memberships`”. Asimismo, en la Figura 68 se muestra la frecuencia de funcionamiento estimada para cada uno de esos módulos tras procurar alcanzar la frecuencia objetivo.

Timing Information (MHz)				
Compute Unit	Kernel Name	Module Name	Target Frequency	Estimated Frequency
kmeans_1	kmeans	proc_memberships5	149.925034	205.465378
kmeans_1	kmeans	proc_new_centers	149.925034	205.465378
kmeans_1	kmeans	kmeans	149.925034	205.465378
kmeans_2	kmeans	proc_memberships5	149.925034	205.465378
kmeans_2	kmeans	proc_new_centers	149.925034	205.465378
kmeans_2	kmeans	kmeans	149.925034	205.465378

Figura 68: Frecuencia de funcionamiento estimada de los distintos módulos

Adicionalmente, también se informa sobre la latencia inherente a cada módulo, expresada tanto en función de los ciclos de reloj, como de manera absoluta indicada en milisegundos. La

estimación de la latencia se presenta para el peor y el mejor resultado esperable durante el funcionamiento del acelerador *hardware*. Además, se indicará cuál es el valor promedio de dicho intervalo. En la Figura 69 y la Figura 70 se exponen los resultados obtenidos con respecto a la latencia.

Latency Information				
Compute Unit	Kernel Name	Module Name	Start Interval	Best (cycles)
kmeans_1	kmeans	proc_memberships5	8766248 ~ 9476679	8766248
kmeans_1	kmeans	proc_new_centers	8617852 ~ 8618134	8617852
kmeans_1	kmeans	kmeans	8766249 ~ 9476680	9479617
kmeans_2	kmeans	proc_memberships5	8766248 ~ 9476679	8766248
kmeans_2	kmeans	proc_new_centers	8617852 ~ 8618134	8617852
kmeans_2	kmeans	kmeans	8766249 ~ 9476680	9479617

Figura 69: Latencia de los módulos (I)

Latency Information					
Compute Unit	Avg (cycles)	Worst (cycles)	Best (absolute)	Avg (absolute)	Worst (absolute)
kmeans_1	9121464	9476679	58.445 ms	60.813 ms	63.181 ms
kmeans_1	8617993	8618134	57.455 ms	57.456 ms	57.457 ms
kmeans_1	9479758	9479899	63.201 ms	63.202 ms	63.202 ms
kmeans_2	9121464	9476679	58.445 ms	60.813 ms	63.181 ms
kmeans_2	8617993	8618134	57.455 ms	57.456 ms	57.457 ms
kmeans_2	9479758	9479899	63.201 ms	63.202 ms	63.202 ms

Figura 70: Latencia de los módulos (II)

Además de los aspectos temporales, otra característica de gran relevancia en un diseño de *hardware* programable es el consumo de recursos FPGA que implica cada uno de sus módulos. De esta manera, en la Figura 71 se especifica la cantidad de recursos de cada tipo que comprende la implementación de los módulos del acelerador *hardware*, la cual se limita a bloques de memoria RAM, celdas *flip-flops* y LUTs.

Area Information							
Compute Unit	Kernel Name	Module Name	FF	LUT	DSP	BRAM	URAM
kmeans_1	kmeans	proc_memberships5	19285	44149	0	112	0
kmeans_1	kmeans	proc_new_centers	3291	7003	0	16	0
kmeans_1	kmeans	kmeans	49478	65149	0	254	0
kmeans_2	kmeans	proc_memberships5	19285	44149	0	112	0
kmeans_2	kmeans	proc_new_centers	3291	7003	0	16	0
kmeans_2	kmeans	kmeans	49478	65149	0	254	0

Figura 71: Recursos FPGA requeridos por cada módulo

En cuanto a los informes de “*Guidance*” en los cuales se notifica cualquier error o incidencia relativa a la implementación del diseño *hardware* según lo establecido por el usuario en el código fuente. Como puede observarse en la Figura 72, la implementación de los *kernels* ha podido efectuarse y el acelerador *hardware* se encuentra operativo.

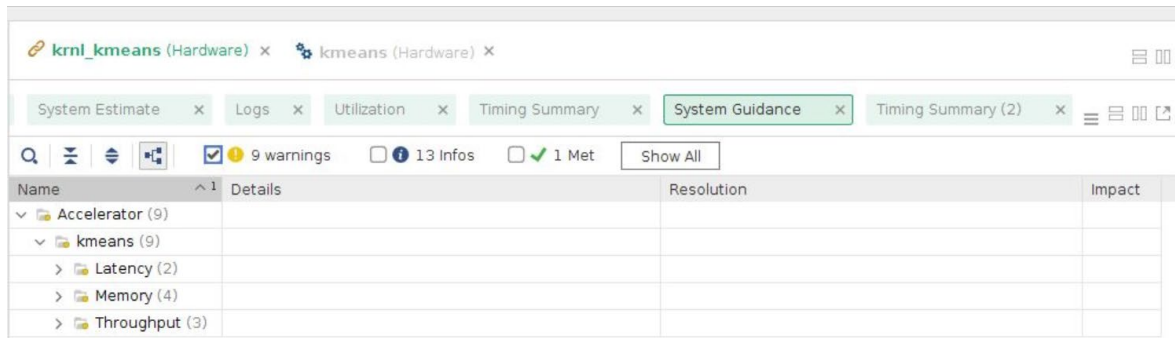


Figura 72: Errores o incidencias en la implementación de los kernels

En relación a los resultados obtenidos del análisis temporal, se generan dos informes de resumen temporal (“*Timing Summary*”). En la Figura 73 se muestra la configuración utilizada para el análisis temporal (“*Timer Settings*”). Como se puede apreciar se hace un análisis de múltiples esquinas para tener en cuenta la dispersión de los parámetros temporales de los dispositivos.

```

| Timer Settings
| -----
-----

Enable Multi Corner Analysis      : Yes
Enable Pessimism Removal          : Yes
Pessimism Removal Resolution      : Nearest Common Node
Enable Input Delay Default Clock  : No
Enable Preset / Clear Arcs        : No
Disable Flight Delays              : No
Ignore I/O Paths                  : No
Timing Early Launch at Borrowing Latches : No
Borrow Time for Max Delay Exceptions : Yes
Merge Timing Exceptions            : Yes

Corner  Analyze  Analyze
Name    Max Paths Min Paths
-----
Slow    Yes     Yes
Fast    Yes     Yes

```

Figura 73: Contenido del apartado *Timing Settings*

El siguiente apartado destacable que se encuentra en el “*Timing Report*” es la verificación temporal realizada que se resume en el informe “*Check Timing*”. En dicho apartado, se estudian las restricciones de temporización establecidas en el diseño implementado, con el objetivo de detectar si falta alguna por declarar, o si se sospecha que los resultados obtenidos de alguna de ellas es incorrecta. En la Figura 74 se muestra que no existe ninguna incidencia en el diseño, desde el punto de vista temporal, como manifiesta el resultado “(0)” presente al final de cada uno de los “*checkings*” declarados.

```

Table of Contents
-----
1. checking no_clock (0)
2. checking constant_clock (0)
3. checking pulse_width_clock (0)
4. checking unconstrained_internal_endpoints (0)
5. checking no_input_delay (0)
6. checking no_output_delay (0)
7. checking multiple_clock (0)
8. checking generated_clocks (0)
9. checking loops (0)
10. checking partial_input_delay (0)
11. checking partial_output_delay (0)
12. checking latch_loops (0)
    
```

Figura 74: Aspectos analizados en el apartado Check Timing

En el apartado “*Design Timing Summary*” se resume si el diseño *hardware* satisface las restricciones temporales impuestas, hecho que se confirmaría si TNS (*Total Negative Slack*), THS (*Total Hold Slack*) y TPWS (*Total Pulse Width Negative Slack*) adquieren un resultado no negativo. Este hecho implicaría que las transiciones de las señales se mantendrían en todo momento dentro de los intervalos preestablecidos. Asimismo, en la Figura 75 y la Figura 76 se demuestra que el cumplimiento por parte del diseño de las restricciones temporales es absoluto.

```

-----
| Design Timing Summary
| -----
WNS(ns)  TNS(ns)  TNS Failing Endpoints  TNS Total Endpoints  WHS(ns)  THS(ns)
-----
0.419    0.000    0                      353822                0.010    0.000
    
```

Figura 75: “*Design Timing Summary*” (I)

```

-----
| Design Timing Summary
| -----
-----
THS Failing  THS Total  WPWS   TPWS   TPWS Failing  TPWS Total
Endpoints   Endpoints  (ns)   (ns)   Endpoints     Endpoints
-----
0           353822    0.167  0.000  0             104497
    
```

Figura 76: “*Design Timing Summary*” (II)

En el apartado “*Clock Summary*” se describen las características de todas las señales de reloj presentes en la arquitectura *hardware* (Figura 77).


```

-----
| Clock Summary
| -----
-----

```

Clock	Waveform(ns)	Period(ns)	Frequency(MHz)
clk_pl_0	{0.000 5.000}	10.000	100.000
zcu102_base_i/clk_wiz_0/inst/clk_in1	{0.000 5.000}	10.000	100.000
clk_out1_zcu102_base_clk_wiz_0_0	{0.000 3.333}	6.667	150.000
clk_out2_zcu102_base_clk_wiz_0_0	{0.000 1.667}	3.333	300.000
clk_out3_zcu102_base_clk_wiz_0_0	{0.000 6.667}	13.333	75.000

Figura 77: Clock Summary

Además de los apartados anteriormente vistos, el “*Timing Summary*” posee una serie de apartados orientados a exponer la evaluación del cumplimiento de las restricciones temporales en las señales del diseño, presentando los terminales donde comienzan las rutas analizadas. Uno de estos apartados es el de “*Pulse Width Checks*” donde se analiza si el ancho, así como el periodo de los pulsos de transmisión de información, se ajustan adecuadamente a los requisitos temporales. En la Figura 78 se muestra un ejemplo de este apartado obtenido en la compilación de la aplicación.

```

Pulse Width Checks
-----
Clock Name:          zcu102_base_i/clk_wiz_0/inst/clk_in1
Waveform(ns):       { 0.000 5.000 }
Period(ns):         10.000
Sources:            { zcu102_base_i/clk_wiz_0/inst/clk_in1 }

```

Check Type	Corner	Lib Pin	Reference Pin	Required(ns)	Actual(ns)
Min Period	n/a	MMCME4_ADV/CLKIN1	n/a	1.071	10.000
Max Period	n/a	MMCME4_ADV/CLKIN1	n/a	100.000	10.000
Low Pulse Width	Fast	MMCME4_ADV/CLKIN1	n/a	3.000	5.000
Low Pulse Width	Slow	MMCME4_ADV/CLKIN1	n/a	3.000	5.000
High Pulse Width	Slow	MMCME4_ADV/CLKIN1	n/a	3.000	5.000
High Pulse Width	Fast	MMCME4_ADV/CLKIN1	n/a	3.000	5.000

```

Pulse Width Checks
-----
Clock Name:          zcu102_base_i/clk_wiz_0/inst/clk_in1
Waveform(ns):       { 0.000 5.000 }
Period(ns):         10.000
Sources:            { zcu102_base_i/clk_wiz_0/inst/clk_in1 }

```

Slack(ns)	Location	Pin
8.929	MMCM_X0Y2	zcu102_base_i/clk_wiz_0/inst/mmcme4_adv_inst/CLKIN1
90.000	MMCM_X0Y2	zcu102_base_i/clk_wiz_0/inst/mmcme4_adv_inst/CLKIN1
2.000	MMCM_X0Y2	zcu102_base_i/clk_wiz_0/inst/mmcme4_adv_inst/CLKIN1
2.000	MMCM_X0Y2	zcu102_base_i/clk_wiz_0/inst/mmcme4_adv_inst/CLKIN1
2.000	MMCM_X0Y2	zcu102_base_i/clk_wiz_0/inst/mmcme4_adv_inst/CLKIN1
2.000	MMCM_X0Y2	zcu102_base_i/clk_wiz_0/inst/mmcme4_adv_inst/CLKIN1

