Implementación paramétrica altamente segmentada del algoritmo k-means para la clusterización de imágenes hiperespectrales sobre hardware reconfigurable

Borja Morcillo Salgado

Trabajo de Fin de Máster

Máster en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

Madrid, junio de 2022

Directores:

Daniel Báscones García
José Manuel Mendías Cuadros

Implementación paramétrica altamente segmentada del algoritmo k-means para la clusterización de imágenes hiperespectrales sobre hardware reconfigurable

Highly pipelined parametric implementation of the k-means algorithm for hyperspectral image clustering on reconfigurable hardware

---

Autor:

Borja Morcillo Salgado

Directores:

Daniel Báscones García

José Manuel Mendías Cuadros

Trabajo de Fin de Máster

Facultad de Informática

Universidad Complutense de Madrid

Convocatoria: Junio 2022

Calificación:

24 de junio de 2022

Autorización de difusión

El abajo firmante, matriculado en el Máster de Ingeniería Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: "Implementación paramétrica altamente segmentada del algoritmo k-means para la clusterización de imágenes hiperespectrales sobre hardware reconfigurable", realizado durante el curso académico 2021-2022 bajo la dirección de Daniel Báscones García y José Manuel Mendías Cuadros del Departamento de Arquitectura de Computadores y Automática, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Borja Morcillo Salgado

Madrid, a 24 de junio de 2022

Esta obra está bajo una
Licencia Creative Commons
Atribución-NoComercial-CompartirIgual 4.0 Internacional.

Agradecimientos

A ti lo primero, que te has dignado a abrir este cacho de tocho impresionante.

Aunque también a los que me han ayudado a escribirlo: Mendi y Carlos, que incluso sabiendo de antes lo cansino que soy, me han vuelto a ayudar. Lo de Carlos no lo digas por ahí, que debe permanecer en secreto.

Y más que por supuesto a Dani, quien me ha demostrado que los matemáticos son en realidad buena gente.

A los buenos chavales de clase, pues de no ser por ellos a ver quién hubiese aguantado en el máster...

A mi bar de confianza: "El Tropezón", en San Rafael. Y al Nobel pequeño, por meterme en la familia.

A los grupos emergentes que solo aspiran a divertirse y grabar maquetas para Capitán Demo. Y a Franco por presentarme a muchos de ellos.

A Chiqui y Ana, para que me mantengan hasta que pueda vivir de mis hijos. Y a Vera por dejarme en paz de una vez.

A Ana, que me sigue el ritmo en los conciertos.

A Mendi otra vez.

Índice general

Índice general I

Índice de figuras VII

Índice de tablas X

Resumen XI

Abstract XII

1. Introducción 2

1.1. Motivación . . . 2

1.2. Objetivos . . . 4

1.3. Antecedentes . . . 5

1.4. Plan de trabajo . . . 6

1.5. Organización de esta memoria . . . 9

2. Contexto tecnológico 11

2.1. Imágenes hiperespectrales . . . 11

2.1.1. Sensores . . . 12

2.2. FPGA . . . 14

2.2.1. Componentes básicos de una FPGA . . . 14

2.2.2. Xilinx VC709 . . . 16

2.3. Algoritmo k-means . . . 17

2.3.1. Pseudocódigo . . . 19

2.3.2. Aprendizaje automático . . . 20

2.3.3. Uso en procesado de imagen 20
2.4. Adaptación *hardware* de *k*-means 21
2.4.1. Distancia Manhattan 22
2.4.2. Aritmética y representación de valores 23
2.4.3. Sustitución de operaciones de división 24
2.4.4. Cálculo de valores acumulados 24
2.4.5. Operaciones multibanda 24
2.4.6. Inclusión de un módulo de control 25
2.4.7. Establecimiento de un sistema de parada 26
2.4.8. Realimentación rápida de centroides 26
2.4.9. Preprocesado de imágenes hiperespectrales (reducción dimensional) . 26
2.5. Desbordamiento 27
2.5.1. Saturación 28
2.5.2. Ajuste de anchura 28
2.6. Arquitectura en *pipeline* 29
2.7. Array sistólico 31
2.8. Bus AXI 31
2.8.1. AXI-Stream 32
2.9. Generación de valores aleatorios 33
2.9.1. LFSR 34
2.10. *IP soft cores* 35
2.10.1. *Divider Generator* 35
2.10.2. *Integrated Logic Analyzer* 36
2.11. Herramientas *software* 36
2.11.1. Implementación *software* del algoritmo 37
2.11.2. Descripción *hardware* 37
2.11.3. EDA y simulación *hardware* 38

2.11.4. Generación de documentación . . . 38
2.11.5. Gráficos vectoriales . . . 39
2.11.6. Control de versiones . . . 39
2.12. Herramientas *hardware* . . . 39

3. Arquitectura *hardware* **42**
3.1. Características y parámetros del diseño . . . 43
3.1.1. Parámetros manualmente configurables . . . 43
3.1.2. Parámetros autoconfigurables . . . 44
3.2. *Pipeline* y vista general del diseño . . . 45
3.2.1. Vista general . . . 46
3.2.2. Comunicación entre módulos . . . 48
3.2.3. Cese de operaciones . . . 49
3.2.4. Rendimiento del *pipeline* . . . 49
3.3. Módulo de clasificación (*Pipelined Classifier*) . . . 52
3.3.1. Array sistólico . . . 53
3.3.2. Mecanismos de inicialización pseudoaleatoria . . . 54
3.4. Módulo de acumuladores (*Acc Updater*) . . . 57
3.4.1. Actualizador de acumuladores y contadores (*Cluster Acc Updater*) . . . 59
3.5. Módulo de actualización de centroides (*Divider*) . . . 60
3.5.1. Divisores multibanda (*Multiband Divider*) . . . 61
3.6. Módulo de control (*Main Control*) . . . 63
3.6.1. Fin de la ejecución . . . 64
3.6.2. Detección de errores . . . 65
3.6.3. FSM de control . . . 65
3.7. Envoltorio elemental para pruebas en FPGA . . . 68
3.8. Otros submódulos . . . 69
3.8.1. *Full-adder* multibanda . . . 69

3.8.2. LFSR multibanda 71
3.8.3. Sincronizador 72

4. Validación y verificación **75**
4.1. Cuprite 75
4.2. Implementación *software* 76
4.3. Viabilidad de la adaptación *hardware* 77
4.3.1. Reducción dimensional 78
4.4. Comportamiento básico en *hardware* 79
4.4.1. Verificación de módulos genéricos 79
4.4.2. Flujo del *pipeline* 79
4.4.3. Módulo de control principal 80
4.5. Comportamiento general del algoritmo 81
4.5.1. Pruebas con imágenes sintéticas simples 81
4.5.2. Pruebas con imágenes sintéticas aleatorias 81
4.6. Diferencias entre *hardware* y *software* 82
4.7. Convergencia a mínimo local 83
4.8. Pruebas con imágenes reales 84
4.9. Simulaciones post-síntesis y post-implementación 86
4.10. Interfaz con FPGA 87
4.11. Verificación del funcionamiento en FPGA 87

5. Resultados **90**
5.1. Objetivos logrados 90
5.1.1. Implementación *software* 90
5.1.2. Diseño *hardware* 92
5.1.3. Simulación y depuración *hardware* 92
5.2. Rendimiento del núcleo 94

5.2.1. Rendimiento del *pipeline* 95
5.2.2. Impacto del número de clústers en el rendimiento 96
5.2.3. Impacto del número (máximo) de píxeles en el rendimiento 97
5.2.4. Resumen de rendimiento 98
5.2.5. Funcionamiento en tiempo real 98
5.2.6. *Speed-up* respecto al *software* 100
5.3. Utilización de recursos 101
5.3.1. Análisis del uso de recursos para la configuración sobredimensionada 104
5.3.2. Impacto de la configuración del núcleo en el uso de recursos 105
5.4. Consumo energético 106

6. Conclusiones y trabajo futuro **110**
6.1. Conclusiones 110
6.1.1. Implementacion *software* 110
6.1.2. Modificaciones sobre el algoritmo original 111
6.1.3. Arquitectura en *pipeline* 111
6.1.4. Desarrollo *hardware* 112
6.1.5. Propiedades del núcleo 113
6.1.6. Conveniencia del diseño *hardware* 114
6.2. Trabajo futuro 116
6.2.1. Reductor de bandas 116
6.2.2. Generación de un *IP soft core* 116
6.2.3. Integrar el núcleo en un codiseño *hardware-software* 117
6.2.4. Experimentación en profundidad 117
6.2.5. Automatizar la generación de *pipelines* en HDL 118

Bibliografía **121**

A. Introduction 123

A.1. Motivation . . . 123

A.2. Objectives . . . 125

A.3. Background . . . 126

A.4. Work plan . . . 127

A.5. Organisation of this report . . . 129

B. Conclusions 132

B.1. Software implementation . . . 132

B.2. Modifications to the original algorithm . . . 132

B.2.1. Resource optimization . . . 133

B.3. Pipeline architecture . . . 133

B.4. Hardware development . . . 134

B.4.1. Simulation . . . 134

B.5. Core's properties . . . 135

B.5.1. Performance . . . 135

B.5.2. Power consuption . . . 135

B.5.3. Resource usage . . . 135

B.6. Hardware design convenience . . . 136

B.6.1. Veredict . . . 137

Índice de figuras

1.1. Espectro visible. Fuente: Wikipedia [1]. 2
1.2. Cubo hiperespectral. Fuente: Wikipedia [2]. 3

2.1. Aplicaciones de imágenes hiperespectrales. 12
2.2. Imagen hiperespectral de un aguacate. Fuente: *European Space Agency* . . . 13
2.3. Sensores de escaneo espectral. Fuente [3]. 13
2.4. Esquemático simple de una FPGA. 15
2.5. Xilinx VC709. Fuente: Xilinx, Inc 16
2.6. k-means, ejemplo de ejecución. Fuente: Wikipedia [4]. 19
2.7. Diferencia ilustrada entre *hardware* y *software*. Fuente de ambas imágenes: Wikimedia Commons. 22
2.8. Árbol de sumadores. 25
2.9. Ejemplo de reducción dimensional. 27
2.10. Ilustración comparativa sobre la segmentación en un *pipeline*. 30
2.11. Ejemplo de array sistólico. 32
2.12. Dos módulos conectados mediante AXI-Stream. 33
2.13. LFSR de 16 bits. 35

3.1. Vista general de la arquitectura del núcleo de procesamiento. 47
3.2. Cronograma del *pipeline*. 51
3.3. Ilustración con el problema del desfase en la actualización del centroide. . . . 52
3.4. Arquitectura del clasificador segmentado. 53
3.5. Arquitectura de los procesadores sistólicos del clasificador. 55
3.6. Arquitectura del actualizador de acumuladores. 58

3.7. Arquitectura de un nodo del actualizador de acumuladores. 59
3.8. Arquitectura del módulo de división, que actualiza los centroides. 60
3.9. Arquitectura de un divisor multibanda. 62
3.10. Máquina de estados finitos para el control del sistema. 66
3.11. Arquitectura de la interfaz con FPGA. 67
3.12. Arquitectura del *Full-adder* multibanda. 70
3.13. Arquitectura del LFSR multibanda. 71
3.14. Arquitectura del sincronizador. 73

4.1. Visualización a color (RGB) de la imagen empleada para pruebas. 76
4.2. Clusterización por *software* con la imagen hiperespectral completa. 77
4.3. Clusterizaciones *software* de la imagen hiperespectral reducida. 78
4.4. Diferencias de clusterización entre imagen completa y reducida. 79
4.5. Imágenes sintéticas. 82
4.6. Ejecución del *script* de verificación. 84
4.7. Clusterización realizada por el núcleo *hardware*. 85
4.8. Cronograma de simulación. 86

5.1. Simulación del núcleo. 94
5.2. Impacto de la inicialización en el rendimiento. 95
5.4. Impacto del número de clústers en la obtención de resultados. 96
5.6. Impacto del número de píxeles en la obtención de resultados. 97
5.8. Impacto del número de píxeles y clústers a identificar en la obtención de resultados. 99
5.9. Utilización de recursos con configuración básica. 101
5.10. Utilización de recursos con configuración amplia. 102
5.11. Utilización de recursos con configuración extremadamente sobredimensionada. 103
5.13. Utilización de recursos sin reducción dimensional. 103

5.14. Estimación de consumo del núcleo. 107

5.15. Comparación de una piscina olímpica con 80 barreños. 108

Índice de tablas

3.1. Parámetros del diseño. 43
3.2. Parámetros autoconfigurables del diseño. 44

5.1. Longitud del *pipeline*. 93
5.2. Comparativa entre *hardware* y *software*. 100
5.3. Resumen de utilización de recursos. 104

6.1. ¿Merece la pena el *hardware*? 115

Resumen

Este proyecto consiste en el diseño e implementación *hardware* del algoritmo de clusterización k-means, aplicado a imágenes hiperespectrales. El objetivo es lograr el cómputo en tiempo real, gracias a las técnicas de aceleración aplicadas.

El diseño cuenta con una arquitectura en *pipeline* altamente segmentado, compuesto por tres módulos retroalimentados entre sí, los cuales procesan de forma iterativa los datos de entrada hasta que los resultados convergen a valores estáticos.

Dicho *pipeline* opera con multitud de píxeles simultáneamente, además todas las bandas de cada píxel hiperespectral se tratan en paralelo. En consecuencia, este logra un rendimiento excepcional: es capaz de procesar a razón de un píxel hiperespectral completo por ciclo de reloj, de forma constante.

El diseño es completamente paramétrico, tal que es posible adaptarlo de forma automática a imágenes de distintas características o alterar las propiedades de la clusterización. También cuenta con una interfaz AXI Stream para facilitar su reutilización e integración en diversos sistemas.

Los resultados muestran una velocidad de procesamiento de aproximadamente 100 millones de píxeles hiperespectrales (con 224 bandas cada uno) por segundo con un discreto uso de recursos en FPGA. Se logra así un amplio margen para la ejecución del algoritmo en tiempo real, dejando holgura para los crecientes flujos de datos que trae la tecnología.

Palabras clave

Hiperespectral, Procesado de imagen, Clusterización, K-means, *Pipeline*, Diseño hardware, RTL, FPGA, HDL, VHDL

Abstract

This project consists of creating a hardware design which implements the k-means clustering algorithm, applied to hyperspectral images. The goal is to achieve real-time computation, thanks to the acceleration techniques applied.

The design has a deep pipelined architecture, made up of three modules with feedback between them, which iteratively process the input data until the results converge to static values.

The *pipeline* operates with multiple pixels simultaneously, meanwhile all bands of each hyperspectral pixel are processed in parallel. As a result: it attains an exceptional performance, being able to reach a constant throughput of one full hyperspectral pixel per clock cycle.

The design is fully parametric, so is possible to automatically adapt it to different kinds of images or to alter the clustering properties. Furthermore, it has an AXI Stream interface in order to increase its reusability and make it easier to integrate into different systems.

The results reveal a processing speed of approximately 100 million hyperspectral pixels (with 224 bands each) per second with a discrete use of FPGA resources. This provides a wide margin for real-time execution of the algorithm, while leaving extra room for the increasing data flows brought by the newest technology.

Keywords

Hyperspectral, Image processing, Clustering, K-means, Pipeline, Hardware design, RTL, FPGA, HDL, VHDL

Capítulo 1

Introducción

1.1. Motivación

Llevamos haciendo lo mismo desde hace por lo menos trescientos mil años, cuando aparecimos como especie en este planeta. La ambición que tenemos los seres humanos por ir más allá de nuestras capacidades no conoce límites. Nuestras manos nos resultaban insuficientes, así que creamos lanzas para cazar; tampoco nos gustaba tener sed e inventamos el botijo; y ahora mucho menos nos conformamos con andar y hemos terminado por desarrollar automóviles.

Lo mismo nos sucede con nuestros ojos: queremos ver más allá de lo que podemos. Y eso ha propiciado la aparición de nuevas herramientas y técnicas para obtener información visual burlando ampliamente nuestros límites naturales. En concreto, los humanos solo somos

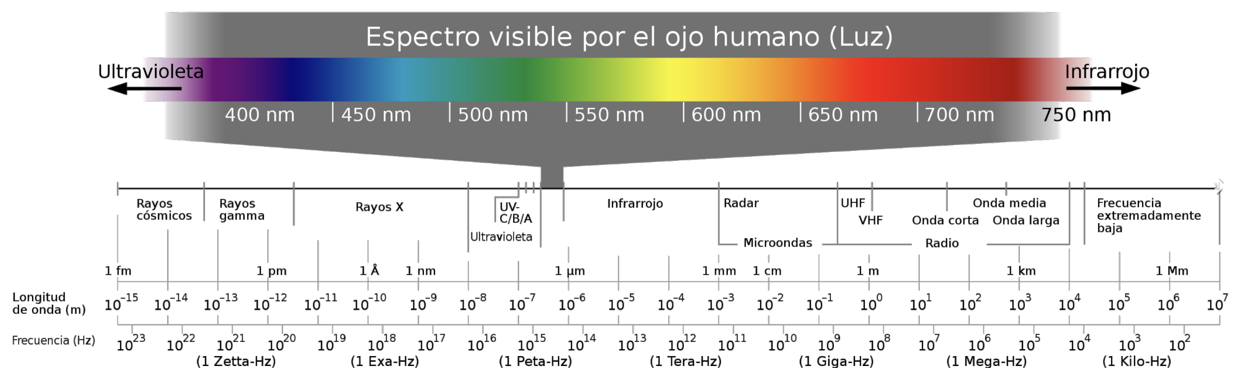


Figura 1.1: Espectro visible. Fuente: Wikipedia [1].

Figura 1.2: Cubo hiperespectral. Fuente: Wikipedia [2].

capaces de percibir una pequeña región del espectro electromagnético (también conocida como espectro visible y mostrada en la figura 1.1); es decir, que hay luz, hay colores, que no podemos ver.

O al menos no podíamos hacerlo hasta que aparecieron los primeros sensores espectrales, que desde entonces han permitido realizar grandes avances en multitud de ámbitos, tales como el estudio de minerales, la agricultura, la astronomía o el procesado de alimentos (por citar algunos). Así pues, al ser capaces de captar más ondas de distinta longitud irradiadas por el objeto de análisis, se amplía inmensamente la información obtenida sobre él. En la figura 1.2 se muestra un cubo hiperespectral, típicamente usados para representar imágenes hiperespectrales.

No obstante, obtener ingentes cantidades de información no deja de resultar abrumador para nuestro entendimiento y a menudo necesitamos un paso previo para que nuestro cerebro pueda sacarle partido. Por ello suele ser conveniente someter los datos hiperespectrales a algún tipo de procesamiento para facilitar la formulación de hipótesis y la obtención de conclusiones a partir de los mismos.

Dichos procesamientos son a menudo tareas bien definidas, compuestas por una serie de pasos claramente pautados y por tanto susceptibles de ser automatizados. En otras palabras, son lo que hoy en día llamamos algoritmos. Solemos definir esos algoritmos mediante *software*, con secuencias de instrucciones que algún ordenador deberá de ejecutar una y otra vez, constantemente, sin parar, como un esclavo digital incapaz de sentir nada, sin más propósito que acatar las órdenes recibidas.

La impaciencia nos consume, queremos resultados cuanto antes. Pero los únicos responsables de la lentitud de la máquina somos nosotros, lo que desemboca en una terrible frustración tan agotadora como desmoralizante. Sin embargo, la solución es simple: invertir un poco más de tiempo en la implementación del algoritmo, para luego ahorrarlo en cada una de las incontables veces que sea ejecutado. Y sin duda, la forma más directa de lograrlo es implementar el algoritmo en *hardware*.

Y llegado este punto, es fácil deducir el propósito de este trabajo: resolver problemas en el ámbito de las imágenes hiperespectrales de una forma rápida y eficiente, utilizando un diseño *hardware* creado específicamente para dicho fin.

1.2. Objetivos

El principal objetivo de este trabajo es elaborar un diseño *hardware* a nivel de transferencia de registros (RTL) que implemente el algoritmo de clusterización "k-means". Este estará orientado al funcionamiento en tiempo real sobre FPGA.

Para la consecución del mismo, será necesario realizar previamente una aproximación *software*, así como una serie de simulaciones que garanticen la viabilidad del proyecto.

Por otra parte, el diseño debe ser totalmente paramétrico, para que sea posible adaptarlo de forma inmediata a soluciones concretas; sin importar las características del sensor hiperespectral que obtiene la imagen de entrada ni las propiedades necesarias en la salida obtenida.

A pesar del gran volumen de información que contienen las imágenes hiperespectrales se

pretende lograr un funcionamiento en tiempo real del sistema.

Con vistas a satisfacer los requisitos de rendimiento, la arquitectura constará de un *pipeline* retroalimentado, el cual explotará diversas técnicas de paralelización para maximizar el rendimiento.

No obstante, también será imprescindible encontrar el equilibrio más óptimo posible entre utilización de recursos *hardware* y rendimiento, tal que sea posible emplazar el diseño en una FPGA.

1.3. Antecedentes

Existe un amplio abanico de trabajos previos en lo que se refiere al empleo de FPGAs para procesar imágenes hiperespectrales [5, 6], aunque aún más frecuentes son las implementaciones *software* en este mismo ámbito; sin embargo, estas últimas no resultan relevantes de cara a este proyecto.

En concreto, el algoritmo de clusterización k-means fue originalmente propuesto por Stuart Lloyd en 1957 [7], aunque también publicado por Eduard W. Forgy [8]. La propuesta de Lloyd enfocaba el algoritmo a la modulación PCM, sin embargo, hoy en día k-means es una de las técnicas más utilizadas para clasificación de imágenes en *Machine Learning*.

Gracias a técnicas como la síntesis de alto nivel o la generación automática de código HDL a partir de *software*, en los últimos años ha proliferado enormemente el número de soluciones que involucran FPGAs. No obstante, hoy por hoy estas técnicas no permiten alcanzar un funcionamiento tan óptimo como las descripciones RTL, donde se tiene un control total sobre el flujo de datos. La referencia [9] aborda el tema en detalle.

Es frecuente que este tipo de investigaciones y trabajos concluyan en que el uso de FPGAs para tratar imágenes hiperespectrales favorece inmensamente a la velocidad de procesado, proporcionando también una excelente escalabilidad. Algunos con ejemplos con HLS son las referencias [10–12], o también en OpenCL [13].

En cuanto al propio k-means, es frecuente encontrar implementaciones en GPU (al igual

que sucede con otros algoritmos de procesado de imagen), tales como [14, 15].

También existen implementaciones que han tratado de acelerar k-means utilizando CPUs, como la propuesta en la referencia [16].

El artículo [17] propone una implementación mixta de k-means, combinando un computador de propósito general y una FPGA para acelerar la primera parte del procesamiento. Esta se basa en una array sistólico de procesadores (más información en la sección 2.7) muy simple, que tiene la ventaja de apenas requerir mecanismos de control. Sin embargo, el resto de operaciones que no están aceleradas suponen un importante "cuello de botella".

1.4. Plan de trabajo

El desarrollo de este proyecto ha abarcado desde noviembre de 2021 a junio de 2022. Este ha sido organizado por fases secuenciales, dando mayor prioridad a las que más dependencias originaban. A continuación, se listan dichas fases por orden cronológico:

1. **Planteamiento del proyecto.**

   Antes de comenzar el desarrollo, se determinaron los rasgos fundamentales del proyecto: implementar sobre FPGA un algoritmo de procesado de imágenes hiperespectrales.

2. **Búsqueda del algoritmo a implementar y análisis de viabilidad.** *(2 semanas).*

   Ya determinada la índole del proyecto, comenzó un proceso de búsqueda de posibles algoritmos a implementar. Para cada uno de ellos se valoraban distintas opciones de implementación para comprobar que era factible completarlo en el tiempo disponible. Finalmente fue seleccionado k-means.

3. **Implementación *software* del algoritmo.** *(2 semanas).*

   En primer lugar, se implementó k-means en *software*. Esto sirvió de toma de contacto con el tratamiento de imágenes hiperespectrales, así como con el propio algoritmo.

4. **Validación *software*.** *(1 semana).*

En primer lugar, era imprescindible comprobar que la implementación *software* era correcta, por lo que fueron programados varios *scripts* para validar los resultados.

Por otra parte, en el diseño *hardware* habrá previsiblemente una serie de restricciones implícitas, por ejemplo la generación de números aleatorios mediante LFSR o la pérdida de precisión al prescindir de aritmética en punto flotante. Por ello fue necesario probar los efectos de las mismas en la precisión y convergencia del algoritmo.

5. **Diseño de la arquitectura.** *(6 semanas).*

   Una vez implementada y validada la versión *software*, se inició el diseño del *hardware* a nivel RTL, módulo por módulo. En esta fase se establecieron primero las bases generales de la arquitectura: un *pipeline* segmentado que procese un píxel por ciclo. También fueron fijados los mecanismos de sincronización e intercambio de datos entre módulos.

   Para cada uno de los módulos se sucedieron múltiples alternativas de diseño, ajustando en ellos la compartición de recursos, el paralelismo, la segmentación y el control interno. De esta forma, hubo variantes que ofrecían mayor rendimiento, algunas menor uso de recursos y otras un control más simple.

6. **Implementación del *pipeline*.** *(3 semanas).*

   Después de cerrar el diseño al completo, arrancó el proceso de implementación en HDL del *pipeline*, módulo por módulo. Este proceso incluye también el desarrollo de los componentes compartidos y su simulación, para garantizar que funcionen correctamente una vez integrados.

   Dado que el diseño ya había sido elaborado, esta fase consistió esencialmente en describirlo en VHDL. En particular, los módulos del *pipeline* son el clasificador, el actualizador de acumuladores y el divisor.

7. **Implementación del módulo de control.** *(2 semanas).*

Después del *pipeline*, fue descrito el módulo de control global. El proceso es más laborioso que el anterior porque la lógica es más compleja, de hecho en él también se incluye la máquina de estados principal.

8. **Integración del núcleo y depuración de errores.** *(3 semanas).*

   Al estar completamente implementados los módulos, se procedió a su integración, conformando el núcleo al completo.

   Seguidamente, comenzó el largo proceso de depuración de errores en *hardware*, primero hasta lograr que toda la descripción fuese sintetizable, y posteriormente hasta obtener resultados coherentes en la simulación.

9. **Verificación del núcleo.** *(4 semanas).*

   A continuación, se llevó a cabo la verificación de los resultados obtenidos del núcleo. Para ello se siguió un método progresivo, utilizando varios tipos de imágenes tanto sintéticas como reales, y simulando todo el diseño para validar su comportamiento. Fue imprescindible realizar las pruebas de forma escalonada, de lo contrario no habría sido posible localizar y corregir los errores.

   En el proceso se recurrió a la primera implementación *software*, así como a una serie de *scripts* elaborados para verificar los resultados.

10. **Implementación de un envoltorio elemental para pruebas en FPGA.** *(2 semanas).*

    Cuando el núcleo estaba completo y validado, fue creado un diseño para poder suministrar imágenes hiperespectrales al núcleo desarrollado, así como arrancar su procesamiento en *hardware*.

11. **Simulación y verificación del envoltorio.** *(2 semanas).*

    Al igual que con el núcleo aislado, se trató nuevamente de validar mediante simulación

post-implementación el sistema completo, para asegurar su funcionamiento previo a su bajada a placa.

1.5. Organización de esta memoria

El resto de esta memoria, está estructurado en una serie de capítulos cuyos contenidos quedan descritos a continuación:

- **Contexto tecnológico**: recopila todos los medios que han sido necesarios para llevar a cabo este proyecto. Contiene información técnica sobre los componentes del núcleo, protocolos, estándares, algoritmos implementados y herramientas utilizadas.

- **Arquitectura *hardware***: se describen los detalles arquitectónicos del núcleo implementado y son justificadas las decisiones de diseño. En el capítulo se muestra el funcionamiento del *pipeline* y los mecanismos de control, para lo cual se han incluido múltiples diagramas.

- **Verificación**: contiene las pruebas realizadas para garantizar el correcto funcionamiento de todos los componentes desarrollados.

- **Resultados**: en él se detallan los objetivos alcanzados y se analizan los resultados obtenidos. Contiene las propiedades del diseño final: rendimiento, uso de recursos, consumo, etcétera.

- **Conclusiones y trabajo futuro**: incluye las conclusiones que se han obtenido a raíz del trabajo realizado, así como las posibles vías de continuación.

- **Bibliografía**: relación de fuentes referenciadas en la memoria (artículos, libros, trabajos previos...).

Capítulo 2

Contexto tecnológico

El presente capítulo describe los algoritmos y técnicas involucradas en el desarrollo de este trabajo, así como los medios técnicos y tecnológicos empleados. Esto incluye también las herramientas *software* y *hardware* utilizadas.

2.1. Imágenes hiperespectrales

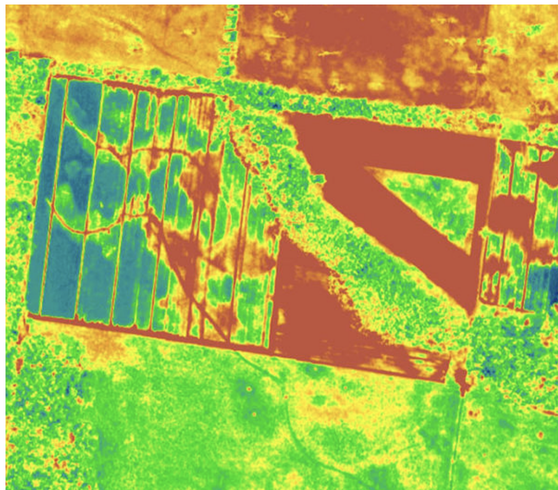
Las imágenes o fotografías convencionales pueden definirse como una matriz de puntos equidistantes entre sí, donde cada uno de ellos representa una intensidad luminosa. Y la luz no es otra cosa que la parte visible de la radiación electromagnética (consultar figura 1.1 de la introducción). De este modo, las imágenes a menudo se emplean para representar visualmente elementos de la realidad.

Por otro lado, las imágenes hiperespectrales atienden a la misma definición del párrafo anterior, pero salvando la diferencia de no estar únicamente restringidas a la parte visible del espectro electromagnético. Es decir, las imágenes hiperespectrales almacenan información que los seres humanos no podemos percibir de forma natural.

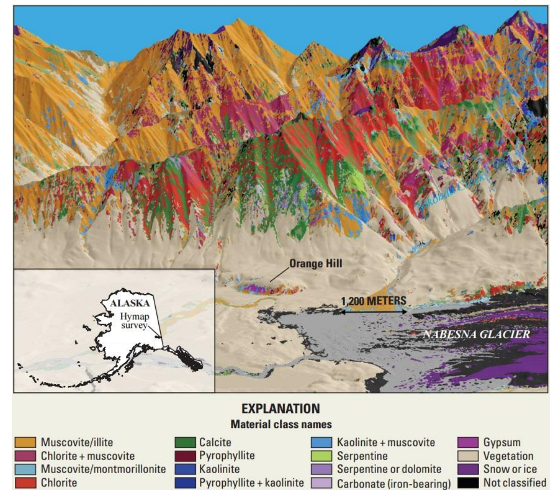
Al igual que ocurre con las fotografías corrientes, las representaciones hiperespectrales digitales no permiten registrar el espectro electromagnético de forma continua, exigen muestrearlo. Por ello, la información se presenta dividida en bandas. No obstante, en las hiperespectrales habrá multitud de bandas (el número depende del sensor), mientras que en las fotografías convencionales hay típicamente tres: una para el rojo, para el azul y para el

verde (que se corresponden con los colores primarios).

Uno de los propósitos más habituales de esta tecnología es identificar fácilmente objetos o materiales que reflejan radiación electromagnética no visible. Es frecuente encontrar imágenes hiperespectrales de la superficie terrestre por el interés que suscita en la búsqueda de minerales 2.1b o en la agricultura de precisión 2.1a.



(a) Imagen hiperespectral de un campo de cultivo. Fuente: *Am. for Ben-Gurion University*.



(b) Análisis geológico con información hiperespectral. Fuente: *USGS*.

Figura 2.1: Aplicaciones de imágenes hiperespectrales.

No obstante se aplica también en otros campos, por ejemplo en astronomía, para medir el comportamiento de astros o galaxias situadas a años luz de distancia; o en controles de calidad de alimentos 2.2.

2.1.1. Sensores

Para la captura de imágenes hiperespectrales existen principalmente tres tipos de sensores: de escaneo espacial (capturan todas las bandas de una pequeña región y se van desplazando para cubrir todo el área); de escaneo espectral (capturan todo el área de la imagen a la vez, pero cada una de las bandas se van obteniendo individualmente) e instantáneos (capturan todas las bandas de todo el área al mismo tiempo).

Dentro de los sensores de escaneo espectral se distinguen dos subtipos: *whisk broom* y

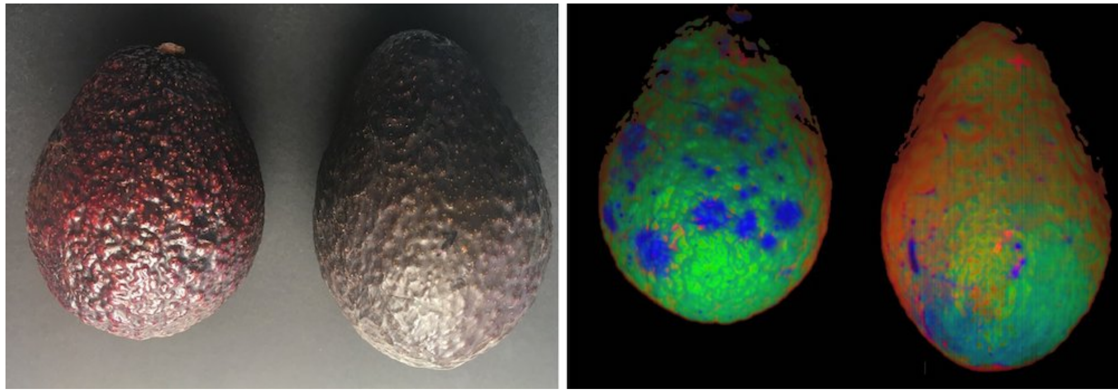
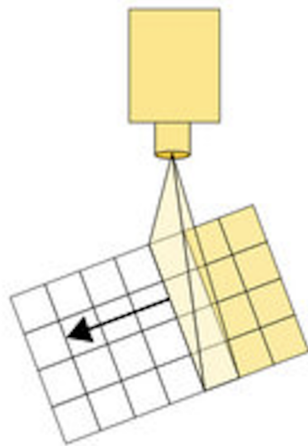
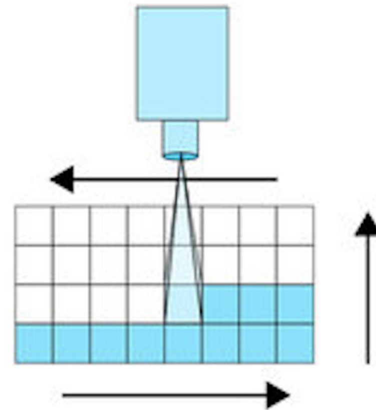


Figura 2.2: Imagen hiperespectral de un aguacate. Fuente: *European Space Agency*

push broom. La diferencia es que los *whisk broom* capturan los pixeles individualmente, y los *push broom* son capaces de capturar filas enteras (más rápido). La imagen 2.3 ilustra los dos tipos de sensor.



