UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
Instituto Universitario de Microelectrónica Aplicada
Sistemas de información y Comunicaciones

Máster en
Tecnologías de Telecomunicación

Trabajo Fin de Máster

Metodología para el sintetizado hardware del algoritmo Vertex Component Analysis (VCA) implementado en Matlab

Autor: Javier Martín Abasolo
Tutor(es): Gustavo Marrero Callicó
Sebastián López Suárez
Fecha: 20 -julio - 2011

IUMA
t +34 928 451 086 | iuma@iuma.ulpgc.es | Campus Universitario de Tafira
f +34 928 451 083 | www.iuma.ulpgc.es | 35017 Las Palmas de Gran Canaria

Versión: 2.0 – 14 jun 2011

UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
Instituto Universitario de Microelectrónica Aplicada
Sistemas de información y Comunicaciones

Máster en
Tecnologías de Telecomunicación

Trabajo Fin de Máster

Metodología para el sintetizado hardware del algoritmo Vertex Component Analysis (VCA) implementado en Matlab

HOJA DE FIRMAS

Alumno/a: Javier Martín Abasolo Fdo.:

Tutor/a: Gustavo Marrero Callicó Fdo.:

Tutor/a: Sebastián López Suárez Fdo.:

Fecha: 20 -julio - 2011

IUMA
t +34 928 451 086 | iuma@iuma.ulpgc.es
f +34 928 451 083 | www.iuma.ulpgc.es
Campus Universitario de Tafira
35017 Las Palmas de Gran Canaria

Versión: 2.0 – 14 jun 2011

UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
Instituto Universitario de Microelectrónica Aplicada
Sistemas de información y Comunicaciones

Máster en
Tecnologías de Telecomunicación

Trabajo Fin de Máster

Metodología para el sintetizado hardware del algoritmo Vertex Component Analysis (VCA) implementado en Matlab

HOJA DE EVALUACIÓN

Calificación: .....

Presidente: Fdo.:

Secretario: Fdo.:

Vocal: Fdo.:

Fecha: 20 -julio - 2011

IUMA
t +34 928 451 086 | iuma@iuma.ulpgc.es | Campus Universitario de Tafira
f +34 928 451 083 | www.iuma.ulpgc.es | 35017 Las Palmas de Gran Canaria

Versión: 2.0 – 14 jun 2011

INDICE

INDICE DE FIGURAS

INDICE DE TABLAS

1. Introducción y objetivos

El avance de la tecnología en los sensores electrónicos y en la óptica experimentado en los últimos años ha permitido el desarrollo de sensores hiperespectrales [1] que logran obtener imágenes de multitud de bandas del espectro electromagnético. La creciente demanda de datos hiperespectrales en el campo de la teledetección para su utilización en aplicaciones medioambientales y de recursos minerales ha impulsado el uso de este tipo de sensor en la teledetección aerotransportada y espacial.

Los sensores hiperespectrales tienen una gran utilidad para la detección y clasificación de los diferentes materiales que forman parte de la superficie terrestre. La radiación solar que atraviesa la atmósfera interactúa con la superficie terrestre, encontrándose con diferentes materiales. Cada tipo de elemento de la superficie reacciona de una forma diferente a la radiación solar absorbiendo la radiación en determinadas longitudes de onda y reflejándola en otras, a este fenómeno se le denomina firma espectral [2]. Gracias a esta firma espectral se pueden identificar los diferentes materiales presentes en la superficie terrestre. Por lo tanto, la utilización de sensores hiperespectrales que toman centenares de muestras del

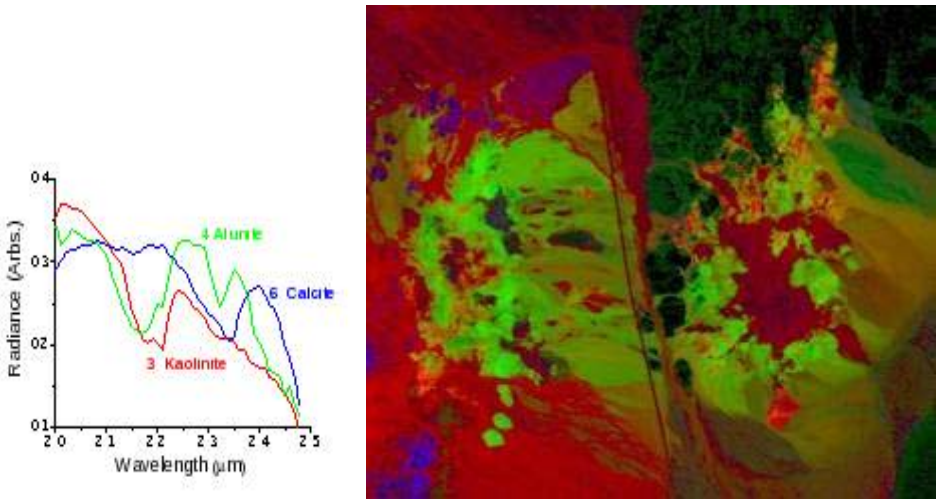
espectro electromagnético permite obtener la firma espectral de los diferentes elementos contenidos en una imagen.

Figura 1. Imagen hiperespectral, donde el eje z representa las diferentes longitudes de onda. Fuente: *http://rst.gsfc.nasa.gov/Sect13/Sect13_9.html*

La detección de los diferentes materiales de la superficie se complica cuando los elementos observados se visualizan mezclados por causa de limitaciones de la resolución espacial.

Para lograr mitigar el efecto de la mezcla de los píxeles de una imagen se han implementado algoritmos que permiten determinar los píxeles puros que forman parte de la superficie muestreada del mismo material respecto a los píxeles mezclados que representan el promediado lineal de diferentes superficies fronterizas.

El algoritmo de extracción de *Endmembers* [3] permite obtener los píxeles puros de los mezclados en las imágenes hiperespectrales. Uno de los algoritmos más utilizados en la extracción de *Endmembers* es el *Vertex Component Analysis* (VCA) [4]. Este algoritmo matricial requiere de una gran cantidad de computación y teniendo en cuenta su gran utilidad en la utilización de sensores hiperespectrales resulta de gran interés la síntesis hardware de estos algoritmos implementados de forma habitual en Matlab [5] para su integración en los sensores hiperespectrales.



(a) (b)

Figura 2. Endmembers correspondientes a minerales (a), abundancia de estos materiales en una imagen del sensor AVIRIS (b). Fuente: *http://www.spectral.com/remotesense.shtml*

El objetivo de este Trabajo Fin de Máster (TFM) es desarrollar una metodología para la conversión de un código Matlab a un lenguaje de descripción hardware como Verilog o VHDL [6], que nos permita comparar los resultados obtenidos en la síntesis desde un lenguaje de alto nivel y de rápido prototipado con los resultados obtenidos por una descripción hardware ad-hoc de mayor complejidad y de mayor tiempo de desarrollo. La ejecución de estos algoritmos implementados en Matlab en un dispositivo hardware reprogramable tiene limitaciones computacionales que requiere de la utilización de múltiples herramientas software para su correcta conversión. Para la realización de esta metodología se ha de utilizar tanto las herramientas que proporciona Matlab para la obtención de código C/C++ embebido (*Embedded C Toolbox*) [7] como herramientas para la utilización de aritmética de punto fijo más adecuada para dispositivos electrónicos como la librería matemática de punto fijo de Matlab (*Fixed-Point Toolbox*) [8] u otras librerías de punto fijo para el lenguaje C/C++ [9]. Mediante estas herramientas se pretende lograr un código C/C++ independiente que haga uso de la aritmética de punto fijo adecuada para la utilización de dispositivos electrónicos como FPGAs.

Para obtener el código de descripción hardware que pueda ser ejecutado en FPGAs es necesario un último paso, que es la utilización de herramientas de conversión de código

C/C++ a código de descripción hardware RTL como Verilog/VHDL. La herramienta que vamos a utilizar se denomina Catapult C [10] y permite la obtención de lenguajes de descripción hardware a partir de código C/C++ independiente.

El objetivo del TFM es obtener una metodología base para la generación de código de descripción hardware mediante modificaciones de configuración necesarias en las librerías matemáticas de punto fijo para lograr mayor eficiencia en la ejecución del algoritmo, procurando no perder resolución numérica apreciable en su resultado. El resultado óptimo será un compromiso entre la resolución numérica del punto fijo y la velocidad de ejecución. Como objetivo complementario del TFM se pretende realizar una comparación entre los resultados obtenidos de eficiencia y resolución de la librería de punto fijo de Matlab y las librerías de punto fijo en C++.

2. Resumen del estado del arte

Este capítulo trata del análisis de imágenes hiperespectrales para la obtención de información sobre el tipo de los materiales presentes en el área de estudio. Para ello se estudiaran los algoritmos de procesamiento de imágenes hiperespectrales en concreto la extracción de *Endmembers* haciendo uso del modelo lineal de mezcla.

Posteriormente se continuará con el estudio de la implementación hardware de los algoritmos de análisis hiperespectrales como puede ser el VCA.

2.1. Imágenes hiperespectral

Una imagen hiperespectral es aquella en que cada píxel se define por un vector de valores espectrales que se corresponde con la radiancia o la reflectividad obtenida para esa longitud de onda. Los sensores hiperespectrales contienen un número superior a las decenas e incluso centenas de bandas. Los canales del espectro electromagnético que obtienen estos sensores típicamente comienzan en el ultra violeta y terminan en el infrarrojo térmico, abarcando un ancho de banda superior a 10 micras.

Usualmente se suele representar los datos hiperespectrales en modo de cubo (hipercubo), compuesto por una pila de imágenes correspondientes a cada canal del espectro. En la siguiente Figura 3 se muestra un esquema del hipercubo.

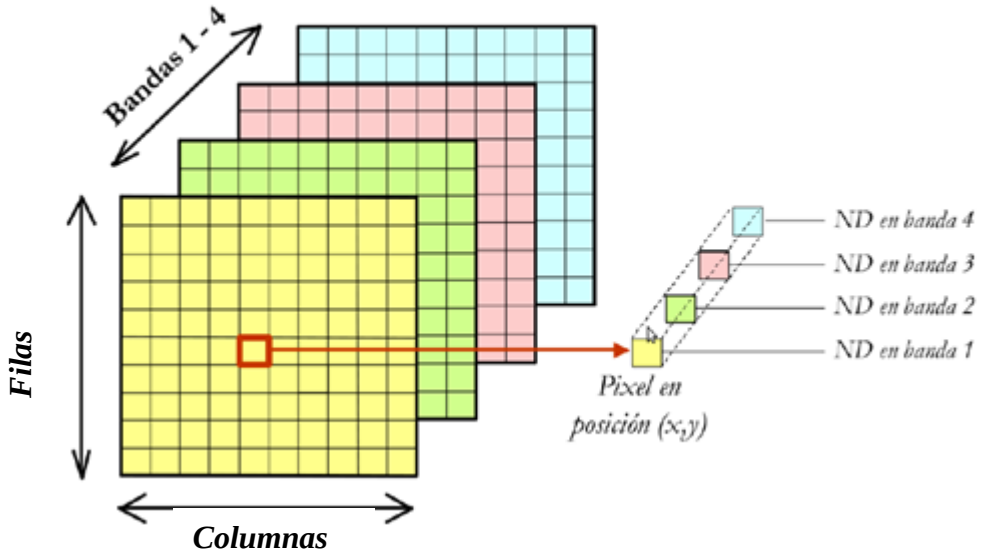


Figura 3. Imagen hiperespectral (hipercubo)

El muestreo de las diferentes longitudes de onda viene determinado por la tecnología del sensor que se emplea. A menor separación entre bandas se suele tener mejores resultados en las aplicaciones gracias a que se puede obtener una firma espectral más fidedigna de los elementos observados.

Uno de los conceptos más importantes de las imágenes hiperespectrales es la existencia de mezclas a nivel de subpíxel, por lo que a grandes rasgos podemos encontrar dos tipos de píxeles: lo píxeles puros y los píxeles mezcla [11]. Se puede definir un píxel mezcla como aquel en el que coexisten diferentes materiales. Este tipo de píxel es el que constituye la mayor parte de la imagen, debido a la insuficiente resolución espacial, que se traduce en píxeles de gran tamaño proclives a mayores mezclas, aunque el fenómeno de mezcla tiene lugar incluso a niveles microscópicos. En la Figura 4 se presenta un ejemplo de una imagen hiperespectral y de los píxeles puros y mezclas obtenidos.