Figura 78: Pulse Width Checks

Otros apartados en los que se efectúa un análisis más específico de las señales transferidas a través de los pines del diseño son los denominados “*Max Delay Paths*” y “*Min Delay Paths*”. El primero de ellos estudia la parte “*setup*” de una señal y comprueba que el tiempo implicado en esta transición no exceda el máximo permisible. El apartado “*Min Delay Paths*” evalúa la contraparte “*hold*” de la señal, verificando si una vez iniciada la captación del dato este se mantiene durante un tiempo superior al mínimo exigido para asegurar su correcta interpretación. En la Figura 79 puede observarse la parte inicial, más genérica, de uno de los múltiples apartados “*Max Delay Paths*” resultantes.

```

Max Delay Paths
-----
Slack (MET) :                0.419ns (required time - arrival time)
Source:
zcu102_base_i/kmeans_1/inst/proc_memberships5_U0/1_clusters_7_U/kmeans_proc_m
memberships5
                                _1_clusters_0_ram_U/ram_reg/CLKARDCLK
                                (rising edge-triggered cell RAMB18E2 clocked by
clk_out1_zcu102_base_clk_wiz_0_0
                                {rise@0.000ns fall@3.333ns period=6.667ns})
Destination:
zcu102_base_i/kmeans_1/inst/proc_memberships5_U0/mul_33s_33s_64_1_1_U51/kmean
s_mul_33s_33s_64_1_1_Multiplier_4_U/p_0/DSP_A_B_DATA_INST/A[7]
                                (rising edge-triggered cell DSP_A_B_DATA clocked
by clk_out1_zcu102_base_clk_wiz_0_0 {rise@0.000ns fall@3.333ns
period=6.667ns})
Path Group:                clk_out1_zcu102_base_clk_wiz_0_0
Path Type:                Setup (Max at Slow Process Corner)
Requirement:             6.667ns (clk_out1_zcu102_base_clk_wiz_0_0
rise@6.667ns - clk_out1_zcu102_base_clk_wiz_0_0 rise@0.000ns)
Data Path Delay:         5.646ns (logic 1.375ns (24.354%) route 4.271ns
(75.646%))
Logic Levels:           6 (CARRY8=2 LUT2=1 LUT6=1 MUXF7=1 MUXF8=1)
Clock Path Skew:        -0.281ns (DCD - SCD + CPR)
  Destination Clock Delay (DCD):    3.863ns = ( 10.530 - 6.667 )
  Source Clock Delay (SCD):         3.657ns
  Clock Pessimism Removal (CPR):    -0.487ns
    
```

Figura 79: Ejemplo de parte inicial de apartado “*Max Delay Paths*”

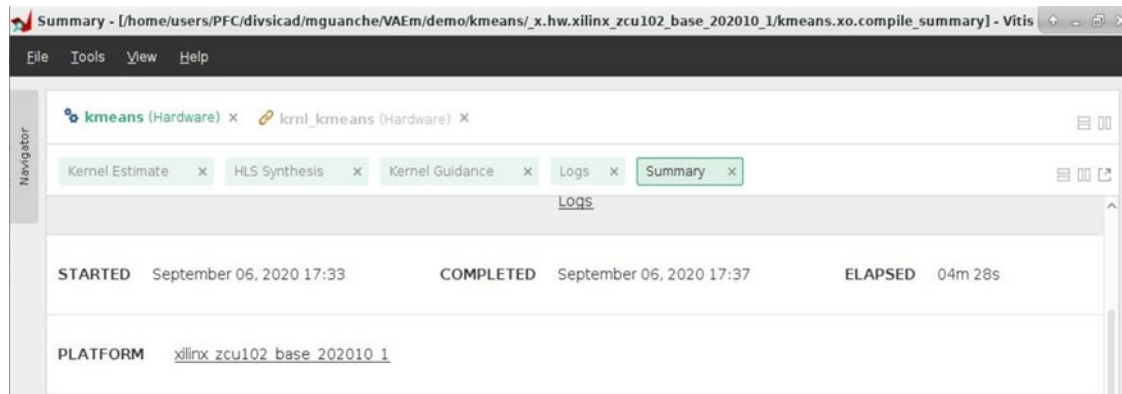
Por otra parte, en la Tabla 7 se muestra el informe “*Utilization*”. En dicho informe se detallan las cantidades de recursos PL de cada tipo que son ocupados en el diseño del acelerador *hardware*, efectuándose la comparativa entre dichas cantidades y las disponibles según las limitaciones impuestas por el dispositivo o las restricciones de utilización indicadas por el diseñador. Esta comparativa se especifica en el apartado “*User Budget*”. Además, se desglosan los recursos implicados en cada uno de los módulos principales que conforman la arquitectura implementada, los cuales, en este caso, son los 2 *kernels* “*kmeans*” [34].

Tabla 7: Informe "Utilization"

Name	LUT	LUTAsMem	REG	BRAM	DSP
Platform	8132 [2.97%]	683 [0.47%]	17990 [3.28%]	48 [5.26%]	0 [0.00%]
User Budget	265948 [100.00%]	143317 [100.00%]	530170 [100.00%]	864 [100.00%]	2520 [100.00%]
Used Resources	72368 [27.21%]	10108 [7.05%]	76992 [14.52%]	158 [18.29%]	812 [32.22%]
Unused Resources	193580 [72.79%]	133209 [92.95%]	453178 [85.48%]	706 [81.71%]	1708 [67.78%]
kmeans	72368 [27.21%]	10108 [7.05%]	76992 [14.52%]	158 [18.29%]	812 [32.22%]
kmeans_1	36184 [13.61%]	5054 [3.53%]	38496 [7.26%]	79 [9.14%]	406 [16.11%]
kmeans_2	36184 [13.61%]	5054 [3.53%]	38496 [7.26%]	79 [9.14%]	406 [16.11%]

Como último informe relativo a la compilación del diseño del conjunto del acelerador *hardware* se encuentra el informe "Logs", cuyo objetivo es indicar el conjunto de acciones ejecutadas en el proceso de compilación de los *kernels*.

Los informes comentados hasta el momento poseen como aspecto común que evalúan el conjunto de la arquitectura que conforma el acelerador *hardware*. Sin embargo, también se encuentran una serie de informes destinados a evidenciar solamente los resultados de la implementación de los *kernels*. Dichos informes se agrupan en la pestaña "kmeans (*hardware*)" ubicada en la ventana de inicio. Uno de estos informes se denomina "Summary", cuyo contenido es el mostrado en la Figura 80. En este informe se indican aspectos generales de la compilación de uno de los *kernels* "kmeans", al igual que ocurría en el "Summary" de la compilación del acelerador *hardware*.



```

COMMAND LINE
-DPARALLEL_POINTS=96
-t hw
--platform xilinx_zcu102_base_202010_1
--save-temps
--temp_dir ./_x.hw.xilinx_zcu102_base_202010_1
-c
-k kmeans
-Isrc
-o _x.hw.xilinx_zcu102_base_202010_1/kmeans.xo src/krn1_kmeans.cpp
    
```

Figura 80: Informe "Summary" del kernel "kmeans"

Otra analogía con respecto a los informes de "krnl_kmeans" y los de "kmeans" se encuentra en el caso del informe "Kernel Estimate" presente en "kmeans". Su contenido principal puede observarse en la Figura 81, y coincide con la información expuesta en el informe "System Estimate" así como el formato de este.

Timing Information (MHz)				
Compute Unit	Kernel Name	Module Name	Target Frequency	Estimated Frequency
kmeans_1	kmeans	proc_memberships5	149.925034	205.465378
kmeans_1	kmeans	proc_new_centers	149.925034	205.465378
kmeans_1	kmeans	kmeans	149.925034	205.465378

Latency Information				
Compute Unit	Kernel Name	Module Name	Start Interval	Best (cycles)
--	--	--	--	--
kmeans_1	kmeans	proc_memberships5	8766248 ~ 9476679	8766248
kmeans_1	kmeans	proc_new_centers	8617852 ~ 8618134	8617852
kmeans_1	kmeans	kmeans	8766249 ~ 9476680	9479617

Latency Information					
Compute Unit	Avg (cycles)	Worst (cycles)	Best (absolute)	Avg (absolute)	Worst (absolute)
kmeans_1	9121464	9476679	58.445 ms	60.813 ms	63.181 ms
kmeans_1	8617993	8618134	57.455 ms	57.456 ms	57.457 ms
kmeans_1	9479758	9479899	63.201 ms	63.202 ms	63.202 ms

Area Information							
Compute Unit	Kernel Name	Module Name	FF	LUT	DSP	BRAM	URAM
-----	-----	-----	----	-----	---	----	----
kmeans_1	kmeans	proc_memberships5	19285	44149	0	112	0
kmeans_1	kmeans	proc_new_centers	3291	7003	0	16	0
kmeans_1	kmeans	kmeans	49478	65149	0	254	0

Figura 81: Kernel Estimate

Un informe clave para entender el proceso de transformación del código C/C++ en *hardware* es el denominado “HLS Synthesis”. En dicho informe se resumen los resultados, tanto desde el punto de vista de la latencia del *kernel* como de los recursos estimados que utilizará finalmente en su implementación. Como se aprecia en la Tabla 8, se identifican las funciones principales y los bucles utilizados en cada función, mostrando las latencias, qué bucles ha sido implementados siguiendo un estilo *dataflow* y la capacidad de introducir segmentación en su ejecución. Las columnas “Trip Count”, “iteration latency” e “Interval” dan información del número de iteraciones en intervalos de inicio de una nueva iteración.

Tabla 8: Informe HLS Synthesis: Latencias

Name	Latency (cycles)	Latency (ns)	Iteration Latency	Interval	Trip Count	Pipelined
kmeans	9479899	6,32E+10		9476680		dataflow
proc_memberships5	9476679	6,32E+10		9476679		no
ld_clusters	213	1,42E+06	3	1	212	yes
VITIS_LOOP_275_1	9476359	6,32E+10	3683		2573	no
ld_features	205	1,37E+06	3	1	204	yes
calc_indexes_VITIS_LOOP_117_2	3402	2,27E+07	3	1	3400	yes
proc_new_centers	8618134	5,75E+10		8618134		no
init_centers	3408	2,27E+07	1	1	3408	yes
VITIS_LOOP_320_1	8614404	5,74E+10	3348		2573	no
calc_centers_VITIS_LOOP_156_1	3269	2,18E+07	7	1	3264	yes
st_members	6	40.002	2	1	6	yes
st_centers	213	1,42E+06	3	1	212	yes

Tabla 9. Informe HLS Synthesis: Recursos

Name	BRAM	BRAM (%)	DSP	DSP (%)	FF	FF (%)	LUT	LUT (%)
kmeans	254	13	406	16	49478	9	65149	23
proc_memberships5	112	6	398	15	19285	3	44149	16
ld_clusters								
VITIS_LOOP_275_1								
ld_features								
calc_indexes_VITIS_LOOP_117_2								

Name	BRAM	BRAM (%)	DSP	DSP (%)	FF	FF (%)	LUT	LUT (%)
proc_new_centers	16	~0	8	~0	3291	~0	7003	2
init_centers								
VITIS_LOOP_320_1								
calc_centers_VITIS_LOOP_156_1								
st_members								
st_centers								

Por último, cabe destacar que dentro del apartado “kmeans (*Hardware*)” existe otro informe “*Logs*”. En dicho informe se muestran los pasos seguidos durante el proceso de síntesis, así como de implementación del diseño *hardware* de los *kernels*, partiendo de la descripción de alto nivel ofrecida por el usuario. Además, este informe notificará las advertencias oportunas acerca del diseño *hardware* y su compilación.

Adicionalmente, puede consultarse el proyecto Vivado generado durante la compilación para evaluar el consumo de potencia implicado en el diseño tal como se muestra en la Figura 82.

Summary

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power: 6.592 W
Design Power Budget: Not Specified
Power Budget Margin: N/A
Junction Temperature: 31.4°C
 Thermal Margin: 68.6°C (68.4 W)
 Effective θ_{JA} : 1.0°C/W
 Power supplied to off-chip devices: 0 W
 Confidence level: Medium

[Launch Power Constraint Advisor](#) to find and fix invalid switching activity

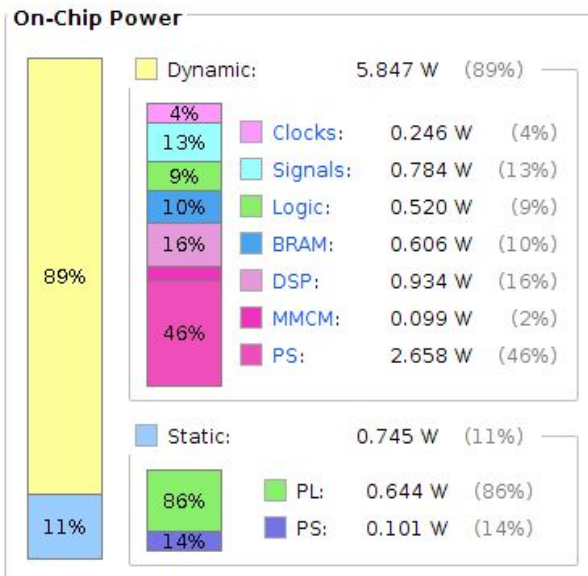


Figura 82: Informe de potencia generado en la compilación “hw”

En los resultados de la Figura 82 se observa que el consumo de potencia total es de 6.592 mW. Cabe destacar que la mayor parte de la potencia requerida por la aplicación es potencia dinámica, la cual constituye el 89 % del total, lo que se traduce en una potencia de 5.847 mW. La potencia consumida por el PS (2.658 mW) representa casi la mitad de la potencia total del acelerador.

Otro aspecto evaluado mediante la creación del proyecto Vivado es la distribución en el MPSoC de los distintos módulos consumidos para la implementación del proyecto. Este aspecto se muestra visualmente en la Figura 83. En dicha figura, los recursos implicados en la implementación de los *kernels* *kmeans_1* y *kmeans_2* presentan color verde y azul oscuro respectivamente. Por otra parte, el *host* se señala con color amarillo y los elementos de interconexión AXI se destacan en color rojo.