(a) *Push broom*.



(b) *Whisk broom*.

Figura 2.3: Sensores de escaneo espectral. Fuente [3].

AVIRIS

Para la realización de este proyecto se han empleado fundamentalmente imágenes capturadas por el sensor AVIRIS (*Airborne Visible / Infrared Imaging Spectrometer*) de la agencia

nacional espacial de Estados Unidos, la NASA.

Se trata de un sensor de escaneo espectral, de tipo *whisk broom*, capaz de distinguir 224 bandas contiguas, con longitudes de onda comprendidas desde los 400 hasta los 2500 nanometros.

Para obtener imágenes del AVIRIS, puede recurrirse a la web habilitada por la NASA para ello [18].

2.2. FPGA

Las FPGAs (o *field programmable gate array*, por sus siglas en inglés) son un tipo de *hardware* reconfigurable. Es decir, son circuitos integrados en los que podemos redefinir el comportamiento de sus elementos funcionales así como alterar su interconexionado interno para implementar cualquier diseño digital.

Dado que el proceso de programación es totalmente reversible, las FPGAs facilitan en gran medida la creación de sistemas *hardware* de propósito específico, así como realizar pruebas con prototipos del diseño y depurar posibles errores antes de alcanzar la implementación final.

Pero su principal ventaja es que ponen el desarrollo *hardware* a disposición del público general. Al ser fabricadas en masa sin una aplicación específica, hacen que el coste de implementación de un circuito digital en esta tecnología sea muchísimo menor (en pequeñas tiradas) que el de fabricación de un ASIC (*Application Specific Integrated Circuit*).

2.2.1. Componentes básicos de una FPGA

Para apoyar la explicación, en la imagen 2.4 se muestra un esquema general del diseño interno de una FPGA.

Típicamente, estos dispositivos se componen de matrices de celdas (etiquetadas como CLB en el diagrama), las cuales son programadas para realizar funciones lógicas; estas además son capaces de mantener su estado (mediante biestables). Dichas celdas se comunican

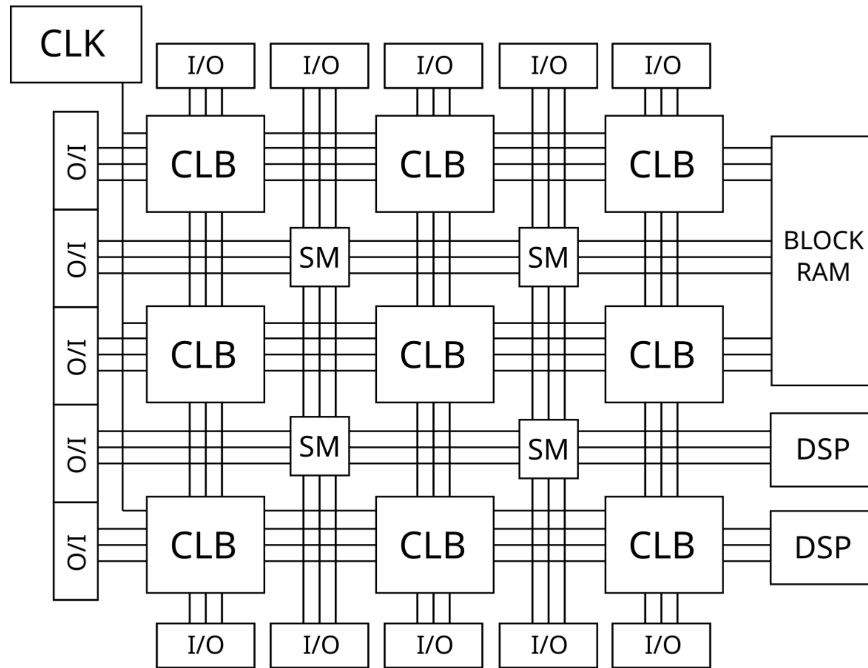


Figura 2.4: Esquemático simple de una FPGA.

entre sí a través de matrices de interconexión (SM en el esquema), que igualmente deben ser configuradas para establecer las rutas.

También en ellas existen elementos de memoria, que permiten instanciar bancos de registros o almacenar datos; es lo que se conoce como *block RAM*. La principal ventaja de este tipo de memoria es su velocidad, pues su tiempo de acceso es de un solo ciclo de reloj.

Por otro lado, cuentan con una red que distribuye el reloj a los distintos elementos de la FPGA.

Además en algunos modelos existen componentes adicionales, tales como DSPs para procesado de señal, que facilitan la implementación de operaciones complejas, por ejemplo la MAC (*Multiply-Accumulate Operation*).

En último lugar estarían los puertos de entrada/salida, que permiten el intercambio de información entre la FPGA y el exterior.

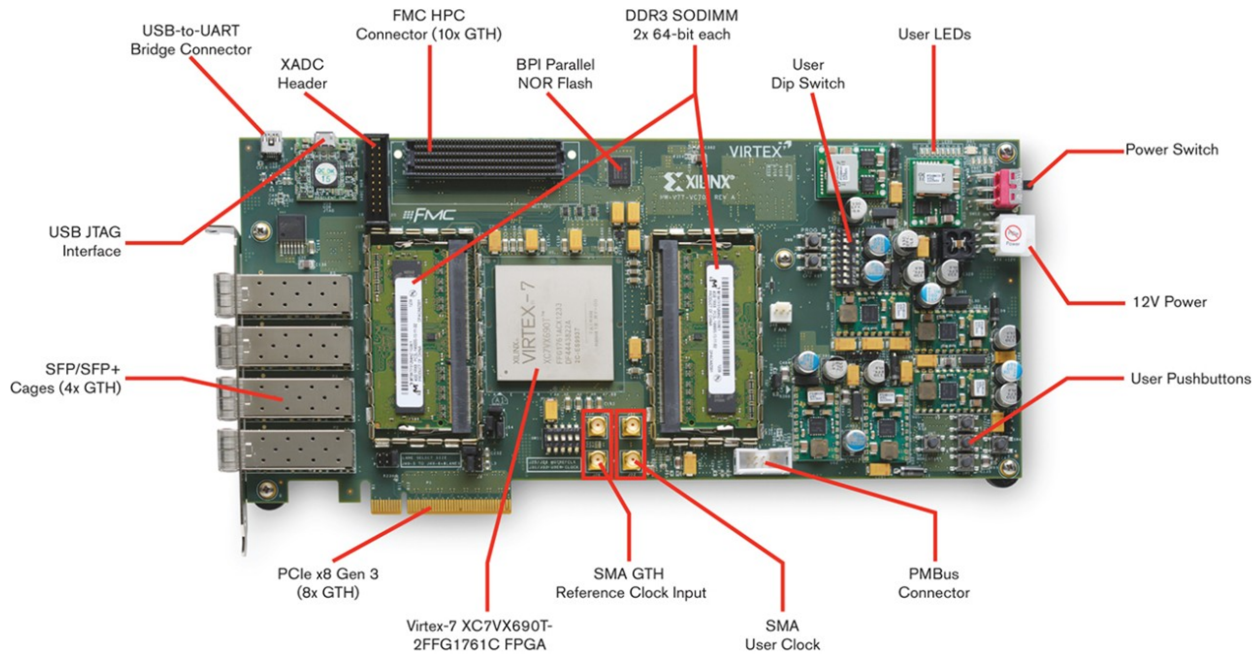


Figura 2.5: Xilinx VC709. Fuente: Xilinx, Inc

2.2.2. Xilinx VC709

A la hora de emplear una FPGA en proyectos como este, lo más habitual es recurrir a una placa de prototipado con FPGA integrada. Se trata de soluciones comerciales orientadas a usuarios finales que constan de un circuito impreso el cual integra la FPGA, así como todos los componentes necesarios para programarla y hacerla funcionar directamente.

En este caso, la plataforma escogida para el proyecto es la placa VC709 fabricada por Xilinx. Esta contiene una FPGA Virtex-7 VX690T, también fabricada por Xilinx; además incorpora una interfaz DDR3, con dos ranuras para conectar un máximo de 4 GB de memoria RAM en cada una (de hasta 1866 Mbps de ancho de banda, a 933 MHz de reloj). También dispone de conectividad PCIe Gen3 x8, pares SMA (para el reloj), UART y cuatro puertos SPF+.

La figura 2.5 muestra una imagen de la VC709, con anotaciones indicando la ubicación de sus componentes.

Se ha escogido esta plataforma por los buenos resultados obtenidos con ella en trabajos

previos del mismo ámbito.

Xilinx Virtex-7

Tal y como se ha citado anteriormente, la VC709 emplea una FPGA Xilinx Virtex-7, fabricada con fotolitografía de 28 nm. En particular el modelo es el XC7VX690T-2FFG1761C.

Esta FPGA cuenta con 108.300 slices, cada uno con ocho *flip-flops* y cuatro LUTs (que son tablas de verdad para implementar funciones lógicas de hasta seis entradas). No obstante estas características son comunes a todas las FPGAs de la serie 7 de Xilinx.

Además esta equipada con 3.600 DSPs para facilitar la implementación de algunos cálculos aritméticos. Cada uno de ellos cuenta con un *pre-adder*, un multiplicador de hasta 25 por 18 bits, un sumador y un acumulador.

Por otro lado, contiene 52.920 Kb de memoria *block RAM* divididos en 1.470 bloques de 36 Kb cada uno.

Toda la información relativa a esta FPGA puede consultarse en el *datasheet* de especificaciones [19].

2.3. Algoritmo *k*-means

El algoritmo k-means sirve para agrupar puntos de un espacio n-dimensional en subconjuntos (también llamados grupos o clústers), de forma automática. El criterio de clasificación consiste en asignar el mismo grupo a aquellos elementos próximos entre sí.

Para ello se calcula el punto medio de todos los elementos que pertenecen a un clúster, así es posible comparar los elementos a clasificar de forma individual con cada clúster. A dicho valor medio se le conoce como centroide. Expresado matemáticamente (siendo n el número de elementos y a_i el valor de cada uno):

$$centroide = \frac{1}{n}\sum_{i=1}^{n} a_i = \frac{a_1 + a_2 + \cdots + a_n}{n}$$

Después de comparar el elemento a clasificar con todos los centroides calculados, este se asignará al grupo cuyo centroide presente el valor más cercano a sí mismo. Para ello es necesario calcular la "distancia" entre ambos puntos n-dimensionales. Hay diversos métodos para ello, por ejemplo la distancia Manhattan o la Euclídea (ambos son discutidos más adelante, en la sección 2.4.1).

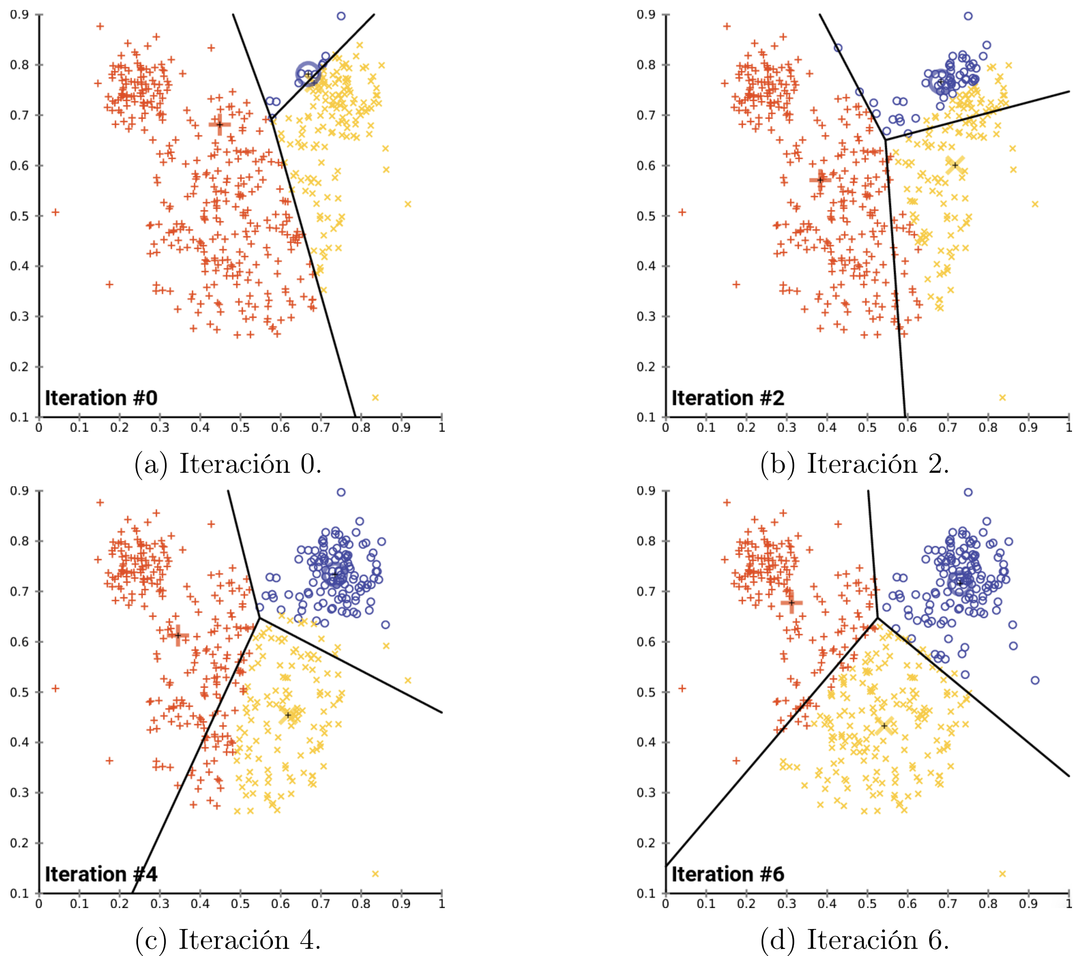
La asignación de un elemento a uno de los subconjuntos repercutirá en el valor medio (centroide) del mismo, por lo que es frecuente que algunos de los elementos que ya pertenecían al subconjunto en cuestión, pierdan la afinidad que guardaban con él. En otras palabras, el valor de los elementos antiguos puede distanciarse del centroide al añadir elementos nuevos al grupo.

Por tanto, todos los elementos del conjunto inicial deben ser evaluados múltiples veces; es decir, el proceso de clasificación se realizará de forma iterativa.

A medida que van sucediendo las iteraciones, los subconjuntos asignados a cada elemento convergerán hacia valores estáticos. De este modo, la ejecución concluye cuando tras una iteración no se producen cambios significativos entre los subconjuntos asignados a los elementos. Dicho de otra manera, el algoritmo termina cuando los elementos dejan de moverse de un subconjunto a otro.

Y para terminar, volvamos al principio. ¿Cómo es posible calcular los centroides en la primera iteración, si al inicio no se conoce que elementos pertenecen a cada clúster? Pues simplemente asignando de forma aleatoria un clúster a cada píxel, tal que las sucesivas iteraciones se encarguen de clasificarlos correctamente.

En la figura 2.6 se observa un ejemplo visual de ejecución del algoritmo k-means: de izquierda a derecha y arriba a abajo se van sucediendo progresivamente las iteraciones, mostrando como evolucionan los centroides y la pertenencia de los datos a cada uno de los tres clústers.



(a) Iteración 0. (b) Iteración 2.

(c) Iteración 4. (d) Iteración 6.

Figura 2.6: k-means, ejemplo de ejecución. Fuente: Wikipedia [4].

K-means es un algoritmo computacionalmente costoso, de hecho pertenece a la clase de complejidad NP-difícil (*NP-Hard* en inglés).

2.3.1. Pseudocódigo

A continuación se muestra el pseudocódigo tomado como referencia para la implementación del algoritmo. Este se ha obtenido del artículo [17] escrito por Dominique Lavenier.

```
while (pixel_move != 0) {
  pixel_move = 0;
  for (i=0; i<NB_PIXELS; i=i+B) {
    for (b=0; b<B; b++) {
      min = MAX_INT;
```

```c
      for (k=0; k<NB_CLASS; k++) {
        if (N_CENTER[k] != 0) {
          for (d=0; d<NB_BANDS; d++)
            dist += ABS(PIXEL[i+b][d] - CENTER[k][d]);
          if (dist < min) {
            min = dist;
            idx[b] = k;
          }
        }
      }
    }
    for (k=0; k<NB_CLASS; k++) change[k] = false;
    for (b=0; b<B; b++)
      if (CLASS[i+b] != idx[b]) {
        cntr_acc += 2; pixel_move++;
        k = CLASS[i+b]; N_CENTER[k]--; change[k] = true;
        for (d=0; d<NB_BANDS; d++) ACC[k][d] -= PIXEL[i+b][d];
        k = idx[b]; CLASS[i+b] = k; N_CENTER[k]++; change[k] = true;
        for (d=0; d<NB_BANDS; d++) ACC[k][d] += PIXEL[i+b][d];
        CLASS[i+b] = idx[b];
      }
    for (k=0; k<NB_CLASS; k++)
      if (N_CENTER[k] != 0 && change[k] == true) {
        cntr_center++;
        for (d=0; d<NB_BANDS; d++) CENTER[k][d] = ACC[k][d]/N_CENTER[k];
      }
  }
}
```

2.3.2. Aprendizaje automático

K-means pertenece a la familia de algoritmos no supervisados de aprendizaje automático, lo cual significa que es el propio algoritmo el que determina la formación de los conjuntos. No hay pautas previas que determinen cómo han de agruparse los elementos.

Sin embargo, es imprescindible especificar en cuántos subconjuntos han de clasificarse los datos de entrada. El uso de un mayor número de clústers supondrá también un aumento en el coste computacional del algoritmo.

2.3.3. Uso en procesado de imagen

Como ya se introdujo en la sección 1.3 (dedicada a los antecedentes de este trabajo), las primeras versiones del algoritmo k-means datan de mediados del siglo XX. Originalmente

fue planteado para su uso en procesado de señal pero nada impide aplicarlo a otros ámbitos, siempre y cuando el objetivo sea clasificar elementos cuantificados.

De hecho, es bastante frecuente encontrar soluciones en visión por computador que hagan uso de k-means; en ellas los elementos a clasificar son a menudo los píxeles de la imagen, los cuales se asocian para tratar de identificar regiones de similar naturaleza.

2.4. Adaptación *hardware* de *k*-means

Todas las fuentes que se han tomado como referencia para la elaboración de este proyecto orientaban su visión del algoritmo a ser implementado total (referencias [7] y [8]) o parcialmente (referencia [17]) en *software*.

Esto significa que el proceso que realiza el algoritmo está ideado como una secuencia de instrucciones, y por tanto no es posible una traducción directa a su equivalente en *hardware* de propósito específico.

Tratar de replicar en *hardware* el comportamiento del *software* de forma idéntica, instrucción a instrucción, aceleraría enormemente la ejecución del algoritmo. Lo mismo ocurriría con otras técnicas que involucran el uso de FPGAs con el mismo propósito: e.g. la síntesis de alto nivel o el uso de herramientas como OpenCL para generar *hardware* sintetizable de forma automática.

No obstante, todas estas soluciones conllevan un enorme desaprovechamiento de las posibilidades que ofrece el *hardware* de propósito específico; por ello en este proyecto se ha optado por un implementar el sistema desde cero, en *hardware* puro.

De esta manera es imprescindible considerar que al trabajar con *hardware* el diseño maneja un flujo de datos, sobre el cual se aplican simultáneamente distintas operaciones, las cuales corresponden a distintas etapas del proceso. Por el contrario, los diseños *software* consisten en un flujo de instrucciones, las cuales deben ejecutarse individualmente sobre los datos de entrada.

Podemos comparar de esta manera una implementación *hardware* con una cadena de

montaje de automóviles (veáse la figura 2.7a), donde cada coche es sometido a la vez a distintas operaciones; no obstante, continuamente (y sin mucha demora) salen nuevas unidades de la fabrica, a pesar de la complejidad del proceso.

Por otra parte, el símil para el *software* sería la elaboración de una receta de cocina (figura 2.7b): a cada ingrediente se le aplica una serie de operaciones de una en una (cortar, batir, pelar...), para después ir mezclándolo con el resto hasta conseguir el plato final. En este caso no importa que las tareas sean simples y cotidianas, porque igualmente cocinar lleva bastante tiempo.

(a) Cadena de montaje del Ford Modelo T.

(b) Cocinar requiere tiempo...

Figura 2.7: Diferencia ilustrada entre *hardware* y *software*. Fuente de ambas imágenes: Wikimedia Commons.

A continuación se describen brevemente las modificaciones más notables que ha sido preciso realizar (así como algunas consideraciones de diseño) para poder trasladar el algoritmo a *hardware*.

2.4.1. Distancia Manhattan

Para clasificar píxeles hiperespectrales (con múltiples bandas) se trabaja en varias dimensiones. De modo que será necesario medir la distancia entre dos puntos n-dimensionales: los píxeles y los centroides.

Para ello se ha empleado la conocida como distancia Manhattan. Esta consiste en calcular una por una las diferencias entre el valor de cada banda del píxel y el centroide a comparar. Posteriormente se sumarán todas las diferencias (una por banda) para así obtener la distancia entre pixel y clúster. Con ello ya sería posible determinar a qué clúster debe asignarse un pixel.

$$distanciaManhattan(\boldsymbol{a}, \boldsymbol{c}) = \frac{1}{n} \sum_{i=1}^{n} |a_i - c_i|$$

Esto es lo que se conoce como distancia Manhattan. Una alternativa sería la distancia Euclídea, la cual trabaja con el cuadrado de la diferencia. El uso de esta última de cara al proyecto ha sido descartado por el mayor coste computacional que supone. El artículo [20] discute y compara los efectos de ambas en el algoritmo k-means.

2.4.2. Aritmética y representación de valores

La aritmética en punto flotante permite operar con gran precisión en la parte decimal, sin embargo no es nada frecuente encontrarla en implementaciones *hardware* por el elevado uso de recursos que exige, así como la complejidad de su gestión. Es por ello que se prefieren soluciones de otro tipo, por ejemplo el punto fijo (FXP) o más recientemente la representación posit.

No obstante, con imágenes es perfectamente posible trabajar bajo la abstracción de valores enteros sin signo sin sacrificar precisión, de hecho las propias muestras emplean este formato para representar la información.

Por tanto, en este proyecto se ha empleado aritmética de enteros sin signo, prestando especial atención a las anchuras seleccionadas para la representación de los distintos tipos de datos. Asimismo ha sido necesario gestionar el problema del desbordamiento en las operaciones (veáse 2.5).

Además, gracias a que no es necesario almacenar valores con signo, se consigue ahorrar un bit a la hora de representar el valor de las muestras.

2.4.3. Sustitución de operaciones de división

Implementar la operación de división en *hardware* supone un altísimo uso de recursos, así como un consumo también elevado de tiempo (de ciclos de reloj) a la hora de procesarla. Por este motivo se han remplazado todas las divisiones simples por otras alternativas como desplazamientos de bits hacia la derecha. En un caso sí ha sido necesario utilizar divisores completos, como de describe más adelante 3.5 en la sección dedicada al módulo que realiza la última etapa del cálculo de centroides.

A costa de sacrificar precisión (que era totalmente irrelevante en los casos afectados por esta optimización), se obtiene un rendimiento mucho mayor, además de que el impacto en el uso de recursos es ínfimo.

2.4.4. Cálculo de valores acumulados

Para el cálculo de la suma de todos los componentes de un vector, en *software* se suelen emplear bucles de instrucciones. Para lograr una implementación *hardware* de dicho tipo de operación se suele recurrir a estructuras de sumadores "en árbol", como puede verse en la figura 2.8.

Este tipo de solución exige emplear múltiples sumadores en lugar de reutilizar uno solo, pero a cambio el rendimiento es muy superior.

2.4.5. Operaciones multibanda

Para que el diseño pueda procesar píxeles enteros directamente, también es necesario implementar módulos aritméticos específicos para ello; pues cabe recordar que al tratarse de imágenes hiperespectrales, cada píxel contiene múltiples bandas.

Esencialmente, dichos módulos constan de una serie de unidades aritméticas que operan en paralelo, de forma sincronizada e independiente para cada banda.

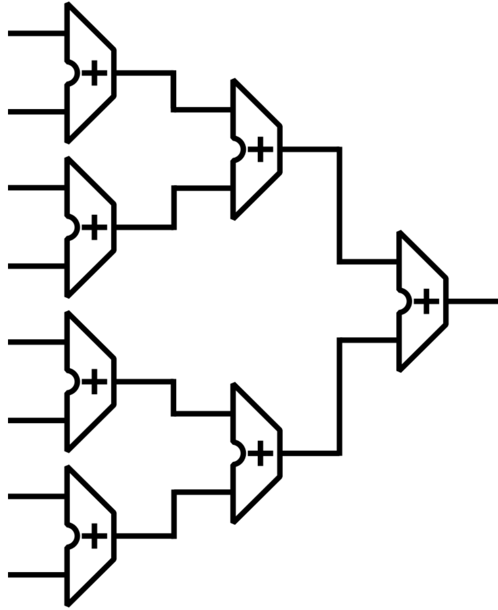


Figura 2.8: Árbol de sumadores.

En particular, se han diseñado módulos multibanda para los sumadores/restadores (*full-adder*), divisores y generadores de números aleatorios.

Esto es necesario a causa de la alternativa de implementación seleccionada, que consiste en procesar la imagen a razón de un ciclo de reloj por píxel. Los motivos que han impulsado a tomar esta decisión sobre el diseño están expuestos más adelante, en el capítulo 3 dedicado a la arquitectura.

2.4.6. Inclusión de un módulo de control

Para orquestar el funcionamiento del sistema, así como gestionar su interacción con la entrada/salida, ha sido necesario integrar un módulo dedicado exclusivamente a ello.

Este emplea una máquina de estados finitos (FSM, Finite State Machine) para controlar la sucesión de iteraciones, dado que el *hardware* se reutiliza en cada iteración. También se encarga de actualizar las métricas que monitorizan el rendimiento del algoritmo.

2.4.7. Establecimiento de un sistema de parada

Para garantizar que el algoritmo termina en algún momento, se ha introducido un sistema que anticipa la convergencia y finaliza el proceso en un menor número de iteraciones.

Consiste en el cálculo de un umbral de movimiento de píxeles (fijado al tres por ciento del total de pixeles de la imagen) y un contador de iteraciones (que limita el máximo número de iteraciones que pueden suceder al doble del número de clústers).

2.4.8. Realimentación rápida de centroides

Por cuestiones de rendimiento, en *software* no es viable actualizar constantemente los centroides de los clústeres, por ello la imagen se procesa en bloques (puede consultarse el código en detalle en la sección 2.3.1). En cada uno de ellos se evalúa cierto número de píxeles para posteriormente actualizar los centroides, tal que el siguiente bloque a procesar pueda servirse de dicha información en lugar de trabajar con datos demasiado desactualizados.

La modificación realizada consiste en prescindir totalmente del sistema de actualización por bloques y reemplazarlo por un mecanismo de realimentación capaz de actualizar los centroides cada vez que se procesa un pixel. Así se acelera notablemente la convergencia del algoritmo, al aumentar la precisión con la cual se clasifican los píxeles.

2.4.9. Preprocesado de imágenes hiperespectrales (reducción dimensional)

Teniendo en cuenta la naturaleza del algoritmo k-means y las imágenes hiperespectrales, se ha optado por realizar una preparación previa de los datos de entrada para maximizar la precisión y el rendimiento.

Por un lado, partiendo de que el espectro de radiación electromagnética capturado en cada pixel de la imagen hiperespectral es continuo (es decir, que las bandas están equiespaciadas), tendremos que las bandas próximas entre sí presentaran valores muy similares. Sin embargo, en algunos casos pueden aparecer manchas o cambios bruscos en su valor.

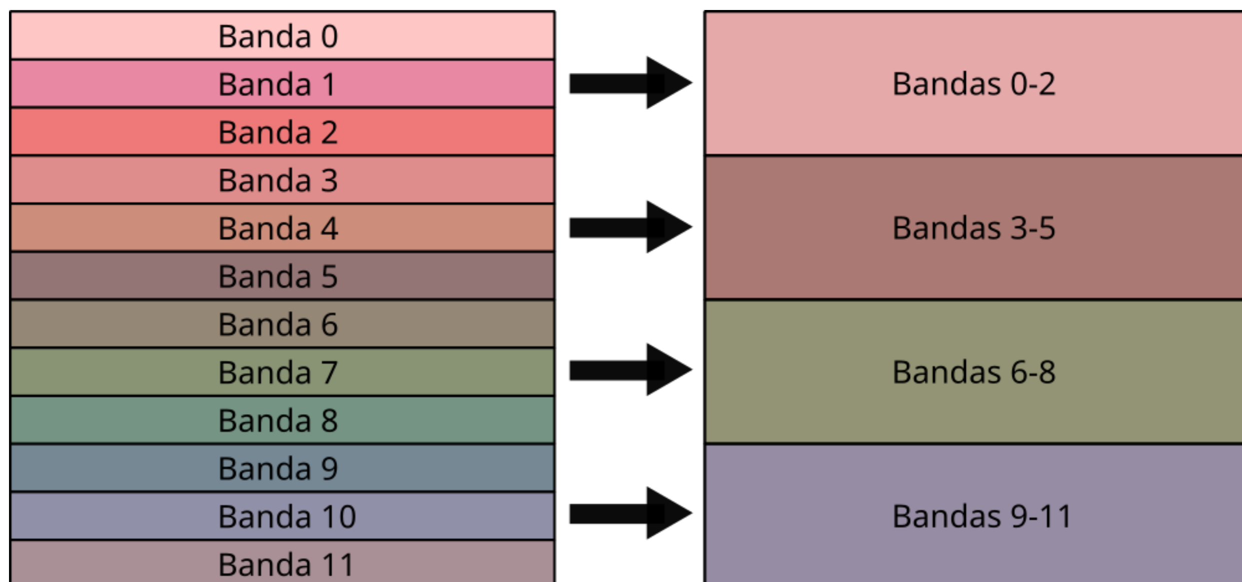


Figura 2.9: Ejemplo de reducción dimensional.

Por otra parte, lo que buscamos clasificar en la imagen son regiones comunes a todas las bandas, por lo que no necesitamos estrictamente la información individual de cada banda. No obstante, es habitual que los sensores hiperespectrales capturen cientos de bandas (224 en caso del AVIRIS), lo cual afecta negativamente al tiempo y recursos empleados para procesar las imágenes.

Entonces, combinando los dos hechos expuestos, surge la idea de fusionar grupos de bandas de la imagen (la figura 2.9 ilustra el concepto): así se consigue suavizar los valores de las muestras, logrando mayor precisión al analizar los datos, pues se mitigarán las desviaciones. Y además al disminuir el número de bandas se facilita la manipulación de las imágenes y se reduce notablemente el uso de recursos *hardware* en la FPGA.

2.5. Desbordamiento

En algunas operaciones aritméticas puede ocurrir que el resultado sea un valor más grande (o más pequeño) de lo que puede representarse con el número de bits empleado. Este fenómeno se conoce como desbordamiento y en caso de suceder durante la ejecución del algoritmo, puede traer consigo consecuencias catastróficas, anulando por completo la validez

del resultado obtenido. Por suerte existen diversas formas de mitigarlo, en este proyecto se ha recurrido a dos de ellas: saturación y ajuste de anchura de las muestras.

2.5.1. Saturación

Consiste en analizar el resultado de la operación aritmética en cuestión (ya sea suma, resta, multiplicación...) y verificar si se ha producido desbordamiento. En caso afirmativo, el resultado será reemplazado por el máximo valor representable, cuando el desbordamiento se produzca por exceso; o el mínimo valor representable, cuando el desbordamiento se produzca por defecto.

No obstante, debe tenerse presente que esta medida acarrea limitaciones asociadas a la pérdida de precisión y abusar de ella no soluciona el problema del desbordamiento. De cara a este proyecto también hay valores que son especialmente sensibles y nunca deben saturarse; por ejemplo, los contadores.

La operación puede expresarse matemáticamente como sigue:

$$sat(x) = \begin{cases} l_i, & x \leq -l_i \\ x, & -l_i \leq x \leq l_s \\ l_s, & x \geq l_s \end{cases}$$

2.5.2. Ajuste de anchura

Otra alternativa más precisa para salvar el problema del desbordamiento consiste en calcular matemáticamente cuál es el máximo o mínimo valor que se necesita representar, considerando todas las posibilidades que pueden darse en los datos de entrada. En función del resultado obtenido se establece la anchura necesaria para darle cabida.

Sin embargo no siempre es posible recurrir a esta alternativa, dado que en algunas situaciones no es posible calcular el máximo o mínimo valor que se va a presentar en la salida. Por ejemplo, al calcular el valor acumulado de un numero indeterminado de muestras. Otra limitación aparece ligada a los recursos *hardware* disponibles, pues no siempre se dispone de capacidad para instanciar registros u operadores del tamaño deseado en todo el diseño.

La anchura (número de bits) necesaria para representar un valor (x) puede calcularse de la siguiente forma:

$$anchura(x) = \lfloor log_2(x) \rfloor + 1$$

2.6. Arquitectura en *pipeline*

A pesar de todas las adaptaciones citadas en el apartado anterior, sigue siendo necesario un soporte en el cual implementar la funcionalidad del algoritmo. Un chasis en el cual integrar todos los componentes, donde se realizan las operaciones a las cuales se someterá la imagen hiperespectral recibida. Este componente no es otro que la arquitectura, que en este caso consiste en un *pipeline*.