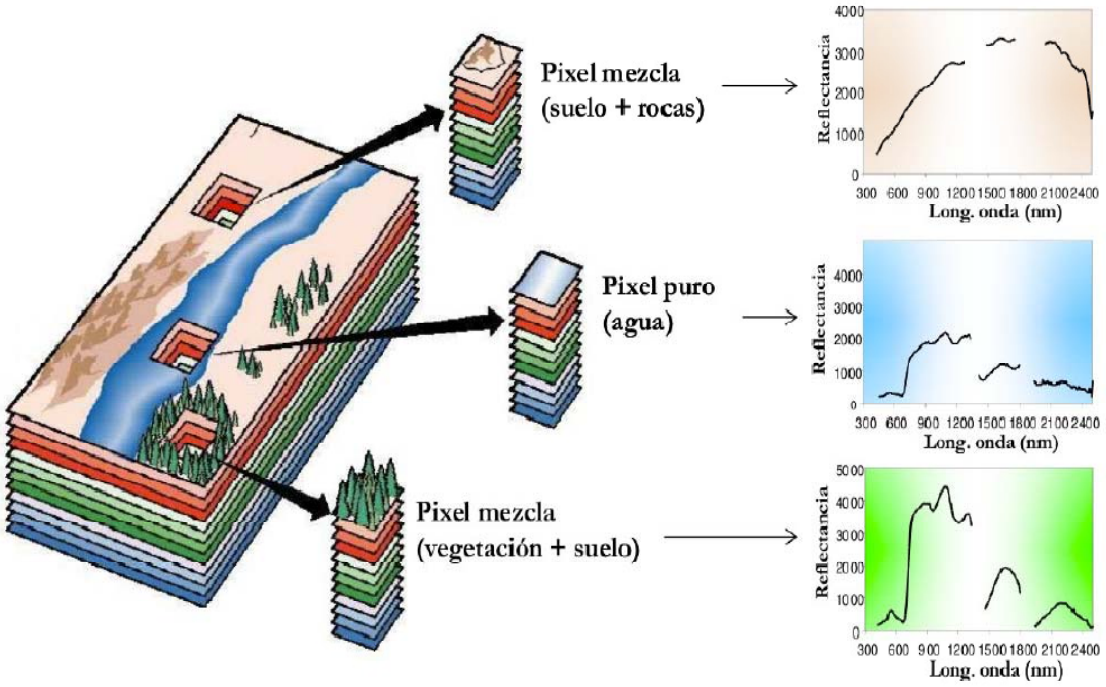


Figura 4. Tipos de píxeles y su firma espectral de una imagen hiperespectral

La obtención de los *Endmembers* y el estudio de la relación existente entre los píxeles puros y las mezclas han supuesto un avance importante en el procesamiento de imágenes hiperespectrales.

2.2. Técnicas de análisis hiperespectrales

La práctica totalidad de las técnicas de análisis hiperespectral desarrolladas hasta la fecha presuponen que la radiación obtenida por el sensor en un determinado píxel viene dado por la contribución de diferentes materiales que residen a nivel sub-píxel. Las técnicas basadas en este modelo son altamente costosas desde el punto de vista computacional. A continuación se detallan las características genéricas de estas técnicas basadas en el modelo lineal de mezcla.

2.2.1. Modelo lineal de la mezcla

El modelo lineal de mezcla es el más utilizado en el análisis de imágenes hiperespectrales, debido a su mayor sencillez. Los píxeles mezcla se representan como una combinación

lineal de firmas asociadas a componentes espectralmente puros. Este modelo ofrece resultados satisfactorios cuando los componentes que residen a nivel subpíxel aparecen espacialmente separados, situación en donde los fenómenos de absorción y reflexión de la radicación incidente pueden ser caracterizados siguiendo un modelo lineal.

El modelo lineal de mezcla se puede interpretar como un espacio bidimensional tal como se presenta en la siguiente Figura 5. En este plano se puede apreciar como todos los puntos de la imagen quedan englobados dentro del triángulo formado por los tres puntos más externos (correspondientes con los elementos puros). Los vectores asociados a dichos puntos constituyen un nuevo sistema de coordenadas con origen en el centroide de la nube de puntos, de forma que cualquier punto de la imagen puede expresarse como combinación lineal de los puntos más externos, siendo estos puntos los candidatos a ser *Endmembers*. Para ello se identifican los elementos extremos de la nube de puntos N-dimensional.

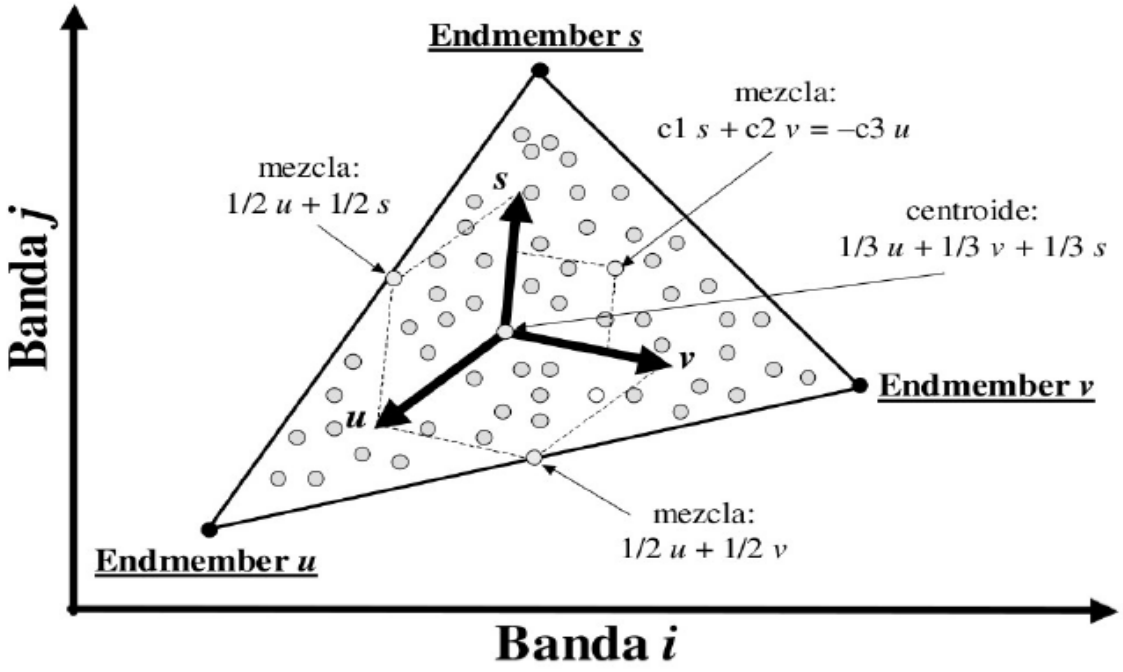


Figura 5. Interpretación del modelo lineal de mezcla

*2.2.2. **Procesado de imágenes hiperespectrales***

El procesado de imágenes hiperespectrales tal y como se ve en la Figura 6 se realiza en varias etapas, primero se realiza una adaptación de los datos obtenidos por el sensor para obtener un formato de datos normalizado como la reflectancia, posteriormente se realiza una etapa de procesado en donde se realiza una reducción de bandas mediante algoritmos como el *Principal Components Analysis* (PCA) que generan combinaciones lineales de intensidades de los píxeles mutuamente incorreladas y de máxima variancia. La salida de esta etapa es utilizada en la extracción de *Endmembers* que obtiene las firmas espectrales de los posibles píxeles puros. Finalmente mediante la segmentación y clasificación de los *Endmembers* se obtienen los píxeles puros de la imagen.

La extracción de *Endmembers* es la parte más importante del proceso ya que su utilización proporciona como resultados los posibles píxeles puros, en este TFM nos vamos a centrar en la implementación de un algoritmo de extracción de Endmembers sin tener en cuenta las demás partes del proceso.

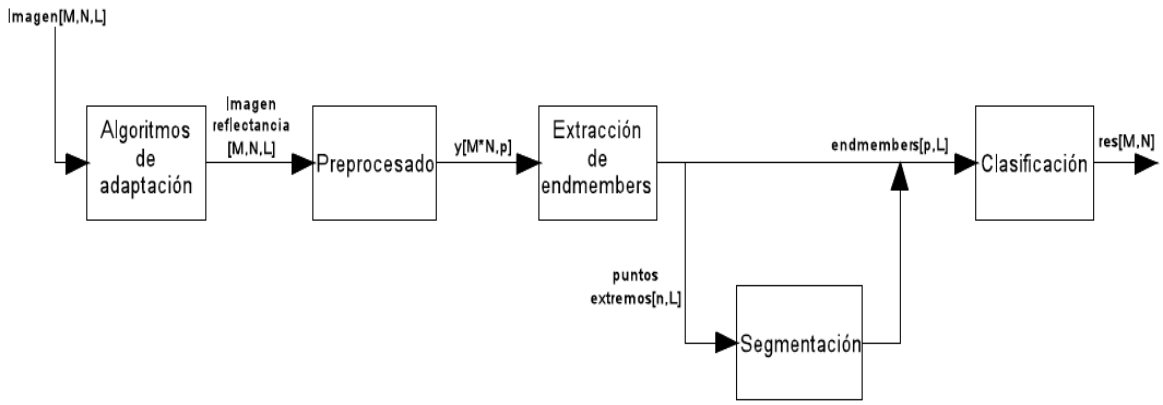


Figura 6. Diagrama de bloques del procesamiento de imágenes hiperespectrales

La extracción de Endmembers se basa en el modelo de composición lineal que se puede representar de la siguiente manera:

$$r = Ms + n$$

- Donde r es una matriz con cada uno de los píxeles de la imagen
- n es la componente de ruido aleatorio que existe en cada píxel
- M es la matriz de *Endmembers* M = [m1, m2, …, m_p] (siendo p el número de Endmembers
- *s* son las abundancias multiplicadas por el factor de escala de iluminación

Por lo tanto es necesario obtener los valores de los Endmembers en el cálculo de este modelo. De esta manera han aparecido numerosos algoritmos matemáticos de los cuales destacan los tres siguientes:

- ✓ *Pixel Purity Index* (PPI) [12]. En este algoritmo se proyecta la imagen un número de veces muy elevado (entre 1000 y 10000), sobre vectores aleatorios que se construyen para cada proyección almacena, al valor mayor de la proyección en valor absoluto. Mediante el cálculo del valor más votado se designan los puntos extremos. Una vez que obtenemos estos *Endmembers* se realiza una clasificación para reducir el número de *Endmembers*.
- ✓ *N-FINDR* [13]. Partiendo de la base de que un símplex es equivalente en n dimensiones a un triángulo. Este algoritmo se basa en buscar un simplex de volumen máximo y un número de vértices p, basándose en la suposición de que en aquel simplex de mayor volumen que encierre a todos los puntos, sus p vértices serán los p *Endmembers* buscados.
- ✓ *Vertex Component Analysis* (VCA) [14]. Se obtiene cada uno de los Endmembers al realizar una proyección completa de la imagen sobre un vector perpendicular al anterior *Endmember* obtenido, realizándose por tanto p proyecciones hasta encontrar p Endmembers.

2.2.3. ***Vertex Component Analysis***

VCA trata de descomponer espectralmente y de manera lineal los vectores mezcla de la imagen en sus correspondientes componentes puras. Se trata de un algoritmo basado en dos aspectos, por un lado los *Endmembers* son los vértices de un simplex, y por otro lado la

transformación afín de un simplex es también otro simplex. El algoritmo funciona tanto con imágenes proyectadas a otro espacio y reduciendo la dimensión espectral de la imagen, como con las imágenes originales que no sufren un preprocesado previo. El método consiste en que de manera iterativa se proyectan los valores de la imagen sobre una dirección perpendicular al subespacio determinado por los *Endmembers* ya calculados. El nuevo *Endmember* se determina como el valor extremo de esta proyección. Se producen tantas iteraciones como *Endmembers* se deseen calcular, es decir, *p* iteraciones. Este hecho supone que computacionalmente es mucho menos costoso que los dos algoritmos precedentes. En la siguiente Figura 7 se muestra gráficamente el funcionamiento del algoritmo VCA.