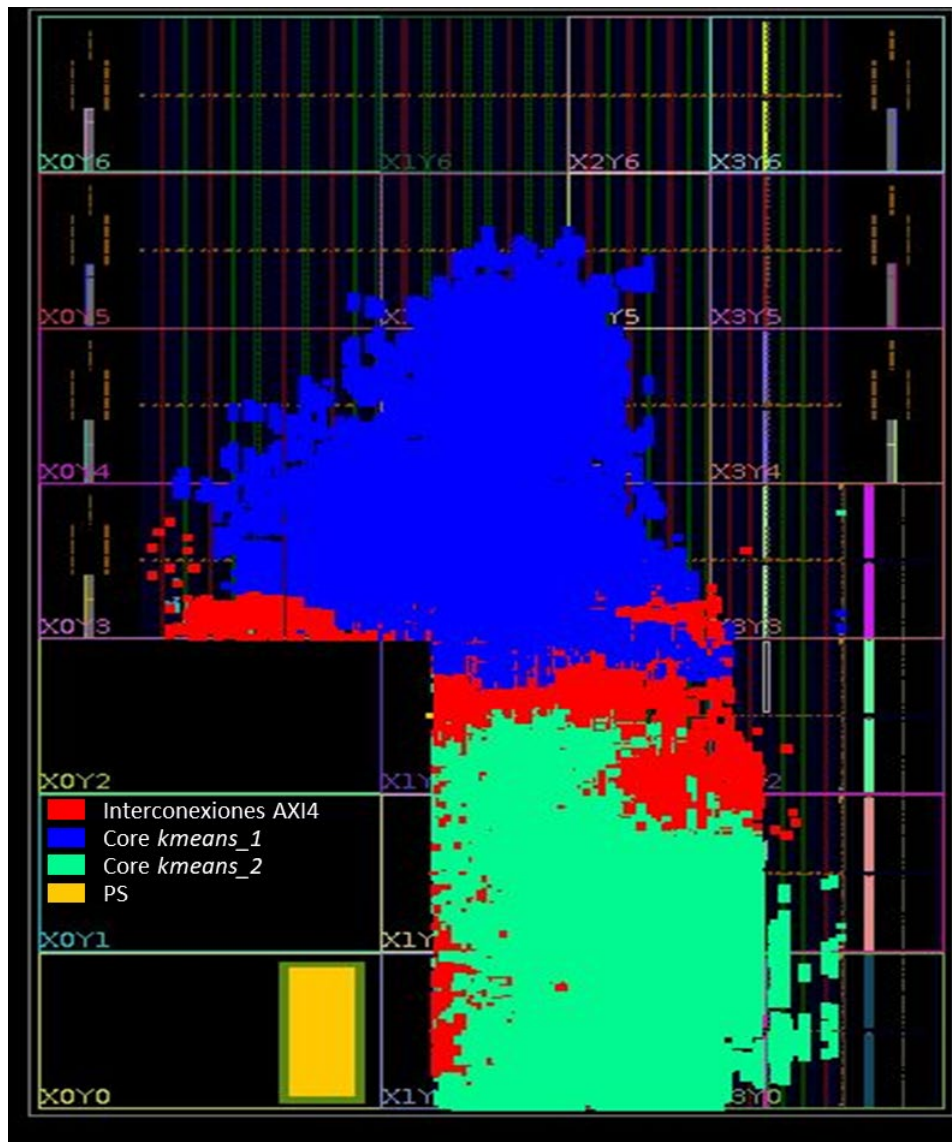


Figura 83: Distribución de recursos del MPSoC en la implementación de *kmeans*.

Con respecto a la búsqueda del peor camino existente en el diseño se obtiene que este presenta un retardo de 5.646 ns (Figura 84). Desde el punto de vista de la algorítmica, este camino se correspondería a la medida de la distancia de separación entre el centroide de un *cluster* y una muestra en una determinada dimensión. Pese a su valor se cumplen las restricciones temporales, ya que el retardo máximo permitido es de 6.667 ns, según los datos proporcionados en Vivado.

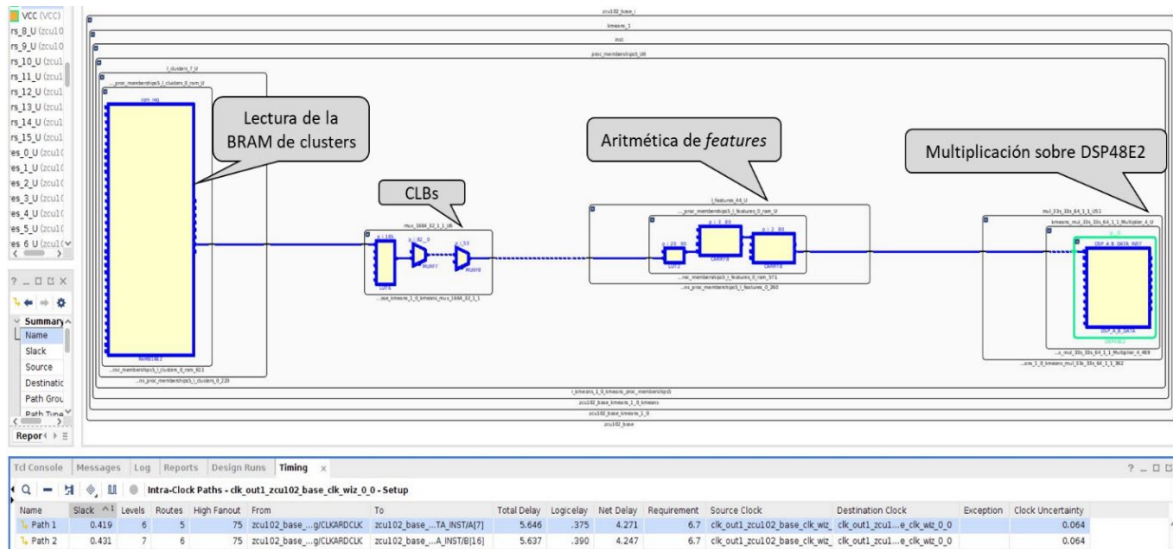


Figura 84. Determinación de la ruta crítica según Vivado

6.5. Conclusiones

En este capítulo se han tratado todos aquellos aspectos relativos a la compilación del diseño tanto para las emulaciones como para la ejecución en placa. Esto incluye una descripción de las reglas presentes en el Makefile, realizando especial hincapié en la regla build ya que es esta última la que define el proceso de compilación. Además, se han tratado las modificaciones realizadas en el contenido del Makefile original con el objetivo de adaptarlo a las necesidades de este proyecto que difieran de las originalmente planteadas.

Por otra parte, se han estudiado las prestaciones de la arquitectura de la aplicación en términos temporales, así como de consumo de recursos *hardware* y de potencia. Además, se verifica la factibilidad de la aplicación. Por último, se analiza la organización de la arquitectura de la aplicación resultante de la compilación para ejecución en la placa de prototipado.

Capítulo 7. Resultados

7.1. Introducción

En este capítulo se comienza detallando el proceso de ejecución de las emulaciones *hardware* y *software* una vez finalizaron sus respectivas compilaciones. Con ello se obtienen unos resultados de las emulaciones que verifican la adecuada funcionalidad del proyecto. A continuación, se aborda el proceso posterior a la compilación de la aplicación para ejecución en placa, indicándose los pasos seguidos para culminar con la implementación de la aplicación en la ZCU102 y ejecutarla con casos de clasificación reales.

7.2. Emulaciones software y hardware

Al lograrse la compilación del proyecto tanto para la ejecución en placa como para las emulaciones, se procede a ejecutar la emulación del mismo. Para ello, se invocará la opción “check” descrita en el archivo “Makefile”. La ejecución de la emulación será efectuada en última instancia por la aplicación QEMU, la cual es un emulador y virtualizador de código abierto, capaz de emular en un PC la funcionalidad, así como el rendimiento, de una arquitectura *hardware* partiendo de los archivos de compilación [35]. El comando utilizado es el que se expone en el Código 65 para la emulación *software*, y en el Código 66 para la emulación *hardware*.

```
host% make --debug check TARGET=sw_emu HOST_ARCH=aarch64
DEVICE=xilinx_zcu102_base_202010_1 EDGE_COMMON_SW=/opt/xilinx/xilinx-zynqmp-co
mmon-v2020.1
```

Código 65: Ejecución de la emulación software

```
host% make --debug check TARGET=hw_emu HOST_ARCH=aarch64
DEVICE=xilinx_zcu102_base_202010_1 EDGE_COMMON_SW=/opt/xilinx/xilinx-zyn qmpco
mmon-v2020.1
```

Código 66: Ejecución de la emulación hardware

Como puede observarse en los comandos anteriores, la ejecución mediante la opción “check” del *makefile* requerirá los mismos parámetros que se precisaron en la compilación

mediante la opción “build”. Adicionalmente, la ejecución de “check” incorpora la activación de la opción “--debug”, con el objetivo de especificar que se exponga una descripción más detallada de los procesos realizados.

La invocación de la ejecución de las emulaciones realizada por medio de las instrucciones descritas en el *makefile* deberá incluir la especificación de un archivo que contenga los elementos a clasificar, además de otro archivo que se utilizará para comprobar los resultados obtenidos en la clasificación. Opcionalmente, esta invocación podrá indicar el número de *clusters* en los que el algoritmo del proyecto deberá organizar los elementos de entrada, suponiendo que no se desee la opción por defecto de 5 *clusters* indicada en el código fuente escrito en C++ del proyecto. En estas emulaciones se utilizarán las opciones especificadas originalmente en el archivo “Makefile” del proyecto. Dichas opciones implican que se utilice el archivo “100” procedente de la carpeta “data” del proyecto “kmeans”, así como el archivo de comprobación “100.gold_c10” ubicado en la misma carpeta. El número de *clusters* a obtener en la clasificación será 10.

Al finalizar la ejecución de la emulación *software*, se obtiene la información expuesta en el Código 67. Dicha información indica que la clasificación resultante de los elementos de entrada concuerda con la especificada en el archivo *GoldenFile*. Además, se muestran las prestaciones temporales obtenidas en distintos procesos de la arquitectura emulada en *software*. De manera análoga, en el Código 68 se muestran los resultados obtenidos en la emulación *hardware*.

```
Number of iterations : 9
PASSED:Based on Golden File: Points membership with Match Rate 100.0000 and
mismatches only 0 compare to GoldenFile.
-----
Performance Summary
-----
Device Initialization      :    1581.2886 ms
Buffer Allocation         :         2.2797 ms
-----
Compute Memberships       :    1674.2755 ms
Update Delta              :         0.1491 ms
Update Centers            :         0.5196 ms
Update Clusters           :         0.4412 ms
Total K-Means Compute Time :    1765.3131 ms
-----
```

Código 67: Resultados de ejecución de la emulación software

```

Number of iterations : 9
PASSED:Based on Golden File: Points membership with Match Rate 100.0000 and
mismatches only 0 compare to GoldenFile.
-----
Performance Summary
-----
Device Initialization      :    4544.7967 ms
Buffer Allocation         :     65.3061 ms
-----
Compute Memberships       :   121835.3185 ms
Update Delta              :      0.1767 ms
Update Centers            :      0.5086 ms
Update Clusters           :      0.6321 ms
Total K-Means Compute Time :  121926.1932 ms
-----
Number of iteration(s)   : 1

```

Código 68: Resultados de ejecución de la emulación hardware

7.3. Ejecución sobre acelerador hardware Xilinx ZCU102

Tras finalizar la ejecución de la aplicación utilizando mecanismos de emulación, especialmente en el caso de la emulación hardware utilizando la plataforma virtual definida, se procedió a la implementación en la placa de prototipado del proyecto. Para ello, se ejecuta en primer lugar la opción “check”, especificando “hw” como “TARGET”. Como resultado de esta ejecución, se crea la imagen “sd_card.img” dentro de la carpeta “package.hw”. Esta imagen se conforma a partir del kernel de Linux, los programas de *booting*, la aplicación “kmeans” y los ficheros de datos de entrada. Todos los archivos se vuelcan la tarjeta SD como archivo de imagen “sd_card.img”. Esta transferencia se efectúa utilizando el comando “dd” de Linux, como se indica en el Código 69. El directorio “/dev/sdb” representa el dispositivo de la tarjeta SD conectada al ordenador.

```
host% sudo dd if=/dev/sdb of=$HOME/VAEm2/demo/kmean s/package.hw/sd_card.img
```

Código 69: Comando para transferencia de los datos para la ejecución del proyecto a la tarjeta SD

A continuación, se configura el arranque de la placa de prototipado para que utilice la tarjeta SD. Esta configuración se realizará con la placa desactivada y colocando los *switches* del componente SW6 en la situación expuesta en la Figura 85 (SW1=1, resto a 0).



Figura 85: Configuración del switch SW6 para arranque desde la tarjeta SD

El proceso de arranque desde la SD se señala mediante el LED Init B en color rojo. Asimismo, la finalización del arranque de la placa se indicará con este LED puesto en verde, como se muestra en la Figura 86.