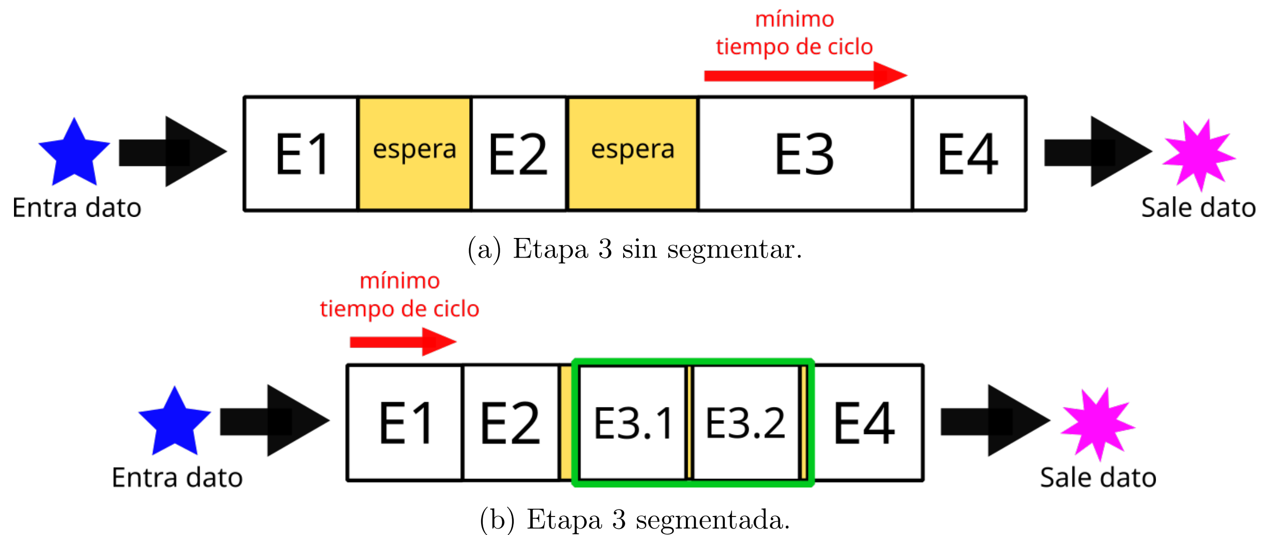
Un *pipeline* es esencialmente un conjunto de elementos de procesado de datos conectados en serie, tal que la salida de cada módulo constituye la entrada del siguiente. Nótese que este tipo de arquitectura encaja perfectamente con el paradigma de diseño hardware (basado en un flujo de datos) descrito en la sección 2.4.

Volviendo al símil con la cadena de montaje de automóviles, en el *pipeline* habrá simultáneamente distintos datos (los coches) fluyendo a través de cada etapa, pasando de una a otra. Cada una de ellas podría compararse con instalar el motor, acoplar la transmisión, colocar la columna de dirección... etcétera; tal que todas dependen entre sí y deben llevarse a cabo en el orden establecido.

Dado que el *pipeline* permite que en cada una de sus etapas se este procesando un elemento, cada vez que avance la cadena se obtendrá un dato en la salida. Esto es mucho más rápido que esperar a que cada dato sea completamente procesado antes de introducir el siguiente.

En las fábricas, la etapa más lenta condiciona el tiempo total del montaje del coche, pues

es necesario esperar a que sea completada para poder avanzar en la cadena. Exactamente lo mismo ocurre en el *pipeline*, así que para mitigar el problema, típicamente se recurre a segmentar internamente los módulos en varias etapas más simples. El objetivo principal es evitar a toda costa las paradas en el flujo de datos y no demorar en más de un ciclo de reloj el paso a la siguiente etapa. Lo que condiciona la duración de las etapas del *pipeline* son los caminos combinacionales presentes en ellas.



(a) Etapa 3 sin segmentar.

(b) Etapa 3 segmentada.

Figura 2.10: Ilustración comparativa sobre la segmentación en un *pipeline*.

En la figura 2.10 puede observarse la diferencia entre un mismo *pipeline*, con su tercera etapa (la más larga) segmentada (2.10b) y sin segmentar (2.10a). La flecha roja indica el tiempo mínimo de ciclo, que está condicionado por la etapa más larga, y que se ve notablemente reducido al segmentar.

Gracias a la segmentación es posible reducir el tiempo de ciclo (es decir, tener ciclos de reloj más cortos) y así aumentar el rendimiento del sistema. También debe tenerse en cuenta que por el contrario, la segmentación excesiva supone *pipelines* con mayor número de etapas y por tanto con mayor latencia (de modo que transcurrirá más tiempo entre la entrada y la salida de un mismo dato). Además incrementar la frecuencia de reloj (reducir el tiempo de ciclo) supone un aumento lineal en el consumo energético, acorde a la siguiente fórmula de la potencia dinámica (donde f es la frecuencia):

$$P_D \propto V^2 \cdot f$$

Por ello es muy relevante diseñar cada etapa de un *pipeline* sin perder de vista el conjunto global para así obtener un sistema lo más óptimo posible, bien equilibrado y con buena escalabilidad (para que sea fácil extenderlo en un futuro).

2.7. Array sistólico

Se conoce por array sistólico a una red homogénea de elementos de procesamiento de datos (también llamados nodos o *cores*). Cada uno de los nodos realiza una parte del cómputo y envía su salida al siguiente. Al final del array sistólico se obtiene el resultado, después de que los datos de entrada hayan atravesado todos los nodos. Este paradigma de funcionamiento guarda un similitud notable con el del *pipeline* (descrito en la sección anterior), por ello su integración dentro del sistema resulta bastante limpia y conveniente.

Además, este tipo de estructura presenta una excelente escalabilidad, debido a que todos los nodos que lo componen son idénticos y para ampliarlo basta con incorporar otros nuevos en la cadena.

La figura 2.11 muestra un ejemplo esquemático un array sistólico, con sus núcleos idénticos conectados en cadena.

En este proyecto se ha empleado un array sistólico unidimensional en el módulo clasificador (descrito en la sección 3.3) para determinar a qué clúster pertenece un pixel recibido a la entrada. En este caso, añadir nuevos nodos al array sistólico permite aumentar el número de clústers a clasificar en la imagen.

2.8. Bus AXI

AXI (*Advanced eXtensible Interface*) es un protocolo de transferencia de información destinado a la comunicación interna en diseños *hardware*. Permite conectar módulos entre sí de forma estandarizada, para garantizar la compatibilidad entre distintos diseños, *IP soft*

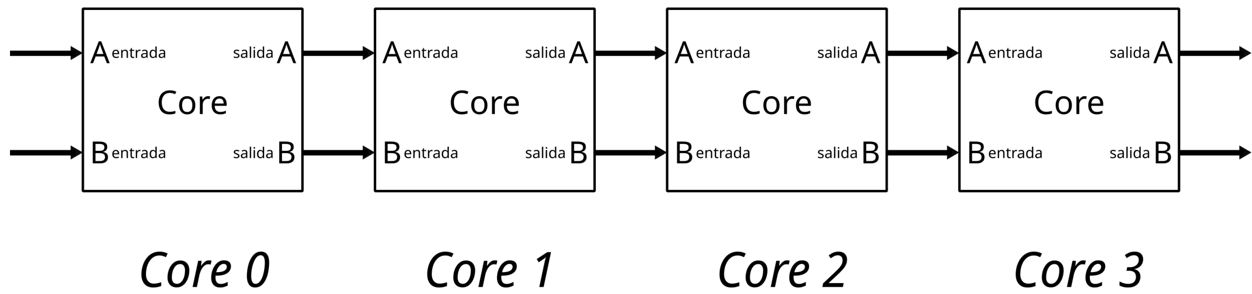


Figura 2.11: Ejemplo de array sistólico.

cores (véase la sección 2.10), interfaces externas, memorias, etcétera. Además de facilitar las implementaciones y favorecer la reutilización de los diseños, esto resulta especialmente útil para coordinar las peticiones y envíos de datos, así como sincronizar las transferencias de información.

La especificación del mismo es desarrollada por la empresa ARM y su uso está libre de *royalties*, por lo que resulta bastante popular en el ámbito del *hardware* y en particular de las FPGAs. Para su empleo en el proyecto se ha consultado la especificación oficial [21].

Con vistas a evitar el sobrecoste de emplear AXI en comunicaciones simples, existe una serie de subconjuntos de la especificación completa (es decir, versiones reducidas de AXI): AXI-Lite y AXI-Stream.

2.8.1. AXI-Stream

En concreto, el protocolo de comunicación que se ha utilizado entre los módulos de este proyecto (así como de cara al exterior) es AXI-Stream. Este prescinde del sistema de peticiones de datos y direccionamiento que utilizan tanto AXI como AXI-Lite para comunicaciones mapeadas en memoria y se enfoca en transmisiones unidireccionales punto a punto. Por otra parte, también es muy eficiente ya que las transferencias se realizan a razón de un dato por ciclo.

Considerando las recién citadas características, AXI-Stream resulta idóneo para la arquitectura en *pipeline* que da soporte a los distintos componentes del sistema.

A continuación se presenta una relación de las señales que intervienen en un bus AXI-

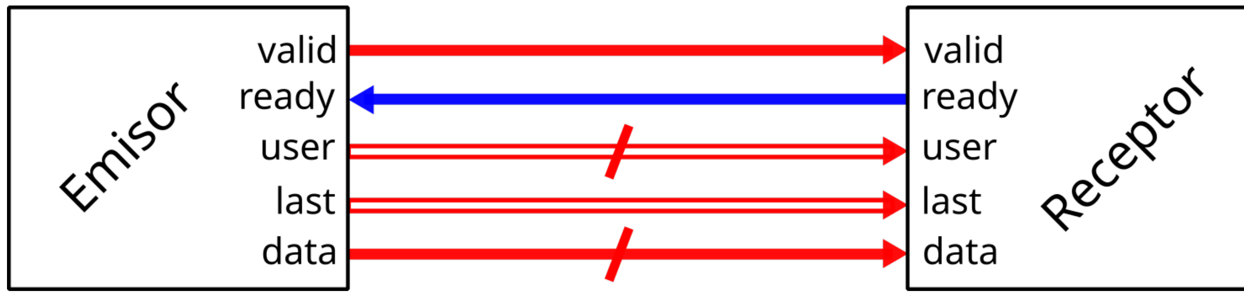


Figura 2.12: Dos módulos conectados mediante AXI-Stream.

Stream. Las señales de control (*valid, ready, last*) siempre hacen referencia al ciclo de reloj actual.

- ***Valid***: Bit que indica la validez del dato presente a la salida del emisor.
- ***Ready***: Bit que indica la disposición del receptor para recibir el dato.
- ***User***: Señal opcional definida por el usuario, sirve para para transmitir información específica de la implementación. Su anchura también la define el usuario.
- ***Last***: Para transmisiones en ráfaga, es un bit que indica la finalización del envío del dato. Su uso es opcional y está sujeto a las características específicas de la implementación.
- ***Data***: Contiene la información a transmitir, su anchura depende de la implementación.

La figura 2.12 ilustra una comunicación entre dos módulos a través de AXI-Stream.

2.9. Generación de valores aleatorios

La generación de valores aleatorios en *hardware* supone un problema moderadamente complejo de resolver, puesto que internamente el comportamiento es totalmente determinista (salvo que se produzcan errores, lo cual constituye otro problema en sí mismo) y no siempre se dispone de una fuente externa para alimentar la aleatoriedad.

Un ejemplo de implementación consistiría en emplear un conversor analógico-digital (ADC) que capture el ruido del ambiente (analógico) y obtener de ahí el valor digital correspondiente, que debería ser razonablemente aleatorio. Sin embargo, esta solución incrementa la complejidad y el coste del sistema, además no siempre se dispone de los medios necesarios para llevarla a cabo.

Por ello, cuando la obtención de un valor aleatorio no es absolutamente crucial (como lo sería, por ejemplo, en los juegos de azar de un casino), se suele recurrir a generadores pseudoaleatorios. Estos suelen partir de un valor inicial: la semilla, el cual es sometido a un proceso para obtener a continuación una sucesión de valores distintos.

De esta forma, los generadores pseudoaleatorios siempre proporcionaran la misma salida (en distintas ejecuciones) ante una misma semilla, lo cual en caso de este proyecto resulta bastante ventajoso, ya que facilita inmensamente las tareas de validación y depuración.

2.9.1. LFSR

La solución *hardware* a la que se ha recurrido para la generación de aleatorios es un LFSR (acrónimo del inglés *Linear-Feedback Shift Register*). Como el propio nombre indica, consiste en un registro de desplazamiento en el cual algunos de sus bits emplean para realimentar su entrada, pasando por una función lógica, habitualmente XOR o XNOT.

La semilla del LFSR la constituye su estado inicial, es decir, el valor que presenta a su salida antes de comenzar a funcionar.

No obstante, el principal inconveniente de los LFSR es que los valores en la salida se repiten de forma cíclica, pasando por todas las posibles combinaciones. De igual modo, cuanto mayor sea la anchura del registro, más tiempo tardarán en repetirse los valores.

En la figura 2.13 está representado esquemáticamente un LFSR de 16 bits.

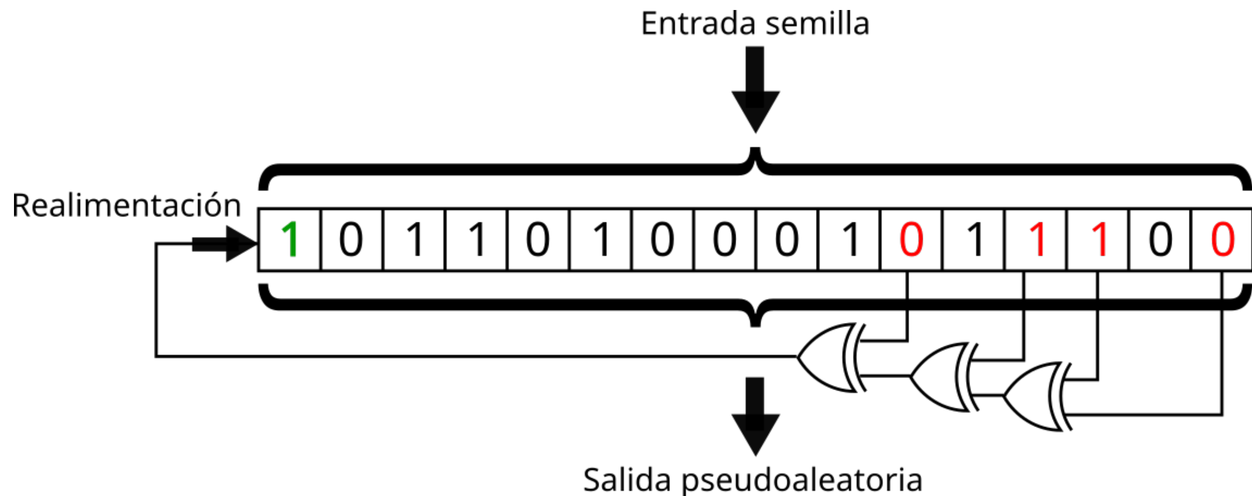


Figura 2.13: LFSR de 16 bits.

2.10. *IP soft cores*

Aunque los *IP cores* no constituyan ningún componente material, tampoco dejan de ser módulos *hardware*: se trata de diseños de circuitos cargables en la FPGA. Sus siglas IP provienen de *Intelectual Property*, puesto que constituyen módulos cerrados (no es posible ver su diseño) y típicamente son proporcionados por los fabricantes, quienes se encargan de diseñarlos, verificarlos y distribuirlos.

Estos módulos agilizan en gran medida multitud de tareas; por ello en este proyecto se ha recurrido a los siguientes, todos desarrollados por *Xilinx Inc.*.

2.10.1. *Divider Generator*

Este *IP soft core* permite realizar la operación aritmética de división en *hardware*. Resulta especialmente útil puesto que las implementaciones de dicha operación son moderadamente complejas y llevarlas a cabo supone un esfuerzo considerable, así como consumir una cantidad elevada de tiempo en verificar su correcto funcionamiento.

Para configurar las distintas instancias del *Divider Generator* e integrarlas en el sistema se ha recurrido al manual proporcionado por el fabricante [22].

2.10.2. *Integrated Logic Analyzer*

Depurar *hardware* es una tarea desesperante y laboriosa que requiere ingentes cantidades de paciencia; además exige que por diseño, el sistema este preparado para depuración. Esto puede lograrse con mecanismos específicos para extraer el contenido de los registros o utilizando buses específicos que permitan leer el estado del *hardware* en momentos concretos.

Por suerte, para FPGAs existe un *IP core* que permite leer algunas señales del sistema en momentos concretos e incluso monitorizarlas durante algunos ciclos de reloj: el *Integrated Logic Analyzer* (o también llamado ILA, por sus siglas).

Asimismo, con él es posible depurar los diseños una vez cargados en la FPGA a través de una interfaz *software*, la cual recibe los datos desde la FPGA para imprimirlos por pantalla.

Sin embargo, presenta numerosas limitaciones, ya que solo es posible capturar las señales conectadas al ILA mediante la activación de uno de los disparadores previamente configurados. Además, como ya he mencionado, solo es posible capturar una reducida ventana de tiempo (como máximo y únicamente bajo ciertas condiciones, son 1024 ciclos de reloj). Una vez completada la captura debe procederse a analizar los datos obtenidos manualmente.

De cara a la detección de errores también está limitado, pues funciona asociado a la frecuencia de reloj del sistema y por ello no es posible analizar completamente el comportamiento de las señales (por ejemplo para identificar rebotes o *glitches*).

Para la configuración e integración del ILA en el sistema, se ha tomado como referencia el *datasheet* proporcionado por el fabricante [23].

2.11. Herramientas *software*

En esta sección se describen las herramientas *software* empleadas a lo largo de todo el proceso de desarrollo del proyecto, incluyendo pruebas, análisis, diseño, implementación, depuración, validación de resultados y elaboración de esta memoria.

El principal sistema operativo empleado para ejecutar las herramientas mencionadas en este apartado es Manjaro KDE (versión 21.2), basado en el kernel Linux.

2.11.1. Implementación *software* del algoritmo

En las primeras etapas del desarrollo se realizó una implementación *software* del algoritmo, para la cual fue empleado Python en su versión 3.9. En complemento, fueron utilizadas las siguientes bibliotecas:

- **Numpy**: Para operar con arrays y matrices de datos de forma eficiente.
- **SpectralPython**: Para la carga en memoria de imágenes hiperespectrales con distintos formatos.
- **PyLFSR**: Para generación de números aleatorios mediante un LFSR. Solo se empleó para pruebas, en la versión final se optó por una implementación propia de mayor rendimiento.

Verificación de resultados

Estas mismas herramientas también se han utilizado también para implementar varios *scripts* que automatizan la verificación de resultados obtenidos o contienen utilidades como la preparación y conversión de datos, generación de imágenes de prueba, etcétera.

En conjunto, se ha utilizado la aplicación *LibreOffice Calc* para analizar algunos resultados y realizar cálculos rápidos.

2.11.2. Descripción *hardware*

La descripción *hardware* (a nivel RTL) ha sido realizada principalmente en VHDL (*Very High Speed Integrated Circuit Hardware Description Language*). Para los módulos funcionales y la arquitectura se ha empleado el estándar de 1993; mientras que para los *testbenches* de simulación se ha optado por el estándar de 2008, puesto que incluye algunas comodidades que resultan de utilidad [24].

Por otra parte, también se ha empleado el lenguaje System Verilog para la descripción de un tipo de memoria ROM. El motivo de ello se debe a la facilidad que presenta (respecto a VHDL) para inicializar la ROM desde un fichero [25].

2.11.3. EDA y simulación *hardware*

Las herramientas EDA (*Electronic Design Automation*) asisten al diseño de circuitos integrados a través de un computador. En particular la EDA empleada para la realización de este proyecto es Vivado (en su versión 2020.2), desarrollada por Xilinx.

De este modo, se ha empleado Vivado para realizar la síntesis e implementación de los diseños *hardware*. La síntesis es el proceso por el cual se obtienen diseños en términos de componentes interconectados a partir de las descripciones a nivel RTL (de transferencia de registro). Esta traducción va ligada a un complejo análisis que verifica la corrección del diseño a distintos niveles. Además somete las descripciones a una profunda optimización con vistas (entre otros) a ahorrar recursos, área (es decir, el espacio físico que ocupa el diseño), reducir caminos lógicos y satisfacer todas las restricciones.

Del mismo modo Vivado también incluye las herramientas necesarias para cargar el diseño elaborado en la FPGA.

Por otra parte, los *IP soft cores* (véase la sección 2.10) empleados en el proyecto también han sido obtenidos y configurados mediante Vivado.

Además, todas las labores de simulación *hardware* (previas a la implementación física) se han realizado igualmente con Vivado. Gracias a este paso previo es posible verificar el comportamiento del diseño sin necesidad de ejecutar la síntesis e implementación (pues requieren bastante tiempo), o incluso sin tener una FPGA disponible.

Igualmente, dicha herramienta ha servido para realizar todas las tareas de depuración del diseño, así como la extracción de resultados para su posterior validación.

Finalmente, Vivado también ha servido para obtener los análisis de tiempos, las estadísticas de utilización de recursos en la FPGA y la estimación de consumo energético del diseño.

2.11.4. Generación de documentación

Esta memoria ha sido escrita con Vim, un editor de texto libre y gratuito.

Se ha utilizado el sistema de composición tipográfica LaTeX para dar formato al texto y generar los documentos finales. Para la gestión de referencias bibliográficas se ha empleado BibTex.

Dichas herramientas han sido utilizadas a través de la aplicación TeXShop, que las integra en un mismo entorno gráfico.

2.11.5. Gráficos vectoriales

Para la elaboración de los gráficos presentes en esta memoria se ha recurrido al *software* libre de diseño gráfico vectorial Inkscape.

Dichos gráficos incluyen los esquemáticos del diseño, diagramas arquitectónicos, formatos de trama, ilustraciones de procesos, etcétera.

2.11.6. Control de versiones

Para llevar a cabo un control de versiones del proyecto se ha utilizado el sistema Git. Los repositorios se han alojado en el servicio en línea GitLab.

Por motivos de seguridad, se ha sometido a dicho control de versiones la totalidad del material generado como consecuencia de este trabajo. Esto incluye el código con la descripción RTL del proyecto, los ficheros de simulación, las implementaciones *software*, los *scripts* de verificación o el contenido de esta misma memoria con sus fuentes y figuras.

Intercambio de archivos

Para coordinar la revisión de la memoria entre alumno y tutores, se ha recurrido a Overleaf. Esta es una plataforma en línea que permite leer y editar de forma colaborativa documentos en formato LaTeX.

2.12. Herramientas *hardware*

Además de la placa de prototipado Xilinx VC709 (ya introducida en la sección 2.2.2), también se ha utilizado un computador doméstico en el desarrollo del proyecto. Este ha sido

necesario para ejecutar las herramientas *software* descritas en el apartado anterior.

El equipo en cuestión está equipado con microprocesador *Intel Xeon E5 v2* (arquitectura *Ivy Bridge*) y 32 GiB de memoria principal.

Capítulo 3

Arquitectura *hardware*

En el presente capítulo se abordan todos los detalles relativos al diseño *hardware* elaborado, incluyendo una introducción a sus características; una perspectiva general de la arquitectura, donde se muestra cómo está enfocado el *pipeline*; y la descripción de los módulos que conforman dicho *pipeline*.

Cabe remarcar que este sistema ha sido creado con vistas a facilitar su reutilización e integración en otras soluciones, por ello, tanto escalabilidad como adaptabilidad han estado muy presentes a la hora de diseñarlo. En consecuencia, el resultado final es altamente escalable y proporciona una gran flexibilidad a la hora de funcionar con imágenes de distinta resolución o número de bandas; del mismo modo, permite alterar las propiedades de la clasificación realizada. Todo ello sin sacrificar rendimiento[1] ni disparar el uso de recursos en la FPGA (para más información, consultar la sección 5.3).

En el repositorio `https://gitlab.com/hw-kmeans` se encuentra la descripción RTL completa del proyecto, junto con los distintos recursos empleados para la simulación *hardware*, implementaciones *software*, *scripts* de validación... etcétera.

[1]Respecto al planteamiento de procesar un píxel por ciclo.

Parámetros del diseño			
Característica	Nombre	Impacto en rend.	Por defecto
Núm. de bandas	N_BANDS	No	10 bandas
Anchura de las muestras	SAMPLE_LEN	No	16 bits
Núm. máximo de píxeles	N_MAX_PIXELS	No	Según la imagen
Núm. clústers a identificar	N_CLUSTER	Sí	Según la imagen

Tabla 3.1: Parámetros del diseño.

3.1. Características y parámetros del diseño

Algunos aspectos del núcleo son paramétricos, lo cual significa que es posible adaptarlo a distintos requisitos de funcionamiento por medio de modificar ciertos valores constantes de la descripción en HDL. Todos ellos están listados en la tabla 3.1.

También hay otros parámetros, cuyo valor es autoconfigurable a partir de los primeros, estos quedan recogidos en la tabla 3.2.

3.1.1. Parámetros manualmente configurables

Número de bandas y anchura de muestras

A la hora de configurar dichos parámetros conviene tener en cuenta diversas consideraciones. En primer lugar, modificar el número de bandas o la anchura de la imagen repercute en infinidad de componentes dentro del núcleo, por ello al aumentar dichos valores, también incrementará el empleo de recursos en la FPGA. Sin embargo, en ningún caso supondrá una penalización en el rendimiento del sistema, gracias a su diseño *hardware* capaz de procesar en paralelo todas las bandas de cada píxel. Este hecho pone de manifiesto la excelente escalabilidad horizontal del núcleo.

Número máximo de píxeles

El número máximo de píxeles (por imagen) que puede procesar el sistema es igualmente configurable. Esto sirve para garantizar la precisión del algoritmo, por medio de evitar la

Parámetros autoconfigurables del diseño		
Característica	Nombre	Función
Anchura de acumuladores	ACCUMULATOR_LEN	$\log_2(2^{\texttt{SAMPLE_LEN}}*\texttt{N_MAX_PIXELS})$
Anchura de contadores de píxeles	PIXELCOUNT_LEN	$\log_2(\texttt{N_MAX_PIXELS})$
Anchura de identificadores de clúster	CLUSTER_LEN	$\log_2(\texttt{N_CLUSTER})$
Anchura de la distancia	DISTANCE_LEN	$\log_2(2^{\texttt{SAMPLE_LEN}}*\texttt{N_BANDS})$

Tabla 3.2: Parámetros autoconfigurables del diseño.

saturación (véase sección dedicada al problema del desbordamiento 2.5). Al igual que en el caso anterior, este parámetro no afecta al rendimiento, aunque sí al uso de recursos de la FPGA.

Cabe remarcar que es perfectamente posible operar con imágenes más grandes de lo establecido por dicho "número máximo de píxeles" y en muchos casos es una muy buena opción para ahorrar recursos, ya que la autoconfiguración (tabla 3.2) está diseñada para garantizar la precisión en el peor caso posible, aun siendo probable que este no llegue a tener lugar. Incluso en caso de suceder, los mecanismos de saturación mitigarían el problema.

Número de clústers a identificar

Por otra parte, el "número de clústers a identificar" configuran el comportamiento del algoritmo para que identifique en los datos de entrada tantas regiones como sea le especificado. Esto impacta en el uso de recursos por la necesidad de añadir (o eliminar) procesadores al array sistólico del clasificador (ver secciones 2.7 y 3.3). Al contrario que en los casos anteriores, alterar el número de clústers a identificar repercute también en el rendimiento, dado que teóricamente serán necesarias más iteraciones del algoritmo hasta obtener el resultado. No obstante, el *pipeline* seguirá procesando a la misma velocidad.

3.1.2. Parámetros autoconfigurables

Los parámetros autoconfigurables, reflejados en la tabla 3.2, pueden ser calculados a partir de los parámetros citados en la sección anterior. Igualmente, es posible configurar de

forma manual este tipo de parámetro, a pesar de ser totalmente innecesario.

En la misma tabla 3.2 se muestra la función empleada para obtener el valor de cada parámetro. Sin embargo hay una limitación importante a tener en cuenta: algunos de estos valores afectan al *IP Core* empleado para la división (consúltese la sección 2.10.1). Por dicho motivo resultará imprescindible reconfigurar el *IP Core* y el módulo divisor que calcula los centroides (en concreto su parámetro de latencia); para más detalles, en la sección 3.5.1 se describe el funcionamiento del divisor multibanda.

3.2. *Pipeline* y vista general del diseño

Como ya se ha introducido en capítulos anteriores, el objetivo principal de este trabajo es desarrollar un núcleo *hardware* que implemente el algoritmo k-means.

A modo de repaso, conviene recordar que k-means es un algoritmo iterativo, que recorre la imagen varias veces, y va asignando a cada píxel el clúster con el centroide más cercano. Cuando un píxel cambia de clúster (se mueve de uno a otro), se recalcula el centroide de ambos clústers (al que va y del que procede). Finalmente, al cesar el movimiento de píxeles el algoritmo concluye su operación.

En fases previas al diseño se valoraron múltiples opciones de implementación, sin embargo todas ellas consisten fundamentalmente en un *pipeline*, pues al tratarse de un algoritmo iterativo es posible reutilizar el mismo *hardware* en cada iteración. A continuación se listan las alternativas que fueron valoradas para la implementación del núcleo:

- **Procesar un píxel por ciclo**: Es el método más rápido. Todas las bandas de los píxeles se tratan en paralelo, mientras que el *pipeline* segmentado procesa a la vez un píxel en cada etapa. También requiere más recursos que los otros métodos.

- **Procesar una muestra por ciclo**: En este método las muestras de los píxeles se procesan de forma secuencial. Esto supone que para una imagen de 200 bandas tardará 200 veces más que en el caso anterior, aunque también permitiría elevar la frecuencia

de reloj (al no haber operaciones de reducción en el *pipeline*). Por otro lado, también reduce el uso de recursos muy notablemente respecto al primer método.

- **Procesar varias muestras por ciclo**: Otra opción sería establecer un término medio entre las dos anteriores, procesando varias muestras en paralelo. No obstante, la complejidad de la implementación se vería incrementada en gran medida.

- **Procesar un centroide por ciclo**: Al igual que en la primera, en esta alternativa todas las bandas de los píxeles se procesan en paralelo. Sin embargo, aquí sería posible trabajar con centroides totalmente actualizados a costa de sacrificar rendimiento y complicar el control.

Tras valorar cuidadosamente todas las opciones, se optó por desarrollar la primera de la lista: procesar un píxel por ciclo. El principal motivo que propició esta decisión es que se trata de la opción más rápida con diferencia y no se han encontrado publicaciones o registros de implementaciones anteriores que alcancen semejante rendimiento.

El inconveniente del uso de recursos se ha solventado por medio de aplicar una reducción dimensional a los datos de entrada (cuyo fundamento está descrito en la sección 2.4.9). La principal ventaja de esta técnica es que apenas se sacrifica precisión en los resultados.

Por otra parte, la desventaja respecto a la opción de procesar un centroide por ciclo, fue totalmente desestimada porque el ahorro de recursos no justifica la pérdida de rendimiento ni el aumento de complejidad del control.

Los detalles sobre el diseño final del *pipeline* pueden consultarse a continuación, en el apartado 3.2.4.

3.2.1. Vista general

En la figura 3.1 se muestra una vista simplificada de los componentes del núcleo. Contorneado por una línea discontinua se encuentra el *pipeline* con sus tres módulos segmentados, que son orquestados por el componente *Main Control*.

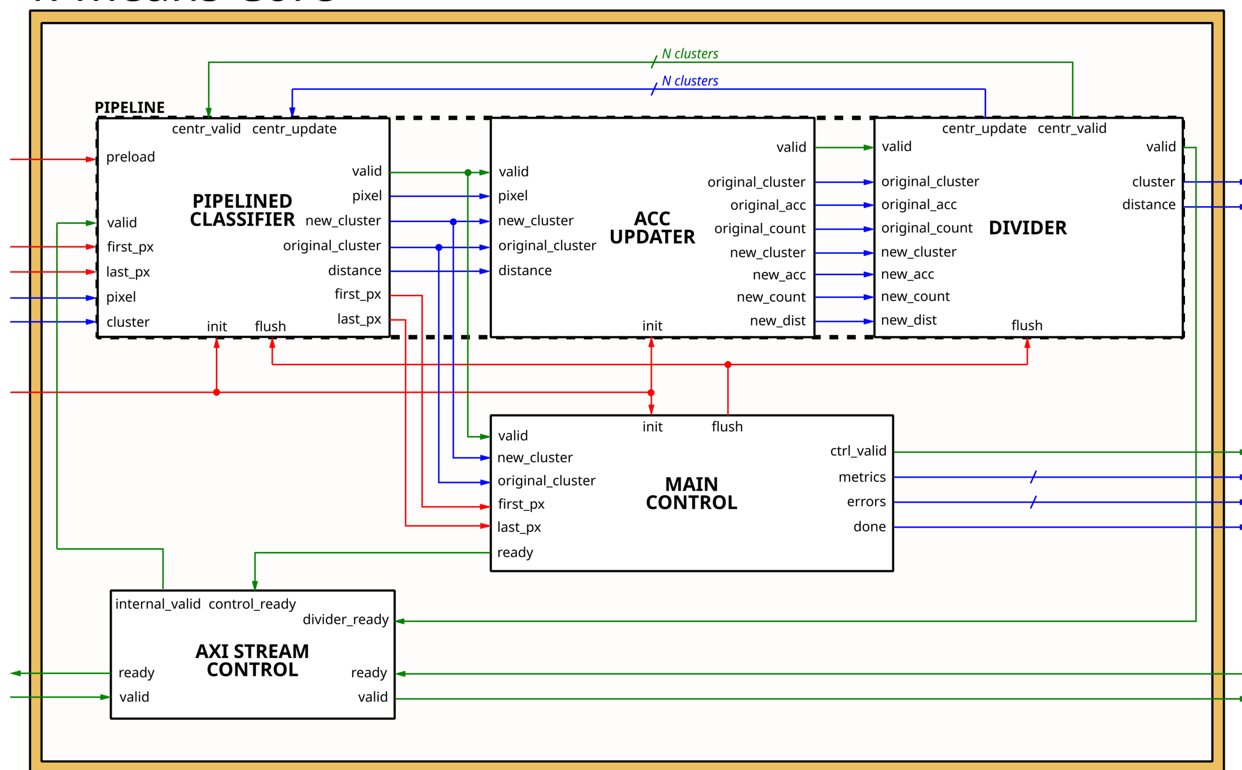


Figura 3.1: Vista general de la arquitectura del núcleo de procesamiento.

Téngase en cuenta que con vistas a facilitar la lectura de esta memoria todos los diagramas han sido simplificados. Por ese motivo, en ellos se han omitido algunas señales, incluyendo las destinadas al reloj o al *reset*. De este modo se pretende centrar la importancia en el flujo de datos a través del núcleo y las señales fundamentales de control, los cuales están siempre presentes en los diagramas y las descripciones que los acompañan.

El primer módulo del *pipeline* es el *Cluster Classifier*, que asigna a cada píxel el clúster más cercano; después *Acc Updater* recalcula el valor de los acumuladores de los clústers en base a los cambios establecidos en la etapa anterior; finalmente el *Divider* realiza la división entre el valor acumulado y el número de píxeles, para así calcular el centroide y retroalimentar al *Cluster Classifier*.