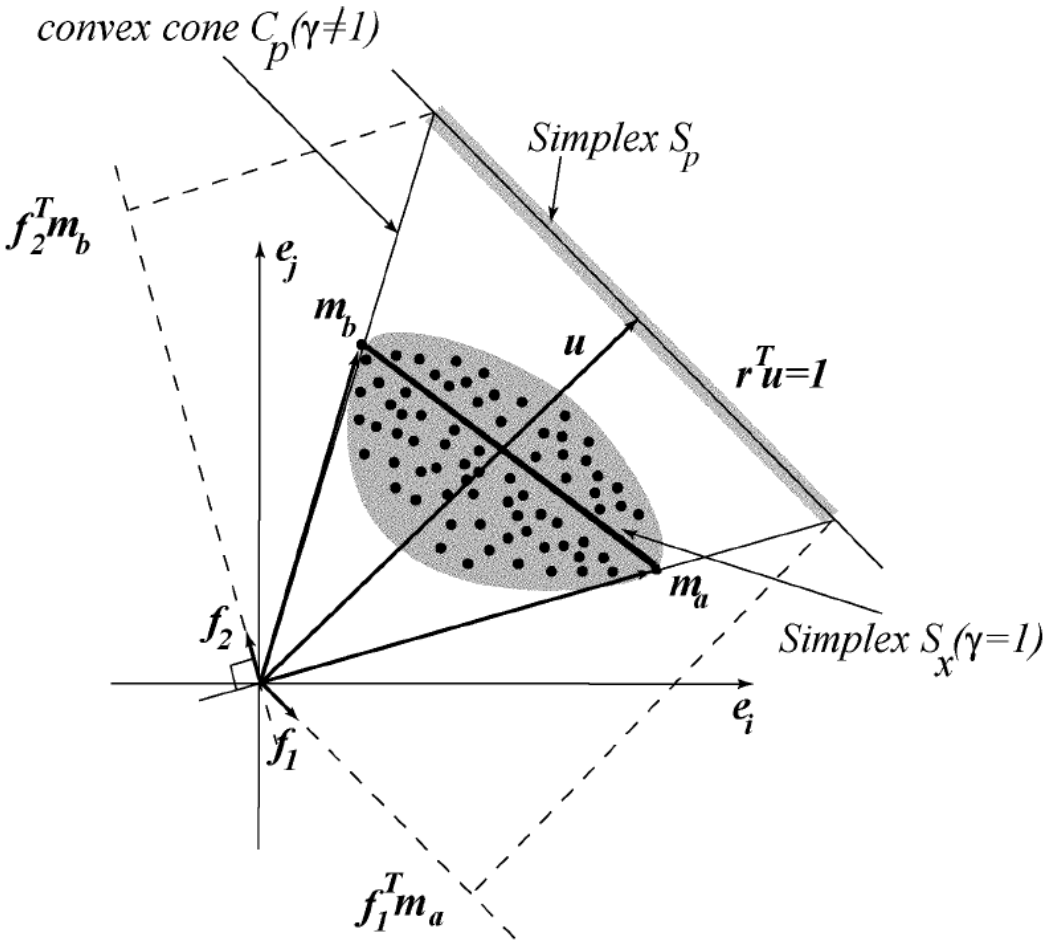


Figura 7. Representación gráfica del algoritmo VCA

La nube de puntos que conforma la imagen se encuentra contenida en un cono convexo C_p a su vez perteneciente al subespacio E_p de dimensión p. El algoritmo VCA se encarga de determinar este subespacio E_p mediante SVD (Singular Value Decomposition) y proyectar los puntos del cono sobre este subespacio, $y=r/(r^Tu)$, formándose un simplex S_p. Este simplex se encuentra contenido en un espacio de dimensiones p-1. En el caso de que la proyección se realizara sobre un subespacio E_d de mayor dimensión $p \leq d \leq L$, la proyección de C_p sobre E_d, seguido de una proyección adecuada es también un simplex con los mismos vértices.

Si la relación señal a ruido disminuye, el factor de escala que se aplicaba en el caso anterior, $r/(r^Tu)$, aumentaría el ruido presente en la imagen, que en este caso es superior. Por este motivo, es preferible no usar este factor de escala, y determinar el nuevo espacio sobre el que se quiere proyectar la imagen, mediante el método PCA. En este caso, el nuevo espacio será de dimensión 'p-1'.

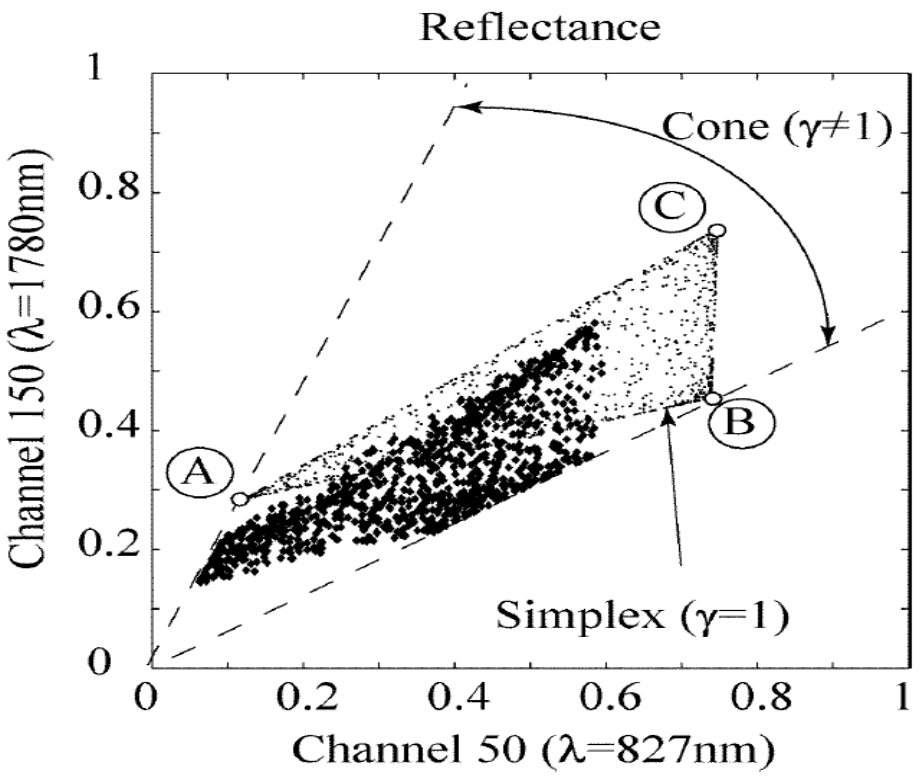


Figura 8. Scatterplot en dos dimensiones de píxeles mezcla formados por 3 Endmembers

Algorithm 1: Vertex Component Analysis (VCA)

INPUT p, $\mathbf{R} \equiv [\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_N]$

1: $\text{SNR}_{\text{th}} = 15 + 10\log_{10}(p)$ dB

2: **if** SNR $>$ SNR_{th} **then**

3: $d := p$;

4: $\mathbf{X} := \mathbf{U}_{\mathbf{d}}^{\mathbf{T}}\mathbf{R}$; {$\mathbf{U_d}$ obtained by SVD}

5: $\mathbf{u} := \text{mean}(\mathbf{X})$; {$\mathbf{u}$ is a $1 \times d$ vector}

6: $[\mathbf{Y}]_{:,j} := [\mathbf{X}]_{:,j}/([\mathbf{X}]_{:,j}^{T}\mathbf{u})$; {projective projection}

7: **else**

8: $d := p - 1$;

9: $[\mathbf{X}]_{:,j} := \mathbf{U}_{\mathbf{d}}^{\mathbf{T}}([\mathbf{R}]_{:,j} - \overline{\mathbf{r}})$; {$\mathbf{U_d}$ obtained by PCA}

10: $c := \arg\max_{j=1\ldots N} \|[\mathbf{X}]_{:,j}\|$;

11: $\mathbf{c} := [c|c|\ldots|c]$; {$\mathbf{c}$ is a $1 \times N$ vector}

12: $\mathbf{Y} := \begin{bmatrix}\mathbf{X}\\ \mathbf{c}\end{bmatrix}$

13: **end if**

14: $\mathbf{A} := [\mathbf{e}_u|\mathbf{0}|\ldots|\mathbf{0}]$; {$\mathbf{e}_u = [0, \ldots, 0, 1]^T$ and $\mathbf{A}$ is a $p \times p$ auxiliary matrix}

15: **for** $i := 1$ to p**do**

16: $\mathbf{w} := \text{randn}(0, \mathbf{I}_p)$; {$\mathbf{w}$ is a zero-mean random Gaussian

17: $\mathbf{f} := ((\mathbf{I} - \mathbf{A}\mathbf{A}^{\#})\mathbf{w})/(\|(\mathbf{I} - \mathbf{A}\mathbf{A}^{\#})\mathbf{w}\|)$; {$\mathbf{f}$ is a vector orthonormal to the subspace spanned by $[\mathbf{A}]_{:,1:i}$.}

18: $\mathbf{v} := \mathbf{f}^T\mathbf{Y}$;

19: $k := \arg\max_{j=1,\ldots,N} |[\mathbf{v}]_{:,j}|$; {find the projection extreme.}

20: $[\mathbf{A}]_{:,i} := [\mathbf{Y}]_{:,k}$;

21: $[indice]_i := k$; {stores the pixel index.}

22: **end for**

23: **if** SNR $>$ SNR_{th} **then**

24: $\widehat{\mathbf{M}} := \mathbf{U_d}[\mathbf{X}]_{:,indice}$; {$\widehat{\mathbf{M}}$ is a $L \times p$ estimated mixing matrix}

25: **else**

26: $\widehat{\mathbf{M}} := \mathbf{U_d}[\mathbf{X}]_{:,indice} + \overline{\mathbf{r}}$; {$\widehat{\mathbf{M}}$ is a $L \times p$ estimated mixing matrix}

27: **end if**

Figura 9. Pseudo-código del algoritmo VCA.

Las líneas de la 1 a la 12 realizan una proyección de la matriz de entrada a un espacio calculado o bien empleando la transformación SVD a un espacio de *p* dimensiones o mediante la transformación PCA a un espacio de *p-1* dimensiones. Esto se hace en función de que la relación señal a ruido (SNR) de la misma sea o no superior a un umbral, que se calcula en la primera instrucción del algoritmo. Esta primera fase que se ha descrito constituye la fase de preprocesado de la imagen. Destacar que los pasos 4 y 9 aseguran que ninguno de los productos escalares entre $[X]_{:,i}$ y u sean negativos, aspectos crucial para el correcto funcionamiento del algoritmo VCA.

A partir de la línea 14, comienza el algoritmo VCA propiamente dicho con la inicialización de la matriz A, matriz sobre la que se irán almacenando las proyecciones de los *Endmembers* que se vayan encontrando, de la forma en que se muestra, todos sus columnas a cero, exceptuando la primera que está ocupada por el vector $e_u = [0,0,\ldots,1]'$. Esta inicialización de la matriz A produce la reducción de una de las dimensiones, obteniéndose todos los puntos proyectados en un espacio de 'p-1' dimensiones. Este hecho tiene gran importancia, ya que se recuerda que el simplex se obtiene proyectando los datos sobre un espacio de estas características.

Una vez inicializada la matriz, se entra en el bucle, línea 15 que calcula en cada iteración un vector 'f', ortonormal a la matriz A, mediante la creación de un vector aleatorio, que es de esta naturaleza para evitar realizar una proyección sobre las columnas de la matriz A de un vector nulo. Una vez calculado 'f', en la línea 18 se realiza la proyección de la matriz 'y' sobre este vector, y se almacena en 'v', de tal forma que en la siguiente línea se calcula el índice del valor mayor absoluto del vector, y se almacena en k.

En la línea 20, con este índice calculado, se selecciona de la matriz 'Y' el vector, que supondrá la proyección del nuevo *Endmember*, y posteriormente se guarda el índice k en un vector de 'p' componentes. Las líneas 23 a la 27 se encargan de reconstruir la imagen al espacio original de 'L' bandas.