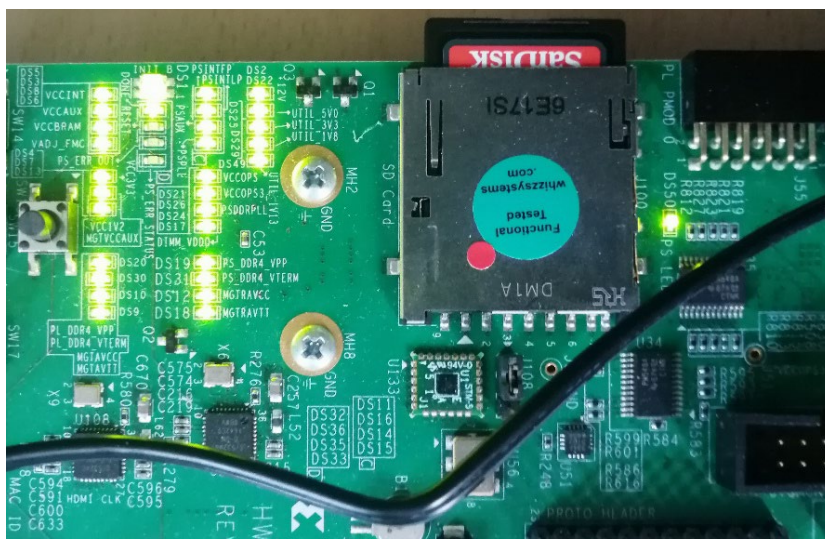


Figura 86: Señalización de la placa de finalización del proceso de arranque

Como se mencionó con anterioridad, las placas de prototipado MPSoC Zynq UltraScale+, entre las que se incluye el modelo utilizado en este trabajo, contienen una serie de microprocesadores que permiten la ejecución de Linux. Este hecho implica también la implementación en la propia placa de una interfaz gráfica de usuario (GUI) usada en este TFG para invocar la ejecución del proyecto en la propia placa. La visualización de esta interfaz se realiza conectando una pantalla al puerto “DisplayPort” de la placa. Por otra parte, se conecta a la placa un teclado inalámbrico que incorpora *touchpad*, utilizando el puerto físico USB 2.0 ULPI PHY provisto en la propia placa, aunque con un adaptador a puerto físico USB estándar. Cabe destacar que, para que la placa detectase el teclado, fue necesario activar el correspondiente *jumper* (Figura 87). Dicha configuración habilita que el puerto físico tenga soporte OTG (*On The Go*), posibilitando la conexión a nivel *software* de periféricos hacia la placa. De esta manera, se establece una conexión directa del teclado con la placa.

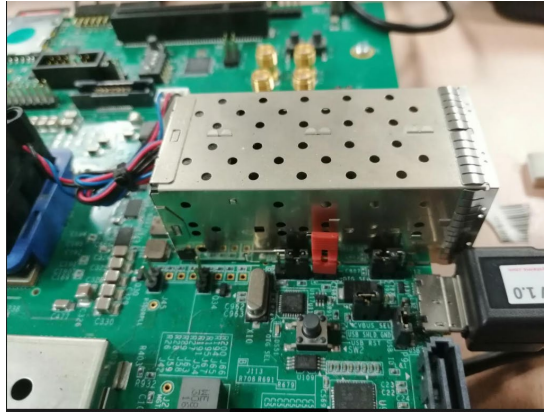


Figura 87: Habilitación de conexión USB para utilización OTG

Una vez configurada la GUI con la placa, se efectúa la ejecución del proyecto. Para ello se abre un terminal y se ejecuta el *script* “run_app.sh” desde el sistema de ficheros. Dicho *script* forma parte del conjunto de archivos que componen la imagen “sd_card.img” previamente descargada y generada como resultado de la ejecución de la opción “check” del archivo “Makefile”. El código descrito en este *script* ejecuta la aplicación “kmeans”, obteniéndose los resultados mostrados en el Código 70 si se utilizan los mismos archivos con datos de entrada a la aplicación que en la ejecución de las emulaciones, además de la misma parametrización. En la Figura 88 se muestra la configuración final del sistema físico, conformado por la placa y los dispositivos periféricos, en el que se ejecuta la aplicación, además de un ejemplo de dicha ejecución. Por otra parte, también se expone en la pantalla izquierda el diagrama de bloques que constituye la aplicación implementada en el MPSoC.

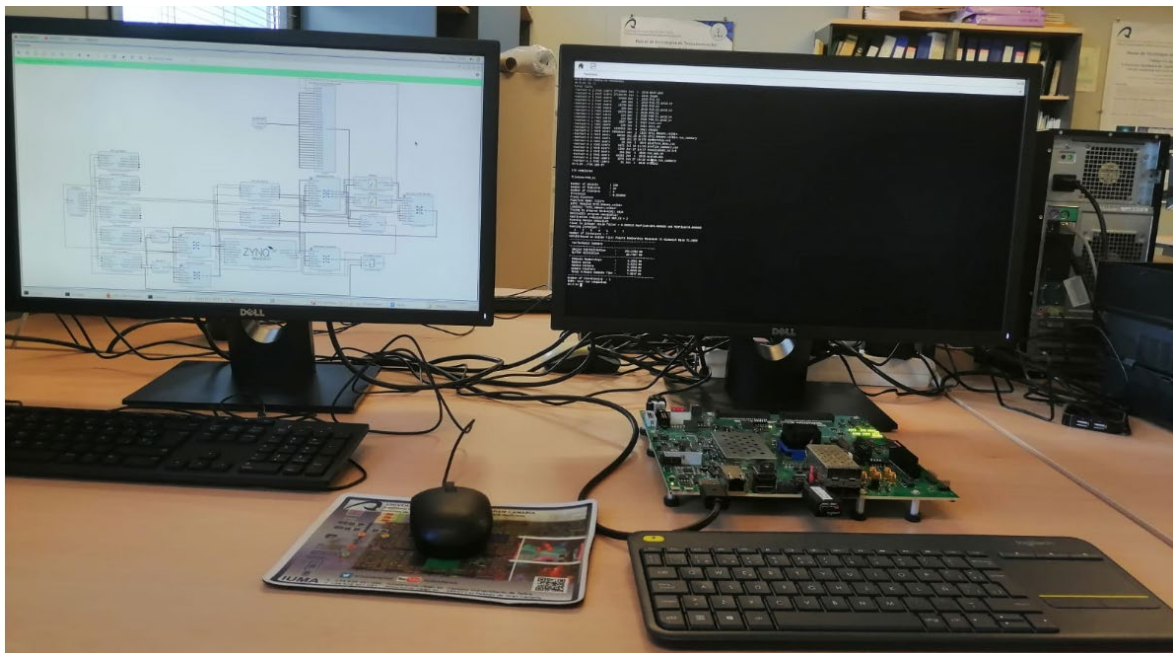


Figura 88: Sistema implementado

```
I/O completed
fileName=100
Number of objects      : 100
Number of features    : 34
Number of clusters    : 10
Threshold              : 0.001000
Found Platform
Platform Name: Xilinx
INFO: Reading krnl_kmeans.xclbin
Loading: 'krnl_kmeans.xclbin'
Trying to program device[0]: edge
Device[0]: program successful!
Application compiled with NUM_CU = 2
Running device execution
Float to integer scale factor = 0.008979 MaxFloat=150638.000000 and
MinFloat=0.000000
Running iteration :
   1   2   3   4   5   6   7   8   9
Number of iterations : 9
PASSED:Based on Golden File: Points membership with Match Rate 100.0000 and
mismatches only 0 compare to GoldenFile.
-----
Performance Summary
-----
Device Initialization      :      192.0474 ms
Buffer Allocation         :      13.5111 ms
-----
Compute Memberships       :      7.2658 ms
Update Delta              :      0.0296 ms
Update Centers            :      0.3894 ms
Update Clusters           :      0.2520 ms
Total K-Means Compute Time :      9.3545 ms
-----
Number of iteration(s)   : 1
INFO: host run completed.
```

Código 70: Resultado ejecución con los archivos de entrada "100" y "100.gold_c10"

7.4. Clasificación de imágenes hiperespectrales

Con el objeto de demostrar que la aplicación funciona con datos reales, se ha probado con un conjunto de imágenes hiperespectrales. Una imagen hiperespectral es una imagen tridimensional, con dos dimensiones espaciales y una espectral [36]. En esta última dimensión se distribuyen tantas longitudes de ondas como el sensor sea capaz de capturar. A diferencia de una RGB que captura tres longitudes, un píxel de la imagen hiperespectral contiene múltiples longitudes de ondas formando una firma espectral, según se evidencia en la Figura 89.

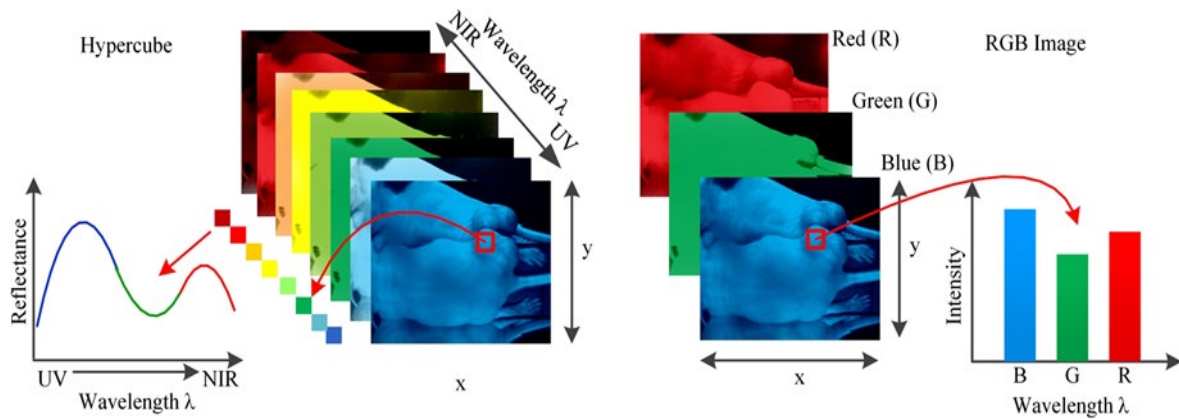


Figura 89: Representación de imagen hiperespectral (izquierda) y de imagen RGB (derecha) [36]

El conjunto de imágenes utilizado está constituido por 10 imágenes de cáncer de piel con una resolución espacial de 50 x 50 píxeles y una cantidad de 115 bandas espectrales que cubren el espectro visible e infrarrojo cercano (VNIR) desde 450 a 950 nm. En la Figura 90 se muestran las imágenes RGB obtenidas con un dermoscopio digital, las cuales corresponden con las imágenes hiperespectrales empleadas en este trabajo.

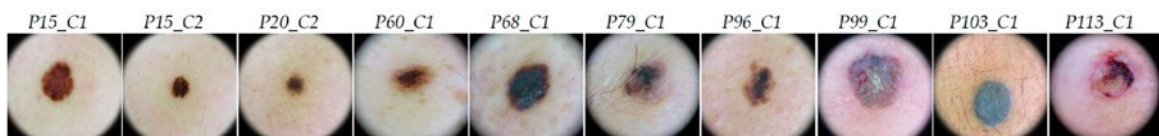


Figura 90: Imágenes RGB de cáncer de piel obtenidas con un dermoscopio digital.

Para la adecuación de estas imágenes a las especificaciones de entrada del algoritmo, fue necesario transformar dichas imágenes, dotándolas de un formato que se adapte al funcionamiento de la aplicación. En primer lugar, se realiza un escalado espacial de las imágenes aplicando un factor de escala de 0.2, por lo que se reduce el tamaño original de las imágenes de 50 x 50 a 10 x 10 píxeles. Esta acción se realiza con la función de Matlab *imresize*, sobre el conjunto de imágenes hiperespectrales `preProcessedImage`. En el Código 71 se detalla esta acción.

```
preProcessedImage = imresize(preProcessedImage,0.2);
```

Código 71: Escalado espacial de las imágenes en Matlab

Las imágenes utilizadas en la aplicación "kmeans" contienen 28 bandas espectrales, por lo que se considera un espectro reducido. Con el objetivo de lograr que las bandas espectrales seleccionadas resulten lo más representativas posibles, durante el proceso de reducción se han tomado una cada cuatro bandas, de manera que el rango espectral recorrido permanezca siendo el mismo. Esta modificación se ejecuta en Matlab mediante la sentencia descrita en el Código 72 y efectuada sobre el conjunto de imágenes `preProcessedImage` indicado. Cada imagen posee una

organización matricial tridimensional, siendo las dos primeras las dimensiones espaciales de largo y ancho, mientras que la última representa la dimensión espectral.

```
preProcessedImage = preProcessedImage(:, :, 1:4:end);
```

Código 72: Selección de bandas espectrales de la imagen

Dado que k-means opera únicamente con matrices bidimensionales, se requiere la conversión del cubo hiperespectral que constituye cada imagen a dos dimensiones. Para lograr este redimensionamiento, se utiliza la ejecución en Matlab expuesta en el Código 73. Con esta última acción, se finaliza el proceso de obtención de los datos a clasificar en k-means a partir de las imágenes hiperespectrales.

```
preProcessedImage = reshape(preProcessedImage, 10*10, 28);
```

Código 73: Redimensionamiento de las imágenes

Mediante esta transformación, cada píxel de la imagen se convierte en una fila dentro del archivo con las muestras a clasificar e introducidas en k-means. Este hecho equivale a decir, debido a la algorítmica de la aplicación, que cada píxel se convierte en una muestra que será clasificada. Por otra parte, las columnas del archivo con las muestras representan las bandas espectrales previamente escogidas. Estas columnas se corresponden en la ejecución de la aplicación con los atributos de las muestras a clasificar.