Main Control es el módulo que determina el estado del algoritmo y monitoriza si se ha alcanzado la convergencia. Va conectado a la salida del *Cluster Classifier* ya que solo necesita conocer los movimientos entre píxeles, así se logra obtener la salida antes, al no tener que esperar a que los datos recorran todo el *pipeline*.

Por otra parte, el *AXI Stream Control* gestiona la comunicación con el exterior del núcleo mediante el bus AXI Stream (descrito en la sección 2.8). Puesto que todo el *pipeline* está diseñado para funcionar de forma continua, a velocidad de un píxel completo por ciclo de reloj, es este módulo el que gestiona también las posibles paradas; puesto que en caso de producirse alguna, será únicamente provocada por un componente externo al núcleo (por ejemplo, un controlador de memoria que tarde varios ciclos en proporcionar un dato).

3.2.2. Comunicación entre módulos

Los distintos módulos del núcleo se conectan mediante señales de tres tipos: sincronización, datos y control.

En primer lugar, las señales de datos se utilizan tanto para introducir en el sistema la información a procesar, como para transferir los resultados totales o parciales de las operaciones a las que se van sometiendo. Algunos ejemplos (pueden observarse en la figura 3.1,

marcadas en azul) son las señales *cluster*, *pixel*, *new_acc* o *centr_update*, entre muchas otras.

Para sincronizar las transferencias de información, se emplean señales etiquetadas como *valid* (en el diagrama 3.1 aparecen en color verde) que utilizan el mismo paradigma que AXI Stream. Es decir, que mientras estén activas, el dato a la salida es válido.

No obstante se ha prescindido de las señales *ready* de AXI Stream, puesto que por diseño los módulos siempre están listos para recibir datos a la entrada. En cualquier caso, sí se utilizan señales *ready* para la comunicación con el exterior del núcleo. De este modo, si se produce algún bloqueo por causas externas, el núcleo esperará a que se libere el bus antes de continuar emitiendo información.

El otro tipo de señal, las de control, envían pulsos para notificar eventos y administrar el comportamiento de otros módulos. Algunas de ellas (en rojo en la figura 3.1) son *preload*, *init*, *flush*...

3.2.3. Cese de operaciones

Una vez el algoritmo ha finalizado, es necesario vaciar el *pipeline*. Pues si vuelve ponerse en marcha para otra imágen de entrada, los datos residuales de la ejecución anterior (que siguen marcados como válidos) se presentarían en la salida nada más empezar.

Además de emitir resultados erróneos, esto originaría problemas de sincronización entre el núcleo y el módulo que reciba su salida.

Por ello existe la señal *flush*, gestionada por el módulo principal de control, que permite limpiar los valores antiguos que han quedado "atrapados" dentro del *pipeline*.

3.2.4. Rendimiento del *pipeline*

Como ya se ha introducido, el *pipeline* es capaz de funcionar sin paradas, procesando un píxel por ciclo. Esto es posible gracias a que todos sus módulos están segmentados internamente, y por tanto, dentro de cada uno de ellos se procesan al mismo tiempo varios píxeles.

No obstante, hasta llenar al completo todas las etapas del *pipeline* no se obtendrá un dato a la salida del mismo. El tiempo que transcurre desde que entra el primer dato hasta obtener el primer resultado es lo que se conoce como latencia.

De forma desglosada por módulos, el retardo del *pipeline* es el siguiente:

- ***Pipelined Classifier***: un ciclo por cada clúster a detectar.
- ***Acc Updater***: siempre un ciclo.
- ***Divider***: depende de la anchura de los acumuladores y por ende, de la configuración del *IP Core* divisor. Para una imagen de 100 megapíxels, tardaría 43 ciclos; mientras que para otra de 1 megapíxel, tardaría 30.

Por ejemplo, para detectar 7 clústers en una imagen de 16 megapíxeles, el *pipeline* tendría un retardo de 48 ciclos (7 del *Pipelined Classifier*, 1 del *Acc Updater* y 40 del *Divider*); lo que es un coste absolutamente despreciable para una imagen con dieciséis millones de píxeles, dado que se procesa un píxel por ciclo. De hecho, cuanto mayor sea el tamaño de la imagen, menor impacto en el rendimiento supondrá el coste de inicialización del *pipeline*.

La figura 3.2 muestra el cronograma correspondiente al *pipeline* configurado para clasificar 5 clústers en imágenes de hasta 1.000 píxeles de 10 bandas, con muestras de 8 bits. Nótese que hay tantos píxeles procesádose a la vez como etapas tiene el *pipeline* segmentado (26 en total). También se aprecia visualmente la ausencia de paradas en la ejecución y la salida constante de un píxel procesado por ciclo.

Actualizaciones de los centroides

Asimismo es muy importante considerar otro factor derivado de este retardo que induce el *pipeline*: el desfase de las actualizaciones de los centroides. Dado que el último dato en salir es el valor calculado de un centroide, los píxeles no serán clasificados en base a información completamente actualizada (puesto que aún se está calculando). En la figura 3.3 se ilustra el

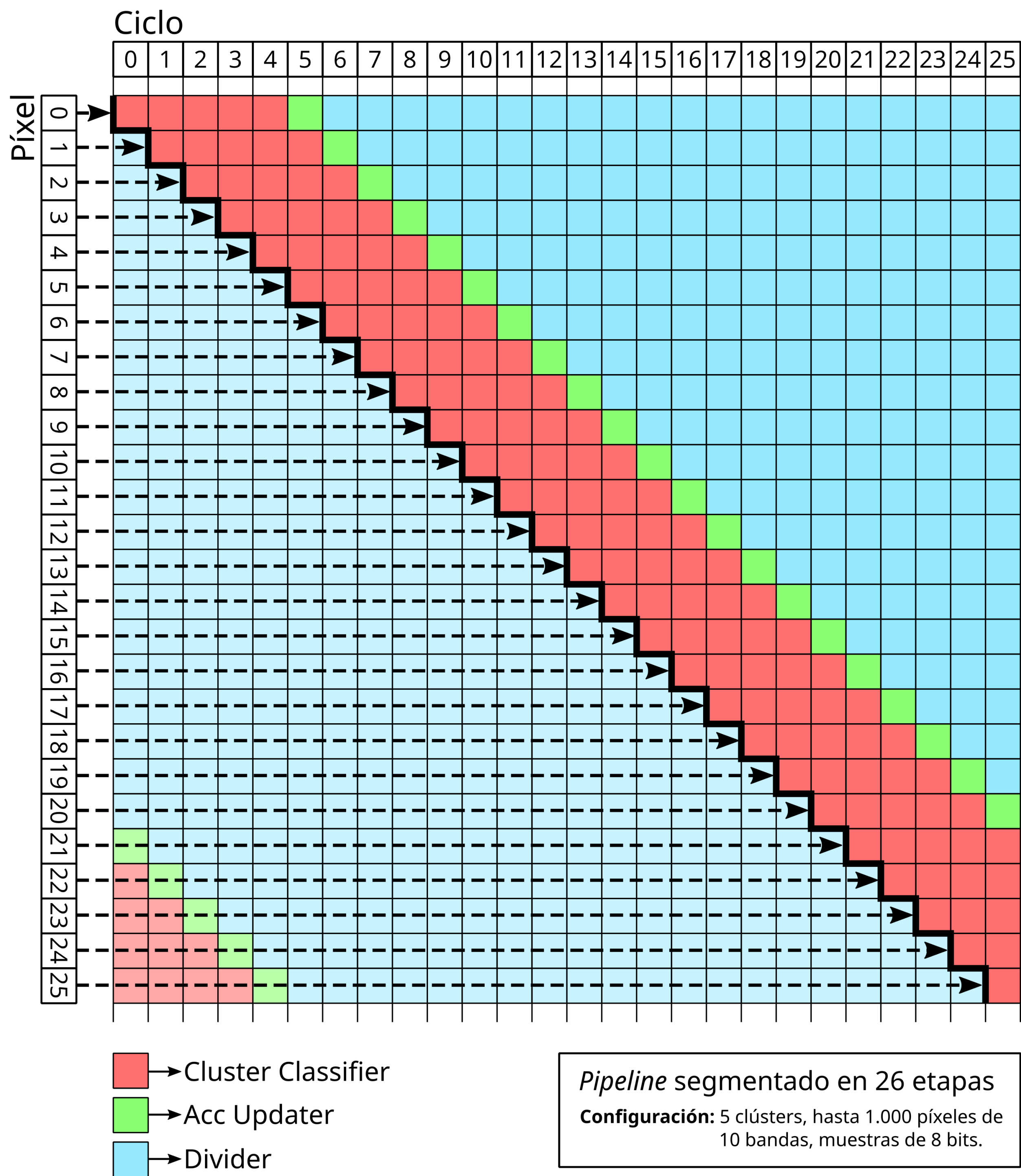


Figura 3.2: Cronograma del *pipeline*.

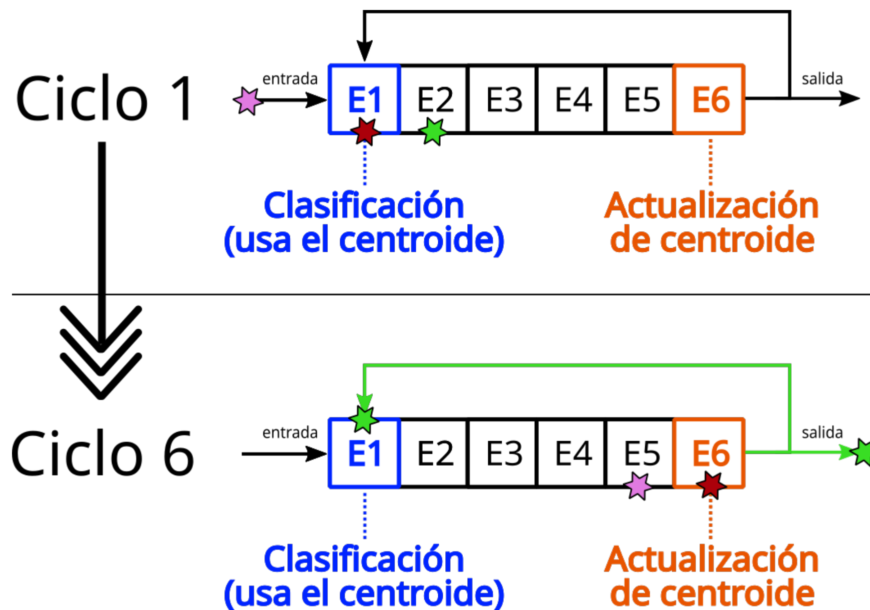


Figura 3.3: Ilustración con el problema del desfase en la actualización del centroide.

problema: hasta varios ciclos después no se actualiza el centroide, mientras siguen entrando datos que son procesados utilizando valores antiguos.

Este desfase en la actualización debe ser lo más pequeño posible, para que el algoritmo alcance la convergencia más rápido. Por ello, la implementación *hardware* de k-means (y en particular este diseño en *pipeline*) supone una gran mejora respecto a sus homólogos en *software*; en estos últimos el procesado se realiza por bloques (de un número prefijado de píxeles) para mitigar los sobrecostes asociados al acceso a memoria. Por contra, este diseño *hardware* recalcula los centroides por cada píxel procesado y apenas tarda unos ciclos en hacer efectiva la actualización.

3.3. Módulo de clasificación (*Pipelined Classifier*)

El módulo de clasificación (*Pipelined Classifier*) es el primero del *pipeline*. Es el encargado de asignar a cada píxel el clúster cuyo centroide se encuentre más próximo (al píxel).

La figura 3.4 muestra un esquema con la arquitectura del módulo. En la parte superior se aprecia el array sistólico, con tantos procesadores como clústers se desee obtener en la

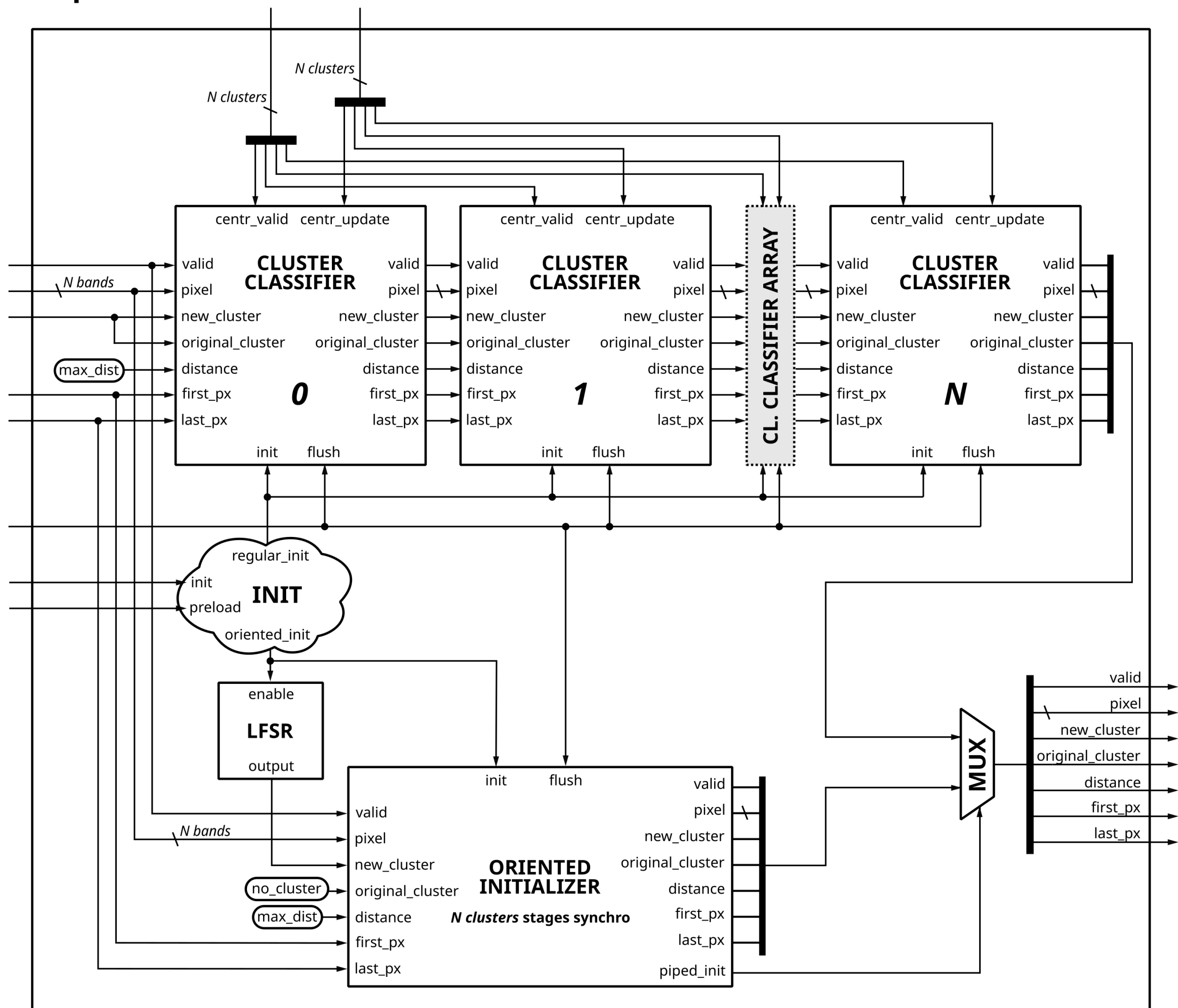


Figura 3.4: Arquitectura del clasificador segmentado.

clasificación de la imagen de entrada. Justo debajo del array se encuentra el sistema de inicialización orientada.

3.3.1. Array sistólico

El array sistólico está compuesto por múltiples procesadores idénticos, cada uno asociado a un clúster de los que se quieren identificar en la imagen. De esta manera, cada procesador almacena el valor del centroide de su clúster. En la figura 3.5 está representada

la arquitectura interna de los procesadores sistólicos.

A la entrada de cada procesador se recibe un pixel, para el cual se calculará su diferencia con el centroide almacenado. Como el píxel hiperespectral tiene múltiples bandas, la diferencia se calcula con un sumador multibanda (*Multiband Full-Adder*).

A continuación se calcula el valor agregado de la diferencia entre todas las bandas del píxel y el centroide, para obtener la distancia entre ambos expresada en un único valor. Esto es lo que se conoce como distancia de Manhattan.

Por otro lado también está el valor de distancia recibido a la entrada del procesador, el cual corresponde con el último clúster asignado al píxel. Ambas distancias (la recibida y la calculada) serán comparadas entre sí.

Si la nueva distancia es menor, significa que el clúster actual es el más próximo, por lo que la señal *new_ cluster* tomará como valor el identificador del clúster actual. Asimismo, *distance* representaría la nueva distancia calculada. Este proceso lo realizan los multiplexores y el comparador que se observan en la figura 3.5.

De este modo, al final del array cada píxel habrá sido comparado con todos los centroides, y por tanto asignado a aquel que le resulte más cercano.

Otro componente relevante en el procesador sistólico es el *Output Synchronizer* de una etapa (descrito más adelante en la sección 3.8.3), que en este caso sirve simplemente para registrar la salida. De no estar presente se formaría un camino combinacional a lo largo de todo el array, que probablemente derivaría en violaciones de temporización.

En el procesador sistólico también hay señales de entrada para la actualización del centroide, puesto que esta se realiza en una etapa más avanzada del *pipeline*.

3.3.2. Mecanismos de inicialización pseudoaleatoria

Inicialmente los clústers están vacíos, es decir, no contienen ningún píxel. Esto es un problema, ya que todos los centroides tendrán el mismo valor (cero) al principio. Esto puede causar que todos los píxeles se concentren en unos pocos clústers, lo cual derivaría en una

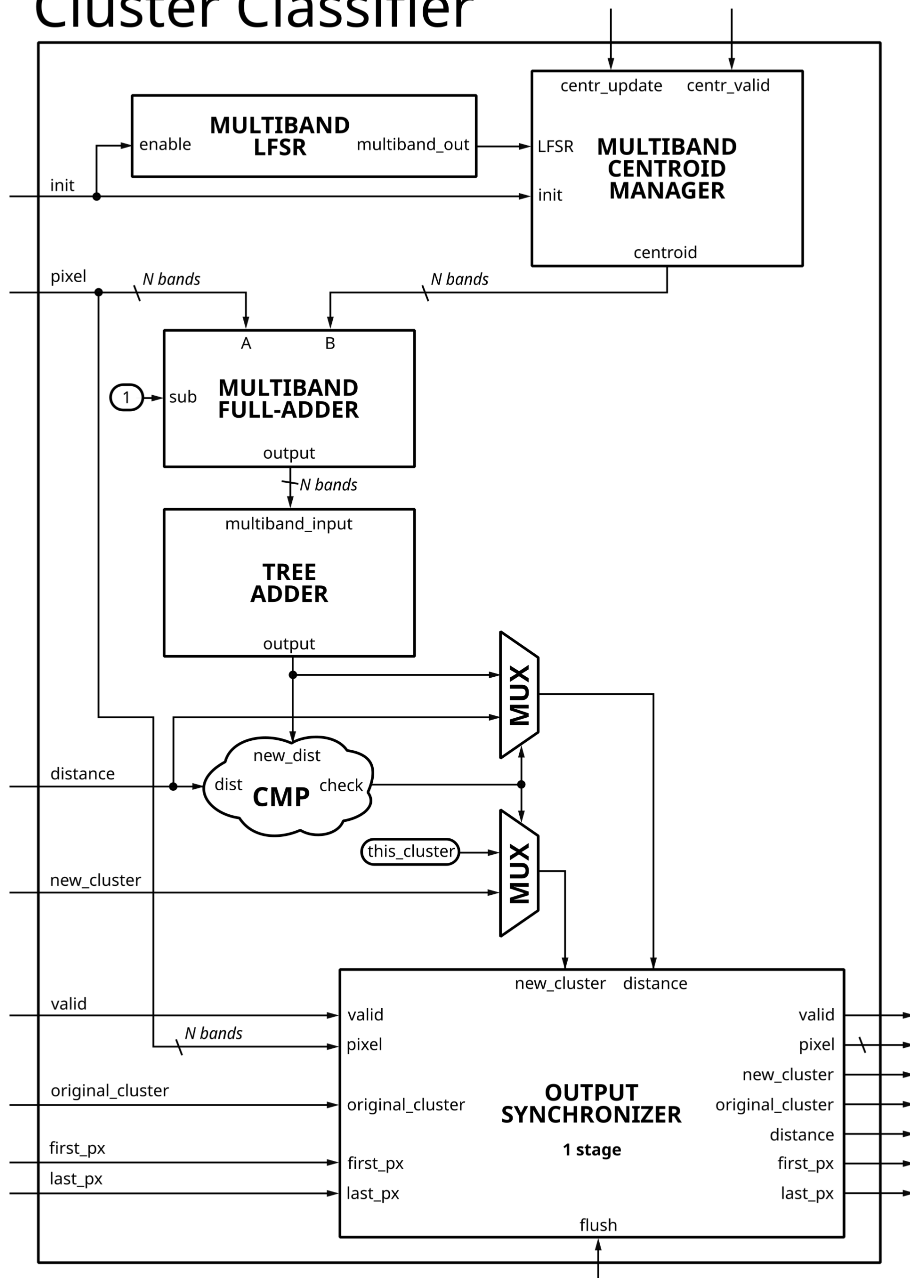


Figura 3.5: Arquitectura de los procesadores sistólicos del clasificador.

nefasta clasificación.

Por ello es imprescindible inicializar los centroides antes de comenzar a procesar los píxeles, así que se han desarrollado dos métodos distintos para dicha tarea: la inicialización orientada y la inicialización forzada. Cada uno resulta conveniente en función de los datos de entrada o lo que se pretenda conseguir en la salida.

Inicialización forzada

Consiste en emplear un módulo LFSR multibanda (descrito en la sección 3.8.2) para cargar valores aleatorios en los centroides. Las semillas utilizadas en cada módulo son distintas, para lograr que cada centroide inicializado aleatoriamente cubra una parte distinta del posible rango de valores.

El principal inconveniente se da si los datos de entrada tienen un bajo rango dinámico. En otras palabras, si los datos concentran su valor entorno a un rango de valores concreto. Esto podría provocar que la mayoría de los píxeles se concentren en unos pocos clústers.

Sin embargo, esta técnica permite en general obtener resultados más rápidamente en comparación con la inicialización orientada. Aunque en cualquier caso, el uso de este mecanismo está destinado fundamentalmente a labores de depuración o pruebas, ya que permite que los centroides partan de un valor conocido.

Inicialización orientada

Otra alternativa es realizar una "precarga" en la primera iteración. Esto consiste en introducir los píxeles en el sistema haciendo que la asignación de píxeles sea aleatoria, mediante un LFSR.

Así, los datos no se clasificarán de forma adecuada en la primera iteración, pero los acumuladores y centroides tendrán valores coherentes entre sí. Luego los cambios en el valor de los centroides serán menos bruscos que con la inicialización forzada (especialmente en las primeras iteraciones).

El principal contra de esta técnica es que exige realizar una iteración extra para la pre-

carga, sin embargo realiza una clasificación mucho más fina empleando todos los clústers que hayan sido configurados en el algoritmo. Esto último también puede afectar negativamente al tiempo de convergencia.

Para activar la inicialización orientada se emplea la señal *preload*, la cual debe mantenerse activa durante toda la primera iteración.

Para facilitar el cambio entre iteración de precarga e iteración corriente, se ha empleado un sincronizador de salida (ver figura 3.5) con tantas etapas como procesadores sistólicos haya. Es necesario porque al contrario que la clasificación, el proceso de inicialización orientada solo tarda un ciclo. Así se logra que el cambio entre una y otra sea totalmente transparente de cara al exterior del módulo clasificador, sin necesidad de paradas o esperas en el *pipeline*.

3.4. Módulo de acumuladores (*Acc Updater*)

Una vez el *Pipelined Classifier* haya calculado cuál es el clúster más cercano a un píxel, comienza el proceso de retirarlo del clúster al que pertenecía originalmente e incorporarlo al nuevo calculado.

Esto implica que se verán afectados los centroides del antiguo y el nuevo clúster, y por tanto deben ser actualizados. El proceso se realiza en dos etapas: actualizar acumuladores y contadores, y dividir el valor acumulado entre el número de píxeles. La primera de ellas (la actualización) tiene lugar en el *Acc Updater*, al cual está dedicado este apartado.

En la figura 3.6 se muestra la arquitectura del actualizador de acumuladores (*Acc Updater*). En su parte izquierda se observa una cadena de módulos *Cluster Acc Updater*, los cuales trabajan en paralelo. Cada uno de ellos está asociado a uno de los clústers a identificar en la imagen y la función que realizan es actualizar el valor acumulado y el contador de píxeles de cada clúster.

Por cada píxel procesado se realizarán entre cero y dos actualizaciones de centroides: cero si el píxel no cambia de clúster; una si el píxel no tenía un clúster asignado previamente (sucede en la primera iteración); o dos si el pixel sale de un clúster y entra en otro. Sin

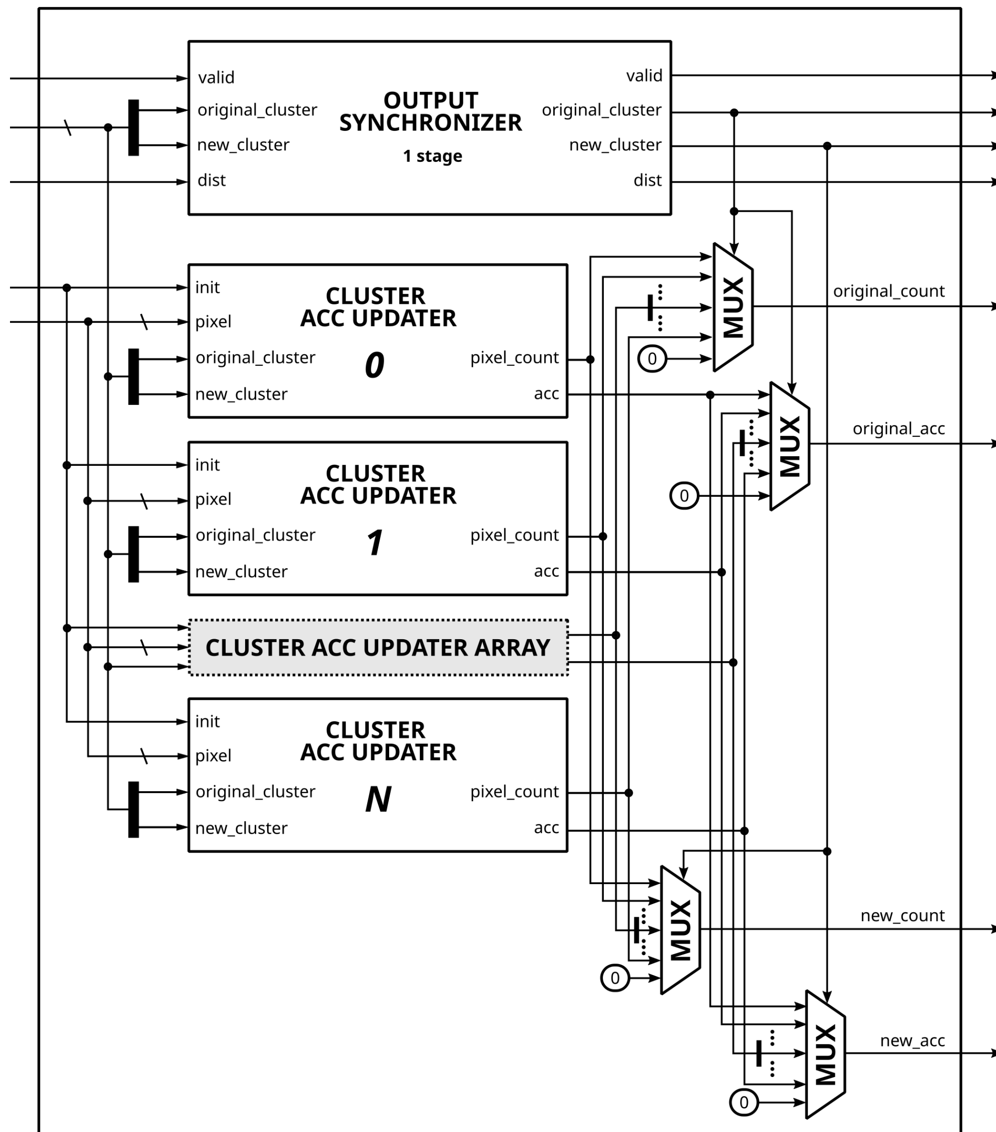


Figura 3.6: Arquitectura del actualizador de acumuladores.

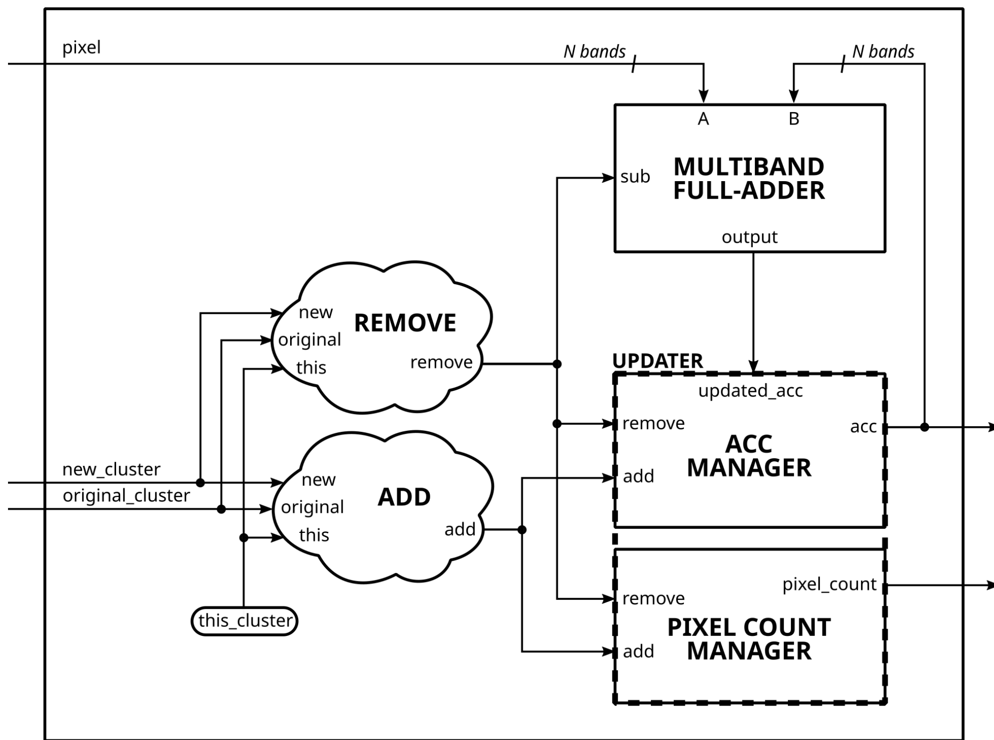


Figura 3.7: Arquitectura de un nodo del actualizador de acumuladores.

embargo, al haber múltiples actualizadores es imprescindible seleccionar cuál de ellos será el que manifieste su valor a la salida del *Acc Updater*. Esta tarea recae sobre los cuatro multiplexores que aparecen a la derecha (figura 3.6).

Al efectuarse en paralelo, la operación de actualización tarda un solo ciclo de reloj, por lo que algunas señales a propagar al siguiente módulo deben sincronizarse con los valores acumulados. Por ello, en la parte superior de la figura 3.6 hay un sincronizador de una etapa.

3.4.1. Actualizador de acumuladores y contadores (*Cluster Acc Updater*)

En la imagen 3.7 se muestra el interior de un *Cluster Acc Updater*. Se compone de dos submódulos actualizadores: uno para el acumulador y otro con el contador de píxeles. El primero de ellos está conectado a un sumador/restador multibanda, para añadir o eliminar

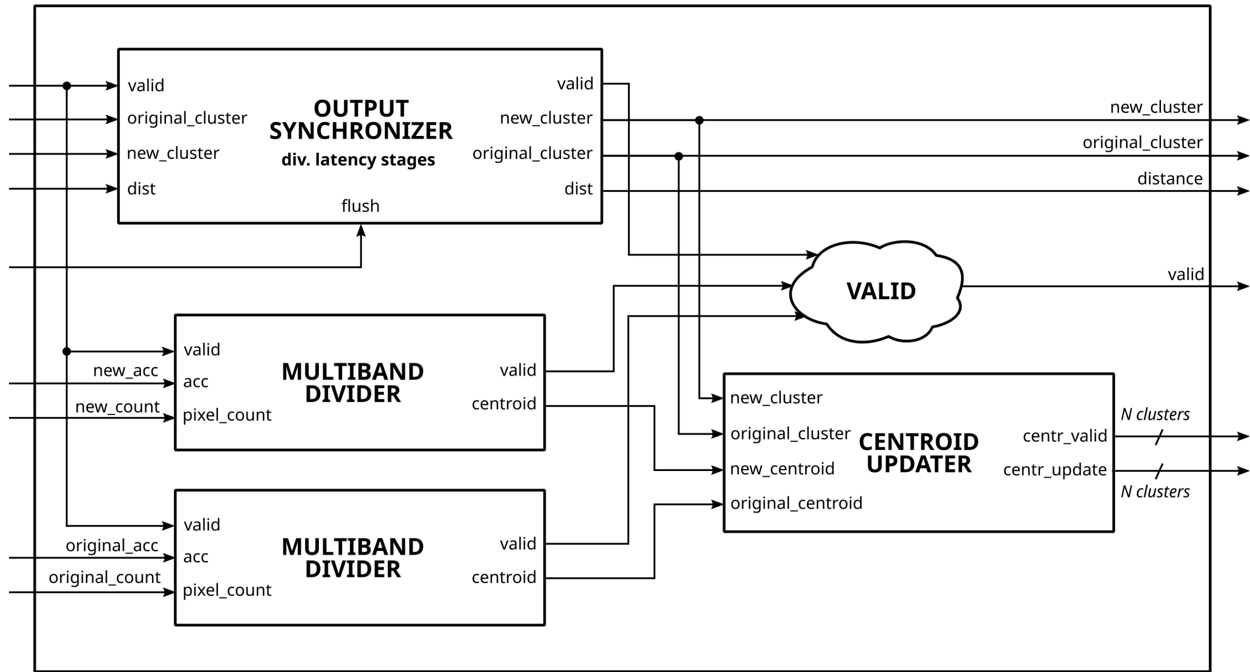


Figura 3.8: Arquitectura del módulo de división, que actualiza los centroides.

el valor del píxel recibido al total acumulado.