Para obtener información sobre la bondad de los resultados respecto a precisión los autores del algoritmo emplean tres ángulos que se calculan a partir del método *spectral angle*. Se destaca que estos ángulos se pueden determinar ya que se trata de imágenes artificiales donde se conoce la firma espectral pura que se debe obtener y por tanto se puede comparar con la que se obtiene con el algoritmo. Las expresiones de los ángulos son los siguientes:

$$\theta_i \equiv (arccos \frac{\langle m_i, \hat{m}_i \rangle}{\|m_i\| \|\hat{m}_i\|})$$

Esta ecuación determina el ángulo espectral que existe entre el *Endmember* estimado $\hat{m}_i$ y el *Endmember* real m_i.

$$\beta_i \equiv (arccos \frac{\langle [S]_{i,:}, \widehat{[S]}_{i,:} \rangle}{\|[S]_{i,:}\| \|\widehat{[S]}_{i,:}\|})$$

En esta ecuación se determinan las diferencias entre las abundancias, que se generan de forma aleatoria a partir de una función de dirichlet, de cada píxel estimadas y las reales, siendo la fórmula para el cálculo de las abundancias $\hat{S} = \widehat{M^{\#}}[r_1, r_2, \dots, r_n]$, que representa la pseudoinversa de la matriz de Endmembers multiplicado por cada uno de los píxeles de la imagen. La última medida que se emplea es el *Spectral Information Divergence* (SID), que compara la similitud de las dos firmas espectrales.

$$SID_{m_i, \hat{m}_i} \equiv D(m_i | \hat{m}_i) + D(\hat{m}_i | m_i)$$

$$D(m_i | \hat{m}_i) \equiv \sum_{j=1}^{L} p_j \, log\left(\frac{p_j}{q_j}\right)$$

Siendo $p_j = \frac{m_{ij}}{\sum_{k=1}^{L} m_{ik}}$ y $q_j = \frac{\hat{m}_{ij}}{\sum_{k=1}^{L} \hat{m}_{ik}}$.

Estas medidas producen p valores (tantos como *Endmembers* se calculen) y por lo tanta para la representación gráfica se calcula el rms de los vectores de valores.

*2.2.4. **Vertex Component Analysis Modificado***

El campo del análisis hiperespectral y sus aplicaciones está en pleno crecimiento lo que ha hecho aumentar en gran medida el interés del mundo científico e investigador. Un ejemplo de este interés lo encontramos en el IUMA que trabaja en este campo. Fruto de este trabajo investigador realizado en este Máster en Tecnologías de Telecomunicación por el alumno Pablo Hostrand Andaluz se ha creado una modificación del algoritmo VCA que permite utilizar una aritmética más sencilla y eficiente en dispositivos electrónicos [15].

A la hora de implementar el algoritmo en hardware, se han de explorar las diferentes posibilidades de modificaciones que permitan minimizar el coste computacional sin producir errores importantes en su ejecución. La parte del algoritmo en el que se realiza el

preprocesado no se modifica, puesto que no va a ser sintetizado en hardware. Por lo tanto las modificaciones se realizan desde la línea 14 hacia delante.

En la línea 16 se establece que el vector 'w', que es utilizado para obtener el vector perpendicular al subespacio generado por 'A', es un vector aleatorio Gausiano de media cero. Este vector puede ser fijado a un valor siempre que sus componentes sean distintas de 0. En la línea 17 se realiza el cálculo de mayor peso computacional del algoritmo, se trata del cálculo de la pseudoinversa, cuyo resultado es además normalizado para obtener un vector ortonormal. Este hecho es superfluo dado que el algoritmo selecciona el índice de la proyección cuyo valor absoluto es mayor por lo que no se necesita la normalización.

En la figura 10 se muestra el algoritmo modificado.

Algorithm: Vertex Component Analysis (VCA) Modifications

INPUT: p, Y=[y_1,y_2,...,y_N] {Y is the projected image calculated in the preprocess}

1. A := [e_u|0| ... |0] ;{e_u := [0, ..., 0,1]^T and A is a px(p+1) matrix}
2. U_b := [0| ... |0] ;{U_b is a pxp matrix}
3. w := [1,...,1] ;{w is a px1 vector}
4. f_{acc} := [0,...,0];
5. [r,s] := arg min $_{j=1,...,p;\ i=1,...,N}$ |[Y]$_{j,i}$| {minimum element Y}
6. **for** i:= 1 **to** p **do**
7. [U_b]$_{:,i}$:= [A]$_{:,i}$;
8. **for** j:= 3 **to** i **do**
9. $c_1(i,j-1) := \dfrac{[A]_{:,i}^T [U_b]_{:,j-1}}{[U_b]_{:,j-1}^T [U_b]_{:,j-1}}$;
10. [U_b]$_{:,i}$:= [U_b]$_{:,i}$ - c_1*[U_b]$_{:,j-1}$;
11. **end for**
12. $c_2(i) := \dfrac{w^T [U_b]_{:,i}}{[U_b]_{:,i}^T [U_b]_{:,i}}$;
13. f_{acc} := f_{acc} + c2* [U_b]$_{:,i}$;
14. f := w - f_{acc}; {f is a ortogonal vector to the subspace spanned by [A]$_{:,1:i}$}
15. **if** i == 1 **then**
16. f_{acc} := [0,...,0]; {Reinitialize the accumulator}
17. **else**
18. m := arg min$_{j = 1, ..., p}$ |[f]$_j$| {minimum element of f}

```
19.     end if
20. v := round(f^T)*round(Y);
21. k := arg max_{j = 1, ..., N} |[v]:,j| {find the projection extreme}
22. [A]:,i+1 := [Y]:,k;
23. [index]_i = k;
24. end for
```

Figura 10. Algoritmo VCA modificado.

En donde el parámetro 'p' es el número de *Endmembers* a calcular, y la matriz 'Y', es la imagen proyectada sobre el espacio calculado E_p, en el preprocesado. Cada uno de los vectores que se presentan en la imagen 'Y', y_1, y_2, …, y_N, es de dimensión 'p', donde 'N' es el número de píxeles de la imagen.

En las primeras cuatro líneas se inicializan matrices y vectores. En este caso la matriz 'A' presenta 'p+1' columnas, ya que el vector de inicialización, 'e_u' se mantiene en la matriz y por tanto se necesita una columna más. La matriz 'u_b' se emplea para almacenar los vectores de la base ortogonal. El vector w queda inicializado a un valor estático igual a 1.

De la línea 7 a la 11 se calcula la ortogonalización de 'f' mediante el método de *Gram-Schmidt* por lo que no es necesario el cálculo de la pseudoinversa, mejorando de una manera importante el número de operaciones matemáticas a realizar en el algoritmo.

En la línea 7 del algoritmo se asigna el nuevo *Endmember* calculado al vector columna de la base 'u_b' de la iteración correspondiente, y posteriormente se entra en el bucle, si se trata de la 3ª o posterior iteración. La primera iteración del algoritmo que calcula el primer *Endmember* se realiza de forma diferente al resto, siendo a partir de la segunda iteración cuando se comienza a generar la base ortogonal. El cálculo del nuevo vector de la base se lleva a cabo en las líneas 9-10. Una vez calculado se procede al cálculo de la constante c_2 que permite calcular el f_{acc} en la línea 13.

En las líneas siguientes el código no varía sustancialmente respecto al algoritmo original.

2.3. Implementación hardware

Las técnicas de análisis hiperespectral se basan en la realización de operaciones matriciales que resultan muy costosas desde el punto de vista computacional. En cambio, el carácter

repetitivo de estas operaciones las hace altamente susceptibles de ser implementadas en arquitecturas paralelas, mejorando de una forma importante su rendimiento computacional. Las técnicas de computación paralela se han utilizado ampliamente en el tratamiento de grandes imágenes, obteniendo tiempos de respuesta muy reducidos. En la actualidad existen diferentes plataformas que permiten realizar el procesado paralelo necesario de estos algoritmos, como las GPUs, o hardware dedicado como las FPGAs y los circuitos integrados de aplicación específica (ASIC). La implementación hardware es la que mejores resultados proporciona, pero es importante destacar que la complejidad de la tarea de implementación aumenta considerablemente por lo que se hace necesario el estudio de alternativas software que permitan una implementación hardware a partir de algoritmos escritos en lenguajes de alto nivel como Matlab.

2.3.1. ***Uso de FPGAs en la implementación de algoritmos hiperespectral***

Las FPGAs (*Field Programmable Gate Array*) son dispositivos programables ideales para la realización de prototipos, en el que los cambios en la implementación son elevados permitiendo realizar múltiples pruebas una vez que se ha programado la placa. Las FPGAs son unos dispositivos que proporcionan un gran equilibrio entre flexibilidad y eficiencia, por lo que se ha elegido este tipo de dispositivo para la implementación del algoritmo VCA.

Las FPGAs están compuestas por una matriz bidimensional de bloques configurables que se pueden conectar mediante recursos generales de interconexión. Estos recursos incluyen segmentos de pista de diferentes longitudes, más unos conmutadores programables para enlazar bloques a pistas o pistas entre sí. Por lo tanto lo que se programa en una FPGA son los conmutadores que sirven para realizar las conexiones entre los diferentes bloques, más la configuración de los propios bloques. En la Figura 11 se muestra una imagen con la estructura básica de una FPGA.

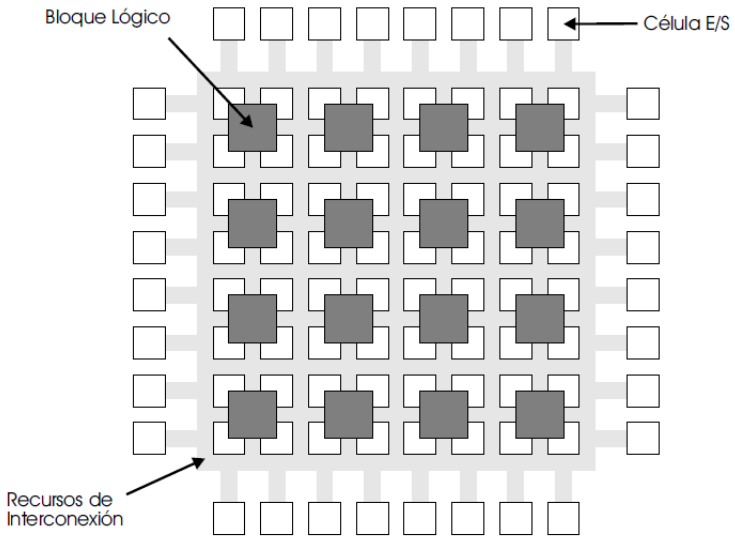


Figura 11. Estructura básica de una FPGA

Los elementos básicos constituyentes de una FPGA, son los siguientes

- Bloques lógicos, cuya estructura y contenido se denomina arquitectura
- Recursos de interconexión, cuya estructura se denomina arquitectura de enrutado
- Memoria RAM, que es cargado en el RESET para configurar bloques y contenidos

Por lo tanto las ventajas más importantes que proporciona el uso de FPGAs son el bajo coste de prototipado y el corto tiempo de producción. Los inconvenientes residen en la baja velocidad de operación y la baja densidad lógica frente a los dispositivos ASIC.

Las FPGAs que existen en el mercado se pueden clasificar en cuatro grandes familias dependiendo de la estructura que adoptan los bloques lógicos que tienen definidas.

1. Matriz simétrica, como son las de XILINX
2. Basada en canales, como son las de ACMEL.
3. PLD jerárquica, como son las de ALTERA o las CPLD's de XILINX
4. Mar de Puertas, como es el caso de ORCA.

*2.3.2. **Sintetizado hardware desde lenguajes de alto nivel***

Los lenguajes de descripción hardware utilizados para la síntesis de las FPGAs tienen una mayor dificultad que los lenguajes de alto nivel como Matlab o C/C++, por lo que desde

hace unos pocos años han aparecido herramientas que permiten la síntesis hardware a más alto nivel como el System-C y otras herramientas que permiten convertir código C/C++ para dispositivos empotrados en código de descripción hardware como el VHDL o Verilog. Gracias a estas nuevas herramientas se logra realizar la síntesis de algoritmos con menor esfuerzo pagando como contrapartida una menor eficiencia en los resultados obtenidos.

La implementación de algoritmos matemáticos suelen realizarse en lenguajes de muy alto nivel, en donde destaca la utilización del Matlab por gran parte de estos investigadores. Matlab permite una programación rápida de prototipos que están muy alejados de implementaciones hardware. Por este motivo Matlab ha trabajado en la realización de una *Toolbox* que permite la generación de código C/C++ para sistemas empotrados. Este hecho abre la puerta a metodologías de sintetizado rápido de códigos en Matlab ya que este código puede ser pasado a C/C++ y posteriormente sintetizado para FPGAs mediante la utilización de herramientas como el Catapult-C.