Para poder verificar que las ejecuciones a realizar ofrecen resultados correctos es necesario sustituir el fichero *GoldenFile* utilizado en ejecuciones anteriores. En su lugar, se emplea un archivo específico para cada ejecución con los resultados de clasificación obtenidos para el mismo caso, pero utilizando una aplicación alternativa. La generación de estos nuevos archivos se efectuó también en Matlab, ejecutando la sentencia indicada en el Código 74 donde numCluster representa la cantidad de *clusters* en los que se realiza la clasificación.

```
output = kmeans(preProcessedImage, numCluster);
```

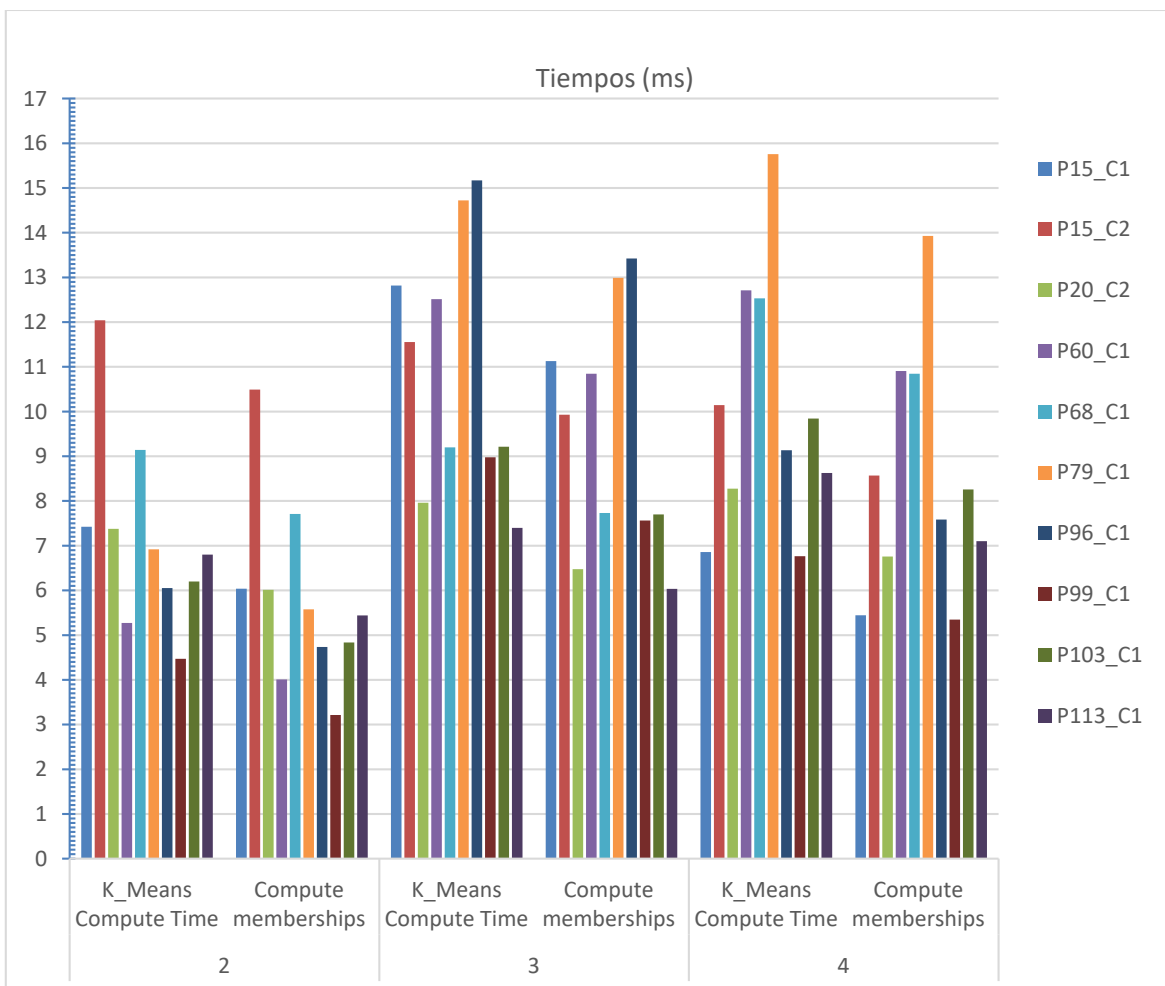
Código 74: Generación de GoldenFiles en Matlab

Para probar distintas opciones de *clustering*, se realizaron clasificaciones en 2, 3 y 4 *clusters* para cada uno de los nuevos *datasets* generados. Las variaciones más significativas entre las nuevas ejecuciones se observaron en el proceso de cálculo de las agrupaciones de las muestras (*Compute memberships*). Este hecho se manifiesta en el tiempo de ejecución total de la aplicación en la placa de prototipado (*K_Means Compute*), ya que este último engloba el proceso *Compute memberships*

además de otros procesos requeridos en la ejecución de la aplicación. Dichas variaciones se representan en la Gráfica 5. De manera análoga a los resultados de las ejecuciones anteriores de la aplicación, esta gráfica indica los resultados del tiempo requerido. Cabe destacar que además del número de muestras y la cantidad de atributos que dichas muestras contienen, las ejecuciones se realizan aplicando un umbral o threshold de 0.001, como se detalla en la Tabla 10.

Tabla 10: Características comunes a todos los casos

Parámetro	Valor
Nº de muestras	100
Nº de atributos	28
Umbral	0.001



Gráfica 5: Tiempos para el cálculo de las agrupaciones de las muestras y para la ejecución de la aplicación en placa

En la gráfica anterior se observa que, en términos generales, las ejecuciones de la aplicación para cada imagen se efectúan con mayor rapidez en una clasificación en 2 clusters, siendo la única excepción el caso de la imagen P15_C2. La razón de ello se encuentra en la firma espectral que

caracteriza los píxeles de las imágenes, es decir, en las longitudes de ondas dentro del espectro electromagnético escogidas para representar los píxeles. Desde la perspectiva de la firma espectral, la distinción más evidente entre los píxeles de las imágenes es de carácter binario según visualicen una región afectada o una región sin afección. Este aspecto también ocurre en la visualización en formato RGB de las imágenes expuestas en la Figura 90, pues el rango espectral VNIR coincide en su mayor parte con el rango del espectro visible. Esto implicaría que una clasificación en 2 *clusters* de los píxeles requiera de menos iteraciones que otras clasificaciones para una cantidad de *clusters* mayor, ya que en la opción binaria existirían menos píxeles de difícil clasificación. En consecuencia, se precisa de un menor tiempo para el cálculo de las agrupaciones de los píxeles y, por lo tanto, la ejecución de la aplicación finaliza antes.

Entre las opciones de clasificación de 3 y 4 *clusters* no aparece ninguna correlación relevante a los tiempos de ejecución que sea indistinta de la imagen considerada. De este aspecto se deduce que ambas opciones presentan grados de dificultad, así como de incertidumbre en su establecimiento muy similares debido a la existencia de píxeles de difícil clasificación.

Las variaciones obtenidas entre los tiempos de ejecución de la clusterización en función del número de *clusters* a establecer no se aproximan a ninguna relación de proporción que sea común a todas las imágenes. Esto implicaría que las características de cada caso de cáncer de piel real presentan una influencia significativa en los tiempos de ejecución de la aplicación, así como la adecuación de la inicialización de los *clusters* a cada caso.

Con el objetivo de obtener una representación visual de los resultados de clasificación tanto de la aplicación kmeans ejecutada en placa como para la ejecución en Matlab, se recurre a la generación de un mapa de segmentación por cada uno de dichos resultados. Para ello, se ejecutan en Matlab las sentencias indicadas en el Código 75, donde output se refiere al resultado de clusterización que se pretende mapear. La organización bidimensional de los mapas generados se corresponde con las dimensiones espaciales de las imágenes de la Figura 90. Asimismo, el eje de ordenadas de los mapas de segmentación equivale al largo de la imagen evaluada, mientras que el eje de abscisas constituye su ancho. La representación de las agrupaciones en *cluster* de los puntos del mapa de segmentación se indica asignando a cada *cluster* un color aleatorio distinto. En la Tabla 11 y la Tabla 12 se exponen los mapas de segmentación resultantes de las ejecuciones de kmeans.

```
img = reshape(output,10,10,1);  
imagesc(img);
```

Código 75: Generación de los mapas de segmentación

Tabla 11: Mapas de segmentación obtenidos en cada ejecución de kmeans (I). Los colores del mapa son aleatorios

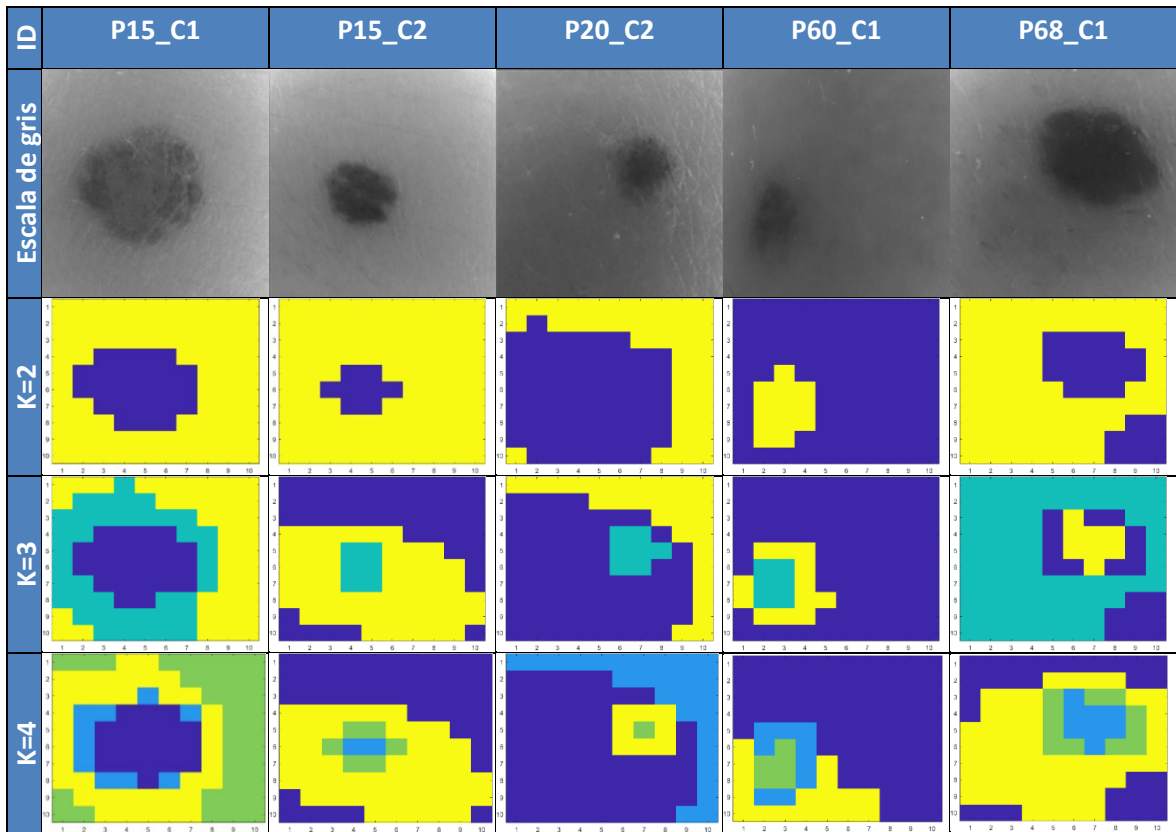
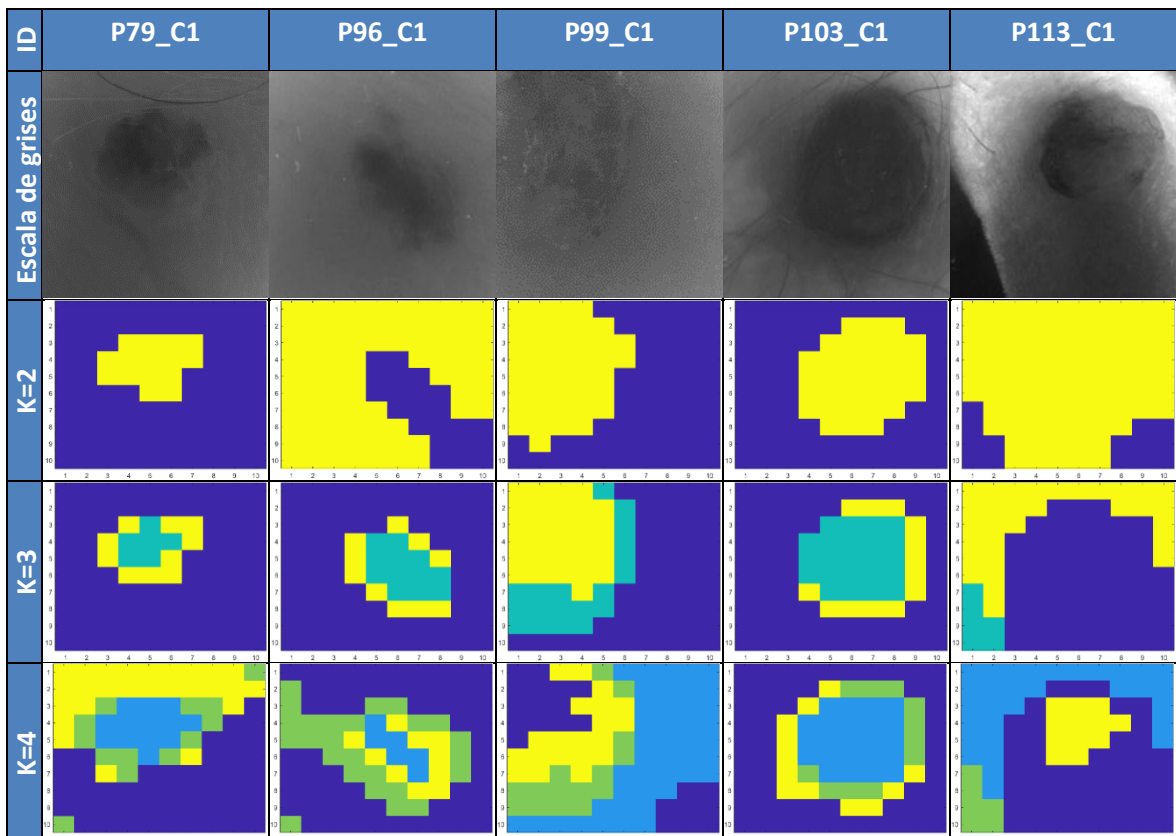