La actuación del módulo está gobernada por la lógica combinacional que hay en él, la cual determina si el clúster al que representa está implicado en la operación actual. Para ello comprueba si su clúster coincide con el nuevo o el antiguo clúster del píxel recibido, para respectivamente incorporarlo o eliminarlo al acumulador (y también registrarlo en el contador).

3.5. Módulo de actualización de centroides (*Divider*)

En la última etapa del *pipeline* se encuentra el módulo divisor, en el cual finaliza el cálculo de los centroides.

Para ello, el *Acc Updater* (descrito en el apartado anterior) proporciona el valor acumulado y el número de píxeles que hay en el clúster original y el nuevo. La tarea que realiza este módulo es dividir el valor acumulado entre el número de píxeles para obtener los centroides

actualizados.

Un esquema de la arquitectura del *Divider* se puede observar en la figura 3.8. En él se aprecian los dos divisores multibanda (*Multiband Divider*) que efectúan el cálculo de los centroides. Puesto que su operación puede tardar varias decenas de ciclos, es necesario sincronizar la salida de estos divisores con las otras señales del módulo; por este motivo hay un *Output Synchronizer* funcionando en paralelo con los divisores.

En este módulo también se encuentra el componente que propaga la actualización de los centroides al clasificador, el cual aparece etiquetado en el diagrama 3.8 como *Centroid Updater*. Dicho submódulo esta conectado directamente a todos los procesadores sistólicos del clasificador mediante las señales *centr_ update* y *centr_ valid*. También gestiona la validez de los centroides calculados, puesto que puede darse el caso en que haya que omitir la actualización; por ejemplo, si no se han movido píxeles o si se ha producido una división entre cero al vaciarse un clúster.

3.5.1. Divisores multibanda (*Multiband Divider*)

Tal y como muestra la figura 3.9, en el interior de los *Multiband Divider* hay un array de *IP Cores* de división. En total se instancian tantos divisores como bandas tengan los píxeles a procesar, todos ellos en paralelo.

Si un divisor encontrase algún error en el cálculo (en general será producido por dividir entre cero), la salida de dicha banda será reemplazada por cero. Está labor la desempeña el multiplexor situado a la derecha en el esquema del módulo.

Configuración del *Divider Generator 5.1*

No es en absoluto trivial la configuración de estos *IP Cores*, puesto que condiciona tanto el rendimiento final del núcleo como el uso de recursos *hardware*.

Por ello, conviene ajustar sus parámetros acorde a lo estrictamente necesario; lo cual no supone ningún problema de cara a su integración en el sistema, gracias a que este es paramétrico y ofrece una gran flexibilidad con las anchuras de los datos y el número de

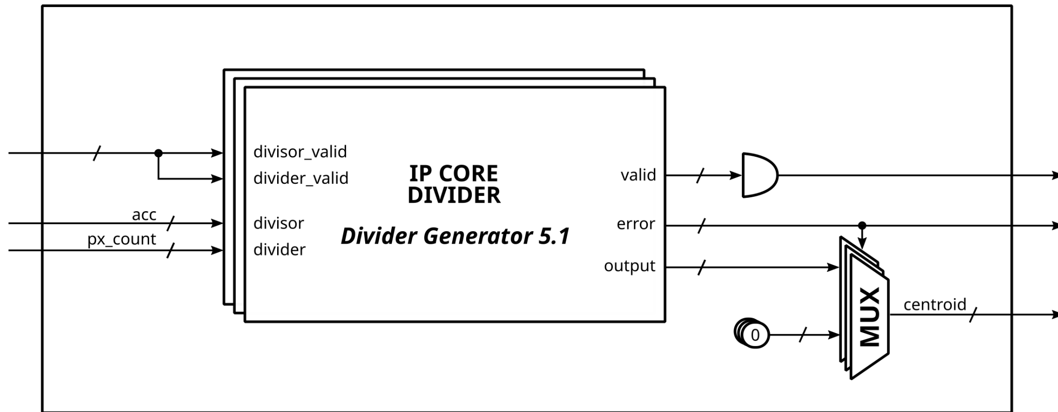


Figura 3.9: Arquitectura de un divisor multibanda.

etapas de los sincronizadores (que son los dos aspectos más relevantes en este caso).

En particular, el *IP Core* ha sido configurado para utilizar el algoritmo *Radix2*, que a pesar de su mayor latencia (no es un problema, puesto que funciona segmentado y no afecta al *throughput* del *pipeline*) destaca por su baja utilización de recursos en comparación con otras alternativas. Si bien es cierto que *LutMult* emplea menos recursos, más no es una opción válida porque no admite anchuras mayores a 17 bits en los datos de entrada.

La aritmética empleada es "sin signo", al igual que en (prácticamente) todo el núcleo.

En relación con las anchuras del dividendo y divisor, estas deben corresponder con las empleadas para representar las muestras de los acumuladores y los contadores de píxeles, respectivamente.

Dado que todas las operaciones se realizan con valores enteros, la salida del divisor debe configurarse para no trabajar con valores decimales, sino únicamente con el resto (*remainder*) de la división. Obtener el resultado en punto fijo carece de sentido, pues su parte decimal será desestimada (provocando así un uso de recursos innecesario).

Además, para reconocer posibles casos de división entre cero, se ha activado la opción *Detect Divide-By-Zero*.

Para igualar el rendimiento del *pipeline*, se ha configurado el número de ciclos por división a uno; de esta manera, el divisor procesará a razón de un píxel por ciclo. Asimismo, este *IP*

Core está segmentado, por lo que su integración en la cadena del *pipeline* es directa.

También se ha deshabilitado el control de flujo (seleccionando el modo *Non Blocking*). Este mecanismo permite que los datos de dividendo y divisor se presenten desincronizados, y emplea un *buffer* para volverlos a sincronizar. No obstante, dicha situación nunca va a suceder en este diseño, por lo que el control de flujo no aporta nada.

Ya en lo relativo a la interfaz del *IP Core*, se utiliza el mismo protocolo que en los otros módulos del núcleo: AXI Stream. No son necesarias señales de tipo *tlast*, puesto que no hay transferencias en ráfaga; ni *tready* ya que el *pipeline* está diseñado para funcionar sin paradas. Sí se utiliza una señal tipo *tuser* para propagar los errores de división entre cero que se citan en el párrafo anterior.

Finalmente, una vez se han ajustado todas las opciones del *Divider Generator*, su herramienta de configuración proporcionará la latencia resultante. Esta indica el número de ciclos que tardan los resultados en alcanzar la salida. Dicho valor es crucial para configurar el número de etapas del sincronizador que hay en el módulo *Divider* (parte superior de la figura 3.8), el cual fue descrito al principio de esta sección.

3.6. Módulo de control (*Main Control*)

El estado del *pipeline* es monitorizado por el módulo *Main Control*, para comprobar el estado del algoritmo y notificarlo al exterior.

En concreto, este módulo recibe la información sobre cómo se están clasificando los píxeles que entran al núcleo, la cual procede del *Pipelined Classifier*. Con ello contabiliza cómo se van sucediendo las iteraciones y mide el número de píxeles que se mueven de un clúster a otro; así identifica cuándo el algoritmo ha alcanzado la convergencia y debe detenerse. También puede detectar posibles errores.

3.6.1. Fin de la ejecución

Main Control considera que la ejecución del algoritmo ha concluido si se da cualquiera de las dos siguientes circunstancias:

- **Menos del 3,125 % de los píxeles se mueve en la última iteración.** Las pruebas realizadas primeramente en *software* revelaron que en las últimas iteraciones tendían a moverse solo los píxeles limítrofes entre clústers sin suponer ningún cambio sustancial en el resultado, pero alargando notablemente la ejecución (hasta lograr que no se moviese ningún píxel). Por este motivo se estableció un umbral (entre el 1 % y el 5 %, según el caso) para considerar que el movimiento de píxeles entre clústers había terminado.

  En primer lugar, para calcular el citado umbral en *hardware* es necesario un contador que permita conocer el número total de píxeles de la imagen. A continuación es necesario realizar una multiplicación o división para obtener el número de píxeles que conforman el umbral; sin embargo, ambas operaciones son demasiado costosas (especialmente la división) como para realizarlas en el mismo ciclo. Por ello se ha optado por la alternativa de desplazar cinco veces a la derecha el valor de la cuenta total de píxeles, de modo que se obtiene el 3,125 % de su valor original. Todo combinacionalmente y con un impacto nulo en la utilización de recursos. Como ya fue descrito en el capítulo 2, cada desplazamiento (de bits) a la derecha supone una división entre dos del valor original.

  Otra forma más sencilla de verlo es, que si se mueve menos de un píxel por cada 32 procesados, entonces se cumple la condición de parada.

- **Han sucedido más iteraciones que el doble del número de clústers.** Esta condición garantiza que el algoritmo termina, aunque el resultado este por encima del umbral descrito en el punto anterior. Usualmente, a mayor número de clústers a identificar, mayor es el número de iteraciones que realiza el algoritmo; de hecho, cuando

se ha iterado tantas veces como clústers hay, el número de píxeles que se mueven suele ser cercano al umbral del 3 %. Por ello, para ir sobre seguro e incrementar ligeramente la precisión, se ha establecido como límite de iteraciones el doble del número de clústers.

3.6.2. Detección de errores

El módulo de control cuenta con un sistema simple de detección de errores que permite identificar dos situaciones no deseadas: error de convergencia y error de cuenta.

La primera consiste en que tras una determinada iteración (distinta a la primera) se produzca más movimiento de píxeles que en la anterior. Esto se considera un "error de convergencia", pues indica que el algoritmo no está evolucionando hacia una situación estable. Puede estar provocado por una mala configuración del algoritmo, si los parámetros (véase la sección 3.1.1) presentan valores demasiado pequeños y se están saturando acumuladores o contadores por encima de los límites de funcionamiento. Otra posible causa sería si los datos de entrada no son coherentes entre distintas iteraciones, ya sea porque se hayan corrompido o desincronizado.

El otro tipo de error, el error de cuenta, tiene lugar al producirse una discrepancia entre el número de píxeles recibidos en distintas iteraciones. Para todas las iteraciones el número de píxeles debería coincidir siempre, dado que el tamaño de la imagen no cambia. En caso de darse este error, sería esperable que estuviese relacionado con una desincronización de los datos de entrada o un fallo en los indicadores *first_px* o *last_px*.

3.6.3. FSM de control

Todo el comportamiento de este módulo está definido mediante una máquina de estados finitos (FSM). Una versión simplificada de la misma puede consultarse en la figura 3.10.

La FSM de control tiene cuatro estados: *Idle*, *Iter*, *Iter Complete*, y *Done*.

- ***Idle.*** Es el estado inicial de reposo. En él los contadores se mantienen a cero a la espera de que lleguen datos válidos para procesar, en cuyo caso pasaría al estado *Iter*.

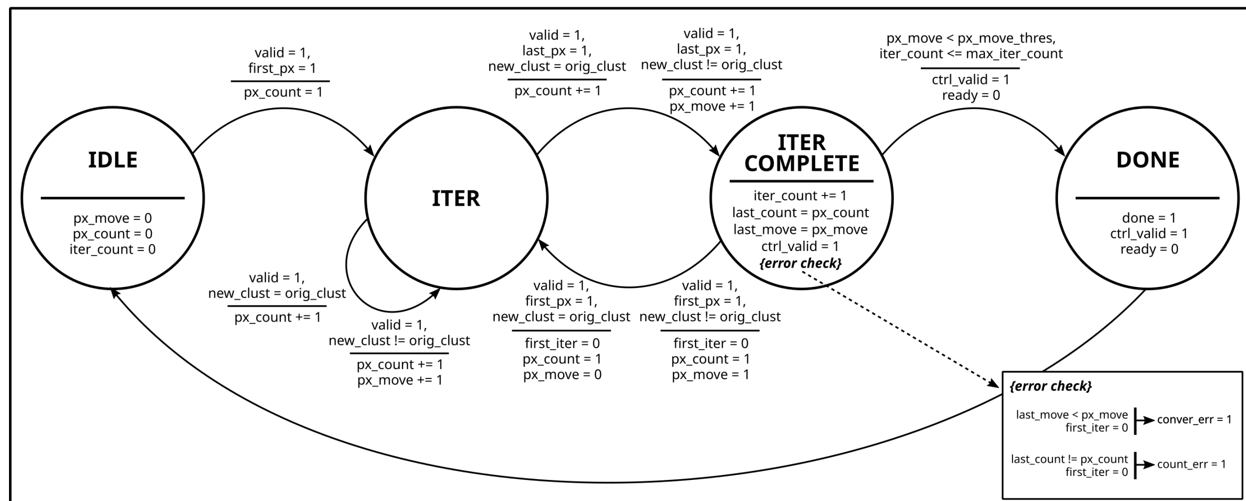


Figura 3.10: Máquina de estados finitos para el control del sistema.

- ***Iter.*** En este estado se permanece durante casi toda la iteración. Cada vez que llega un píxel se actualiza el contador de píxeles, y cada vez que hay un movimiento entre clústers se actualiza el contador de movimiento. Cuando finaliza la iteración (es decir, al llegar el último píxel) pasa al estado *Iter Complete*.

- ***Iter Complete.*** Al finalizar una iteración se comprueba si se ha producido algún error (de convergencia o de sincronización). También se incrementa el contador de iteraciones y se almacenan los contadores de píxeles y movimiento (para compararlos en la siguiente vuelta). En caso de cumplirse alguna condición de parada (ver 3.6.1) avanzará al estado *Done*, en caso contrario regresará a *Iter*.

- ***Done.*** Cuando el algoritmo termina se notifica al resto de módulos. Además dedica un ciclo de reloj a resetear los contadores, reiniciar el sistema y vaciar el *pipeline* (mediante la señal *flush*). Durante ese último ciclo la señal *ready* (del bus AXI) estará a cero. Acto seguido volverá al estado *Idle*.

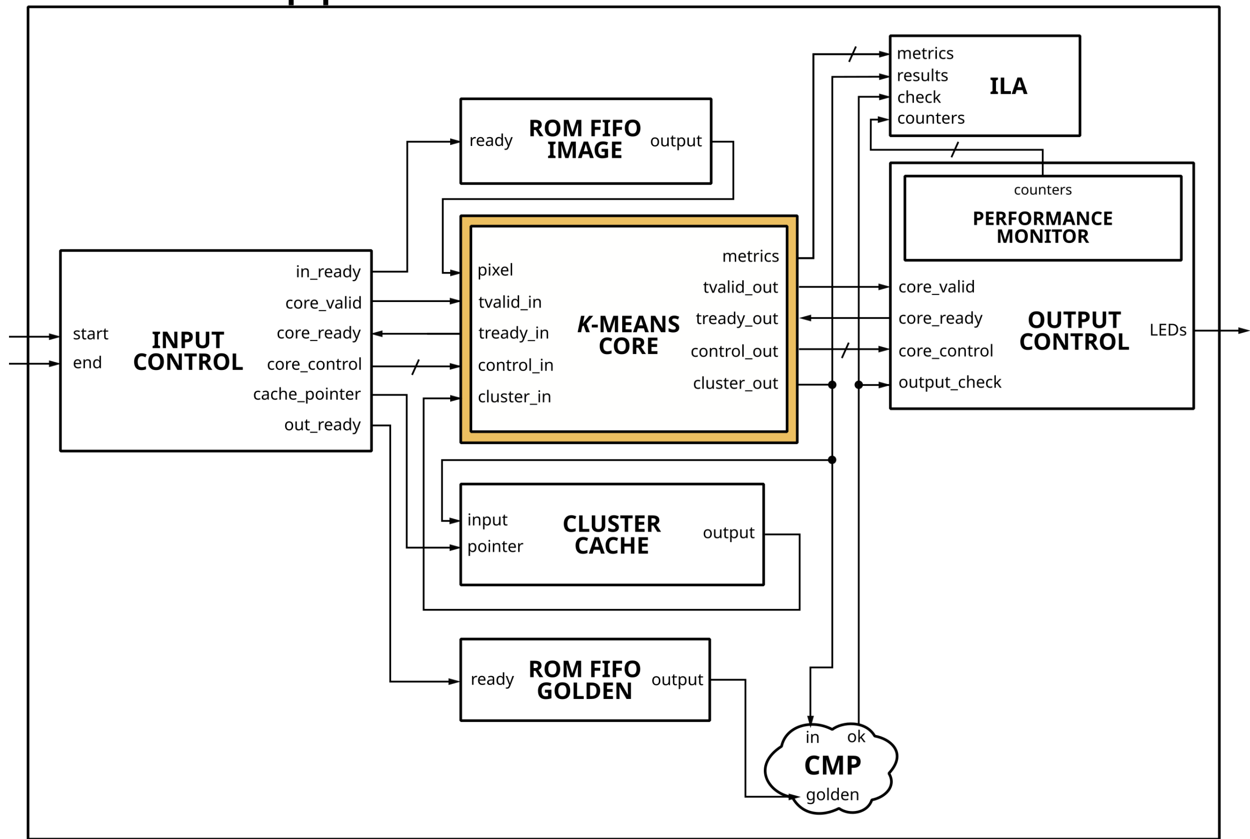


Figura 3.11: Arquitectura de la interfaz con FPGA.

3.7. Envoltorio elemental para pruebas en FPGA

Este proyecto no está orientado a crear un sistema independiente, sino un núcleo que funcione como un coprocesador o como acelerador en la clusterización de imágenes hiperespectrales. Por este motivo, para poder probar su funcionamiento en una FPGA es necesario instanciarlo dentro de un envoltorio o interfaz que le proporcione los datos de entrada y los estímulos necesarios para funcionar.

Con ese mismo propósito se ha diseñado la interfaz de la figura 3.11. Esta no solo proporciona los datos de entrada y los estímulos al núcleo, sino que también verifica la validez de su salida y monitoriza su comportamiento durante la ejecución, a través de un analizador lógico integrado (ILA).

Entrando en detalles, este módulo cuenta con dos memorias ROM que proporcionan los datos almacenados en orden, al igual que una FIFO. Una de ellas (*ROM FIFO Image*), contiene los datos de la imagen hiperespectral, que le serán suministrados al núcleo. La otra (*ROM FIFO Golden*), contiene todos los valores que se espera obtener a la salida del núcleo, los cuales han sido extraídos de simulaciones *hardware* previamente verificadas. Esta última ROM permite validar el funcionamiento del núcleo, para garantizar que no presenta errores después de ser cargado en la FPGA.

También se observa en el diagrama 3.11 el componente *Cluster Cache*. Se trata de un banco de memoria que almacena la clasificación que el algoritmo realiza de forma iterativa.

Para administrar los estímulos de entrada, el módulo *Input Control* implementa una máquina de estados que acciona en orden las señales de control del núcleo. En ella se gestionan los procesos de inicialización; precarga, ya que emplea inicialización orientada (ver 3.3.2; ejecución y terminación del algoritmo.

Por otra parte, también se controla la salida del núcleo con el módulo *Output Control*. Este registra las señales de control del núcleo (errores, terminación, validez...) y monitoriza el rendimiento (cuenta el número de ciclos que emplea en procesar).

Con vistas a obtener información lo más completa posible sobre el funcionamiento del

sistema, se ha instanciado también un ILA (*IP Core*). En él se da visibilidad a la salida del núcleo, sus métricas y señales de control, los contadores de *Output Control* o los estímulos de entrada, por citar algunos.

3.8. Otros submódulos

Esta sección está dedicada a algunos componentes del sistema que aparecen de forma recurrente dentro de los módulos. Estos han sido diseñados para ser fácilmente reutilizables, además de contar con opciones de configuración mediante parámetros.

3.8.1. *Full-adder* multibanda

Dado que el núcleo trabaja con píxeles enteros en sus operaciones ha sido necesario crear operadores multibanda, como es el caso de este *Full-adder* o sumador/restador. Tanto la anchura de sus operandos, resultado y número de bandas son paramétricos; así puede emplearse en distintos tipos de datos, por ejemplo píxeles, acumuladores o contadores, que utilizan distintas anchuras para representar los valores.

En la figura 3.12 se observa un diagrama de su arquitectura. Si se sigue el flujo de datos en orden, se observa que a la entrada los operandos son redimensionados: se debe a que la ALU opera con un bit más para detectar el desbordamiento a la salida sin que este llegue a producirse.

La señal *sub_as_diff* selecciona si el resultado de las restas debe ser el valor absoluto. Se utiliza esta función en el cálculo de la distancia entre píxel y centroide, pues se pretende calcular la diferencia entre ambos.

Finalmente, si se hubiese disparado el detector de desbordamiento, el resultado será saturado al máximo o mínimo valor representable (con el número de bits original, antes del redimensionamiento). Un detalle importante, es que este módulo trabaja con aritmética sin signo (al igual que todo el núcleo), por ello las restas con resultado negativo serán saturadas a cero.

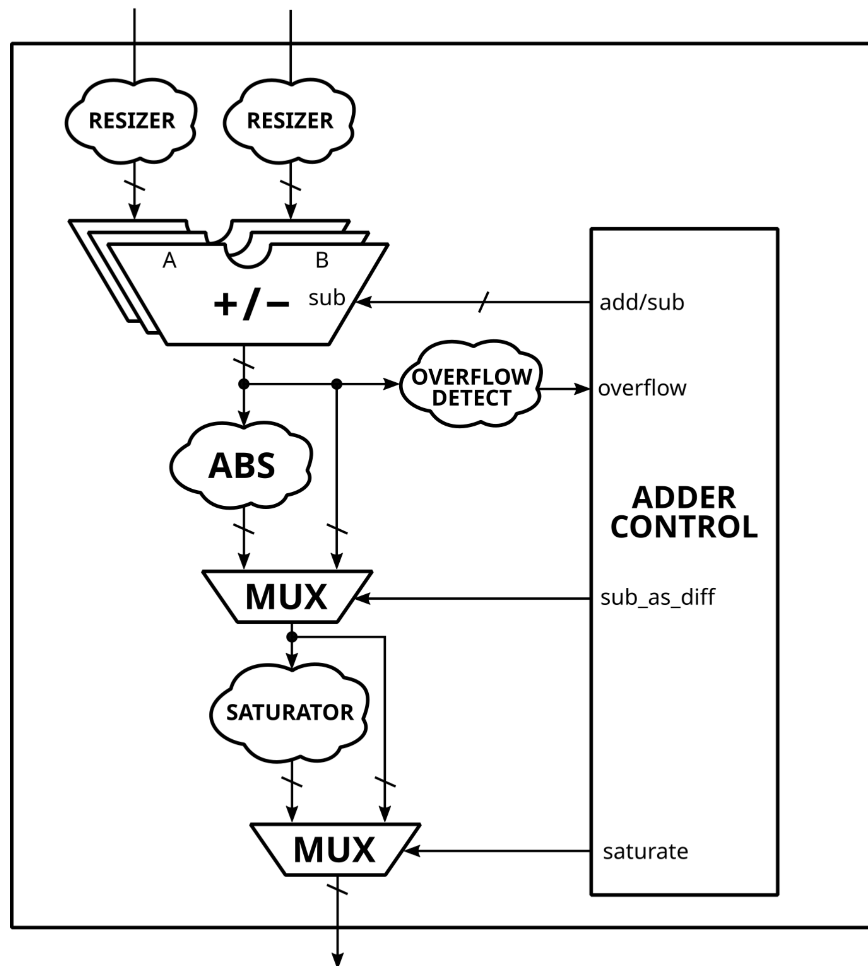


Figura 3.12: Arquitectura del *Full-adder* multibanda.

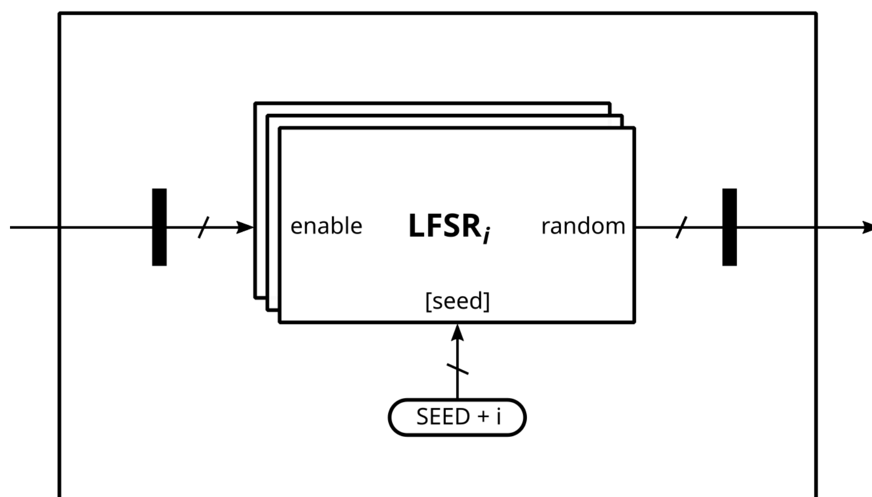


Figura 3.13: Arquitectura del LFSR multibanda.

Toda la operación del sumador multibanda es combinacional, es decir, su salida no está registrada.

3.8.2. LFSR multibanda

El LFSR multibanda permite generar una colección valores aleatorios en paralelo. Tanto el número de valores como su anchura son paramétricos.

En su interior (véase la figura 3.13 hay un array de módulos LFSR de tipo *fibonacci*, que pueden generar un dato pseudoaleatorio nuevo cada ciclo de reloj. Del mismo modo al descrito en la sección 2.9.1, estos implementan un registro de desplazamiento realimentado por una nube combinacional compuesta por puertas XNOR. Este último detalle implica que la semilla nunca debe ser "todo unos", puesto que el LFSR caería en un estado bloqueado y la salida siempre sería la misma.

Cada uno de los LFSR que componen este módulo está inicializado con una semilla distinta, tal que así todos los valores de cada banda sean distintos entre sí.

En total, los LFSR implementados pueden generar valores pseudoaleatorios de entre 2 y 32 bits de anchura. Una consideración importante a la hora de utilizar este módulo es

que los valores generados por los LFSR se repiten cíclicamente, luego cuanto menor sea su anchura, más corto será el periodo de repetición.

3.8.3. Sincronizador

En módulos como el *Divider* o el *Pipelined Classifier* era necesario sincronizar algunas señales con otras que tardaban varios ciclos en procesarse; para ello se hacía referencia al *Output Synchronizer*.

Dicho sincronizador (cuya arquitectura está reflejada en el diagrama 3.14) consiste en una cadena de registros a través de los cuales van pasando los datos, de forma similar a cualquiera de los otros módulos segmentados, pero sin realizar ningún procesamiento en el transcurso.

El número de etapas es paramétrico, para que sea posible adaptarlo sin dificultad a distintos casos de uso.

Al igual que en el resto de módulos del *pipeline* en los que intervienen registros, el sincronizador recibe una señal de *flush* para vaciar sus datos e impedir que se propaguen.

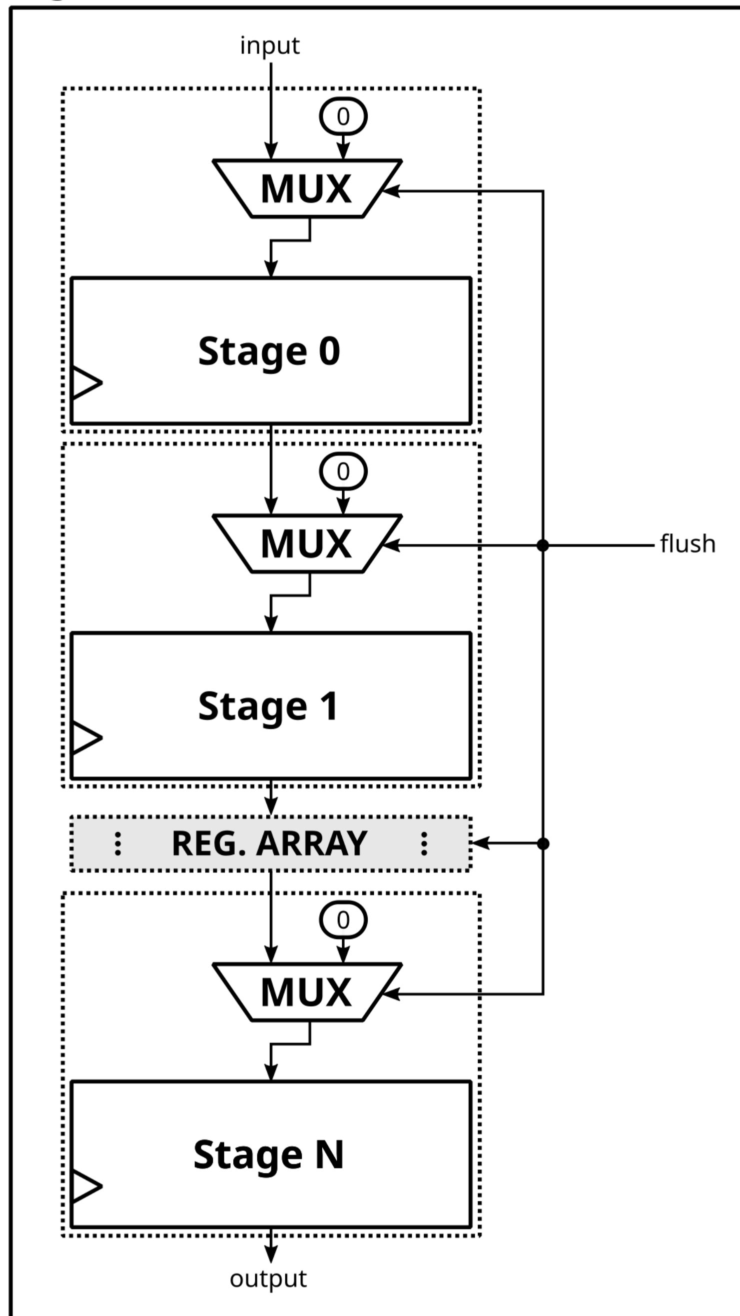


Figura 3.14: Arquitectura del sincronizador.

Capítulo 4

Validación y verificación

Este capítulo describe el largo proceso llevado a cabo para garantizar la validez de los resultados obtenidos. Dicha verificación se ha realizado de forma progresiva, a medida que fueron sucediéndose las distintas fases del desarrollo. Así cada paso en el proyecto se ha dado sobre una base sólida y comprobada.

4.1. Cuprite

Todas las pruebas de en este capítulo que involucran imágenes reales se han realizado alimentando al algoritmo con una hiperespectral (capturada por el sensor AVIRIS) de la región de Cuprite en Nevada (Estados Unidos). Esta es una región bien conocida desde el punto de vista mineralógico que contiene expuestos varios minerales de interés como la alunita, buddingtonita, calcita, caolinita o moscovita.

La figura 4.1 muestra una representación a color (RGB) de la imagen. Esta imagen hiperespectral tiene un tamaño de 256x307 píxeles (78.592 en total); de 224 bandas cada uno (que equivale a 17.604.608 muestras), con longitudes de onda entre 0,4 y 2,5 μm y resolución espectral nominal de 10 nm.

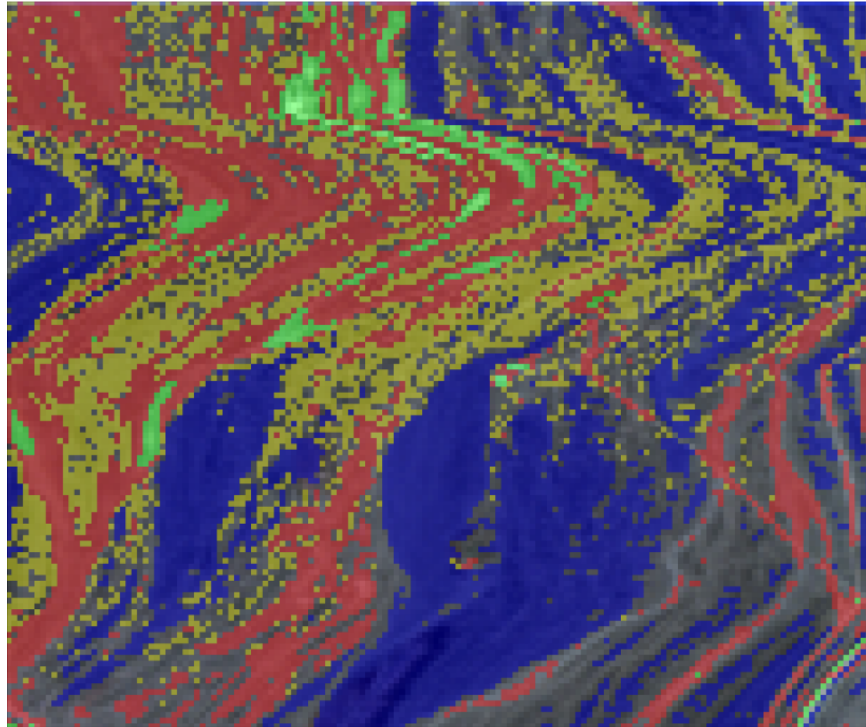
En ella se observa una carretera en vertical, ubicada entre una llanura a su izquierda y una montaña a su derecha. En la parte superior de la misma aparece una distorsión frecuente en este tipo de imágenes, conocida como deriva; sin embargo, para este algoritmo no es necesario corregirla, pues no afecta a la clusterización.

Figura 4.1: Visualización a color (RGB) de la imagen empleada para pruebas.

4.2. Implementación *software*

En primer lugar, para las pruebas preliminares en *software* no fue necesaria una validación demasiado rigurosa, puesto que la especificación del algoritmo venía dada (en pseudocódigo) en el artículo de Lavenier [17]. Por ello fue suficiente verificar que la salida del algoritmo se correspondía con lo esperado. En la figura 4.2 se muestra un ejemplo de clasificación en cinco clústers realizada por el algoritmo implementado en *software*, sobre la imagen hiperespectral de Cuprite. Esta ha sido obtenida con uno de los *scripts* programados para visualizar gráficamente los resultados. Cada uno de los clústers se muestra sobreimpreso en falso color sobre la imagen original en escala de grises.

Para obtener un grado de seguridad mayor respecto a los resultados, se utilizó una implementación ya existente de k-means para comparar su salida con la obtenida anteriormente. Debido a la inicialización aleatoria y las particularidades de cada versión, la salida no resultaba idéntica entre ambas; sin embargo, los clústers identificados se asemejaban muy



224 bandas

Figura 4.2: Clusterización por *software* con la imagen hiperespectral completa.

considerablemente.

4.3. Viabilidad de la adaptación *hardware*

Antes de comenzar el diseño del núcleo era necesario asegurar que las restricciones impuestas por el *hardware* no tendrían un efecto fatal en el funcionamiento de k-means. Para ello se trató de replicar en el *software* ya implementado el uso de números pseudoaleatorios mediante LFSR o la aritmética de enteros (en lugar de en punto flotante), con el objetivo de comprobar cuál era su impacto en la precisión y convergencia del algoritmo.