3. Herramientas utilizadas

Para la síntesis hardware del algoritmo VCA programado en Matlab se ha hecho uso de múltiples herramientas software y de programación que se van a describir en este capítulo. Entre estas herramientas destacan la IDE de programación matemática Matlab con sus *Toolboxes*, la librería para punto fijo de C++ *fixed*, la IDE de programación Code::Block y la herramienta para la síntesis hardware a partir de código C/C++ Catapult-C.

3.1. Matlab

Matlab constituye actualmente un estándar de facto dentro de las herramientas del análisis numérico, tanto por su gran capacidad y sencillez de manejo como por su enorme versatilidad y difusión.

Matlab (*Matrix Laboratory*) es un lenguaje de programación técnico-científico que básicamente trabaja con variables vectoriales y matriciales. Es fácil de utilizar debido a que contiene varias cajas de herramientas con funciones incorporadas (*Toolbox* de procesamiento de señales, herramientas de C embebido, matemática aritmética de punto

fijo, etc.). Dos *Toolbox*es esenciales en el desarrollo de este TFM son el *Embedded* y el *Fixed-Point*, estas herramientas permiten la generación de código C/C++ para sistemas empotrados a partir de código Matlab compatible y la utilización de aritmética de punto fijo en los algoritmos. A continuación se va a proceder a describir estas *Toolboxes* con más detalle.

3.1.1. ***Toolbox Embedded Matlab***

La *Toolbox* de *Embedded* de Matlab permite generar código C embebido para dispositivos empotrados a partir de un subconjunto de funciones compatibles de Matlab [16]. El código generado en C cumple con los requisitos de tiempo real para la memoria y de variables locales.

Para hacer uso del *Embedded* C de Matlab es necesario instalar y configurar un compilador de C++, dependiendo del sistema operativo en que se ejecuta. Para el caso de Windows 64 bits es necesario instalar la versión Visual Studio de Microsoft.

En este TFM se ha utilizado el Visual Studio 2008. Para poder integrar el compilador en Matlab ha sido necesario instalar el Microsoft Visual Studio 2008 Service Pack 1 y posteriormente el Microsoft Windows SDK for Windows 7 y el .NET Framework 3.5 SP1. Una vez instalado el software se ha procedido a la configuración desde Matlab para su uso.

Para configurar el compilador que va a ser utilizado en el *Embedded* de Matlab es necesario configurar las utilidades de compilación de C en Matlab, esto se realiza con el comando *mex*. Tecleando *mex –setup* podemos configurar el compilador que se va a utilizar. Matlab detecta automáticamente los compiladores compatibles instalados en el sistema operativo y te permite seleccionar cual va a ser utilizado.

En la Figura 12 se muestra el prompt de Matlab dónde se puede ver el proceso de configuración del compilador de Visual Studio con el comando *mex*.

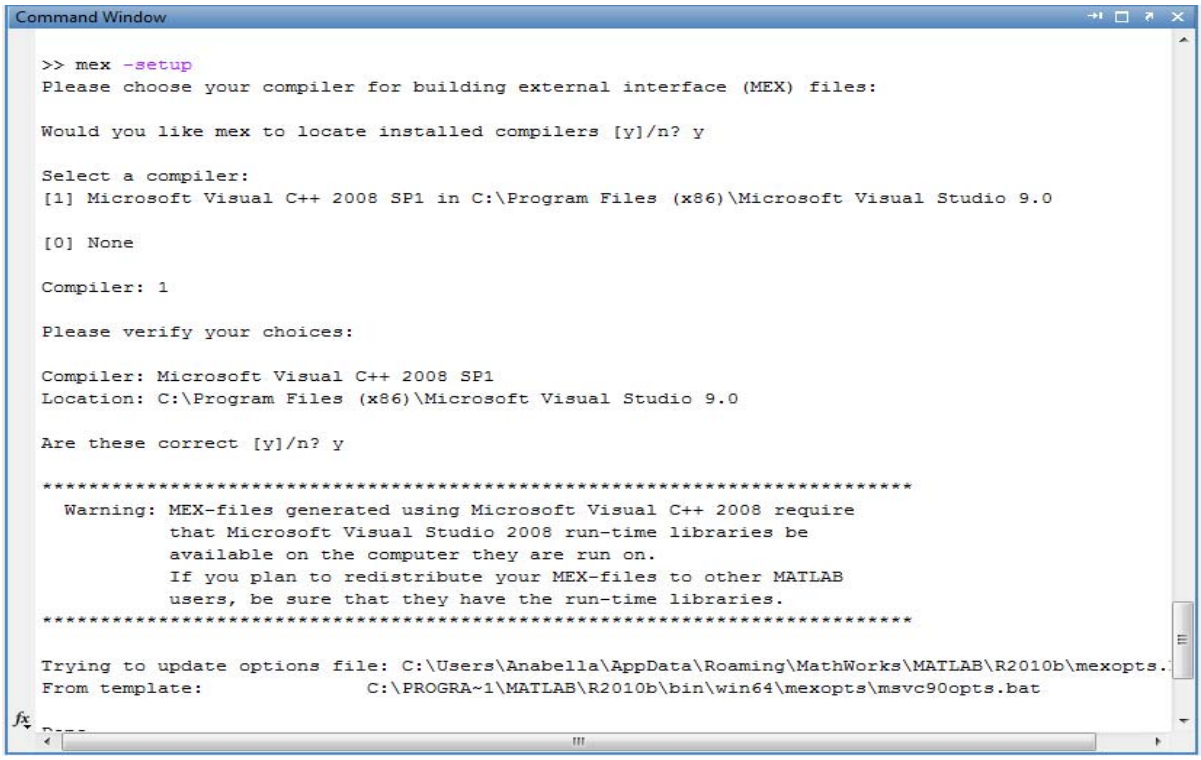


Figura 12. Configuración del compilador de C/C++ utilizado en Matlab mediante el comando mex

Una vez configurado el compilador, se puede generar código C mediante el *Embedded* después de realizar unos pocos pasos de configuración. Primero hemos de crear un objeto de configuración de trabajo en tiempo real *rtwcfg* que se realiza de la siguiente forma:

```
>> rtwcfg = emlcoder.RTWConfig
>> rtwcfg.CustomSource = 'main.c'
```

La primera línea crea el objeto de configuración, mientras que la segunda inserta la ruta de la función principal de C. De esta forma se pueden configurar múltiples parámetros del *Embedded*.

Un ejemplo de ejecución para la obtención de código C a partir de código Matlab se presenta a continuación.

```
>> emlc -s rtwcfg –d test_proj test.m -report -eg {u}
```

El comando *emlc* es el encargado de generar el código C, utilizando la configuración contenida en el objeto *rtwcfg,* el cual generará una carpeta con el nombre test_proj, dónde se almacenan los archivos fuentes. El código C es generado a partir del archivo de código

Matlab llamado test.m. Para finalizar se le introduce el tipo de parámetros de entrada que va a tener la función mediante el *–eg{u}*.

*3.1.2. **Toolbox Fixed-Point***

Un número en punto fijo representa un valor de tipo real mediante un número entero en el que se ha fijado un número de dígitos antes del punto decimal y otro número de dígitos después de este punto. El uso del punto fijo en lugar del más complejo pero eficiente coma flotante es útil para la representación de valores reales cuando el hardware que procesa los datos carece de una unidad de procesado de números en coma flotante (FPU). El punto fijo proporciona mejor rendimiento que la coma flotante en estos dispositivos pequeños de bajo rendimiento.

Un ejemplo que ilustra el funcionamiento del punto fijo es la utilización de enteros decimales para representar números reales de dos decimales. Por lo tanto podemos definir la variable A y B.

Real A = 2,5 → Entero 250 (escala 100)

Real B = 10,00 → Entero 1000 (escala 100)

Para realizar sumas y restas se realiza de la siguiente forma.

Real C = 2,5 + 10 = 12,5 → Entero 250 + 1000 = 1250 (escala 100)

Real C = 10 – 2,5 = 7,5 → Entero 1000 – 250 = 750 (escala 100)

Por lo tanto la suma y las restas en punto fijo son idénticas a su equivalente en entero. Para operar multiplicaciones y divisiones se realiza de la siguiente forma.

Real C = 2,5 * 10 = 250 → Entero 250 * 1000 = 250000/100 = 25000 (escala 100)

Real C = 2,5 / 10 = 0,25 → Entero (100*250/1000) = 25 (escala 100)

Por lo que podemos observar como para el producto se ha de dividir el resultado por la escala y en la división se ha de multiplicar previamente por este valor. Estas operaciones han de ser realizadas en el punto fijo para mantener el número de dígitos decimales almacenados.

Para el caso binario el sistema es idéntico, el único cambio reside en la aritmética binaria. Para una variable de 64 bits se suele dejar un valor de escala suficiente que permita almacenar datos enteros con suficiente resolución. Por ejemplo, si se utiliza un 64x16, indica que los últimos 16 bits están dedicados para la parte decimal por lo tanto podemos

tener una resolución máxima de 2^{-16} = 0,0000152588. Por el lado contrario se puede almacenar en la parte entera 48 bits, o lo que es lo mismo a 2^{48-1} = 140.737.488.355.328, además de un bits de signo. De esta manera se describen los dos problemas de la aritmética de punto fijo, el primero que hay una resolución máxima para la parte decimal, por debajo de ese valor no se puede medir. El segundo problema es el desbordamiento, que indica que si el número es superior al máximo almacenamiento se producirá un error en el uso del punto fijo.

Para definir la escala del punto fijo se utilizan números potencia de dos, dado que se está ejecutando en un dispositivo electrónico el que multiplicar y dividir en potencias de dos se traduce en desplazamientos binarios que son operaciones simples para los dispositivos electrónicos.

Las *Toolbox* de Matlab *Fixed-Point* proporciona el uso de la aritmética de punto fijo para una multitud de funciones compatibles. El punto fijo de Matlab se implementa en una clase que puede almacenar números en punto fijo de tamaños casi ilimitados. Además el punto fijo de Matlab gestiona las operaciones entre las diferentes variables que pueden tener tamaños y escalas diferentes. *Fixed-Point* proporciona herramientas de configuración del comportamiento de los *Fixed-Point* y de la aritmética asociada. Finalmente el uso de *Fixed-Point* está soportado en parte por la *Toolbox* de *Embedded* Matlab, por lo que se puede hacer uso del punto fijo en la generación de código C generado desde Matlab. A continuación se muestran varias formas de inicialización de las variables de punto fijo:

```
A1 = fi(3.1416)
A2 = fi(pi, 1, 8) % word length 8 bits, and fraction length best precision
```

3.2. Librería C++ de punto fijo

En este TFM se ha explorado la posibilidad de hacer uso de librerías de punto fijo para el lenguaje C/C++ que nos permita eliminar el uso de la coma flotante, que es complicada de sintetizar en los dispositivos electrónicos programables FPGAs. Realizando una búsqueda por las librerías existentes en Internet nos encontramos con un amplio conjunto de ellas, de las que destaca la librería en C/C++ Libfixmath y la librería de C++ *fixed* [17].

Estas librerías utilizan el potencial de C++ para proporcionar una envoltura a la aritmética de punto fijo gracias a que C++ permite programar y sobrecargar los diferentes operadores matemáticos y lógicos como la suma, la resta, el producto, la división, la comparación, etc.

De esta forma se puede utilizar la aritmética de punto fijo de la misma forma que se utiliza la aritmética entera o flotante. Las librerías de punto fijo proporcionan operaciones de mayor complejidad como la raíz cuadrada, la potencia u operaciones trigonométricas lo que permite realizar la gran mayoría de los algoritmos matemáticos.

La limitación que introduce el punto fijo respecto al *overflow* y la falta de resolución también está presente en las implementaciones realizadas en C++, más aun si se tiene en cuenta que se utilizan enteros de 32 bits para almacenar los datos. Este tamaño no es adecuado para los cálculos que se pretenden realizar en nuestro algoritmo por lo que se ha reescrito la librería de C++ para que utilice enteros largos de 64 bits, con esto se pueden obtener mejores resultados en la implementación del algoritmo.