Tabla 12: Mapas de segmentación obtenidos en cada ejecución de kmeans (II). Los colores del mapa son aleatorios



A la vista de estos resultados, se comprueba que la aplicación kmeans presenta una buena capacidad a la hora de delimitar la región afectada. Sin embargo, es cierto que se observan algunas discrepancias entre lo observado en la escala de grises y los resultados de la clasificación, como es el caso de la clasificación en 2 clusters de los píxeles de la imagen P113_C1. En este caso particular, la razón de ello se justifica con las malas condiciones en las que fue tomada la captura, la cual presenta una región muy oscura debido a que la lente no estaba en completo contacto con la piel, lo que impide la correcta segmentación de la imagen. Dicha problemática sería fácilmente evitable asegurando que la captura se realice en mejores condiciones y procurando que la iluminación de dicho tejido sea lo más uniforme posible. Pese a esta inconveniencia, la aplicación kmeans ha sido capaz de sortearla, ya que con un ligero incremento del número de clusters se consigue delimitar en el mapa de segmentación resultante la región de la piel afectada.

Por otra parte, en la mayor parte de las clasificaciones para más de 2 *clusters* se encuentran algunas agrupaciones que bordean las regiones más afectadas, siendo dichas regiones abarcadas por un *cluster* distinto. De esta manera, se manifiesta una distribución concéntrica de las agrupaciones alrededor del centro de la región visiblemente afectada, algo esperable teniendo en cuenta la naturaleza de la afección.

La comparativa entre los resultados obtenidos en la ejecución de kmeans y Matlab se realiza visualmente por medio de sus respectivos mapas de segmentación. El motivo para ello se debe a las divergencias existentes entre las dos aplicaciones con respecto a la elección del centroide de los *clusters* iniciales. Sin embargo, ambos procesos de inicialización son válidos, ya que cumplen con las restricciones respecto a la inicialización de los *clusters* establecidas en el arquetipo del algoritmo *k-means*. Como consecuencia de estas diferencias, las dos aplicaciones pueden generar resultados distintos entre sí desde el punto de vista de la identificación numérica otorgada a cada *cluster* y, en menor medida, del contenido de los mismos. Este último caso ocurre especialmente cuando la delimitación de los *clusters* prefijada se encuentra más difuminada. No obstante, las distribuciones de las clasificaciones serán visualmente similares.

De esta manera, podrán contrastarse los resultados de la segmentación realizando una comparativa visual de estos con los resultados obtenidos en la ejecución de Matlab, representados también de la misma manera. En las tablas desde la Tabla 13 hasta la Tabla 15 se exponen algunas comparativas entre algunas de las ejecuciones realizadas donde los colores de los clusters no indican ninguna asociación, ya que son elegidos aleatoriamente.

Tabla 13: Ejecución de la clasificación para 4 clusters en placa y en Matlab para la imagen P113_C1. Los colores del mapa son aleatorios.

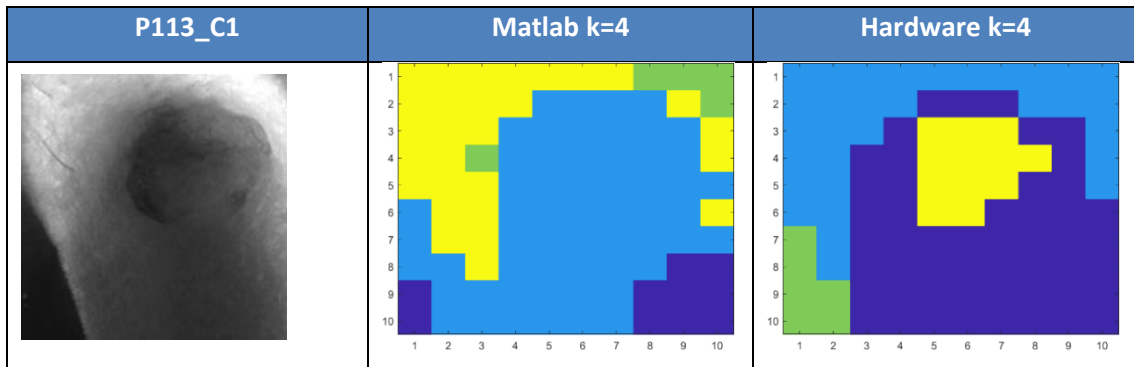


Tabla 14: Ejecución de la clasificación para 2 y 3 clusters en placa y en Matlab para las imágenes P15_C1 y P15_C2. Los colores del mapa son aleatorios.

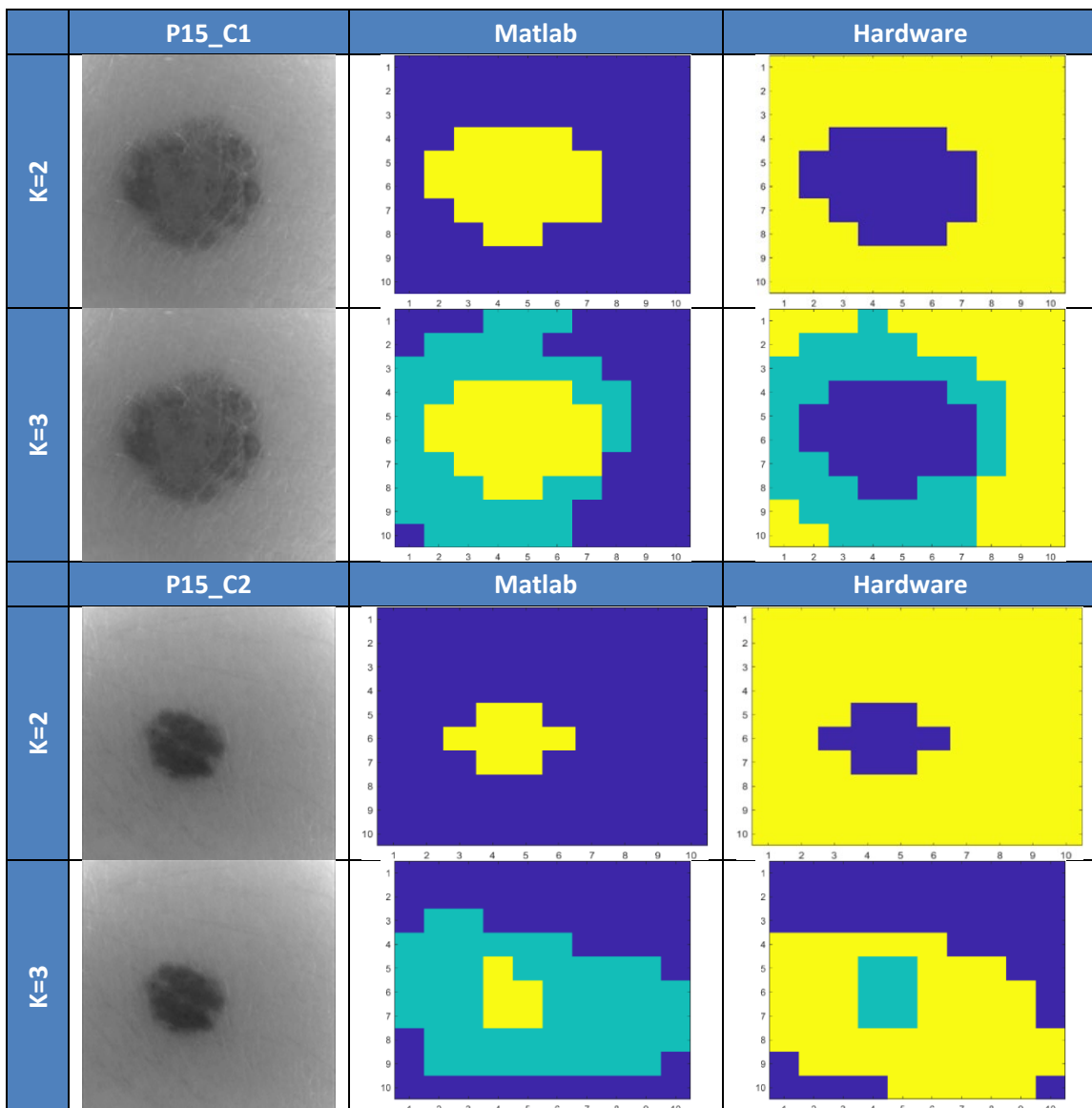
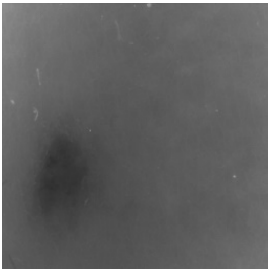
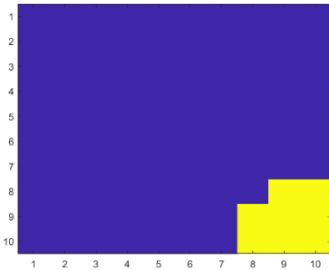
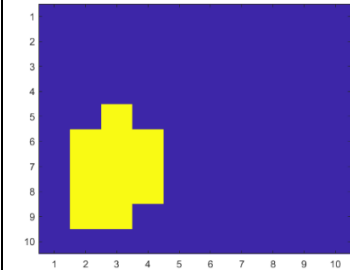
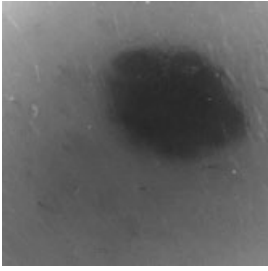
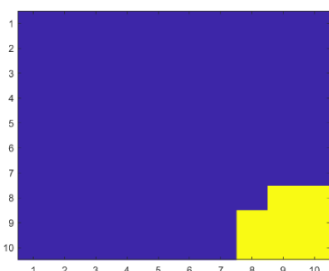
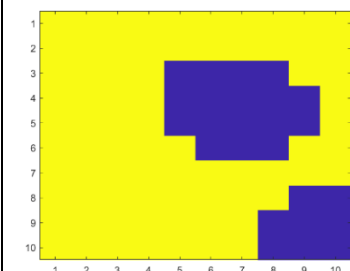
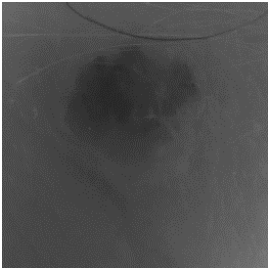
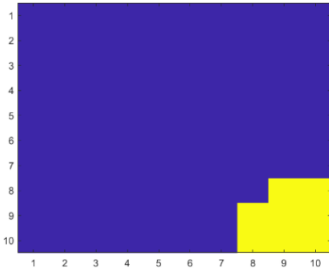
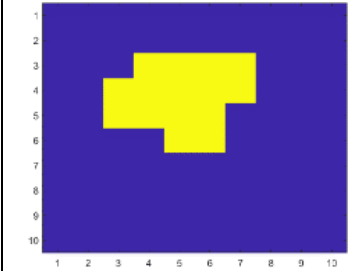
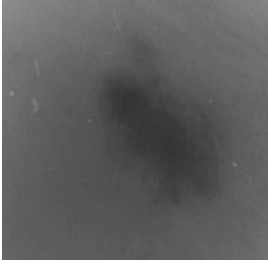
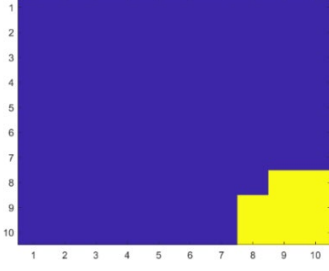
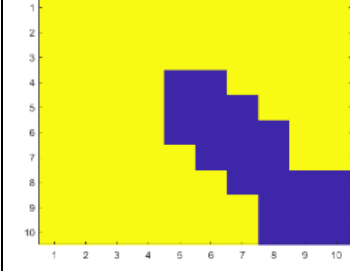
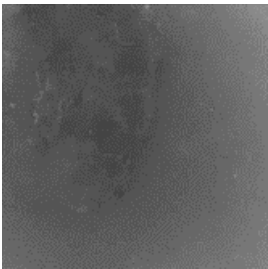
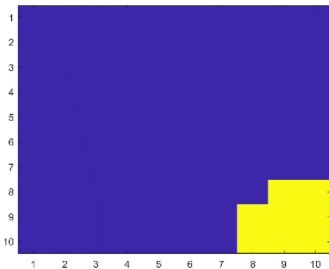
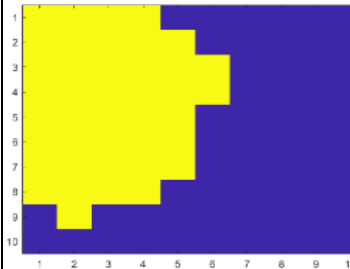
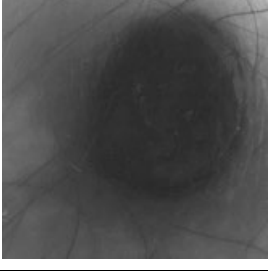
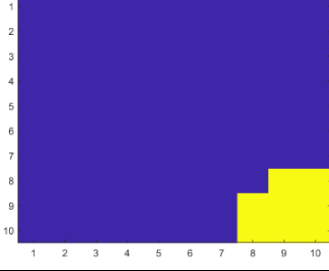
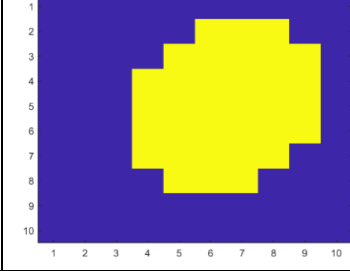


Tabla 15: Ejecuciones de la clasificación para 2 clusters en placa y en Matlab. Los colores del mapa son aleatorios.