En caso de no haber superado con éxito esta fase, habría sido necesario replantear por completo las opciones de implementación. Por ello debía validarse la viabilidad antes de iniciar el diseño *hardware*.

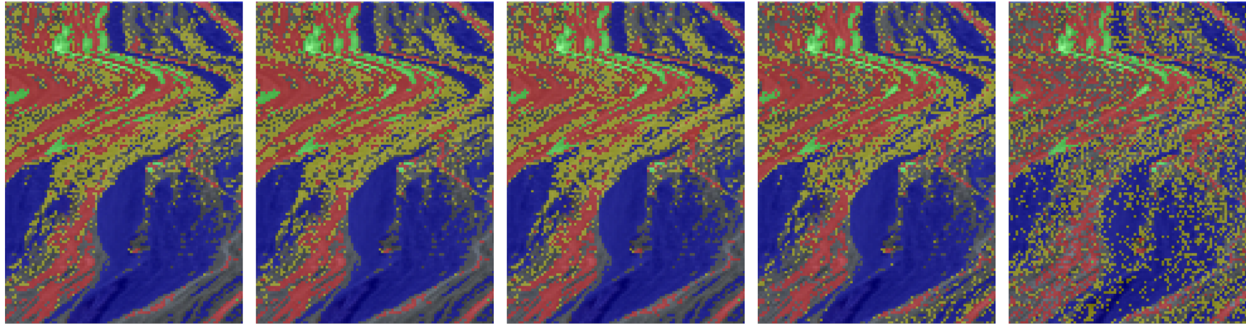


Figura 4.3: Clusterizaciones *software* de la imagen hiperespectral reducida.

4.3.1. Reducción dimensional

A continuación era imprescindible comprobar que la reducción dimensional era viable y no distorsionaba los resultados. Para ello fueron comparados los resultados obtenidos entre ejecutar el algoritmo con todas las bandas y ejecutarlo una vez aplicada la reducción. En la colección de imágenes 4.3 se muestran los efectos de la reducción dimensional en la clusterización; debajo de cada imagen figura el número de bandas resultante. A medida que aumenta la reducción (y disminuye el número de bandas), comienzan a mezclarse los píxeles en las regiones espacialmente contiguas.

Para determinar cuál es el grado de reducción más apropiado, se realizó una comparación entre la clusterización original y las que utilizan la imagen reducida. En la figura 4.4 se muestran en rojo las diferencias entre ambos resultados, superpuestas sobre la imagen original en blanco y negro.

De esta manera fue determinado que 10 bandas eran suficientes, ya que menos de un 10 % de los píxeles clasificados se veían afectados, correspondiendo la mayoría de estos con las regiones donde los clústers aparecen mezclados. Esto implica que las 224 bandas de la imagen original sean combinadas en grupos de 22.

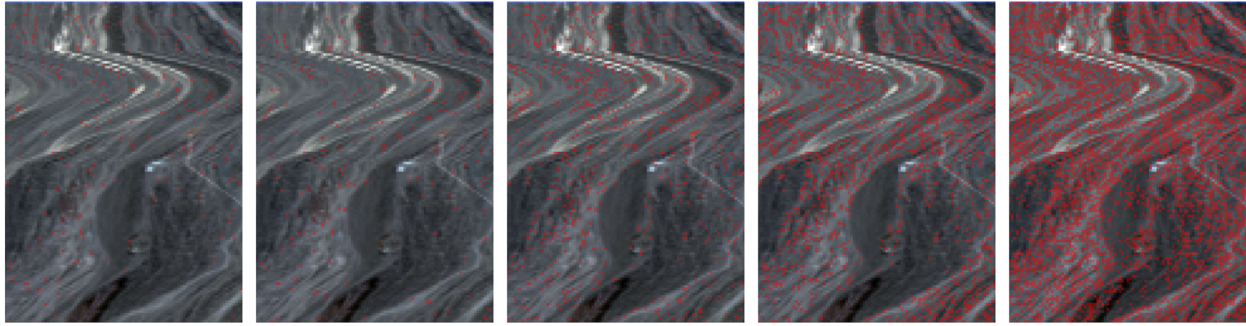


Figura 4.4: Diferencias de clusterización entre imagen completa y reducida.

4.4. Comportamiento básico en *hardware*

A lo largo del desarrollo *hardware* fue verificado mediante simulación el funcionamiento de los distintos componentes del núcleo.

4.4.1. Verificación de módulos genéricos

En primer lugar se procedió a validar algunos módulos aislados como el divisor, el LFSR o el sumador multibanda. Nótese que todos ellos son instanciados múltiples veces y en distintos puntos del proyecto, luego tener la certeza de su correcto funcionamiento es crucial para localizar errores en módulos más grandes y complejos.

Para ello fueron elaborados varios *testbenches* que sometían los citados submódulos a diversos estímulos de forma controlada, tal que la salida esperada fuese conocida, para así poder comprobarla manualmente. En este paso se prestó atención a que los estímulos cubriesen todos los modos de funcionamiento, para asegurar que la lógica había sido correctamente implementada.

4.4.2. Flujo del *pipeline*

Con esta etapa de la validación se pretendía asegurar que el *pipeline* presentaba un comportamiento acorde a lo diseñado. Para ello, a partir de una simulación, se monitorizó

manualmente y paso a paso el avance de un dato de entrada a lo largo de toda la cadena de procesamiento. Para ello fue necesaria una simulación en profundidad, que registrase una gran cantidad de señales internas de cada componente, para así detectar el punto exacto dónde se originaba cada error.

El principal inconveniente de este método es el elevado tiempo necesario para computar toda la simulación con semejante nivel de detalle. Además el problema resultó agravado por la necesidad de lanzar de nuevo toda la simulación (desde el principio) tras realizar una corrección, dado que es imprescindible regenerar el diseño completo después de modificarlo.

No obstante, esta es la forma más directa de asegurar que cada operación de la cadena era realizada correctamente y también que la lógica de control de cada módulo tomaba las acciones pertinentes. Asimismo se pudo comprobar el funcionamiento de los mecanismos de sincronización y realimentación entre los módulos segmentados del *pipeline*.

Por supuesto, este proceso abarca desde la misma entrada del píxel en el núcleo, pasando por el array sistólico, el actualizador de acumuladores, el recálculo del centroide, la actualización de este último (que es enviada al procesador del clúster correspondiente) y finalmente la salida del resultado.

4.4.3. Módulo de control principal

De forma similar al proceso anterior, se realizó un seguimiento al módulo de control. Dado que este componente está conectado a la salida del clasificador y funciona en paralelo con el resto del *pipeline*, fue más conveniente validarlo aparte.

En él la lógica es más compleja, dado que utiliza una única máquina de estados para gestionar todo el núcleo. Esto incluye la gestión de las iteraciones, las paradas en el flujo de datos, los reinicios o el vaciado del *pipeline* a finalizar el algoritmo, por citar algunos.

Además se verificó que las condiciones de parada eran calculadas correctamente, incluyendo el umbral máximo de píxeles intercambiados (en base a la imagen recibida) y el límite de iteraciones (en base a la configuración actual del núcleo).

Por otra parte, también fueron validados los mecanismos de detección de errores que incluye este módulo, tanto para detectar problemas de convergencia como discrepancias en los datos de entrada.

4.5. Comportamiento general del algoritmo

Sin embargo, una vez implementado y parcialmente depurado todo el algoritmo en *hardware* resultó imprescindible comprobar que su comportamiento a la hora de procesar imágenes era correcto. Además es necesario conocer si las diferencias con respecto a la versión *software* afectaban a los principios fundamentales.

Los cambios aplicados al algoritmo para adaptarlo a *hardware* figuran en la sección 2.4.

4.5.1. Pruebas con imágenes sintéticas simples

Las primeras pruebas fueron realizadas con imágenes sintéticas pequeñas, las cuales presentaban grupos de píxeles idénticos, con valores distantes entre sí. El objetivo perseguido con ello era alimentar al núcleo con datos que resultasen fáciles de clasificar, para poder solventar los errores que afectasen al fundamento más básico del núcleo.

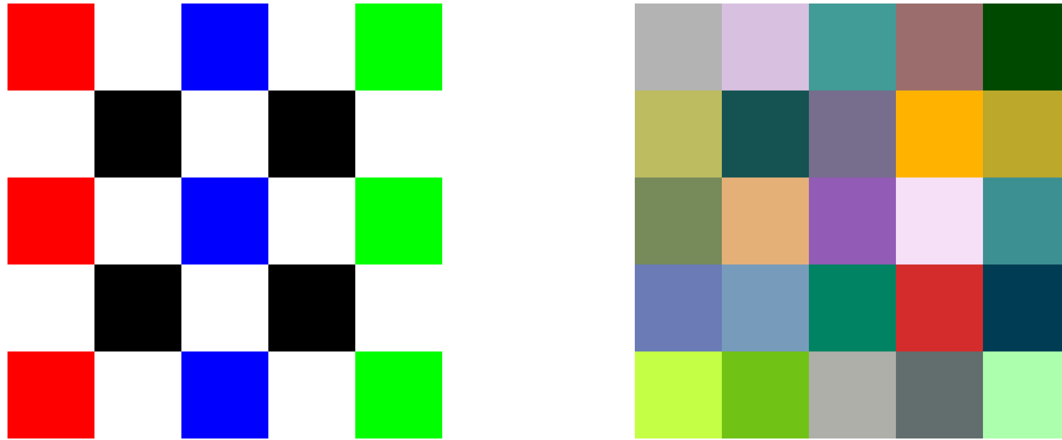
De este modo se esperaba poder verificar dos hechos: que el algoritmo sea capaz de converger y que todos los píxeles iguales entre sí eran asignados al mismo clúster.

En la figura 4.5a se muestra un ejemplo del tipo de imagen enviada a la entrada del núcleo.

4.5.2. Pruebas con imágenes sintéticas aleatorias

Posteriormente se pasó a emplear imágenes cuyos píxeles habían sido generados aleatoriamente (como en la figura 4.5b), mediante uno de los *scripts* elaborados en *software*. De este modo se garantiza que los datos de entrada toman valores distintos, ocupando todo el rango dinámico posible.

Al haber sido ya comprobado que el algoritmo alcanza la convergencia (en el paso ante-



(a) Imagen sintética simple para verificación. (b) Imagen sintética generada aleatoriamente.

Figura 4.5: Imágenes sintéticas.

rior), cabe esperar de esta prueba que la clasificación sea equitativa en todos los clústers. Es decir, que al finalizar el algoritmo, a cada clúster pertenecerá un número similar de píxeles asignados. Así también se prueban datos que resultaría improbable (pero nunca imposible) encontrar en imágenes reales, y por tanto susceptibles de provocar una potencial condición de error que no habría sido detectada en esta fase de verificación.

4.6. Diferencias entre *hardware* y *software*

Dado que la versión *software* de k-means ha sido previamente verificada y además es el punto de partida para el núcleo diseñado, es necesario comprobar que ambos se comportan de la misma manera para así validar la implementación *hardware*.

Cabe recordar que debido a ciertas restricciones del diseño *hardware*, así como por otras modificaciones realizadas en el algoritmo original (para mejorar su rendimiento), el núcleo desarrollado no presenta un comportamiento idéntico a su homólogo *software*. Dichas adaptaciones figuran en la sección 2.4.

En cualquier caso, para garantizar que el comportamiento es equivalente en *hardware* y *software*, se procedió a comparar entre sí la salida obtenida en ambas versiones. Sin embargo, carecería de sentido realizar una comparación directa, pues los mecanismos de inicialización

aleatoria son una de las principales diferencias entre ambos; por ello es imprescindible un paso previo: adaptar los algoritmos para que puedan partir de un estado inicial conocido. Para ello se ha utilizado la llamada "inicialización forzada" (véase la sección 3.3.2) del núcleo, que permite establecer directamente el valor inicial de los centroides. Dicho valor fue el empleado también en la variante *software*.

Una vez realizadas las modificaciones se pudo comprobar cómo los resultados eran muy similares pero no idénticos. En distintas pruebas con imágenes sintéticas, los índices de coincidencia rondan el 90 % y las discrepancias solo se producen entre clústers cercanos.

Más allá de los resultados finales, ambas versiones también difieren en el número de iteraciones empleadas hasta converger, siendo este mayor en la versión *software*.

La causa de dichas diferencias reside en la actualización rápida (y constante) de centroides que realiza la versión *hardware*; pues en contraposición, el algoritmo *software* procesa por bloques y por tanto con datos mucho más desactualizados, lo cual afecta negativamente al tiempo de convergencia.

4.7. Convergencia a mínimo local

A pesar de las conclusiones del apartado anterior, era necesario ir más allá con la verificación; por lo que fue elaborado un nuevo *script* para verificar que el diseño convergía a un mínimo local, y no a un valor erróneo.

Para realizar dicha validación es necesario extraer del núcleo el valor final de los centroides, así como evidentemente los resultados de la clusterización. Dicha labor no resulta especialmente complicada gracias a que las herramientas de simulación empleadas permiten monitorizar las señales internas del diseño. Asimismo, fue elaborado un *testbench* en VHDL 2008 que escribiese en un fichero de texto los resultados de clusterización obtenidos por el núcleo.

El proceso de verificación consiste en comprobar que cada píxel ha sido asignado al centroide más cercano (comparándolo con todos los existentes), y por tanto no es preciso

Figura 4.6: Ejecución del *script* de verificación.

moverlo a otro clúster. La figura 4.6 muestra una ejecución del *script* de verificación para una imagen de 1.000 píxeles.

4.8. Pruebas con imágenes reales

Una vez comprobado y validado el funcionamiento del núcleo se iniciaron las pruebas con imágenes hiperespectrales reales (capturadas por el AVIRIS). En la figura 4.7 se muestra un ejemplo de clusterización realizada por el núcleo *hardware* sobre la fotografía de Cuprite. Dicha imagen tiene 78.592 píxeles, que han sido clasificados en siete clústers distintos (cada uno representado en un color).

Para obtener la imagen con los resultados de clusterización se ha programado nuevamente una utilidad que adapta los resultados proporcionados por el núcleo y los envía al mismo *script* empleado para colorear las primeras pruebas realizadas en *software*.

Para el procesado de la imagen, el algoritmo ha realizado ocho iteraciones (incluida la primera de inicialización), en las cuales ha empleado 628.700 ciclos de reloj. En la figura 4.8 está representada la ejecución completa en un cronograma. En él se aprecia como el algoritmo converge perfectamente y sin errores; si se presta atención a la señal *pixel_move*, se observa

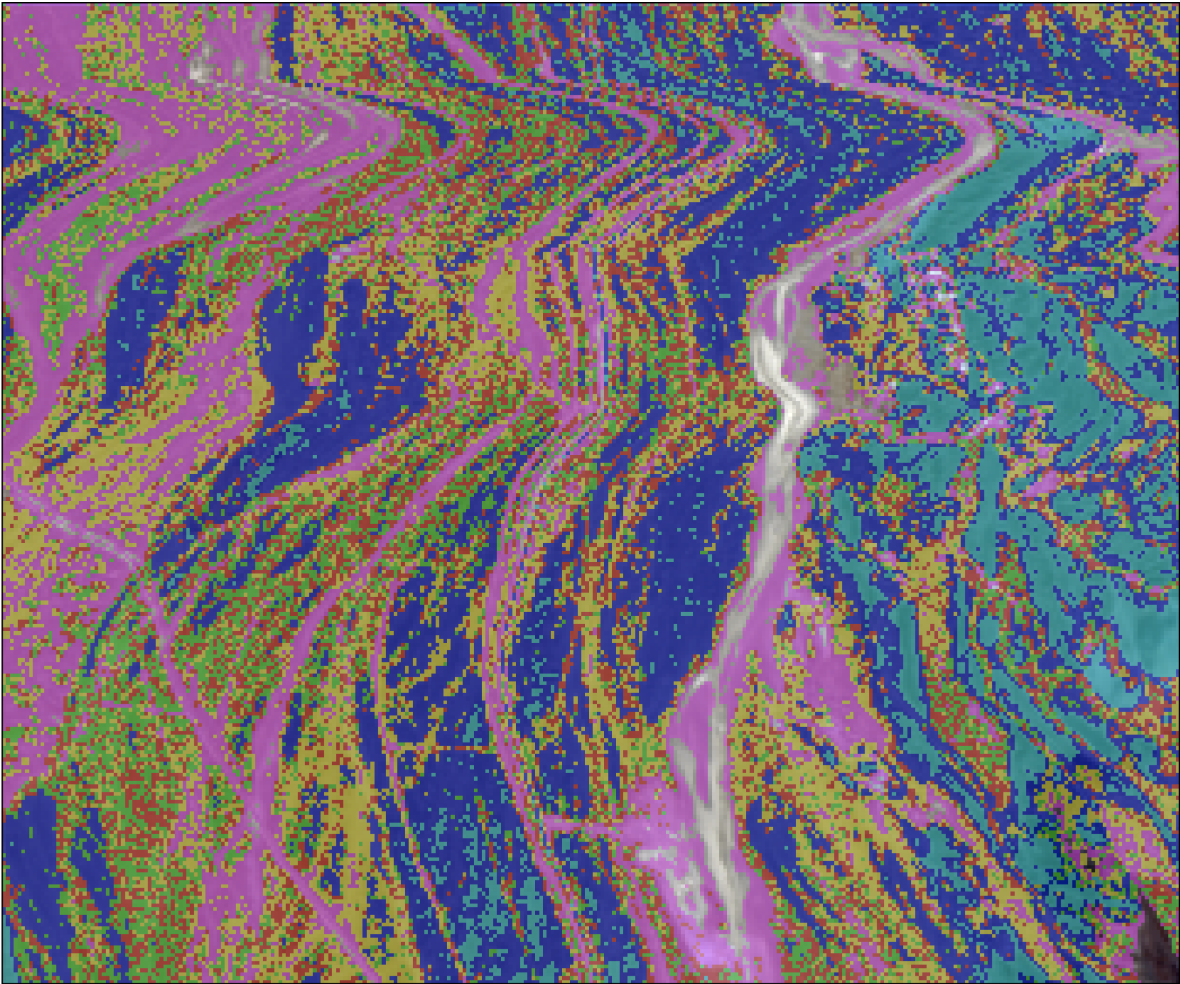


Figura 4.7: Clusterización realizada por el núcleo *hardware*.

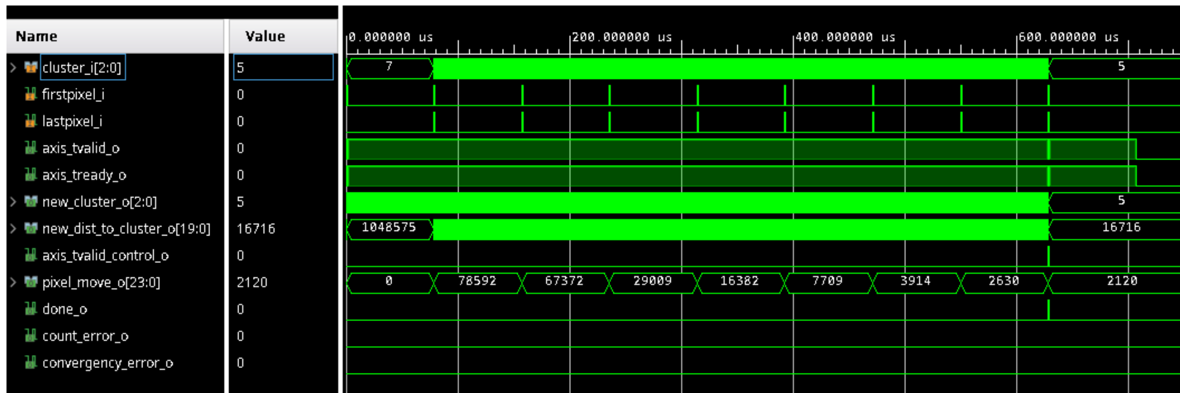


Figura 4.8: Cronograma de simulación.

cómo va disminuyendo el número de píxeles que cambian de clúster (hasta alcanzar el umbral mínimo, fijado al 3,125 %).

4.9. Simulaciones post-síntesis y post-implementación

Llegado este punto se podía dar por finalizada la verificación. Sin embargo, para garantizar que el funcionamiento no se vería afectado por la carga del diseño en la FPGA, se repitieron algunas de las pruebas en simulaciones post-síntesis y post-implementación. Esto quiere decir que la simulación será realizada sobre el diseño obtenido después de las etapas de síntesis y *place and route*; de modo que puede verificarse el comportamiento de aquello que va a cargarse en placa, incluso después de ser optimizado por Vivado (la herramienta EDA).

Los errores presentados en esta fase, así como las discrepancias respecto a las simulaciones de comportamiento que se han realizado con anterioridad, son especialmente complicados de identificar y solucionar. De entrada este tipo de simulación es mucho más costosa que las realizadas anteriormente, y además requieren que el diseño esté completo para que la EDA (Vivado) pueda elaborar el diseño final (hasta la etapa llamada "implementation"). Esto se traduce en más tiempo de espera hasta conocer el resultado. Por otra parte, aquí se tiene una dependencia absoluta con la herramienta y los errores en la mayor parte de los casos están derivados de algún tipo de optimización o comportamiento no deseado, que está fuera

del control del desarrollador. Por ello no solo es difícil de localizar, sino también de corregir la causa del error.

De hecho fue necesario cambiar la descripción de algunas partes del diseño para lograr que los resultados de las simulaciones post-síntesis y post-implementación coincidiesen con las de comportamiento. Igualmente ha sido necesario lidiar con algunas opciones de configuración de la herramienta, que derivaban en resultados no deseados.

4.10. Interfaz con FPGA

Finalmente, con vistas a comprobar del sistema una vez cargado en placa, fue diseñado un envoltorio para el núcleo, el cual actúa como interfaz con la FPGA. En él se compara la salida del núcleo con los datos verificados obtenidos en simulación, que a su vez están almacenados en una ROM interna. Para más detalles, este módulo es descrito en la sección 3.7).

Dicha integración del núcleo en la interfaz de pruebas también ha sido sometida a una simulación post-implementación. El proceso se ha realizado en varias fases, validando primero el comportamiento de la interfaz con un núcleo "vacío" (es decir, sin módulos internos ni *pipeline*); de este modo fue posible rectificar los errores de forma aislada en el envoltorio.

Más adelante fue simulado el conjunto al completo, ya con el núcleo que implementa k-means. Así se comprobó el funcionamiento del núcleo en un entorno completo, incluyendo la comunicación a través del bus AXI.

4.11. Verificación del funcionamiento en FPGA

Al ser completadas con éxito todas las simulaciones, incluso las post-implementación para el núcleo ya integrado dentro de la interfaz, se iniciaron las pruebas en FPGA.

En ellas, el analizador lógico (que está integrado en el citado envoltorio) ha permitido monitorizar el comportamiento del núcleo.

En cualquier caso, para validar el funcionamiento en la FPGA, la propia interfaz ha sido diseñada con un mecanismo incorporado para realizar la verificación de resultados por sí

misma. De este modo, si tras la ejecución la salida al completo corresponde con los datos esperados, se notifica al exterior mediante una señal (que a su vez está conectada a un LED).

Capítulo 5

Resultados

En en presente capítulo quedan recogidos los frutos que ha dado este proyecto. En él se abordan los objetivos logrados y se analizan los resultados obtenidos.

5.1. Objetivos logrados

Esta sección está dedicada a describir la consecución de los objetivos propuestos, tratando los resultados obtenidos después de cada hito.

En el repositorio `https://gitlab.com/hw-kmeans` puede encontrarse todo el contenido elaborado a causa de este proyecto.

5.1.1. Implementación *software*

El primer hito de este trabajo fue lograr una implementación *software* del algoritmo k-means.

Para ello se tomó como referencia la implementación de k-means propuesta por Dominique Lavenier [17], cuyo pseudocódigo puede leerse en la sección 2.3.1.

Dado que el lenguaje empleado fue Python, que es interpretado, las primeras versiones presentaban un rendimiento muy pobre.

Posteriormente, gracias a la optimización del código y el empleo de bibliotecas como Numpy o SpectralPython (descritas en la sección 2.11.1) se logró paliar en gran medida dicha debilidad.

Puesto que depurar y verificar *software* es mucho más rápido y viable que hacerlo en *hardware*, esta primera toma de contacto con el algoritmo trajo consigo un punto de referencia para posteriormente compararla con las implementaciones orientadas a FPGA.

En todo momento, se pretendía acercar lo más posible el comportamiento del *software* a la posterior implementación *hardware*. Por ello, también fue elaborada una segunda versión del algoritmo en Python, la cual imita el modo de inicialización de su equivalente en HDL; pues la implementación *software* original empleaba un mecanismo para generar valores aleatorios radicalmente distinto al LFSR.

Reducción dimensional

En conjunto con el algoritmo, también fue implementado el sistema de reducción dimensional de imágenes hiperespectrales. Así se podía comprobar también la viabilidad de esta técnica y comparar los resultados obtenidos con distintos grados de reducción, todo antes de abordar la fase de diseño *hardware*.

Scripts de verificación

Además del algoritmo, fueron desarrolladas una colección de funciones y *scripts* para poder comprobar la validez de los resultados obtenidos. Entre ellos se incluyen:

- **Utilidades para representar gráficamente las clasificaciones**. Fueron utilizadas para verificar visualmente los resultados de la clusterización, tanto para la versión *software* como *hardware* del algoritmo.

- **Un comparador de resultados**, que mide la diferencia entre dos clusterizaciones. Para verificar en qué grado afectaban las modificaciones y adaptaciones del algoritmo a la salida obtenida.

- **Un verificador de resultados**, empleado para comprobar que el algoritmo converge adecuadamente y alcanza un mínimo local.

- **Un generador de imágenes aleatorias**, para ampliar el abanico de combinaciones a probar en el algoritmo.
- **Un conversor de formato de las imágenes hiperespectrales**, para poder cargarlas en la memoria ROM que hay en la interfaz con la FPGA.

5.1.2. Diseño *hardware*

Acorde a las decisiones de diseño que fueron expuestas en el capítulo anterior (sección 3.2), se alcanzó la arquitectura definitiva del núcleo. Esta presenta las características que ya han sido expuestas, resultando en un *pipeline* altamente segmentado y paramétrico, el cual es capaz de procesar de forma constante a razón de un píxel por ciclo.

En particular, el número de etapas del *pipeline* depende del número de clústers a identificar, la anchura de las muestras y el número máximo de píxeles; sin embargo el número de bandas no afecta, puesto que se procesan todas en paralelo. En el cuadro 5.1 se muestran algunos posibles ejemplos de configuración.

Dicha tabla también pone de manifiesto la excelente escalabilidad del sistema, puesto que el número de etapas del *pipeline* aumenta de forma logarítmica con respecto al número de píxeles y la anchura de las muestras. Si ponemos el foco en los clústers, el incremento es lineal (una etapa más por clúster); no obstante, el número de clústers suele ser reducido.

5.1.3. Simulación y depuración *hardware*

Depurar *hardware* es una tarea muy costosa que requiere un gran esfuerzo y dedicación. El hecho de que todos los componentes funcionen simultáneamente (ya que existen físicamente), ligado a la complejidad de monitorizar su valor, hace que encontrar la causa de posibles errores sea extremadamente difícil.

Para aliviar la situación, se ha recurrido a herramientas de simulación *hardware* que permiten acelerar el proceso de depuración. Para ello fueron elaborados varios entornos de pruebas (también llamados *testbenches*), que permiten estimular módulos aislados (o

Etapas del *pipeline*: ejemplos de configuración			
Máx. número píxeles	Número clústers	Anchura muestras	**Etapas *pipeline***
5.000	5	16	**37**
5.000	8	16	**40**
5.000	8	24	**48**
50.000	8	16	**42**
50.000	7	16	**45**
50.000	10	24	**55**
500.000	9	16	**47**
500.000	11	16	**49**
500.000	13	24	**59**
5.000.000	12	16	**54**
5.000.000	15	16	**57**
5.000.000	17	24	**67**
50.000.000	12	16	**57**
50.000.000	15	16	**60**
50.000.000	17	24	**70**

Tabla 5.1: Longitud del *pipeline*.

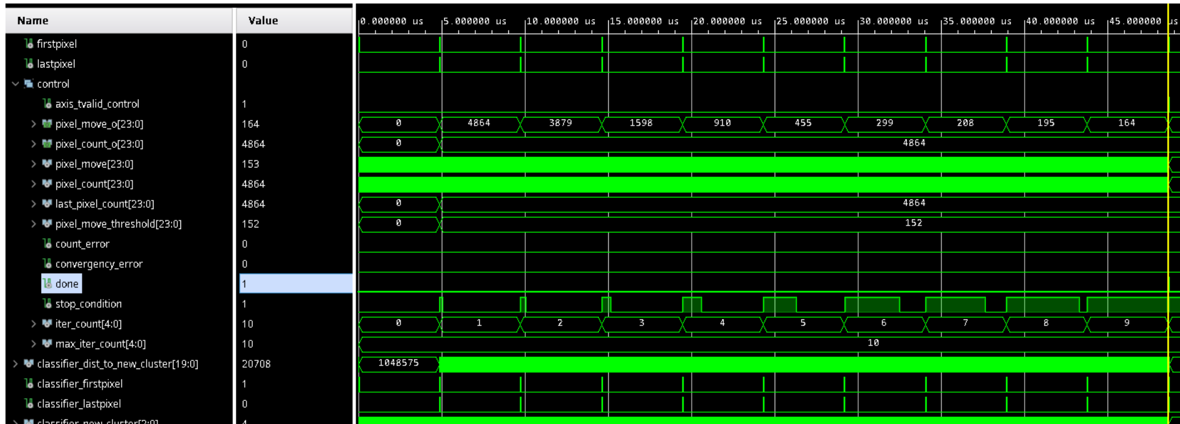


Figura 5.1: Simulación del núcleo.

también el sistema al completo) para evaluar si su comportamiento se corresponde con el especificado.

La gran ventaja de la simulación, es que no es necesario disponer físicamente de la FPGA que albergará el diseño. Gracias a ello ha sido posible probar el funcionamiento del núcleo sin disponer de la FPGA.

Sin embargo, la simulación también presenta una importante desventaja: requiere un uso elevado de recursos computacionales y su ejecución es muy lenta. Con la máquina empleada para este trabajo, las simulaciones completas del algoritmo (de tipo *behavioral*, la más "rápida" y menos rigurosa) han tardado más de 20 horas en finalizar, para imágenes de 300.000 píxeles.

Como resultado de las simulaciones se han obtenido cronogramas detallados con información de las señales internas del núcleo. Un ejemplo se muestra en la figura 5.1.

En el capítulo 4, se describe cuáles son los resultados obtenidos mediante simulación y cómo se han aplicado a la verificación del proyecto.

5.2. Rendimiento del núcleo

En esta sección se discuten los resultados obtenidos en lo que respecta al rendimiento del núcleo *hardware*. Para una mayor consistencia en la comparación de resultados, en ellos

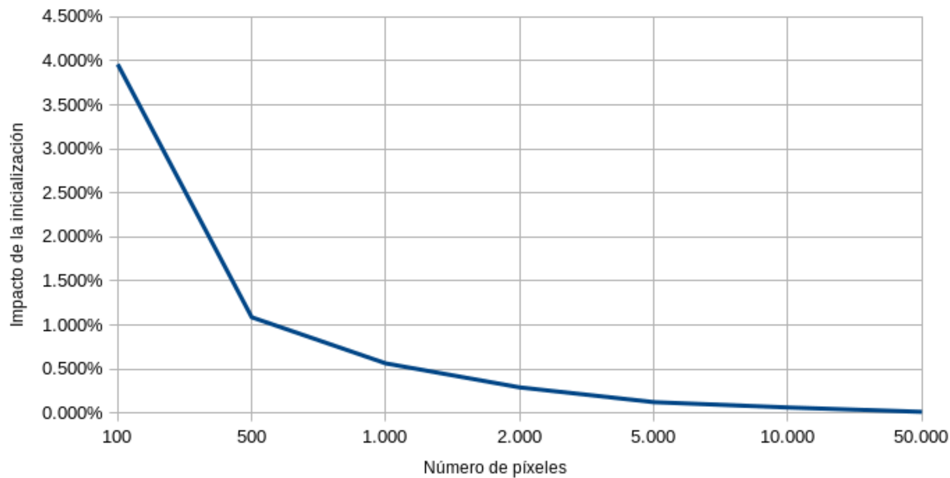


Figura 5.2: Impacto de la inicialización en el rendimiento.

Clasificación en 5 clústers para imagen de 10 bandas con muestras de 16 bits.

se ha utilizado la misma imagen de Cuprite presentada anteriormente, en la sección 4.1.

5.2.1. Rendimiento del *pipeline*

El *pipeline* ha sido diseñado para procesar a velocidad constante de un píxel por ciclo; por tanto, ese es el resultado obtenido. Como la velocidad de reloj establecida para el sistema son 100 MHz, este es capaz de procesar 100 megapíxeles hiperespectrales (224 bandas cada uno, en caso del sensor AVIRIS) por segundo.

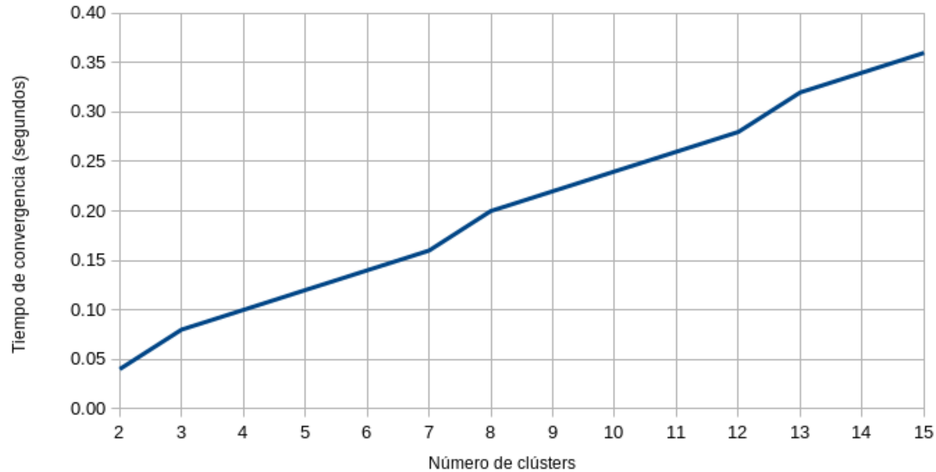
No obstante, aunque el *pipeline* pueda procesar un píxel por ciclo, al ser necesarias múltiples iteraciones hasta obtener el resultado, disminuye el rendimiento global del núcleo.