Se ha elegido la librería *fixed* para su modificación dado que su código es el más sencillo en cuanto a la programación y a que ha sido programado íntegramente en C++. La librería consta únicamente de dos archivos fuentes (.cpp y .h) que define el total funcionamiento del tipo de dato numérico en punto fijo. A continuación se muestran los prototipos de métodos que proporciona la librería para su uso.

```
fixed(const fixed& fixedVal);
fixed(const fixed* fixedVal);
fixed(long long int nVal);
fixed operator++(void);
fixed operator++(int);
fixed operator--(void);
fixed& operator=(fixed fixedVal);
fixed& operator=(int intVal);
bool operator==(fixed fixedVal);
bool operator==(int intVal);
bool operator!=(fixed fixedVal);
bool operator!=(int intVal);
bool operator<(fixed fixedVal);
bool operator<(int intVal);
bool operator<=(fixed fixedVal);
bool operator<=(int intVal);
```

```
bool operator>(fixed fixedVal);
bool operator>(int intVal);
bool operator>=(fixed fixedVal);
bool operator>=(int intVal);
operator double(void);
operator float(void);
operator int(void);
fixed floor(void);
fixed ceil(void);
fixed operator+(int b);
fixed operator-(int b);
fixed operator*(int b);
fixed operator/(int b);
fixed operator+(fixed b);
fixed operator-(fixed b);
fixed operator*(fixed b);
```

Gracias a la implementación de los operadores anteriores la sintaxis de la aritmética en punto fijo para esta librería en C++ queda de la siguiente manera.

```
fixed a = 3.1416;
fixed b = (a+4.5)/2.14;
if(a <= b) b = sin(a);
float c = (float) b;
…
```

3.3. IDE de programación C++ Code::Block

Code::Block es un entorno de programación integrado libre y multiplataforma para el desarrollo de aplicaciones realizados en C++. Permite la utilización de diversas librerías gráficas multiplataforma para la implementación de programas con interfaz gráfica en diferentes sistemas operativos.

Code::Block, al igual que las demás IDEs, soporta varias utilidades de apoyo a la programación como el coloreo de sintaxis, auto completado de los métodos, tabulación inteligente de código, etc. Además, presenta un navegador de proyectos: vista de archivos, símbolos (heredados, etc.), clases y recursos. También dispone de menús de configuración de los proyectos C++, su compilación y ejecución.

Una de las características más importantes de Code::Block son los complementos (*plugins*) que permite que la IDE sea muy dinámica y potente. Code::Block permite enlazarse a una gran variedad de compiladores, desde compiladores como el Microsoft Visual Studio Toolkit, pasando por el compilador de Borland C++ Compiler, Intel C++ Compiler y terminando por el compilador GNU (GCC, G++). Este último en sus versiones para Linux/Unix y Windows (MinGW). En la siguiente Figura 13 se muestra la IDE de programación Code::Block.

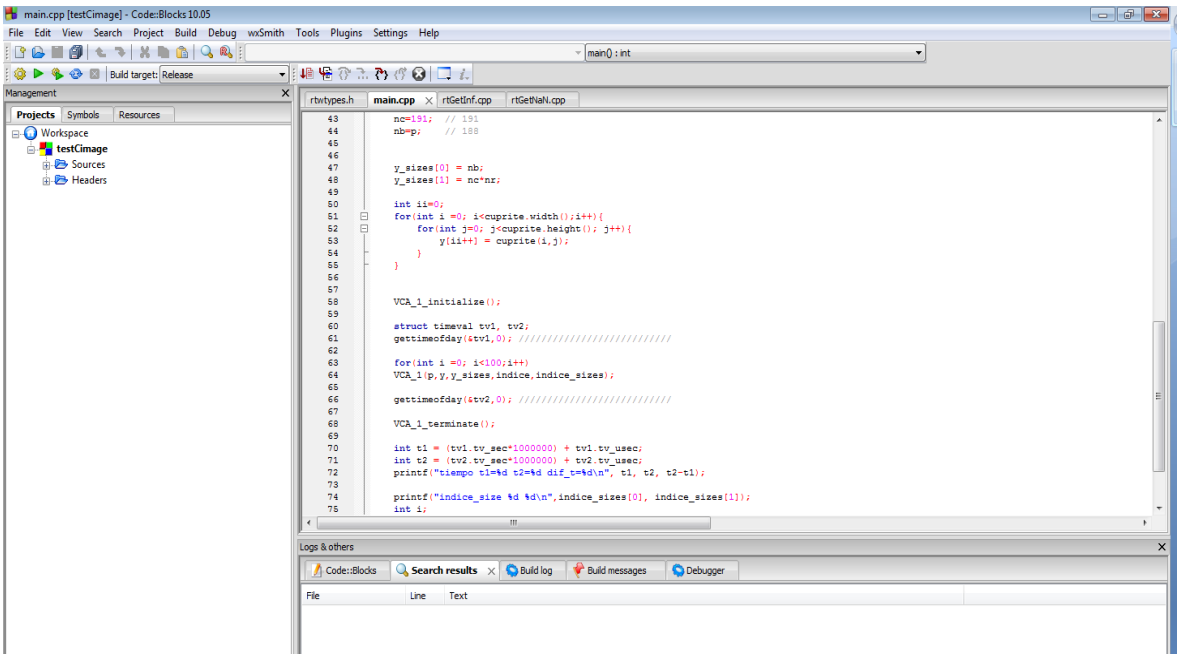


Figura 13. IDE de programación Code::Block

3.4. Herramienta se síntesis de alto nivel Catapult C

Catapult C [18] es una herramienta de sintetizado hardware de alto nivel. Es un software comercial producido por Mentor Graphics para la utilización de código C/C++ y genera salidas RTL en código VHDL y Verilog, entre otros, para la síntesis hardware de FPGAs y ASICs. Catapult C permite a los usuarios indicar las restricciones de tiempo y área indicando la frecuencia de reloj a utilizar y la tecnología de destino.

La utilización básica de Catapult no resulta complicada y sólo se requiere de la inserción de los archivos .cpp para iniciar el proceso. Posteriormente, Catapult C comprueba la corrección del código fuente teniendo que tener especial cuidado con el uso de funciones matemáticas incluidas en la cabecera math.h. En el caso de que alguna función no esté soportada, como por ejemplo *fabs()* se ha de implementar para su posterior uso. Una vez que el código es correcto, se continúa con la configuración del diseño, en donde se selecciona el dispositivo a sintetizar y la frecuencia de reloj que se quiere utilizar. El siguiente paso, Catapult genera las restricciones de la arquitectura seleccionada. Una vez finalizado este paso se pasa al organizador del sintetizado y para finalizar se realiza el paso de generación del lenguaje de descripción hardware RTL seleccionado.

Catapult C proporciona una herramienta llamada *Precision* que permite realizar los pasos necesarios para la síntesis del código RTL generado. Gracias a que obtiene los datos necesarios del dispositivo seleccionado de Catapult C solo se ha de pulsar el botón de sintetizado para generara automáticamente todo el proceso.

Se debe destacar que los tiempos de generación del código RTL y del sintetizado posterior son muy prolongados pudiéndose necesitar días para la finalización de este proceso.

En las figuras 14 y 15 se muestra una captura de pantalla del programa Catapult C y Precision.

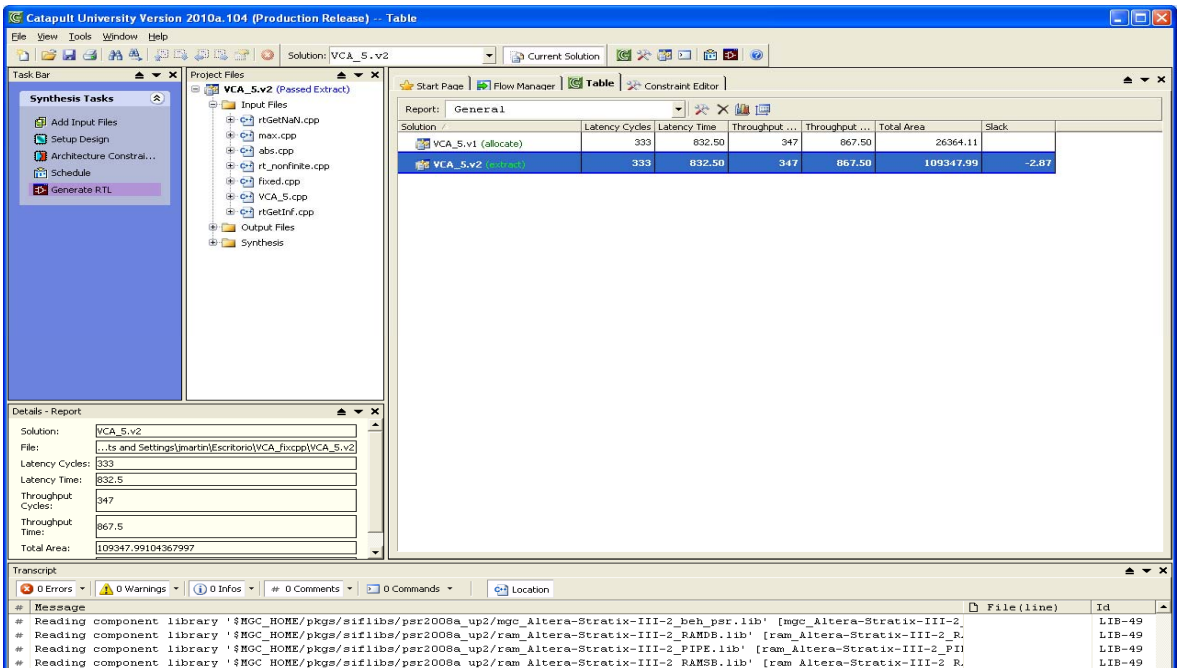


Figura 14. Ejecución del programa Catapult C

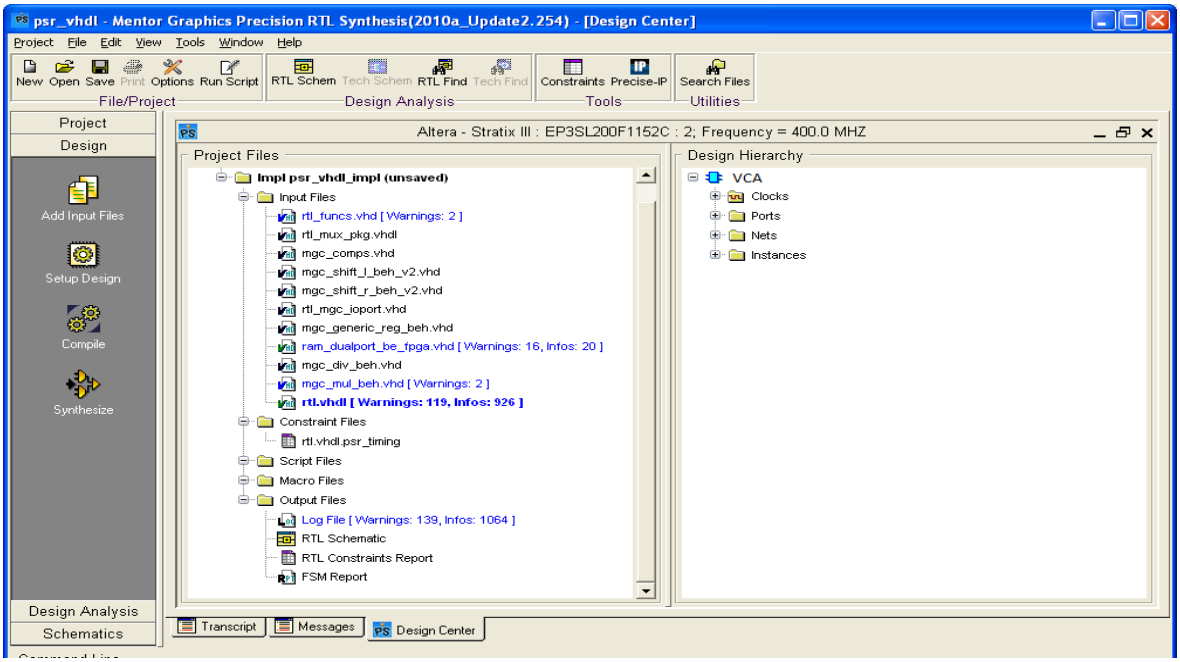


Figura 15. Ejecución del programa Precision

4. Trabajo realizado y descripción de la metodología utilizada

En este capítulo se va a describir de forma detallada el trabajo realizado en el TFM en la obtención del sintetizado hardware del algoritmo VCA a partir de un código fuente escrito en Matlab. Se va a describir las diferentes versiones del código realizado para sintetizar el algoritmo, introduciendo mejoras respecto al código original. Además se han utilizado diferentes estrategias en el uso de la aritmética de punto fijo y en la aritmética entera para intentar mejorar el rendimiento del algoritmo VCA. Recordemos que el objetivo ha sido realizar los cambios necesarios desde los lenguajes de alto nivel, sin introducirse en la complicación de la programación en el lenguaje de descripción hardware, para que de este modo podamos comprobar si los resultados obtenidos en este proceso son comparables a la implementación directa del lenguaje de descripción hardware en VHDL o en Verilog.