ID	Escala de grises	Matlab k=2	Hardware k=2
P60_C1			
P68_C1			
P79_C1			
P96_C1			
P99_C1			
P103_C1			

Uno de los aspectos reseñables que se observa en las comparativas realizadas en las tablas anteriores se encuentra en la Tabla 13. En dicha tabla, se ve que la ejecución de Matlab no consigue delimitar claramente la lesión, algo que sí ocurre si se utiliza en su lugar la aplicación kmeans objetivo de este proyecto para una ejecución en 4 *clusters*.

En los mapas de segmentación de la Tabla 13 se observa que los resultados obtenidos tanto en Matlab como en la placa para la clasificación en 2 *clusters* de los píxeles de la imagen P15_C2 coinciden perfectamente (los colores de los mapas de segmentación son generados de manera aleatoria). Para la imagen P15_C1 la diferencia de resultados entre ambas ejecuciones es de un píxel. Al aumentar el número de *clusters* a 3 se obtienen también resultados similares en ambas ejecuciones.

Por otra parte, es destacable el efecto de la existencia de píxeles muertos en la esquina inferior derecha de las imágenes “P60_C1”, “P68_C1”, “P79_C1”, “P96_C1”, “P99_C1”, “P103_C1” y “P113_C1” como se muestra en la Tabla 15. Este hecho origina que al ejecutar el algoritmo de Matlab con un número de *clusters* igual a 2 se establezca la subdivisión entre estos píxeles y el resto de la imagen. Sin embargo, la aplicación kmeans con aceleración *hardware* en este proyecto identifica perfectamente la lesión, ignorando los píxeles muertos excepto en el caso de la imagen “P113_C1” debido a las condiciones de la imagen descritas con anterioridad. Esto sucede de igual manera usando otros números de *clusters*, por lo que puede concluirse que la implementación propuesta mejora los resultados.

7.5. Comparativa ejecución software y con acelerador hardware

Al analizar los resultados temporales de la ejecución de la aplicación implementada con aceleradores *hardware*, se observan mejoras significativas con respecto a los tiempos requeridos en una ejecución enteramente *software*. Asimismo, en la Tabla 16 se realiza una recopilación de los tiempos de ejecución resultantes para la versión en *software*, así como la versión asistida con aceleradores *hardware*, que permite esclarecer esta comparativa. Los resultados de la ejecución en *software* corresponden a la ejecución de la emulación *software* de la aplicación. En los dos casos expuestos en la Tabla 16, se utiliza el conjunto de elementos a clasificar que se especifica en el fichero “100”, y la clasificación se realiza en 10 *kernels*.

A excepción de la etapa “Buffer Allocation” estos resultados justifican la gran adecuación en la utilización de aceleradores *hardware*, puesto que con ellos se introduce un *speedup* de aproximadamente 189 en el tiempo de ejecución total “*K-means Compute Time*” con respecto a la

ejecución completa en *software*. El cálculo de este *speedup* “p” se indica en la ecuación (17), donde (“ t_{hw} ”) y (“ t_{sw} ”) representa el tiempo de ejecución para la aplicación con y sin acelerador *hardware*.

$$p = \frac{t_{sw}}{t_{hw}} = \frac{1765}{9.35} = 188.77 \cong 189 \quad (17)$$

Tabla 16: Comparativa entre la ejecución *software* y la ejecución con aceleradores *hardware*

	Buffer Allocation	Compute Membership	Update Delta	Update Centers	Update Clusters	K-means Compute Time
Emulación <i>software</i> (Sw)	2.28	1 674.28	0.1491	0.5196	0.4412	1 765.00
Emulación (EHw)	65.31	121 835.00	0.1767	0.5086	0.6321	121 926.00
Hardware (Hw)	13.51	7.27	0.0296	0.3894	0.252	9.35
Mejora (Hw/Sw)	0.17	230.30	5.04	1.33	1.75	188.77

Con respecto a las etapas, la reducción de tiempo más notoria se observa en la etapa de cálculo de la pertenencia de los elementos a cada *clusters*, referenciada como “Compute Membership”. Esto se debe a que concentra la mayor parte de la aritmética de la aplicación, (cálculo de las distancias entre elementos y centroides de *clusters*, recálculo de los centroides, etc), que ha sido enteramente traspasada a la parte PL, es decir, de aceleración *hardware* de la placa de prototipado. Dicha etapa ofrece además grandes oportunidades con respecto al grado de paralelización de sus operaciones, las cuales son relativamente sencillas. Asimismo, esta paralelización es mucho más explotable utilizando *hardware* programable, basado en varios módulos simples (CLBs, DSPs, etc) pero muy replicados en la plataforma física, que utilizando *software*, basado en módulos más complejos (microprocesadores), pero de los que se disponen menos réplicas debido a que su arquitectura física requiere mucha más lógica.

7.6. Conclusiones

En este capítulo se aborda la ejecución del algoritmo sobre un conjunto de imágenes hiperespectrales sobre la placa de prototipado ZCU102 utilizada en este proyecto. En una situación inicial las ejecuciones se realizaron con unos datos de entrada preestablecidos de la aplicación. Sin embargo, la aplicación se ejecutó posteriormente con *datasets* procedentes de un problema de clasificación real, que en este caso consistió en la detección de cáncer de piel y de su distribución en el tejido cutáneo. De esta manera, pudo demostrarse la utilidad de esta aplicación ejecutada con recursos de aceleración *hardware* en la placa de prototipado objetivo de este proyecto.

Capítulo 8. Conclusiones y líneas futuras

8.1. Conclusiones

A lo largo de este Trabajo Fin de Grado, se ha desarrollado el proceso de implementación de una aplicación de *machine learning* sobre la placa de prototipado MPSoC ZCU102 que incluye un dispositivo MPSoC Synq Ultrascale+. En dicho proceso se ha realizado un estudio de los distintos recursos presentes en la placa, abarcando los módulos de lógica programable (DSP, CLB, BRAM, etc), módulos que conforman el sistema de procesamiento (APUs, RTUs, GPUs, etc), módulos para el arranque y gestión de los distintos recursos de la placa (PMU y CSU) elementos de intercomunicación como, por ejemplo, las interfaces AXI y los diversos puertos periféricos de la placa.

Posteriormente, se estudiaron distintas alternativas de algoritmos de *machine learning* como KNN, clasificadores polinómicos, redes neuronales y algoritmos de *clustering*. En esta última categoría, se ha explorado el algoritmo *k-means*, siendo la opción escogida. Tras efectuar la elección, se ha estudiado la implementación *hardware/software* de una aplicación de *machine learning* que se correspondiese a dicha categoría. La arquitectura de esta aplicación se organiza en un sistema heterogéneo que se subdivide en funcionalidad *software*, introducida en el sistema de procesamiento de la placa, y aceleración *hardware*, implementada en la lógica programable de la misma. La programación de esta aplicación se realiza en C/C++.

Tras evaluar el código de referencia, se procede a su implementación y a la obtención de un conjunto de informes en los que se analizan aspectos de la implementación en el dispositivo de la aplicación relativos a tiempo de ejecución en los diversos módulos funcionales de la arquitectura y los recursos de lógica programable requeridos.

Como paso previo a la ejecución en placa, se han realizado distintas pruebas soportando tanto la emulación *software* en la que la arquitectura ARM Cortex A53 se ha emulado sobre QEMU, ejecutando la aplicación con objetivos funcionales. Asimismo, se ha creado una plataforma virtual donde se utiliza igualmente el emulador QEMU para la arquitectura ARM Cortex A53 y un simulador digital para simular los aceleradores *hardware*. Este proceso se conoce como emulación *hardware*.

Sobre dicha plataforma es posible ejecutar la aplicación, con las que se realiza una verificación más detallada de la funcionalidad del diseño. Finalmente, se realiza la ejecución sobre el prototipo *hardware* con distintos conjuntos de datos de entrada, evaluando así las prestaciones de la aplicación. Finalmente, se evalúan distintos casos y se concluye que la solución *hardware/software* mejora la latencia de ejecución 189 veces con respecto a la solución *software* funcionando a una frecuencia de 150 MHz.

8.2. Líneas futuras

Como continuación de la labor realizada en este TFG, se plantean las siguientes líneas futuras para la mejora de la implementación de *machine learning* en plataformas MPSoC Zynq UltraScale+.

1. Estudiar los efectos del ajuste de determinados parámetros del algoritmo (desenrollado de determinados bucles, por ejemplo) sobre el consumo de recursos, sobre la frecuencia de funcionamiento y potencia consumida.
2. Dotar a la aplicación “kmeans” de la capacidad de establecer dinámicamente la cantidad de agrupaciones más adecuada. La adición de esta funcionalidad presentaría la desventaja de aumentar el tiempo de ejecución, así como el consumo de recursos *hardware* de la aplicación. No obstante, también mejoraría la autonomía de la aplicación, ya que esta podría prescindir de una especificación explícita del número de *clusters* a obtener. Este aspecto sería especialmente útil en la integración de la aplicación como submódulo de un sistema más complejo.
3. Estudiar la viabilidad de modificar de forma dinámica el número de *kernels* de procesamiento, explotando la reconfiguración dinámica del dispositivo.
4. Dotar a la aplicación de capacidad de gestionar la existencia de atributos desconocidos. Este hecho, le otorgaría además una doble utilidad, pues aparte de su función en la detección de agrupaciones implícitas, podría estimar valores de atributos desconocidos en determinados elementos a partir del subconjunto de elementos pertenecientes al mismo *cluster*. Este aspecto aumentaría el grado de certeza de la estimación en comparación a si se considerase la totalidad de los elementos a clasificar. Asimismo, se lograría una funcionalidad de preprocesado de valores que podría complementar a otras aplicaciones.
5. Aplicar la metodología utilizada para probar algoritmos de *machine learning* alternativos. De esta manera, podrá realizarse una comparativa entre las prestaciones de las distintas

arquitecturas resultantes en términos temporales y de consumo de recursos, además de evaluar la eficacia predictiva de los diferentes algoritmos.

6. Mejorar los procesos de prototipado, de tal forma que se puedan transferir los datos y aplicaciones con mecanismos más eficientes que la utilización de medios físicos (SD Card), creando una infraestructura para el desarrollo de otros algoritmos (Infrastructure-as-a-Service – IaaS).