También influye negativamente en el rendimiento el llenado inicial del *pipeline*; aunque el impacto es absolutamente despreciable para imágenes de cierto tamaño, y más aún si se necesitan múltiples iteraciones. De hecho, el impacto de la inicialización en el rendimiento será menor a medida que aumente el tamaño de la imagen.

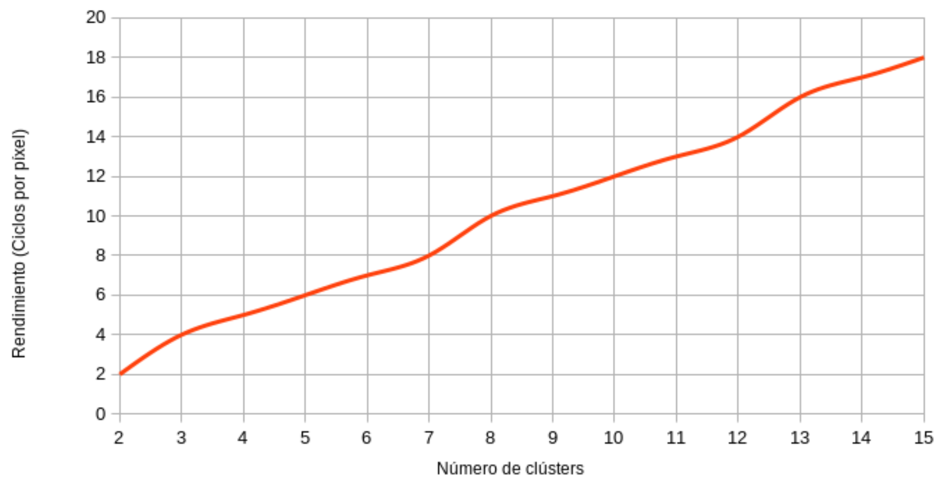
En la figura 5.2 se muestra una gráfica que relaciona el impacto de la inicialización con el tamaño de la imagen. Dicho impacto está expresado en tanto por ciento, referido al número de ciclos sobre el total dedicados al llenado del *pipeline*. Los datos corresponden a una serie de clasificaciones en cinco clústers. Nótese que para imágenes de 50.000 píxeles (el

equivalente a solo 0,05 megapíxeles), el impacto es ya prácticamente nulo.

5.2.2. Impacto del número de clústers en el rendimiento



(a) Impacto del número de clústers a clasificar en el tiempo de ejecución.



(b) Impacto del número de clústers a clasificar en el rendimiento.

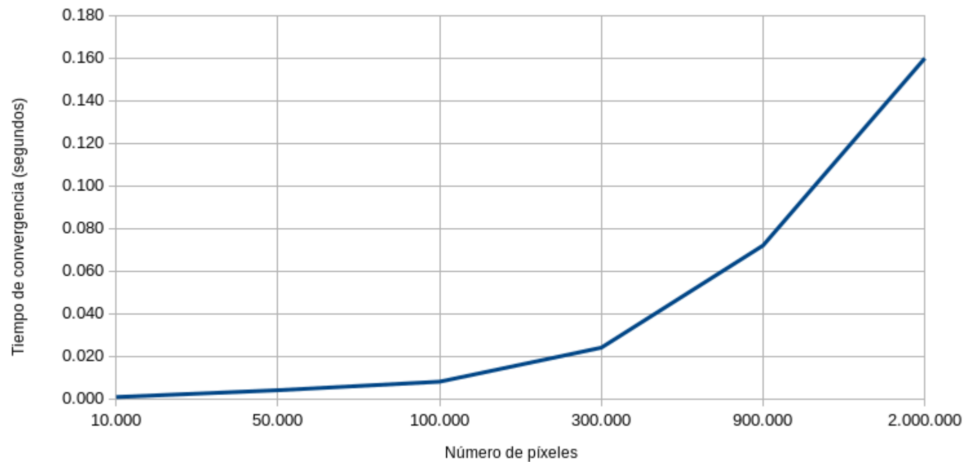
Figura 5.4: Impacto del número de clústers en la obtención de resultados.

Clasificación en imagen de 2 megapíxeles, con 10 bandas y muestras de 16 bits.
(Asumiendo convergencia con $< 3{,}25\,\%$ de píxeles reasignados)

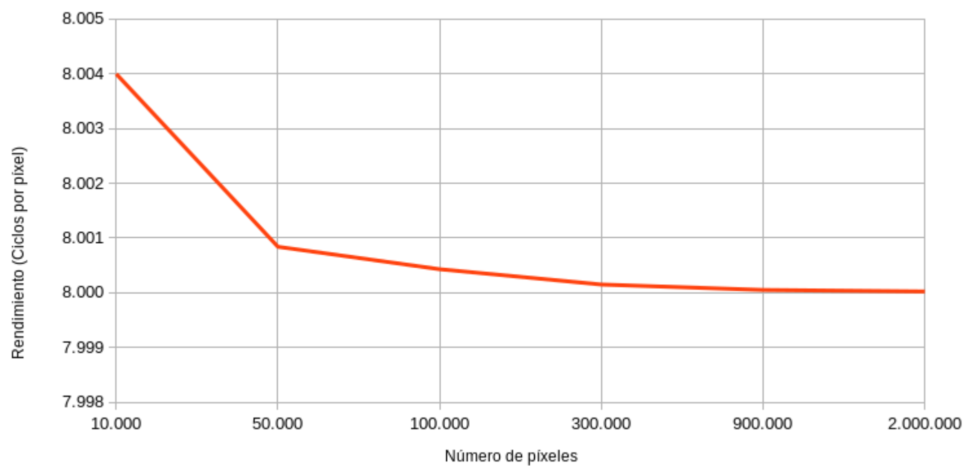
Por otra parte, el número de clústers a identificar en las imágenes afecta al tiempo de ejecución del algoritmo. La figura 5.4 muestra como al incrementar el número de clústers, el rendimiento del sistema disminuye, aumentando así el tiempo de convergencia. Esto se

debe a que para clasificar los datos entre mayor número de clústers, son necesarias más iteraciones. Este comportamiento es exactamente el que cabría esperar del algoritmo, sin ninguna penalización añadida a consecuencia de la implementación *hardware*. Los datos se han calculado para una imagen de 2.000.000 de píxeles.

5.2.3. Impacto del número (máximo) de píxeles en el rendimiento



(a) Impacto del número de píxeles en el tiempo de ejecución.



(b) Impacto del número de píxeles en el rendimiento.

Figura 5.6: Impacto del número de píxeles en la obtención de resultados.

Clasificación en 7 clústers para imagen de 10 bandas con muestras de 16 bits.
(8 iteraciones hasta convergencia)

Al igual que sucede al aumentar el número de clústers, procesar imágenes con más píxeles conlleva un incremento en el tiempo de ejecución. En la figura 5.6a se aprecia el aumento lineal del tiempo de convergencia con respecto al número de píxeles. Nuevamente, el aumento del tiempo de ejecución es proporcional al número de clústers debido al funcionamiento del algoritmo y no al diseño *hardware*.

De hecho respecto a la versión *software*, el *hardware* converge en menor número de iteraciones (gracias al mecanismo de actualización rápida de centroides); luego el impacto del número de clústers en el tiempo de ejecución es menor en el diseño elaborado que en el original.

Mientras tanto, en la figura 5.6b se observa cómo el rendimiento permanece casi constante. El leve descenso (en el rendimiento) ocurre porque al incrementar el número de píxeles, aumenta la anchura de los acumuladores y por tanto la latencia del divisor será también mayor, causando que el *pipeline* sea ligeramente más largo. Los datos se han calculado para una clusterización en siete grupos. En dicha gráfica se observa que el rendimiento es cercano a 8 ciclos por píxel, lo cual se debe a que la convergencia se alcanza en 8 iteraciones; es decir, como el *pipeline* procesa un píxel por ciclo, la imagen completa es procesada 8 veces.

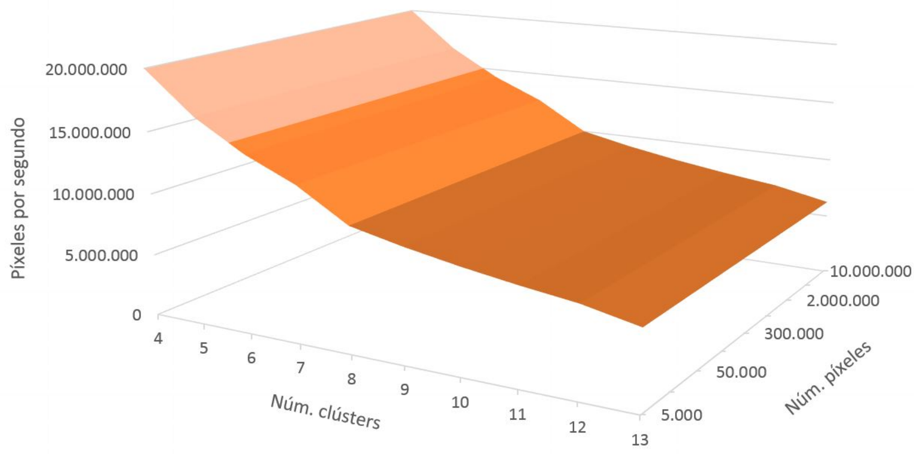
5.2.4. Resumen de rendimiento

A continuación, se muestra de forma resumida el impacto en el rendimiento de las dos variables anteriores: número de clústers y número de píxeles. En la gráfica tridimensional 5.8a se observa cómo varía el rendimiento del núcleo (expresado en píxeles procesados por segundo) en relación con el número de clústers a identificar y el tamaño de la imagen.

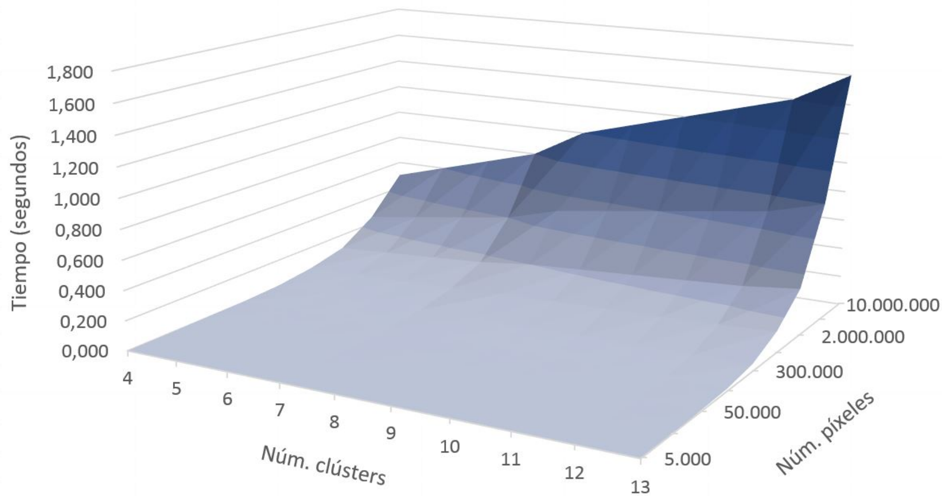
Asimismo, en la figura 5.8b se muestra de forma análoga a la anterior el tiempo de ejecución.

5.2.5. Funcionamiento en tiempo real

Si hubiese restricciones de temporización (por ejemplo, para un funcionamiento en tiempo real) en el procesado de la imagen, existirían algunos límites respecto al número máximo de



(a) Gráfica de rendimiento.



(b) Gráfica de tiempo de cómputo.

Figura 5.8: Impacto del número de píxeles y clústers a identificar en la obtención de resultados.

clústers y píxeles a procesar por el sistema.

Consideraremos como límite de funcionamiento en tiempo real la velocidad a la que el sensor hiperespectral AVIRIS es capaz de tomar los datos: 137.143 píxeles por segundo. Por otra parte, el núcleo está desarrollado para funcionar a al menos 100 MHz de frecuencia de reloj; lo cual permite procesar 100 millones de píxeles en un segundo (individualmente). El rendimiento real será inferior al ser necesarias varias iteraciones, en cada una de las cuales son procesados todos los píxeles. Considerando un caso de uso real (aunque teórico), seria

Comparativa entre *hardware* y *software*					
Versión	Núm. iter.	T. ejec.	*Speed-up* tiempo	T. desarrollo	*Speed-down* des.
Software	9	55,74 s	8.861,7	2 semanas	-14,5
Hardware	8	6,29 ms		29 semanas	

Tabla 5.2: Comparativa entre *hardware* y *software*.

necesario tratar de identificar más de 600 clústers en una imagen para alcanzar los límites de funcionamiento en tiempo real.

5.2.6. *Speed-up* respecto al *software*

Finalmente, para mostrar la ganancia real que proporciona este diseño se han tomado las dos versiones del algoritmo, las cuales han sido configuradas de igual forma (mismo número de bandas y clústers) y alimentados con la misma imagen. En particular, se ha clusterizado en 7 regiones una imagen de 78.592 píxeles, reducida de 224 a 10 bandas.

Como referencia, la implementación *software* ha sido ejecutada en un equipo doméstico con microprocesador Intel Xeon E5 (arquitectura Ivy Brigde) de 3,3 Ghz. Este detalle es bastante relevante, dado que los resultados están sujetos a amplias variaciones en función de la máquina empleada.

Los resultados obtenidos están resumidos en la tabla 5.2. En ella se ha incluido el rendimiento obtenido en ambas variantes, aunque para ofrecer una comparativa más justa también se ha incluido el aspecto más desfavorable de la implementación *hardware*: el tiempo de desarrollo.

De hecho, de todos los resultados expuestos en la sección anterior, puede concluirse que ninguno se ve afectado negativamente por la implementación *hardware*. Dicho de otra forma, todos los incrementos en el tiempo de cómputo al aumentar el número de clústers o tamaño de la imagen son inherentes a la naturaleza del propio algoritmo, no están derivados del diseño *hardware*. La única excepción es la penalización asociada al llenado del *pipeline* (la cual es ínfima).

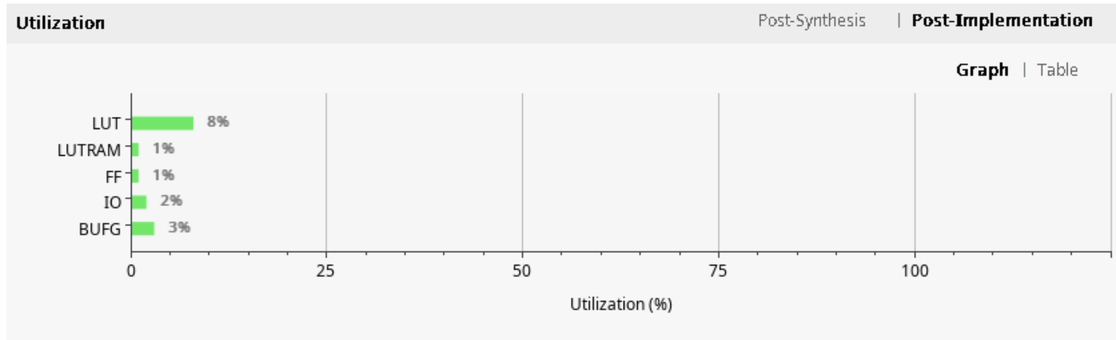


Figura 5.9: Utilización de recursos con configuración básica.

5.3. Utilización de recursos

Una vez completada en su totalidad la descripción *hardware* del sistema, la herramienta de síntesis (Vivado 2020.2) obtuvo los resultados expuestos en esta sección.

Debido a la alta parametrización del sistema, el uso de recursos estará fuertemente condicionado por la configuración empleada. Por ello, para mostrar estos resultados se han establecido tres configuraciones distintas: básica, amplia y sobredimensionada; con vistas a proporcionar una visión amplia sobre la utilización de recursos.

Configuración básica

En primer lugar, para una configuración "básica" se han establecido 10 bandas; 4 clústers a identificar; y soporte para imágenes de hasta 5.000 píxeles con muestras de 16 bits. En este caso, el porcentaje de ocupación ronda el 5 %.

En la figura 5.9 se muestran los resultados de ocupación (*post-implementation*) obtenidos desde la herramienta EDA.

Configuración moderada

Por otro lado, la configuración "amplia" consiste en emplear 10 bandas; 8 clústers a identificar; y soporte para imágenes de hasta 500.000 píxeles con muestras de 16 bits. Con esta configuración se trata de reflejar un caso de uso estándar, aunque con soporte para imágenes hiperespectrales grandes.

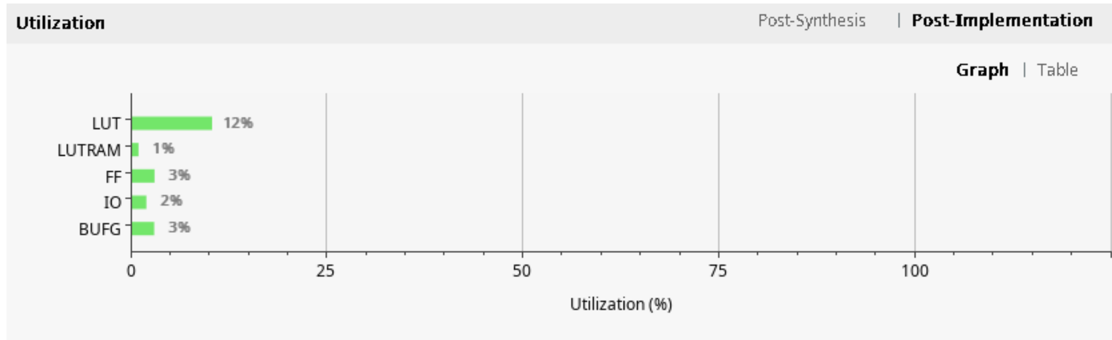


Figura 5.10: Utilización de recursos con configuración amplia.

La figura 5.10 refleja los resultados de ocupación obtenidos para la configuración amplia.

Configuración extremadamente sobredimensionada

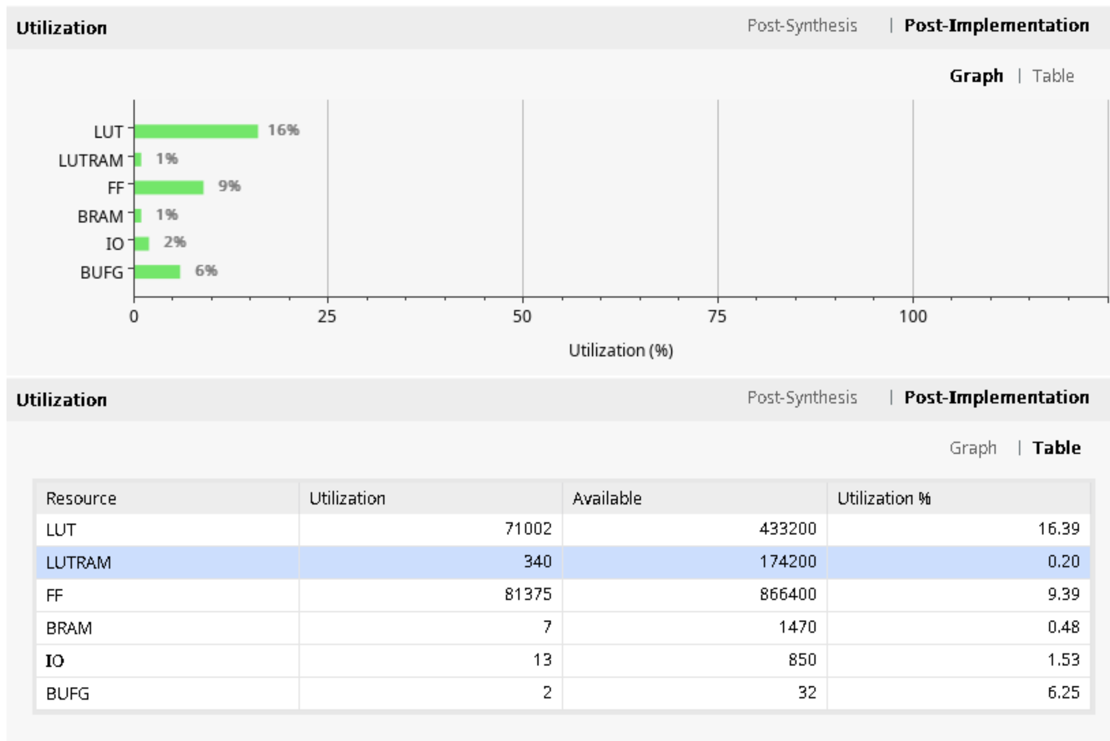
Finalmente para la configuración "extremadamente sobredimensionada" los parámetros del núcleo se han establecido de forma muy ambiciosa, tal que el uso de recursos sea mucho más elevado que en un caso estándar. A pesar de ello, en términos globales el porcentaje de ocupación del diseño en la FPGA es cercano al 10 %.

En particular, se han configurado 40 bandas (gracias a la reducción dimensional es suficiente con 10); 15 clústers (no suele ser necesario identificar más de 7 u 8); soporte para imágenes de 16 megapíxeles (la imagen completa de Cuprite solo tiene 1,36 millones de píxeles); y muestras de 16 bits.

Configuración sin reducción dimensional

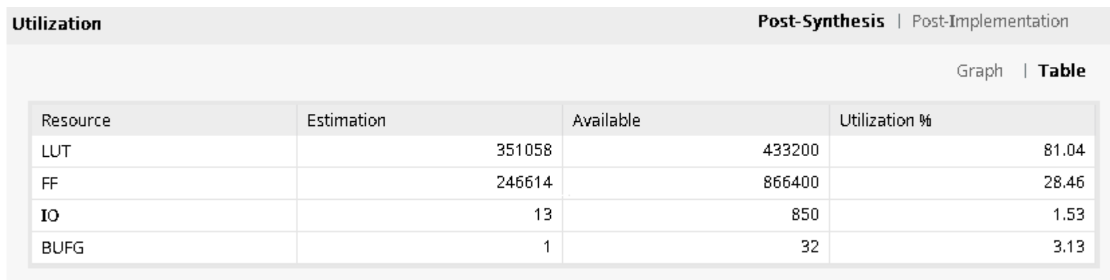
A modo de prueba, fue sintetizado un diseño con 224 bandas (las mismas que una imagen del AVIRIS sin preprocesar). En este caso, la utilización de LUTs se dispara hasta el 81 % de los disponibles, mientras que los *Flip Flops* alcanzan el 30 %. Dada la situación sería difícil emplazar otros componentes en la misma FPGA, por lo que las posibles aplicaciones del sistema se verían reducidas a funcionar únicamente como coprocesador (externo) en un codiseño *hardware-software*.

En la figura 5.13 se muestra la tabla de uso de recursos generada por Vivado para dicha



Resource	Utilization	Available	Utilization %
LUT	71002	433200	16.39
LUTRAM	340	174200	0.20
FF	81375	866400	9.39
BRAM	7	1470	0.48
IO	13	850	1.53
BUFG	2	32	6.25

Figura 5.11: Utilización de recursos con configuración extremadamente sobredimensionada. FPGA objetivo: Xilinx Virtex-7 XC7VX690T.



Resource	Estimation	Available	Utilization %
LUT	351058	433200	81.04
FF	246614	866400	28.46
IO	13	850	1.53
BUFG	1	32	3.13

Figura 5.13: Utilización de recursos sin reducción dimensional.

configuración con 224 bandas.

Con este tipo de configuraciones extremas se pretende demostrar los buenos resultados obtenidos en lo que a consumo de recursos se refiere, puesto que uno de los objetivos propuestos era no excederse en dicho aspecto. Así la implementación no deja de ser viable en otros modelos de FPGA más modestos.

Utilización de recursos con distintas configuraciones					
Configuración	Máx. píxeles	Bandas	Profundidad	Clústers	Uso LUTs
Básica	5.000	10	16 bit	4	8 %
Amplia	500.000	10	16 bit	8	12 %
Sobredimensionada	16.000.000	40	16 bit	15	16 %
Sin reducción	16.000.000	224	16 bit	15	81 %

Tabla 5.3: Resumen de utilización de recursos.

Resumen de configuraciones

La tabla 5.3 muestra un breve resumen sobre el uso de recursos obtenido en cada una de las configuraciones expuestas.

5.3.1. Análisis del uso de recursos para la configuración sobredimensionada

Para hacer un análisis más detallado en En la figura 5.11 se muestran los resultados *post-implementation* de utilización de la FPGA para la configuración sobredimensionada. De entrada, se ve como el uso de recursos es notablemente bajo a pesar de la inmensa cantidad de operaciones que se realizan al ejecutar el algoritmo, sobretodo teniendo en cuenta que los parámetros del núcleo están completamente sobredimensionados.

Esto se debe (en gran medida) al diseño en *pipeline*, que permite reutilizar todo el *hardware* en cada iteración. Asimismo, el alto grado de segmentación en los módulos implica que el aprovechamiento de recursos sea máximo, pues en cada ciclo de reloj se utilizan la totalidad de los componentes del sistema.

Si observamos cuidadosamente dichos resultados de utilización (figura 5.11), se tiene:

- LUTs (*Look Up Tables*): 16,39 %. Debido a la abundante presencia de lógica para las operaciones (que se realizan en paralelo para todas las bandas), son necesarios 400.000 LUTs.

- LUTRAM: 0,2 %. Dado que hay disponibles bastantes *Flip Flops* y *Block RAM*, no es necesario emplear LUTs como memoria.

- FF (*Flip Flops*): 9,39 %. El uso moderado de *Flip Flops* se debe principalmente a la segmentación del *pipeline* y los mecanismos de sincronización.

- BRAM (*Block RAM*): 7 bloques de 36 Kb. Los 252 Kbits de memoria se emplean principalmente para almacenar los datos de muestra y verificación del núcleo, ubicados en el módulo envoltorio que actúa como interfaz con la FPGA.

- IO (*Input-Output*): 13. Apenas se utilizan puertos de entrada-salida, dado que la comunicación con el exterior es mínima.

- BUFG (*Global Clock Buffer*): 2. Se utilizan dos *buffers* de reloj para solventar problemas de *fan out* en el diseño.

5.3.2. Impacto de la configuración del núcleo en el uso de recursos

En esta sección se analiza en qué recursos *hardware* repercute la variación de los parámetros de configuración del núcleo.

Número máximo de píxeles

Como ya se ha expuesto anteriormente, al aumentar el número máximo de píxeles soportado cabe esperar un mayor uso de recursos debido a dos factores: las representaciones de datos más anchas y el alargamiento del *pipeline*.

El primero es evidente, ya que serán imprescindibles contadores más anchos (ya que en ningún caso es viable emplear mecanismos de saturación en ellos), así como acumuladores y centroides (aunque en este caso sí sería factible recurrir a la saturación). El impacto de ello es directo en la utilización de biestables.

Por otro lado el incremento en la longitud del *pipeline* se debe al divisor multiciclo, que requerirá más etapas para procesar los operandos de mayor anchura. Esto implica que

también serán necesarios sincronizadores más largos. Por ello, se verán incrementados el uso de lógica y registros.

Número de clústers

El número de clústers a identificar repercute directamente en los módulos clasificador y actualizador de acumuladores, aunque también en el divisor (pero en menor medida).

Como es evidente el clasificador necesitará un número acorde de procesadores sistólicos en el array, y por tanto más lógica para el cálculo de distancia y el control. En el actualizador de acumuladores, repercutirá en el número de módulos que realizan la actualización en paralelo; de ese modo influye principalmente en la lógica empleada para las operaciones.

Asimismo en el divisor afecta al mecanismo que gestiona la actualización de los centroides, lo cual se verá reflejado en la lógica combinacional.

Además de todo lo anterior, el número de clústers también impacta en el uso de la entrada/salida. Esto se debe a la anchura empleada para representar los identificadores de los clústers, y esto es muy relevante porque el número de puertos de I/O suele ser limitado en las FPGAs. No obstante, si el núcleo se utiliza como módulo interno puede obviarse este hecho.

Número de bandas

El incremento en el número de bandas supone que gran parte del *hardware* será replicado para funcionar en paralelo (dado que las bandas se procesan de forma simultánea), luego el impacto en los recursos utilizados es considerable. Esto repercute tanto en uso de LUTs como de registros.

5.4. Consumo energético

Es habitual que los sensores hiperespectrales se utilicen embarcados en satélites, donde el suministro energético es muy limitado. Por ello es fundamental prestar atención al consumo en proyectos como este.

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

Total On-Chip Power:	**1.913 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**27.2°C**
Thermal Margin:	57.8°C (48.8 W)
Effective θJA:	1.1°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

Launch Power Constraint Advisor to find and fix invalid switching activity

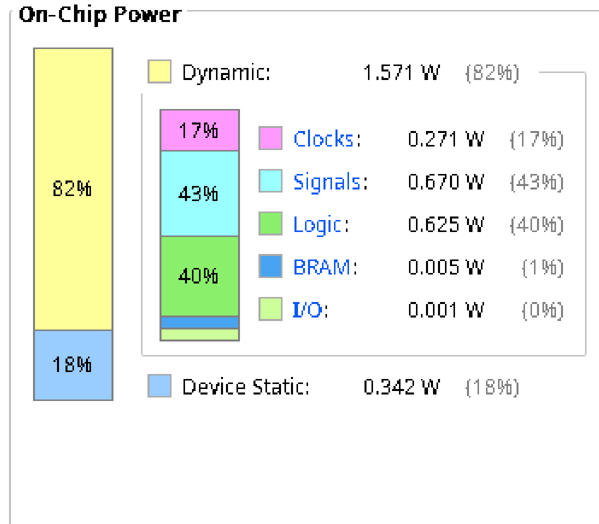


Figura 5.14: Estimación de consumo del núcleo.

Para obtener los datos se ha recurrido a las estimaciones calculadas por Vivado (la herramienta EDA empleada para sintetizar los diseños). Estas últimas se muestran en la figura 5.14. Por valorar un caso medio, la configuración del núcleo ha consistido en 10 bandas, 5 clústers, 16 millones de píxeles y 16 bits para las muestras. La frecuencia del reloj principal son 100 MHz.

Como cabría esperar, prácticamente la totalidad de la potencia disipada es dinámica; correspondiendo en su mayor parte con la lógica combinacional y los registros del sistema. Esto es bastante razonable, ya que la mayor parte de la implementación se compone de LUTs y *Flip Flops*.

También el reloj presenta un consumo notable, lo cual no es de extrañar, puesto que alcanza a todos los componentes secuenciales y por ende está muy presente en todo el diseño implementado.

En total, la potencia disipada son 1,91 watios. Un microprocesador actual de gama alta puede disipar entorno a 150 watios de potencia a pleno rendimiento, por lo que en comparación, este sistema consume cerca de 80 veces menos. Por poner un ejemplo, es similar a comparar el diámetro de un barreño con el largo de una piscina olímpica. La

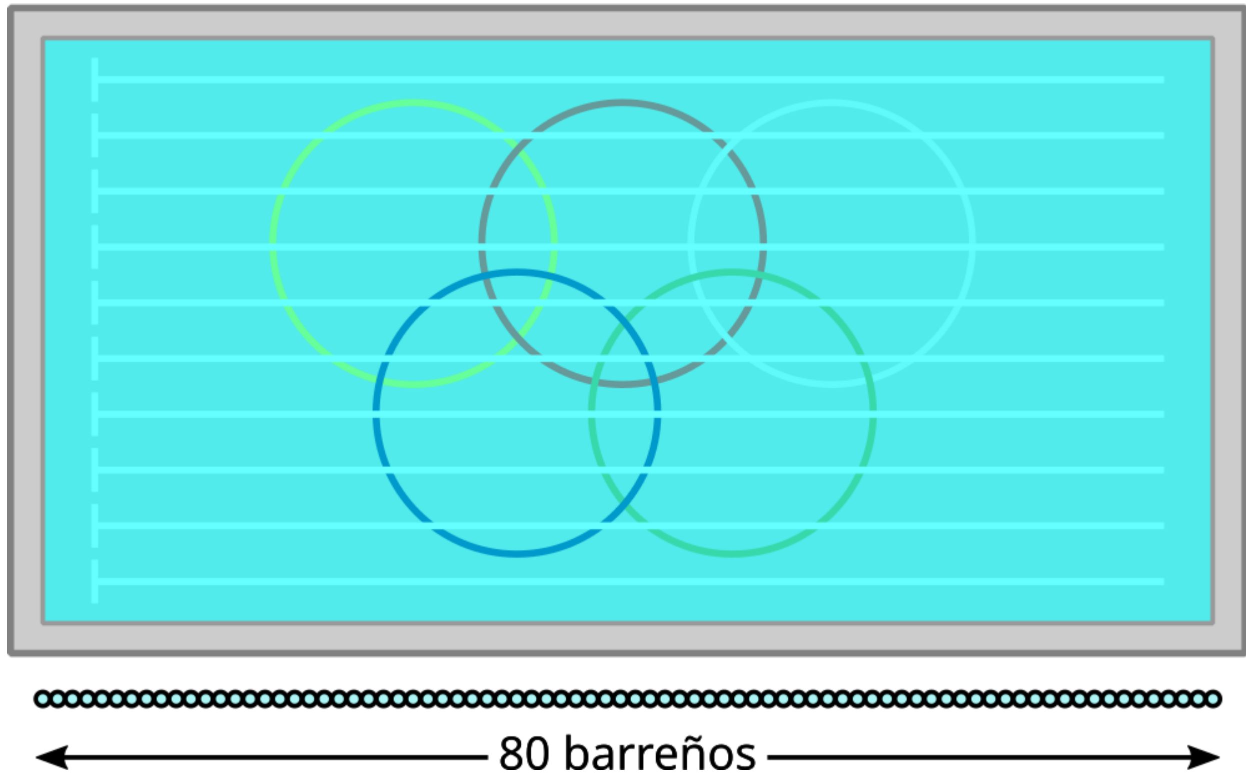


Figura 5.15: Comparación de una piscina olímpica con 80 barreños.

figura 5.15 ilustra la comparativa.

A pesar de la abismal diferencia de consumo entre los dos sistemas, la ganancia de rendimiento respecto al *software* es todavía mayor.

Capítulo 6

Conclusiones y trabajo futuro

Este capítulo cierra la memoria tratando las conclusiones que se han obtenido a raíz del trabajo realizado, así como las posibles vías de continuación.

6.1. Conclusiones

En base a la implementación *hardware* de k-means elaborada y los resultados expuestos en el capítulo anterior, se han obtenido las siguientes conclusiones.

6.1.1. Implementacion *software*

Abordar un diseño *hardware* desde cero es una tarea compleja que requiere bastante esfuerzo. Por ello realizar un primer acercamiento a la implementación mediante *software* con herramientas como Matlab, o en este caso Python, permite conocer la viabilidad de la solución propuesta y adelantar algunos de los problemas a solventar. Asimismo ayuda a conocer el dominio de la aplicación si se carece de experiencia previa, al igual que ha sucedido en este trabajo.