Una vez descrito el trabajo realizado se va a proceder a describir la metodología utilizada para la síntesis hardware a partir del algoritmo VCA programado en Matlab. La metodología, aunque ha sido realizada para un algoritmo en concreto, pretende ser una referencia a la hora de obtener cualquier código de descripción hardware para FPGAs desde algoritmos realizados en el lenguaje Matlab. A la hora de realizar esta metodología

se ha tenido en cuenta limitaciones que existen al trasladar algoritmos implementados en lenguajes de alto nivel que utilizan tipos de datos con coma flotante a arquitecturas empotradas que carecen de unidades especializadas para el procesado de estos datos. En este TFM se presentan algunas alternativas para la implantación de estos algoritmos sin necesidad de utilizar la notación de coma flotante.

4.1. Descripción del trabajo realizado

El trabajo desarrollado en el TFM se ha realizado de una manera incremental, en donde inicialmente se ha sintetizado una versión básica del algoritmo VCA y posteriormente se ha ido modificando para lograr mejoras en este código inicial. Las versiones del algoritmo VCA sintetizadas han sido las siguientes:

1. ***Versión 1. Algoritmo original VCA***. Esta versión contiene el código original sin modificaciones y respetando el tipo de datos utilizados para el cálculo de la VCA para el lenguaje de alto nivel.
2. ***Versión 2. Algoritmo VCA mejorado***. Esta versión contiene un código mejorado que utiliza funciones matemáticas más sencillas para sintetizar hardware, pero mantiene los tipos de datos utilizados en el lenguaje de alto nivel.
3. ***Versión 3. Algoritmo VCA mejorado que hace uso del tipo de punto fijo de Matlab Fixed-Point***. Utilizando el código VCA mejorado se sustituyen las variables de tipo *double* por variables de punto fijo *Fixed-Point* del *Toolbox* de Matlab.
4. ***Versión 4. Algoritmo VCA mejorado que hace uso del tipo de punto fijo para C++ fixed***. Utilizando el código VCA mejorado se sustituye las variables de tipo *double* en el código C/C++ por la librería de punto fijo *fixed*.
5. ***Versión 5. Algoritmo VCA mejorado que hace uso de aritmética entera***. Utilizando el código VCA mejorado se sustituyen las variables de tipo *double* por variables enteras *int* mediante el uso de aritmética entera.

4.1.1. ***Versión 1. Algoritmo original VCA***

En la primera versión del sintetizado del algoritmo VCA se han necesitado realizar los siguientes pasos: generación del código C a partir del código Matlab mediante la *Toolbox*

Embedded, prueba de funcionamiento y adecuación del código C, generación del lenguaje de descripción hardware a partir del código C haciendo uso del programa *Catapult* C. El código original escrito en Matlab del algoritmo VCA se muestra a en la Figura 16.

```
function indice = VCA_1( p, y )
    indice = zeros(1,p);
    A = zeros(p,p);
    A(p,1) = 1;
    for i=1:p
        w = ones(p,1);
        f = w - A*pinv(A)*w;
        f = f / sqrt(sum(f.^2));
        v = f'*y;
        [v_max indice(i)] = max(abs(v));
        A(:,i) = y(:,indice(i));
    end
end
```

Figura 16.Código original del VCA en Matlab

A continuación se describe los pasos necesarios para la síntesis hardware con mayor detalle.

Generación del código C a partir del código Matlab mediante la *Toolbox Embedded*

Una vez estudiada la *Toolbox Embedded* de Matlab, instalado el compilador de C/C++ utilizado por el *Embedded* y configurado los principales parámetros para su utilización, se procedió a realizar la primera generación de código C a partir del código fuente original del VCA. Para ello primero se hizo la siguiente configuración previa:

1. Generación del objeto de configuración del *Embedded*

```
>> rtwcfg = emlcoder.RTWConfig
```

2. Configuración gráfica de los parámetros del compilador. Cambiamos el compilador de C a C++, y eliminamos la opción de generar Makefile, la demás opciones las dejamos por defecto.

```
>> open rtwcfg
```

3. Introducción de la ruta del main.cpp que va a ser utilizado por el código generado en C/C++.

```
>> rtwcfg.CustomSource = 'main_vca_1.cpp'
```

4. Finalmente se ejecuta el comando para la generación del código C/C++, pasando como parámetro el objeto de configuración rtwcfg y se introduce el nombre del proyecto a generar VCA_1. A continuación se introduce el código de la función de Matlab VCA_1.m, y finalmente se introduce un ejemplo de los parámetros de entrada que requiere la función, en este caso un parámetro double, y una matriz de doublés con un tamaño de 5 filas por 10000 columnas.

```
>> emlc -s rtwcfg -d VCA_1 VCA_1.m -eg {'double', emlcoder.egs('double',
[5 10000])} -report
```

Una vez ejecutado el comando se obtiene un error debido a que es necesario acotar el tamaño del parámetro de entrada cuando el código de la función depende de este parámetro, para ello se utiliza la función *assert* en la primera línea, dentro de la función, de la siguiente manera.

```
assert(p <= 20);
```

De esta manera se indica en el código que el primer parámetro de la función *p* no puede tener valores superiores a 20, si eso sucede se producirá un error. Una vez modificado el código y ejecutado de nuevo el comando se obtiene un resultado positivo al cabo de unos pocos segundos de ejecución. Como resultado se obtiene un directorio llamado VCA_1 en el que se ha generado una serie de archivos .cpp y .h que corresponden al código fuente de C++.

Prueba de funcionamiento y adecuación del código C

Una vez que tenemos el código C generado por el *Embedded* de Matlab lo primero que hacemos es comprobar lo que se ha generado para poder probar posteriormente su correcto funcionamiento. El formato por defecto de generación de *Embedded* produce un archivo .cpp y .h por cada función utilizada en el código fuente de Matlab. De esta forma se han generado los códigos correspondientes a las funciones matriciales: *abs, max, priv, power, sum*, y *svd*. Junto a estos archivos fuente se generan también archivos fuentes auxiliares que definen los tipos de datos utilizados en el código, y funciones de inicialización y finalización del algoritmo. Finalmente aparece la función principal que tiene el mismo nombre que la función de Matlab utilizada VCA_1.cpp y VCA_1.h. Abriendo el archivo de

cabecera podemos ver cuál es el prototipo de función generada y cuáles y de qué tipo son los parámetros de entrada y salida generados para la utilización de la función. A continuación se presenta el prototipo de función generada.

```
void VCA_1(real_T eml_p, real_T eml_y_data[50000],int32_T eml_y_sizes[2],
real_T eml_indice_data[20], int32_T eml_indice_sizes[2]);
```

Podemos ver como la estructura de los parámetros de entrada y salida ha cambiado respecto a la función de Matlab. La función de C no retorna ningún dato, utilizando dos parámetros de salida para este fin, `eml_indice_data` y `eml_indice_sizes`. El primero es un vector con los índices calculados, mientras que el segundo es una variable auxiliar que indica el número de entradas utilizadas en el vector. Este segundo parámetro almacena dos valores puesto que está configurado para definir matrices, como en este caso el índice es un vector, el primer valor que indica el número de filas esta siempre fijado a 1, variando únicamente el número de columnas. Podemos ver que el parámetro de entrada que indica el número de *Endmembers* a buscar se mantiene mientras que el valor de la matriz de entrada se ha transformado en dos parámetros, el primero `eml_y_data` que almacena los valores en forma de vector y el segundo `eml_y_sizes` que indica el número de filas y columnas de la entrada.

En cuanto a los tipos de datos podemos ver como Matlab a definido tipos de datos nuevos respecto a los de C/C++ pero que tienen la siguiente correspondencia: `real_T = double`, `int32_T = int`. También podemos ver como *Embedded* ha cambiado el nombre de los parámetros, introduciendo el prefijo `eml_` a todos los parámetros y el sufijo `_data` y `_sizes` a los parámetros matriciales indicando en el primero que es el vector que almacena los datos y en el segundo que es el vector que almacena el tamaño de la matriz.

Una vez visto el prototipo de la función de C, vamos a la función principal *main* para modificar la llamada a la función VCA respetando este formato. Por lo tanto creamos un nuevo proyecto de C++ en la IDE de programación Code::Block incluyendo los archivos fuentes generados y realizando una llamada a la función VCA_1 desde la función principal. En la figura 17 se muestra el contenido del archivo *main* para el código VCA_1.

```
#include <CImg.h>
#include <wingdi.h>
#include <stdio.h>
#include <stdlib.h>
#include "VCA_1.h"
#include "VCA_1_initialize.h"
#include "VCA_1_terminate.h"
using namespace cimg_library;

int main()
{
    CImg<real_T> ima("../ejemplo10000_p5.tiff");
    real_T y[50000];
    int32_T y_sizes[2];
    real_T p;
    real_T indice[30];
    int32_T indice_sizes[2];
    p=5;
    y_sizes[0] = p;
    y_sizes[1] = 100*100;

    int ii=0;
    for(int i =0; i<ima.width();i++){
        for(int j=0; j<ima.height(); j++){
            y[ii++] = ima(i,j);
        }
    }
    VCA_1_initialize(); // función de inicialización VCA
    VCA_1(p,y,y_sizes,indice,indice_sizes); // función VCA
    VCA_1_terminate(); // función de finalización del VCA
    for (int i =0; i< indice_sizes[1]; i++){
        printf("Resultado %g \n",indice[i]);
    }
    return 0;
}
```

Figura 17. Código de la función main utilizada para el testeo del funcionamiento de la función VCA_1 generado por el *Embedded* de Matlab

Para ejecutar el programa de prueba se va a utilizar una librería de imágenes llamada CImg que nos va a permitir cargar el ejemplo guardado en la imagen de formato tiff generado en la demo VCA de Matlab, de esta forma podremos comparar los resultados y ver si coinciden. Para insertar la imagen en el parámetro de entrada de la función tenemos que tener en cuenta que Matlab trabaja en formato filas por columnas, mientras que en las librerías de imágenes de C/C++ se utiliza el sistema de coordenadas x/y, por lo tanto se ha de leer los datos de la imagen por columnas para seguir el formato de Matlab. Además de introducir los datos del vector se ha de introducir el tamaño de la matriz en formato filas por columnas, recordemos que los datos tienen 5 filas correspondientes a la reducción de los canales hiperespectrales mientras que se compone de 10000 columnas correspondientes al número de píxeles de la imagen. El parámetro p indica el número de *Endmembers* que se quieren obtener, este número coincide con el número de canales reducidos.

Una vez que tenemos los parámetros de entrada y hemos generado los de salida nos disponemos a llamar a la función de inicialización de variables que utiliza la función VCA,

posteriormente se llama a la función VCA finalizando el uso del algoritmo con la liberación de los recursos de memoria mediante la función de terminación.

Para finalizar se visualiza el resultado de los índices de los *Endmembers* recorriendo el vector de índices. Se realizaron test con imágenes generadas de forma sintética desde Matlab, y se utilizó una imagen de referencia en el entorno hiperespectral (*Cuprite*) para validar los resultados obtenidos, pudiéndose constatar que se obtenían los mismos resultados tanto en Matlab como en C++. En el capítulo de resultados se describen los test con mayor detalle.