Bibliografía

- [1] BBVA, «“Machine learning”: ¿qué es y cómo funciona?» [Online]. Disponible en: <https://www.bbva.com/es/machine-learning-que-es-y-como-funciona/>. [Accedido: 01-feb-2020]
- [2] FUNIBLOGS, «Cinco ejemplos de la vida cotidiana donde encontrar inteligencia artificial - Funiber Blogs - FUNIBER», 2018. [Online]. Disponible en: <https://blogs.funiber.org/tecnologias-informacion/2018/09/24/funiber-cinco-ejemplos-vida-cotidiana-donde-encontrar-inteligencia-artificial>. [Accedido: 13-ago-2020]
- [3] Xilinx Inc, «Zynq UltraScale+ MPSoC Data Sheet: Overview (DS891)», 2019 [Online]. Disponible en: www.xilinx.com/support/documentation/data_sheets/ds891-zynq-ultrascale-plus-overview.pdf. [Accedido: 14-feb-2020]
- [4] J. A. Belloch, G. León, J. M. Badía, A. Lindoso, y E. San Millan, «Evaluating the computational performance of the Xilinx Ultrascale+ EG Heterogeneous MPSoC», *Journal of Supercomputing*, 2020 [Online]. Disponible en: <https://doi.org/10.1007/s11227-020-03342-7>
- [5] A. Wicaksana, C. M. Tang, y M. S. Ng, «A scalable and configurable Multiprocessor System-on-Chip (MPSoC) virtual platform for hardware and software co-design and co-verification», en *CONMEDIA 2015 - International Conference on New Media 2015*, 2016.
- [6] Towards Data Science, «Introduction to Machine Learning for Beginners». [Online]. Disponible en: <https://towardsdatascience.com/introduction-to-machine-learning-for-beginners-eed6024fdb08>. [Accedido: 24-abr-2020]
- [7] Tutorial And Example, «Hill Climbing Algorithm in AI». [Online]. Disponible en: <https://www.tutorialandexample.com/hill-climbing-algorithm/>. [Accedido: 07-dic-2020]
- [8] M. Kubat, *An Introduction to Machine Learning*. Springer International Publishing, 2015 [Online]. Disponible en: <https://www.springer.com/gp/book/9783319348865>. [Accedido: 26-abr-2020]

- [9] Superprof Diccionario, «Que significa campana de gauss en Matemáticas». [Online]. Disponible en: <https://www.superprof.es/diccionario/matematicas/probabilidades/campana-gauss.html>. [Accedido: 18-jun-2020]
- [10] L. Crockett, D. Northcote, C. Ramsay, F. Robinson, y B. Stewart, *Zynq[®] MPSoC With PYNQ and Machine Learning Applications*. Glasgow, Scotland, UK: University of Strathclyde Glasgow, 2019 [Online]. Disponible en: <https://www.zynq-mpsoc-book.com/>. [Accedido: 16-feb-2020]
- [11] National Instruments, «Las Ventajas de los Dispositivos FPGA de la Serie 7 de Xilinx». [Online]. Disponible en: <https://www.ni.com/es-es/innovations/white-papers/13/advantages-of-xilinx-7-series-fpga-and-soc-devices.html#section--1669561456>. [Accedido: 10-abr-2020]
- [12] Xilinx Inc, «Zynq Migration Guide Zynq-7000 SoC to Zynq UltraScale+ MPSoC Devices» [Online]. Disponible en: www.xilinx.com/support/documentation/user_guides/ug1213-zynq-migration-guide.pdf. [Accedido: 10-abr-2020]
- [13] Taiwan Semiconductor Manufacturing Company, «16/12nm Technology». [Online]. Disponible en: <https://www.tsmc.com/english/dedicatedFoundry/technology/16nm.htm>. [Accedido: 10-abr-2020]
- [14] Xilinx Inc, «ZCU102 Evaluation Board User Guide», 2019 [Online]. Disponible en: www.xilinx.com/boards_and_kits/zcu102/ug1182-zcu102-eval-bd.pdf. [Accedido: 22-feb-2020]
- [15] Xilinx Inc, «Zynq UltraScale+ Device Technical Reference Manual», 2020 [Online]. Disponible en: https://www.xilinx.com/support/documentation/user_guides/ug1085-zynq-ultrascale-trm.pdf. [Accedido: 02-mar-2020]
- [16] Xilinx Inc, «Zynq UltraScale+ MPSoC Embedded Design Methodology Guide UG1228 (v1.0)», 2017 [Online]. Disponible en: www.xilinx.com/support/documentation/sw_manuals/ug1228-ultrafast-embedded-design-methodology-guide.pdf. [Accedido: 23-feb-2020]
- [17] «Introduction to the Zynq Triple Timer Counter Part... - Community Forums». [Online]. Disponible en: <https://forums.xilinx.com/t5/Xcell-Daily-Blog-Archived/Introduction-to-the-Zynq-Triple-Timer-Counter-Part-One-Adam/ba-p/407537>. [Accedido: 22-feb-2020]

- [18] Real Digital, «ARM's global timer and ZYNQ's triple timer counter module». [Online]. Disponible en: <https://www.realdigital.org/doc/8cfcb505117c049c18d31fa4287285ef>. [Accedido: 22-feb-2020]
- [19] J. A. Castillo, «Qué es la memoria caché L1, L2 y L3 y cómo funciona», 2019. [Online]. Disponible en: <https://www.profesionalreview.com/2019/05/02/memoria-cache-l1-l2-y-l3/>. [Accedido: 22-feb-2020]
- [20] Xilinx Inc, «Zynq UltraScale+ Device Technical Reference Manual» [Online]. Disponible en: www.xilinx.com/support/documentation/user_guides/ug1085-zynq-ultrascale-trm.pdf. [Accedido: 02-mar-2020]
- [21] Arm, «ARM Information Center», en *ARM11 processors*, [Online]. Disponible en: <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0338g/Chdhbjjb.html>. [Accedido: 23-feb-2020]
- [22] Xilinx Inc, «Zynq UltraScale+ MPSoC: Software Developers Guide (UG1137)» [Online]. Disponible en: www.xilinx.com/support/documentation/user_guides/ug1137-zynq-ultrascale-mpsoc-swdev.pdf. [Accedido: 14-mar-2020]
- [23] Xilinx Inc, «Zynq UltraScale+ MPSoC: Embedded Design Tutorial A Hands-On Guide to Effective Embedded System Design», 2019 [Online]. Disponible en: www.xilinx.com/support/documentation/sw_manuials/xilinx2019_1/ug1209-embedded-design-tutorial.pdf. [Accedido: 27-mar-2020]
- [24] Xilinx Inc, «Zynq UltraScale+ MPSoC Product Tables and Product Selection Guide», 2016. .
- [25] Xilinx Inc, «UltraScale Architecture Configurable Logic Block», 2015 [Online]. Disponible en: https://www.xilinx.com/support/documentation/user_guides/ug574-ultrascale-clb.pdf. [Accedido: 25-feb-2020]
- [26] Xilinx Inc, «UltraScale Architecture SelectIO Resources User Guide» [Online]. Disponible en: www.xilinx.com/support/documentation/user_guides/ug571-ultrascale-selectio.pdf. [Accedido: 25-mar-2020]
- [27] Xilinx Inc, «Vitis Unified Software Platform Documentation Application Acceleration Development», 2020 [Online]. Disponible en: www.xilinx.com. [Accedido: 15-dic-2020]
- [28] GitHub, «Vitis_Accel_Examples/demo/kmeans at master Xilinx/Vitis_Accel_Examples». [Online]. Disponible en:

- https://github.com/Xilinx/Vitis_Accel_Examples/tree/master/demo/kmeans. [Accedido: 05-abr-2020]
- [29] B. R. Gaster y L. Howes, «OpenCL C++ Wrapper API», 2013 [Online]. Disponible en: <https://www.khronos.org/registry/OpenCL/specs/opencv-cplusplus-1.2.pdf>. [Accedido: 12-jul-2020]
- [30] Xilinx Inc, «HLS Pragmas», 2020 [Online]. Disponible en: https://www.xilinx.com/html_docs/xilinx2019_2/vitis_doc/Chunk1852886277.html#jit1504034365862. [Accedido: 15-jul-2020]
- [31] «cppreference.com». [Online]. Disponible en: https://es.cppreference.com/w/cpp/chrono/high_resolution_clock. [Accedido: 20-jul-2020]
- [32] «SDx Command and Utility Reference Guide» [Online]. Disponible en: www.xilinx.com. [Accedido: 06-sep-2020]
- [33] Xilinx Inc, «AXI Verification IP (VIP)». [Online]. Disponible en: <https://www.xilinx.com/products/intellectual-property/axi-vip.html>. [Accedido: 13-dic-2020]
- [34] «Vivado Design Suite User Guide Design Analysis and Closure Techniques», 2019 [Online]. Disponible en: www.xilinx.com. [Accedido: 11-sep-2020]
- [35] «KVM: virtualización nativa en Linux | Todos hacemos TIC». [Online]. Disponible en: <https://diocesanos.es/blogs/equipotic/2015/08/26/kvm-virtualizacion-nativa-en-linux/>. [Accedido: 08-nov-2020]
- [36] G. Lu y B. Fei, «Medical hyperspectral imaging: a review», *Journal of Biomedical Optics*, vol. 19, n.º 1, p. 010901, ene. 2014 [Online]. Disponible en: <https://www.spiedigitallibrary.org/terms-of-use>. [Accedido: 30-nov-2020]

Presupuesto

Presupuesto

En este apartado se pretende realizar una estimación de los recursos utilizados en la implementación y documentación del sistema descrito por este proyecto. Se generan una serie de costes que serán definidos en este apartado según su naturaleza. De esta manera, en la Tabla 17 se indican los gastos relativos a recursos *hardware*, mientras que en la Tabla 18 y en la Tabla 19 se exponen los gastos propios de los recursos *software* y los recursos humanos.

1.1. Costes recursos hardware

En este apartado se definen los recursos *hardware* utilizados en este Trabajo Fin de Grado. En la Tabla 17 se indica el equipamiento requerido. Cabe destacar que los costes de estos recursos para el proyecto se han determinado como un porcentaje del precio real de los mismos, siendo este precio el especificado en la columna “Coste del recurso (€)” de la Tabla 17. El porcentaje considerado se expone en la columna “% Amortización”, siendo equivalente al porcentaje de la vida útil del recurso que se presupone que se precisará para el desarrollo del proyecto. De esta manera, se obtienen los costes indicados en la columna “Coste de utilización (€)”, que serán los considerados en este presupuesto.

Tabla 17: Presupuesto de los recursos hardware

Recurso <i>hardware</i>	Coste del recurso (€)	% Amortización	Coste de utilización (€)
Estación de trabajo con sistema operativo Linux RedHat 7	2 500.00	6.7	167.50
MPSoC Zynq UltraScale+ ZCU102	2 106.84	6.7	141.16
Ordenador con sistema operativo Windows 10	800.00	6.7	53.60
Teclado con touchpad. Logitech K400.	24.00	6.7	1.61
Pantalla marca Dell 22 pulgadas modelo P2217	125.00	6.7	8.37
Tarjeta SD Marca Kingston SDC10	30.00	6.7	2.01
Total hardware			374.25

1.2. Costes recursos Software

Los recursos *software* se indican en la Tabla 18. Como coste de mantenimiento se ha establecido el coste de la licencia anual.

Tabla 18: Presupuesto de los recursos software

Recursos <i>software</i>	Tipo de licencia	Importe de licencia	Importe de mantenimiento (€)
Microsoft Office 2016	Universitaria	14.00	14.00
Vitis IDE 2020.1	Universitaria	2 700.00	2 700.00
Total software (€)			2 714.00

1.3. Costes recursos humanos

Para poder estimar el coste humano definido en la realización de este trabajo se ha considerado el salario por hora de un ingeniero. En la Tabla 19 se indica el coste para el número de horas estimadas para la realización del proyecto.

Tabla 19: Presupuesto de los recursos humanos

Recursos humanos	Coste/h	horas	Coste total (€)
Ingeniero	12.00	300.00	3 600.00

1.4. Coste total

Además de los costes indicados, que pueden establecerse como directos, deben tenerse en cuenta una serie de costes considerados indirectos. Para poder cubrirlos se considera una cuantía adicional del **25%** con respecto al total de los costes directos. Adicionalmente, debe tenerse en cuenta la tasa de impuestos, para lo que se considera el actual I.G.I.C sobre bienes y servicios del 7%. De esta manera, se obtiene que el presupuesto total requerido para la realización de este proyecto alcanza la suma expuesta en la Tabla 20.

En este apartado se define el coste total del Trabajo Fin de Grado como suma de todos los gastos anteriores, incluyendo tanto costes directos como indirectos. Asimismo, los resultados obtenidos se muestran en la Tabla 20.

Tabla 20: Presupuesto total

	Importe costes directos (€)	Importe costes indirectos (€)
Recursos hardware	374.25	93.56
Recursos software	2 714.00	678.50
Recursos humanos	3 600.00	900.00
Importe costes(€)	6 688.25	1 672.06
Impuestos (7%)	468.17	117.04
Importe costes(€) con I.G.I.C.	7 156.42	1 789.10
Importe total(€)		8 945.52

Yo, Mario Daniel Guanche Hernández, declaro que el presupuesto de este proyecto asciende a ocho mil novecientos cuarenta y cinco euros y cincuenta y dos céntimos (8 945.52 €).

Las Palmas de Gran Canaria, 23 de diciembre del 2020

Fdo. Mario Daniel Guanche Hernández

Pliego de condiciones

Pliego de Condiciones

1.1. Introducción

En este apartado se definen las herramientas necesarias, tanto *hardware* como *software*, para la realización del Trabajo Fin de Grado. En los siguientes apartados se indica el equipamiento que se ha utilizado en la realización de este trabajo.

1.2. Recursos *Hardware*

Ordenador Dell Precision T1650.

Teclado con touchpad. Logitech K400.

Pantalla con conexión DisplayPort marca Dell 22 pulgadas modelo P2217

Tarjeta SD 16 GB. Marca Kingston SDC10/16GB

Kit de Evaluación Xilinx Zynq UltraScale+ MPSoC ZCU102.

1.3. Recursos *Software*

Sistema operativo Linux Red Hat 7.0

Sistema operativo Windows 10.

Vitis IDE 2020.1.

Microsoft Office Word 2016.