Otra ventaja que se ha experimentado a causa de realizar una primera implementación del algoritmo en *software* es que así se obtiene una base sólida para poder verificar posteriormente el comportamiento del *hardware* diseñado.

6.1.2. Modificaciones sobre el algoritmo original

Cambios estructurales

La mayoría de las adaptaciones realizadas al algoritmo para trasladarlo al *pipeline* segmentado han tenido un impacto muy positivo en el rendimiento del algoritmo. Entre ellas se incluyen el procesado en paralelo de todas las bandas de cada píxel, la sustitución de divisiones por desplazamientos de bits o el recálculo constante (y rápido) de centroides. Este último cambio logra que la convergencia se alcance en menos iteraciones, lo cual es muy ventajoso puesto que cada iteración implica procesar todos los píxeles de la imagen.

Optimización de recursos

Por otra parte, también se ha concluido que las adaptaciones realizadas con vistas a lograr un diseño viable (y que no utilice una cantidad desproporcionada de recursos en la FPGA) han tenido un impacto casi nulo en la precisión de los resultados. Ejemplos de ello son la reducción dimensional de las imágenes hiperespectrales, el uso de aritmética de enteros (en lugar de punto flotante) o el empleo de LFSRs para obtener valores aleatorios.

Sin embargo, lo más remarcable de estas últimas medidas son los magníficos resultados logrados en utilización de recursos (expuestos en el capítulo anterior, sección 5.3); puesto que a pesar de sintetizar el diseño con configuraciones totalmente desproporcionadas, se seguían obteniendo circuitos perfectamente emplazables en la FPGA.

6.1.3. Arquitectura en *pipeline*

El diseño en *pipeline* ofrece un rendimiento excepcional en el algoritmo implementado (al igual que en otros de naturaleza similar), dado que su segmentación tan profunda permite procesar multitud de datos simultáneamente.

En este caso el grado de paralelismo es extremadamente alto, puesto que todos los píxeles procesados a la vez en el *pipeline* son hiperespectrales, y por ello todas las bandas de cada uno también son tratadas simultáneamente. De esta manera se explota al mismo tiempo el

paralelismo a nivel de ciclo y de operación.

También es muy remarcable el hecho de que la arquitectura sea paramétrica, ya que así es posible configurarlo automáticamente para utilizar imágenes de distinto tamaño, con distinto número de bandas o distinta profundidad en las muestras; así como para identificar distinto número de clústers.

6.1.4. Desarrollo *hardware*

Tal y como se ha puesto de manifiesto a lo largo de esta memoria, desarrollar *hardware* exige un esfuerzo mucho mayor el *software*. El primero es mucho más "delicado", no puede tomarse una decisión de implementación sin considerar el contexto, pues todas las operaciones ocurren al mismo tiempo en vez de secuencialmente. Ni siquiera es trivial realizar una suma y almacenar su resultado. Tampoco existen grandes bibliotecas como las que hay en el mundo del *software*, con funciones para resolver casi cualquier problema sin apenas programar.

Además, no solo hay que tener en cuenta que el comportamiento especificado sea correcto, sino también que sean satisfechas las distintas restricciones físicas. En caso contrario el diseño sería inválido.

Las FPGAs y herramientas disponibles hoy en día facilitan mucho esta labor, llegando al punto ser posible generar un diseño *hardware* sin escribir una sola línea en HDL. Algunos ejemplos son la síntesis de alto nivel (HLS) o el compilador de OpenCL. Sin embargo, con estas técnicas el desarrollador pierde completamente el control del flujo de datos y a pesar de que la ganancia de rendimiento respecto al *software* sea notable, no es posible lograr ni por asomo diseños tan finos como a nivel RTL.

Simulación

No obstante, la herramienta EDA empleada (Vivado 2020.2) proporciona utilidades de simulación *hardware* muy completas, que agilizan bastante las tareas de implementación y depuración. Gracias a ellas ha sido posible analizar detalladamente los resultados y verificar

el comportamiento del núcleo diseñado.

Aun así el proceso no deja de ser complejo y lento, pues las simulaciones más ambiciosas han tardado cerca de un día entero en completarse. Eso implica que depurar conlleva muchas más horas de lo deseable, debido a la cantidad de tiempo que transcurre hasta obtener los resultados después de una modificación en el código. Para salvar la dificultad se han realizado las pruebas de la forma más acotada y escalonada posible.

6.1.5. Propiedades del núcleo

Aunque en el capítulo 5 se exponen de forma detallada los resultados obtenidos, en esta sección se incluye un breve resumen.

Rendimiento

El *pipeline* segmentado procesa a velocidad constante (sin paradas) de un píxel hiperespectral por ciclo, tal que todas las bandas (224 en caso del sensor AVIRIS) son tratadas en paralelo. Su longitud varía en función de la configuración empleada, pero típicamente oscila entre 40 y 60 etapas.

Dado que la velocidad de reloj establecida son 100 Mhz, el *pipeline* es capaz de procesar 100 megapíxeles hiperespectrales en un segundo. En comparación con los 0,14 megapíxeles por segundo que logra obtener el sensor hiperespectral AVIRIS, puede afirmarse que la velocidad de procesado está muy por encima de los límites de funcionamiento en tiempo real; ya que los datos se obtienen mucho más lento de lo que se procesan.

Por otro lado, gracias al mecanismo de actualización rápida de centroides, los datos son clasificados en base a información más actualizada y por tanto la convergencia se alcanza en menor número de iteraciones (que en la versión *software* original).

Consumo

El núcleo desarrollado no solo ofrece un rendimiento prácticamente inmejorable, si no también un consumo energético muy bajo. La potencia disipada total son 1,91 watios.

Esto lo convierte en un diseño ideal para embarcar en sistemas donde el ahorro energético es crucial; por ejemplo, en satélites aeroespaciales, como los empleados para la captación de imágenes hiperespectrales.

Utilización de recursos

Gracias a las medidas citadas hace unos párrafos (sección 6.1.2), se ha logrado un porcentaje de ocupación de la FPGA (Xilinx Virtex-7) inferior al 10 %, incluso para configuraciones extremadamente sobredimensionadas.

Esto quiere decir que sería posible incluso instanciar varios núcleos en un diseño para que trabajen en paralelo, o integrar uno solo como parte de otro sistema en la misma FPGA.

Otra conclusión es que resulta posible prescindir de la reducción dimensional, ya que el diseño configurado con 224 bandas (las mismas que capta el sensor AVIRIS) sigue siendo emplazable en la FPGA.

6.1.6. Conveniencia del diseño *hardware*

Si los procesos de diseño, implementación y depuración han resultado tan costosos: ¿acaso merecía la pena crear una versión *hardware* de k-means? La respuesta a dicha pregunta es realmente la conclusión final de este trabajo.

Para responderla, en la tabla 6.1 se han recopilado varios de los resultados expuestos en el capítulo 5.

En ella observamos que el tiempo de desarrollo de la versión *hardware* ha requerido cerca de 15 veces más tiempo que la implementada en *software*, luego cabría esperar algo que sea al menos 15 veces mejor, pero no es exactamente así.

Muy muy por encima de las expectativas, la versión *hardware* funciona cerca de 8.900 veces más rápido que el *software*. Además al ser un diseño determinista por naturaleza, procesar una misma imagen con la misma configuración siempre tardará el mismo tiempo.

Otra gran ventaja es que el *hardware* logra un consumo energético cerca de 80 veces menor. Por ilustrar la magnitud de la mejora, supongamos que un teléfono móvil corriente

¿Merece la pena el *hardware*?						
	T. desarrollo	F. mejor.	T. ejecución[1]	F. mejora	Consumo	F. mejora
Software	2 semanas	-14,5	55,74 s	8.861,7	150 w [2]	78,5
Hardware	29 semanas		6,29 ms		1,91 w	

Tabla 6.1: ¿Merece la pena el *hardware*?

consumiese 80 veces menos energía: sería suficiente cargarlo una vez cada dos meses en lugar de todos los días.

En particular para el diseño elaborado, existe un importante factor a su favor: la escalabilidad. Este es más difícil de cuantificar, luego no se ha incluido en la tabla. Pero al ser paramétrico el núcleo, resulta inmediato extenderlo de forma horizontal para adaptarlo a casos de uso concretos; en muchos casos con un impacto nulo (o casi nulo) en el rendimiento. También su interfaz AXI Stream permite integrarlo directamente en otros flujos de procesamiento.

Veredicto

Por todo ello, puede concluirse que diseñar un *pipeline* en *hardware* es la forma de lograr los resultados más óptimos posibles. El tiempo y la dedicación que ello requiere lo convierte en una solución orientada a sistemas de propósito específico, con un ámbito de aplicación muy concreto donde las restricciones de rendimiento sean estrictas.

Aunque actualmente existan herramientas avanzadas que permiten crear diseños cargables sobre FPGA abstrayendo al desarrollador del propio *hardware*, es imposible alcanzar con ellas el mismo grado de optimización que al trabajar a nivel RTL. Por este motivo dichas alternativas quedan descartadas.

En cualquier caso, lo que sí resulta muy ventajoso es el uso de FPGA como plataforma

[1]Tiempo de ejecución del algoritmo para identificar 7 clústers en una imagen de 78.592 píxeles con 10 bandas hiperespectrales y muestras de 16 bits.

[2]Potencia máxima disipada por el microprocesador empleado.

de desarrollo, gracias a la gran flexibilidad que proporciona frente a otras alternativas como los ASICs (que no son dinámicamente reconfigurables).

6.2. Trabajo futuro

A lo largo de esta sección se lista una serie de posibles vías para continuar con el desarrollo del sistema.

6.2.1. Reductor de bandas

Aunque el núcleo puede funcionar con imágenes hiperespectrales sin modificar (como se ha expuesto en la sección 5.3 de los resultados), este ha sido ideado para funcionar con la reducción dimensional ya aplicada en los datos de entrada.

Por ello se propone como posible vía de continuación del proyecto implementar un módulo para acelerar la reducción dimensional. Este iría conectado a la entrada del *pipeline* para suministrar directamente la imagen hiperespectral al núcleo.

6.2.2. Generación de un *IP soft core*

Con vistas a facilitar la reutilización del sistema, se sugiere el empaquetado del núcleo en forma de *IP soft core* configurable. Así el proceso de integración en otros sistemas sería inmediato y no requeriría lidiar con los ficheros que contienen el código en HDL, por tanto se prevendrían potenciales errores derivados del uso de la herramienta EDA.

Además, gracias a que el diseño del sistema es paramétrico, sería posible configurar las características del mismo fácilmente (número de bandas, anchura de muestra, número de clústers...), a través de una interfaz gráfica que asista al usuario en el proceso.

El proceso es viable dado que Vivado proporciona una utilidad para la creación de *IP soft cores* compatibles con su entorno.

6.2.3. Integrar el núcleo en un codiseño *hardware-software*

Otra alternativa, que además permitiría sacar partido al trabajo realizado en este proyecto, sería crear un sistema en el cual integrar el núcleo. Este estaría implementado también en la FPGA y se compondría de un *soft processor* y una interfaz de memoria (para conectar con módulos externos de RAM). De este modo, la flexibilidad del procesador de propósito general permitiría ejecutar distintos programas o volcar al exterior información sobre el núcleo (útil para depuración y obtención de resultados).

Una característica del diseño que facilitaría esta vía de continuación es el uso del bus AXI para la entrada/salida, pues es el mismo estándar que típicamente emplean otros componentes como los citados procesador e interfaz de memoria.

Para llevarlo a cabo se haría uso de la herramienta Vitis, que está orientada a la creación de este tipo de sistema. Un aspecto ventajoso de Vitis es que está desarrollado por la misma empresa que Vivado (la herramienta usada en la implementación del núcleo), luego ambas aplicaciones son compatibles entre sí.

Asimismo, la línea de trabajo propuesta en el apartado anterior ("Generación de un *IP soft core*") ayudaría a la realización de esta otra, por lo que se recomienda realizarlas conjuntamente.

6.2.4. Experimentación en profundidad

En caso de continuar el trabajo de acuerdo a la vía anterior, sería posible construir un entorno para realizar pruebas sobre el núcleo *hardware* de forma extensiva. De esta forma sería posible validar el funcionamiento real del sistema una vez cargado en la FPGA. El principal objetivo es poder lanzar pruebas de forma automática y monitorizar cómo se comporta el núcleo en cada una de ellas, con vistas la realización masiva de experimentos aprovechando el procesamiento en tiempo real que permite el diseño.

Además, esto permitiría tener un banco con múltiples configuraciones del núcleo para así realizar pruebas con imágenes de distinto tamaño, profundidad de bits o número de bandas.

De igual modo podría identificarse diferente número de clústers. Así las pruebas no estarían restringidas a una única configuración.

Esta implementación también requeriría desarrollar una interfaz de memoria DRAM, para poder almacenar las imágenes hiperespectrales y las caches con los datos de la clusterización. De lo contrario no sería posible almacenar los datos en la *block RAM*.

La gran ventaja de todo ello es que el mismo entorno podría reutilizarse para validar o realizar pruebas sobre FPGA para otros diseños similares, más allá del núcleo con k-means.

6.2.5. Automatizar la generación de *pipelines* en HDL

Además de k-*means*, existe una gran multitud de algoritmos aplicables al procesado de imágenes hiperespectrales, los cuales pueden implementarse en un *pipeline* segmentado. Salvando las particularidades de cada implementación, es frecuente encontrar fuertes similitudes entre ellos.

De este modo, se plantea la creación de un entorno automatizado para el desarrollo de arquitecturas de propósito específico, tales como la elaborada en este proyecto. La estrategia de implementación se basaría en la definición de un conjunto de núcleos simples reutilizables extremadamente optimizados que al interconectarlos logren implementaciones altamente segmentadas de gran rendimiento.

El entorno permitirá especificar el grafo de procesamiento del algoritmo hiperespectral en base a un conjunto de núcleos software interconectados, así como de ligaduras. Los núcleos de software se trasladarán a la FPGA sobre núcleos hardware análogos que cumplan con las mismas restricciones.

La idea es similar a las herramientas de síntesis de alto nivel ya existentes, pero particularizadas a un nicho de aplicación, a un tipo concreto de arquitectura objetivo y con un nivel de granularidad mayor.

Bibliografía

[1] Wikipedia. Espectro visible. https://es.wikipedia.org/wiki/Espectro_visible, 2022.

[2] Wikipedia. Hyperspectral imaging. https://en.wikipedia.org/wiki/Hyperspectral_imaging, 2022.

[3] Stefan Paulus and Anne-Katrin Mahlein. Technical workflows for hyperspectral plant image assessment and processing on the greenhouse and laboratory scale. *GigaScience*, 9, 08 2020.

[4] Wikipedia. k-means clustering. https://en.wikipedia.org/wiki/K-means_clustering, 2022.

[5] Daniel Báscones García. Técnicas de compresión de imágenes hiperespectrales sobre hardware reconfigurable. *Universidad Complutense de Madrid*, 2020.

[6] Sergio Torres Martínez. Implementación en FPGA usando SDSoC de un filtro espacio-espectral de mapas de clasificación hiperespectrales para detección de tumores cerebrales. *Escuela Técnica Superior de Ingeniería y Sistemas de Telecomunicación*, 2017.

[7] Stuart P. Lloyd. Least squares quantization in pcm. *IEEE Transactions on Information Theory*, 28:129–137, 1982.

[8] Edward W. Forgy. Cluster analysis of multivariate data: efficiency versus interpretability of classifications. *Biometrics*, 21:768–780, 1965.

[9] Donald G. Bailey. The advantages and limitations of high level synthesis for fpga based image processing. In *Proceedings of the 9th International Conference on Distributed*

Smart Cameras, ICDSC '15, page 134–139, New York, NY, USA, 2015. Association for Computing Machinery.

[10] Fraser D Robinson, Louise H Crockett, William H Nailon, and Robert W Stewart. High-level synthesis for medical image processing on systems on chip: A case study. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, 2016.

[11] Yanjing Bi, Chao Li, and Fan Yang. Very high level synthesis for image processing applications. In *ICDSC 2016*, pages 160–165, 09 2016.

[12] M. Akif Özkan, Oliver Reiche, Frank Hannig, and Jürgen Teich. A Highly Efficient and Comprehensive Image Processing Library for C++-based High-Level Synthesis. In *FPGAs for Software Programmers (FSP 2017)*, September 2017.

[13] F. Muslim, Liang Ma, Mehdi Roozmeh, and Luciano Lavagno. Efficient fpga implementation of opencl high-performance computing applications via high-level synthesis. *IEEE Access*, PP:1–1, 02 2017.

[14] Colin Taylor and Michael Gowanlock. Accelerating the yinyang k-means algorithm using the gpu. In *2021 IEEE 37th International Conference on Data*, pages 1835–1840, 04 2021.

[15] Can Yang, Yin Li, and Fenhua Cheng. Accelerating k-means on GPU with CUDA programming. In *IOP Conference Series: Materials Science and Engineering*, volume 790, page 012036. IOP Publishing, 03 2020.

[16] Alexander Andreev and Egor Smirnov. Accelerate k-means clustering with intel xeon processors. *The Parallel Universe*, 39(1):61–69, 03 2020.

[17] Dominique Lavenier. Fpga implementation of the k-means clustering algorithm for hyperspectral images. *LA-UR 00-3079*, 09 2000.

[18] NASA. Airborne visible / infrared imaging spectrometer. https://aviris.jpl.nasa.gov/, 2022.

[19] Xilinx Inc. *7 Series FPGAs Data Sheet: Overview.*

[20] M. Leeser, J. Theiler, M. Estlick, and J.J. Szymanski. Design tradeoffs in a hardware implementation of the k-means clustering algorithm. pages 520–524, 2000.

[21] ARM. *AMBA 4 AXI4-Stream Protocol.*

[22] Xilinx Inc. *Divider Generator LogiCORE IP Product Guide.*

[23] Xilinx Inc. *Integrated Logic Analyzer.*

[24] Peter J. Ashenden. *The designer's guide to VHDL.* Elsevier Science & Technology, 2008.

[25] Brock J. LaMeres. *Quick Start Guide to Verilog.* Springer, 2019.

Apéndice A

Introduction

A.1. Motivation

We have been doing the same for at least three hundred thousand years, when we appeared on this planet. The ambition we have as humans to go beyond our capabilities is limitless. Our hands were insufficient, so we created spears for hunting; we didn't like to be thirsty, so we invented the *botijo*; and now we are neither satisfied with walking so we ended up developing automobiles.

It is exactly the same with our eyes: we want to see further than we can. And that has led to the emergence of new tools and techniques to obtain visual information overcoming our natural limits. In particular, humans are only able to perceive a small portion of the electromagnetic spectrum (also known as the visible spectrum and shown in the figure A.1);

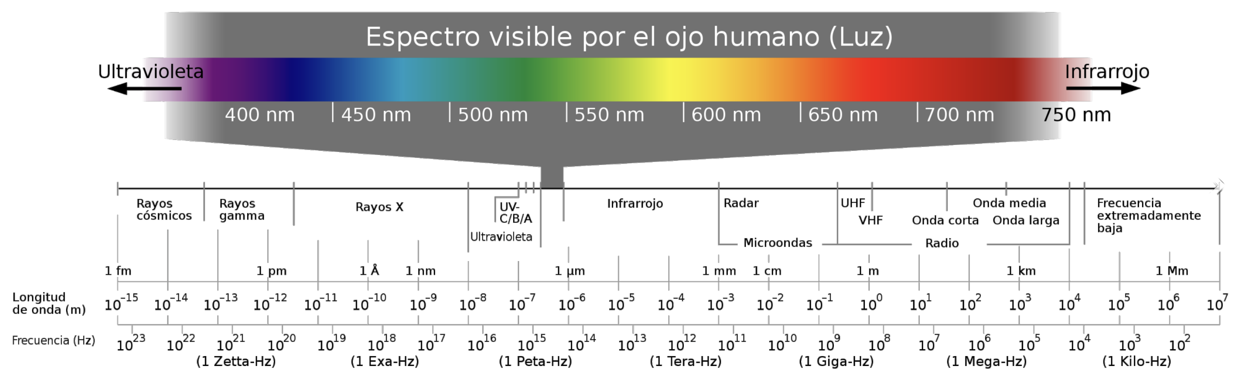


Figura A.1: Visible spectrum. Source: Wikipedia [1].

Figura A.2: Hyperspectral cube. Source: Wikipedia [2].

which means that there is light, there are colors, that we cannot see.

Or at least we could not until the first spectral sensors appeared, which have made possible a lot of advances in many fields, such as the study of minerals, agriculture, astronomy or food processing (to name a few). Thus, by being able to capture more wavelengths radiated by the matter of analysis, the information obtained about it is greatly expanded. Figure A.2 shows a hyperspectral cube, typically used to represent hyperspectral images.

However, obtaining huge amounts of information is still overwhelming for our understanding and we often need a preliminary step for our brain to take advantage of it. That is why it is often desirable to process hyperspectral data first, in order to make easier to formulate hypotheses and then draw conclusions from them.

Such processing is usually a well-defined task, composed of clearly scripted steps and therefore capable of being automated. In other words, those are the so-called algorithms. We usually define algorithms by means of software, with sequences of instructions that some computer will execute over and over, non-stop, without any rest. Just like a digital slave unable to feel anything, whose only purpose is to obey firmly the orders it receives.

We are lost in our own eagerness to get results quickly, but it is also us who created such a slow machine. This leads to a terrible frustration, leaving us exhausted and demoralised. However, the solution is simple: to invest a little more time implementing the algorithm, and then save it in each of the countless times it will be executed. The straightest way to achieve this is creating a hardware algorithm.

At this point, it is easy to guess the purpose of this work: develop a solution for a hyperspectral imaging related problem, by designing a fast and efficient hardware design, specifically aimed for that.

A.2. Objectives

The main objective of this work is to elaborate a hardware design at register-transfer level (RTL) which implements the "k-means" clustering algorithm, this will be oriented to real time operation on FPGA.

To achieve this it will be necessary to carry out a software approach first, as well as simulations, in order to guarantee the plausibility of the project.

On the other hand, the design must be fully parametric, so that it can be easily adapted to specific solutions, regardless of the characteristics of the hyperspectral sensor that obtains the input image or the properties required in the output obtained.

Despite the large volume of information contained in hyperspectral images, it is also intended to achieve real-time operation.

With the aim of meeting performance constraints, the architecture will consist of a pipeline, which will exploit various parallelisation techniques in order to maximise performance.

However, it will also be essential to find an optimal balance between hardware resource utilisation and performance, such that it is possible to place the design on an FPGA.

A.3. Background

There are many previous works that use FPGAs to process hyperspectral images [5, 6]; although software implementations are most frequent, those are not much relevant to this project.

Specifically, the k-means clustering algorithm was originally proposed by Stuart Lloyd in 1957 [7], although it was also published by Eduard W. Forgy [8]. Lloyd's proposal focused the algorithm on PCM modulation, however, today k-means is one of the most widely used techniques for image classification in Machine Learning.

Thanks to techniques such as high-level synthesis or the automatic generation of HDL code from software, the number of solutions involving FPGAs has proliferated enormously in recent years. However, at present, these techniques do not achieve as optimal a performance as RTL descriptions, in which developers have full control over the data flow. Reference [9] discusses that issue in detail.

This sort of researches often conclude that using FPGAs to process hyperspectral images greatly improves processing speed, while also providing excellent scalability. Some examples with HLS are the references [10–12], or in OpenCL [13].

About k-means itself, GPU implementations are common (as with other image processing algorithms), for example [14, 15].

There are also implementations that aimed to accelerate k-means by using CPUs, such as the one proposed in the reference [16].

Article [17] proposes a mixed implementation of k-means, combining a general-purpose computer and an FPGA to speed up the first part of the processing. This is based on a very simple, systolic array of processors (more information in the section 2.7), which has the advantage of hardly requiring any control mechanisms. However, the non-accelerated operations represent a considerable bottleneck.

A.4. Work plan

The development of this project started on November 2021 and finished on June 2022. It has been organised in sequential phases, being prioritised the ones which caused more dependencies. These stages are listed below (in chronological order):

1. **Project approach.**

   Before starting the development, the main objective of this project was set: to implement a hyperspectral image processing algorithm on an FPGA.

2. **Algorithm determination and feasibility analysis.** *(2 weeks).*

   Once project nature was established, a process of searching for algorithms to implement began. For each possible algorithm, different implementation options were evaluated to ensure that the project could be completed in time. Finally, k-means was selected.

3. **Software implementation of k-means.** *(2 weeks).*

   First of all, k-means was implemented in software. This was useful to get to know the application domain (hyperspectral image processing), as well as the algorithm itself.

4. **Software validation** *(1 week).*

   On the one hand, it was essential to check that the implemented software was correct, so several scripts were programmed to validate the results.

   On the other hand, there will foreseeably be many restrictions implicit to hardware design, for example the generation of random numbers by using an LFSR or the loss of precision after replacing floating point arithmetic. It was therefore necessary to test the effects of these on the accuracy and convergence of the algorithm.

5. **Architecture design.** *(6 weeks).*

Right after validating the software version, hardware began to be designed, module by module. In this phase, the main pillars of the architecture were established: a segmented *pipeline* that processes one pixel per cycle. Data exchange and synchronisation mechanisms were also conceived.

For each module, many design alternatives were proposed, considering several variables such as resource sharing, parallelism, segmentation or internal control complexity. This way, there were variants with trade-offs between higher performance, lower resource usage and others simpler control.

6. **Pipeline implementation.** *(3 weeks).*

   Pipeline started to be implemented (module by module) after completing the whole design stage. This process also involved the development of shared components and their validation via simulations, to ensure that they will work properly once integrated.

   At this point, all the design had already been developed, so this phase essentially consisted of describing it in VHDL.

7. **Control module implementation.** *(2 weeks).*

   The main control module was described after the pipeline. This process took more time and effort than the previous, due to its more complex logic. In fact, it also includes the main state machine.

8. **Core integration and debug.** *(3 weeks).*

   Once the modules were fully implemented, they were integrated to build up the core.

   Next step was to debug hardware errors, first until it could be synthesised, then until core-wide simulations gave coherent and reasonable results.

9. **Core verification.** *(4 weeks).*

   To verify the results produced by the core, it was used a progressive method. Various types of images (both synthetic and real) were used in simulations of the entire design

in order to validate its behaviour. It was essential to perform the tests in a stepwise manner, otherwise it wouldn't have been possible to locate and correct the errors.

The first software implementation of k-means was used in this process. Also, several scripts were coded to verify the results.

10. **FPGA wrapper implementation.** *(2 weeks).*

    This stage involved the design and implementation to load and test the core (which had already been debugged and verified) on the FPGA.

11. **FPGA wrapper simulation.** *(2 weeks).*

    Just as previously done with the core alone, a post-implementation simulation was performed on the complete system, in order to ensure that it would work on the board.

A.5. Organisation of this report

This report is structured in several chapters, described below:

- **Technological context**: it compiles all resources that have been essential to accomplish this project. Also contains technical information about core components, protocols, standards, implemented algorithms and used tools.
- **Hardware architecture**: contains the core's hardware architecture, including the pipeline and control mechanisms.
- **Validation**: presents all the testing and simulations performed to ensure the proper operation of the algorithm.
- **Results**: analyses results and details the objectives achieved. It also includes the features of the final system: performance, resource usage, power consumption...

- **Conclusions and future work**: everything that has been concluded during the development of this project. Different ways to keep developing this work are presented in this section too.
- **Bibliography**: list of source materials that are referred to in this report (articles, books, previous works...).

Apéndice B

Conclusions

Based on the k-means hardware implementation and the results shown in the previous chapter, the following conclusions have been drawn.

B.1. Software implementation

Creating a hardware design from the ground up is a complex task that requires quite a lot of effort.

Therefore, a first approach to the implementation using software tools such as Matlab or Python, allows to check the viability of the proposed solution as well as to anticipate some of the problems to be solved. It also helps getting to know the application domain, which is useful when there is a lack of previous experience, as happened in this case.

Another advantage of implementing the algorithm in software is that it provides a solid basis for later verification of the hardware's behaviour.

B.2. Modifications to the original algorithm

Structural changes

Most of the modifications made to the algorithm in order to implement it as a segmented pipeline have had a very positive impact on its performance. These changes include: parallel processing of all bands, replacing divisions by bit shifts and the mechanism for fast centroid

updating. The latter implies that convergence is achieved in fewer iterations, which is very advantageous since each iteration involves processing all the pixels in the image.

B.2.1. Resource optimization

On the other hand, it has also been concluded that the adjustments made to achieve a feasible design (which means not using a disproportionate amount of FPGA resources) have had almost no impact on the accuracy of the results. Examples are dimensional reduction, integer arithmetic (instead of floating point) or using LFSRs to obtain random values.

Nevertheless, the most remarkable aspect of these adaptations is the magnificent results achieved in terms of resource utilisation (described in the previous chapter, section 5.3), since, despite synthesising the design with totally disproportionate configurations, circuits still could be perfectly placed on the FPGA.

B.3. Pipeline architecture

The pipeline design offers exceptional performance for k-means (as in other similar algorithms), since the high degree of segmentation allows processing multiple data simultaneously.

In this case the degree of parallelism is extremely high, since all the pixels processed at the same time in the pipeline are hyperspectral, and therefore all the bands of each one are also processed simultaneously.

The core is highly parallel, since all those concurrently processed pixels are hyperspectral, which means that they have multiple (up to 224) bands, also being processed at the same time. In this way, parallelism is exploited at cycle and operation level alltogether.

It is also remarkable that the architecture is parametric: it can be automatically configured to use images with different sizes, number of bands and sample depths; as well as to identify different number of clusters on it.

B.4. Hardware development

As has been shown throughout this report, developing hardware requires much more effort than software. The former is much more "delicate", an implementation decision cannot be taken without considering the context, as all operations occur at the same time instead of sequentially. It is not even trivial to perform an addition and store its result. Nor there are large libraries like those in the software world, with functions to solve almost any problem without hardly programming.

Moreover, it is not only necessary to take into account that the specified behaviour is correct, but also that the different physical constraints are met. Otherwise the design would not be valid.

The FPGAs and tools available nowadays make this task much easier, to the point where it is possible to generate a hardware design without writing a single line in HDL. Examples are high-level synthesis (HLS) or the OpenCL compiler. However, with these techniques developers completely lose control of the data flow, and although the performance gain over software is significant, it is not possible to achieve a such a fine design as at RTL level.

B.4.1. Simulation

However, the EDA tool used (Vivado 2020.2) provides very rich hardware simulation utilities, which speed up the implementation and debugging tasks noticeably. They have made possible to analyse the results in detail and verify the behaviour of the designed kernel.

Even so, the process is still complex and time-consuming, with the most ambitious simulations taking nearly a full day to complete. This means that debugging takes many more hours than desired, because obtaining results after a code modification is a very slow process. In order to overcome this difficulty, the tests have been carried out in the most gradual and staggered way possible.

B.5. Core's properties

Although a detailed account of results is given in chapter 5, a brief summary is included in this section.

B.5.1. Performance

The pipeline processes at a constant rate (non-stop) of one hyperspectral pixel per cycle, such that all bands (224 for the AVIRIS sensor) are processed in parallel. Its length varies upon the configuration used, but typically ranges between 40 and 60 stages.

Since the clock rate is set at 100 Mhz, the pipeline is capable of processing 100 hyperspectral megapixels in one second. Compared to the 0.14 megapixels per second at which the AVIRIS hyperspectral captures images, it can be stated that the processing speed is well beyond the limits of real-time operation; because data is acquired much slower than it is processed.

On the other hand, thanks to the fast centroid update mechanism, the data is classified using more up-to-date information. Therefore, the algorithm converges in fewer iterations (than in the original software version).

B.5.2. Power consuption

The developed core not only offers an almost ideal performance, but also a very low power consumption. The total power dissipation is 1.91 watts.

This turns it into a perfect design for on-board systems where saving energy is critical, for example in aerospace satellites, such as those used for hyperspectral imaging.

B.5.3. Resource usage

Thanks to the methods mentioned a few paragraphs ago (section B.2), it could be achieved a very low resource usage on a Xilinx Virtex-7. For an oversized setup, less than 10 % of the FPGA is used.

Is hardware worth it?						
	Dev. time	Improvement	Exec. time[1]	Improvement	Power	Improvement
SW	2 weeks	-14.5	55.74 s	8,861.7	150 w [2]	78.5
HW	29 weeks		6.29 ms		1.91 w	

Tabla B.1: Is hardware worth it?

This means that it would even be possible to instantiate several cores in one design (maybe to work in parallel), or to integrate a single core taking part of another system inside the same FPGA.

Another conclusion is that dimensionality reduction is not strictly necessary, as it is still possible to configure the design for 224 bands images (same as the AVIRIS sensor) and fit the design in the FPGA.

B.6. Hardware design convenience

If design, implementation and debug processes have been so time-consuming and complex: was it worth creating a *hardware* version of k-means? The answer to that question is really the final conclusion of this report.

To answer it, table B.1 summarises several results already presented in chapter 5.

On it we see that the development time of the hardware version took about 15 times more than the one implemented in software, so we would expect something that is at least 15 times better, but it is not exactly there.

Far above expectations, the hardware version runs around 8,900 times faster than the software one. Moreover, being (hardware) a deterministic design by nature, processing the same image with the same settings will always take the same amount of time.

[1] Execution time to identify 7 clusters in a 78,592 pixel image with 10 hyperspectral bands and 16-bit samples.

[2] Maximum dissipated power by CPU

Another great advantage is that the hardware achieves around 80 times lower power consumption. To illustrate the magnitude of the improvement, suppose an ordinary mobile phone that consumes 80 times less energy: it would be enough to charge its battery once every two months instead of every day.

Particularly for the elaborate design, there is an important factor in its favour: scalability. This is more difficult to quantify, so it has not been included in the table. But because the core is parametric, it is easy to extend it horizontally to suit specific use cases, in many cases with zero (or almost zero) impact on performance. Its AXI Stream interface also allows it to be integrated directly into other processing flows.

B.6.1. Veredict

For all these reasons, it can be concluded that designing a pipeline in hardware is the way to attain the most optimal results possible. The time and dedication that requires makes it a solution oriented to specific purpose systems, with a very specific scope of application where performance constraints are tight.

Although nowadays there are plenty of advanced tools that can be used to create loadable designs on FPGAs, but fully abstracting the developer from the hardware itself. It is impossible to achieve the same degree of optimisation with them as when working at RTL level. For this reason, these alternatives are rejected.

Anyhow, using an FPGA as development platform is a great choice, thanks to the flexibility it provides compared to other alternatives such as ASICs (which are not dynamically reconfigurable).