Generación del lenguaje de descripción hardware a partir del código C

Una vez que se comprueba que el código C generado por *Embedded* Matlab es correcto se procede a utilizar la herramienta de sintetizado hardware *Catapult* C. Para ello se crea un nuevo proyecto y se introducen todos los archivos de código fuente .cpp generado por *Embedded* sin incluir la función principal main. Una vez que se ha cargado todo el código fuente se introduce el *pragma hls_design top* encima de la función VCA_1 para indicar a *Catapult* que es la función principal que se quiere sintetizar. Realizada la configuración se inicia el proceso de sintetizado haciendo click en el botón de *"Setup Design"*, una vez ejecutado se obtiene múltiples errores relacionados con funciones matemáticas definidas en el archivo math.h. Los errores se deben a que *Catapult* no implementa muchas funciones matemáticas que sí proporciona el compilador de C/C++. Las funciones matemáticas utilizadas en el programa son: *fabs, sqrt, floor, pow*. Por lo tanto se ha de implementar estas funciones para lograr sintetizar este código, las funciones *fabs* y *floor* son muy sencillas de implementar mientras que las funciones *sqrt* y *pow* tienen una mayor dificultad. Las funciones se van a incluir en unos nuevos archivos fuentes llamados math_fun.h y math_fun.cpp. En la figura 18 se presenta el código de las funciones implementadas.

```
#include "rtwtypes.h"
#include "math_fun.h"

real_T fabs(real_T x){
    real_T y;
    if( x<0 ) y = -x;
    else      y = x;
    return y;
}

real_T sqrt(real_T m){
```

```
    int j;
    real_T i=0;
    real_T x1,x2;
    while( (i*i) <= m ) i+=0.1;
    x1=i;
    for(j=0;j<10;j++){
        x2=m;
        x2/=x1;
        x2+=x1;
        x2/=2;
        x1=x2;
    }
    return x2;
}

real_T floor(real_T x){
    int32_T w (int32_T) x;
    return (real_T) w;
}

real_T pow(real_T x, real_T y){
    int i;
    real_T result;
    if( y >= 0){
        result = 1;
        for(i = 0 ; i < y ; i++) result *= x;
    }else{
        result = 1;
        for ( i = -1 ; i >= y ; i--) result /= x;
    }
    return x;
}
```

Figura 18. Math_fun.h, archivo fuente que define las funciones matemáticas del math.h no compatibles en *Catapult* C

Una vez implementadas las funciones y comprobado su buen funcionamiento se procede a incluir los archivos fuentes en *Catapult* C, así como en las fuentes del VCA, comentando los includes de math.h original para evitar errores por múltiples definiciones de las funciones. Con esto volvemos a ejecutar *Catapult* y logramos eliminar estos errores. Pocos segundos más tarde emerge un panel en donde se selecciona el dispositivo utilizado, en nuestro caso la Altera Stratix III, y se configura la frecuencia de funcionamiento (400 MHz). Se pulsa en el botón de aplicar de esta opción y unos 15 minutos (dependiendo del ordenador) finaliza la etapa de *"Setup Design"*. A continuación se pulsa el botón de *"Architecture constraint"* donde se genera la arquitectura del sintetizado teniendo en cuenta sus restricciones. En esta etapa el tiempo necesario de ejecución ronda los 30 minutos. Finalizada esta etapa se permite la ejecución del botón *"Schedule"* en donde se planifica la síntesis hardware, este paso puede tardar entre una y dos horas. Para finalizar se pulsa el botón *"Generate RTL"* en donde se genera el código de descripción hardware

configurado al inicio de *Catapult* (VHDL y/o Verilog). Este último paso es el más costoso del proceso que duró unas 5 o 6 horas.

Finalizada la ejecución del último paso obtenemos el código de descripción hardware objetivo del TFM. Para obtener información de la bondad del código generado es necesario realizar la síntesis hardware del código mediante la utilidad incluida en Catapult denominada Precision. Este software permite sintetizar tanto código VHDL como Verilog para las FPGAs de Xilinx y Altera. Los paso necesarios son la compilación y la síntesis del código. Los datos de configuración por defecto son los definidos anteriormente en *Catapult*. Este proceso puede durar hasta 6 horas, finalizado este tiempo podemos obtener información detallada sobre la frecuencia y latencia del sintetizado así como el área y recursos utilizados de la FPGA.

En la figura 19 se muestra una captura de pantalla de la ejecución del programa *Catapult* C que se encuentra ejecutando el paso *Schedule* necesario para la generación del código de descripción hardware en VHDL y Verilog.

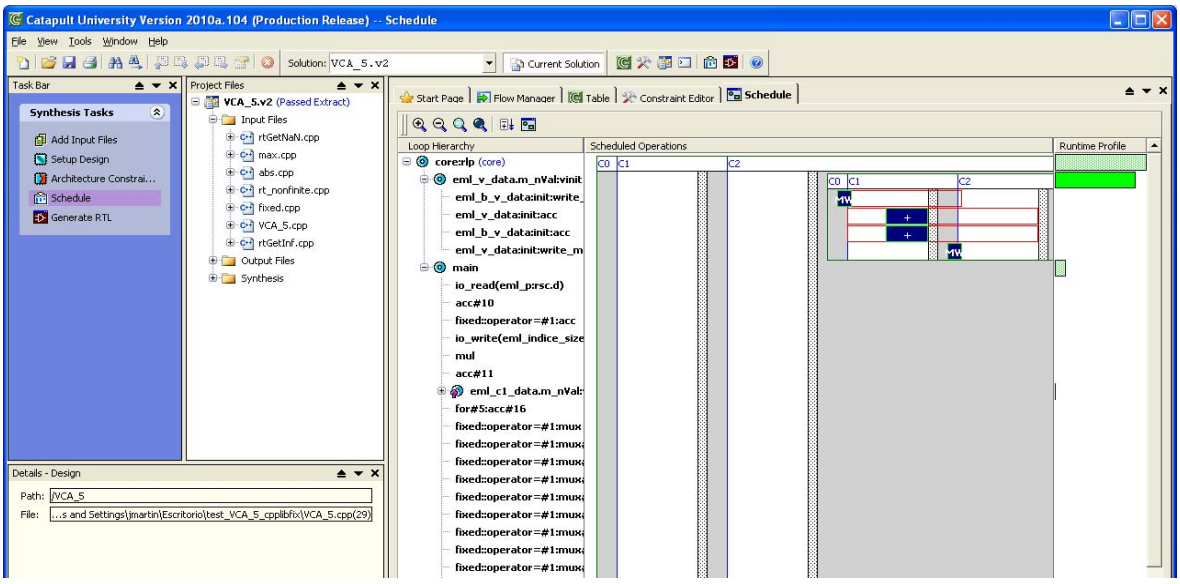


Figura 19. Pantallazo de la aplicación *Catapult* C

4.1.2. Versión 2. Algoritmo VCA mejorado

En la segunda versión del sintetizado del algoritmo VCA mejorado se han necesitado realizar los siguientes pasos: adecuación del código Matlab, generación del código C a partir del código Matlab mediante la *Toolbox Embedded*, prueba de funcionamiento y adecuación del código C, generación del lenguaje de descripción hardware a partir del código C haciendo uso del programa *Catapult* C. A continuación se describen estos paso con mayor detalle.

Adecuación del código Matlab

Para mejorar los resultados iniciales del algoritmo se ha utilizado una versión mejorada que utiliza un cálculo numérico más adecuado para el hardware. Para lograr mejores resultados se ha realizado una adecuación del código que permite la utilización de tipos de datos como enteros en los índices y los iteradores así como un desenrollado del código que permite la simplificación de las funciones matriciales. Además, teniendo la experiencia del proceso inicial, se van a inicializar las variables de forma estática para eliminar errores en el *Embedded* Matlab. En la figura 20 se presentan la función VCA mejorada y en la figura 21 se muestra la correspondiente función adecuada para la utilización del *Toolbox Embedded* Matlab.

```
% Función VCA mejorado original
function indice = VCA_2( p, y )
    assert(p <= 20);
    indice = zeros(1,p);
    A = zeros(p,p);
    A(p,1) = 1;
    for i=1:p
         w = ones(p,1);
         if (i == 1)
           f = w;
           f(p) = 0;
         elseif (i == 2)
            f = [ones(p-1,1);-(sum(A(1:p-1,1))/A(p,1))];
            u(:,1) = A(:,1);
            c2(i-1) = (w'*u(:,1))/(u(:,1)'*u(:,1));
         else
            c1 = (u(:,1:i-2)'*A(:,i-1))'./(sum(u(:,1:i-2).*u(:,1:i-2)));
            u(:,i-1) = A(:,i-1) - sum(c1(ones(1,p),:).*u(:,1:i-2),2);
            c2(i-1) = (w'*u(:,i-1))/(u(:,i-1)'*u(:,i-1));
            f = w -  sum(c2(ones(1,p),1:i-1).*u(:,1:i-1),2);
         end
         v = f'*y;
         [v_max indice(i)] = max(abs(v));
         A(:,i) = y(:,indice(i));
     end
end
```

Figura 20. Código Matlab mejorado del VCA

```
% Función VCA mejorada y adecuada para el Embedded
function indice = VCA_5( p, y ) %#eml
    assert(p <= 20);
    indice = zeros(1,p,'int32');
    A = zeros(p,p,'double');
    A(p,1) = double(1);
    u = zeros(p,p,'double');
    c2 = zeros(1,p,'double');
    f  = zeros(1,p,'double');
    for i=1:p
         w = ones(p,1,'double');
         if (i == 1)
             f = w;
             f(p) = 0;
         elseif (i == 2)
             f(p) = double(0);
             for ii=1:p-1
                f(ii) = double(1);
                f(p) = f(p) - (A(ii,1)/A(p,1));
             end
             u(:,1) = A(:,1);
             c2_1 = double(0);
             c2_2 = double(0);
             for ii=1:p
                 c2_1 = c2_1 + (u(ii,1));
                 c2_2 = c2_2 + ((u(ii,1)*u(ii,1)));
             end
             c2(i-1) = (c2_1)/c2_2;
         else  %%%% i > 2
             c1   = zeros(1,i-2,'double');
             c1_1 = zeros(1,i-2,'double');
             c1_2 = zeros(1,i-2,'double');
             for ii=1:i-2
                for jj=1:p
                    c1_1(ii) = c1_1(ii) + ((u(jj,ii)*A(jj,i-1)));
                    c1_2(ii) = c1_2(ii) + ((u(jj,ii)*u(jj,ii)));
                end
             end
             for ii=1:i-2
                c1(ii) =  (c1_1(ii))/c1_2(ii);
             end
             u_sum = zeros(p,1,'double');
             for ii=1:i-2
                for jj=1:p
                    u_sum(jj) = u_sum(jj) + ((c1(1,ii)*u(jj,ii)));
                end
             end
             for jj=1:p
                u(jj,i-1) = A(jj,i-1) - u_sum(jj);
             end
             c2_1 = double(0);
             c2_2 = double(0);
             for ii=1:p
                c2_1 = c2_1 + u(ii,i-1);
                c2_2 = c2_2 + ((u(ii,i-1)*u(ii,i-1)));
             end
             c2(i-1) = (c2_1)/c2_2;
             f_sum = zeros(p,1,'double');
             for ii=1:i-1
                for jj=1:p
                    f_sum(jj) = f_sum(jj) + ((c2(1,ii)*u(jj,ii)));
                end
             end
             for jj=1:p
                f(jj) = w(jj) - f_sum(jj);
             end
         end
         [nf nc] = size(y);
         v = zeros(1,nc,'double');
         for ii=1:nc
            for jj=1:p
                v(ii) = v(ii) + (f(jj)*double(y(jj,ii)));
```

```
            end
        end
        [v_max indice(i)] = max(abs(v));
        A(:,i) = double(y(:,indice(i)));
    end
end
```

Figura 21. Código Matlab adecuado al *Embedded* C del VCA mejorado

Podemos ver como la adecuación del código ha producido más líneas de código, pero vemos como se han simplificado muchas funciones matriciales que no eran completamente necesarias por bucles anidados que operan solamente sobre los datos requeridos.

También podemos ver como se han utilizado tipos de datos enteros en el uso de los iteradores y en las variables que tienen uso de enteros. Por ejemplo el índice se ha definido como entero puesto que los datos que alberga son las posiciones de los píxeles *Endmembers* de la imagen de entrada.

Generación del código C a partir del código Matlab mediante la *Toolbox Embedded*

Siguiendo el proceso indicado para la primera versión se procedió a realizar la generación de código C a partir del código fuente del VCA mejorado. Los pasos de configuración del *Embedded* fueron idénticos que en la primera versión, los únicos cambios realizados fueron el nombre de la función VCA_2.m y por consiguiente el nombre de la salida VCA_2 y el parámetro de entrada p que pasa a ser un entero de 16 bits sin signo. A continuación se muestra el comando para la generación del código C del *Toolbox Embedded* Matlab.

```
>> emlc -s rtwcfg -d VCA_2 VCA_2.m -eg {'uint16', emlcoder.egs('double',
[5 10000])} -report
```

Una vez ejecutado el comando se obtiene un resultado positivo al cabo de unos pocos segundos de ejecución. Como resultado se obtiene un directorio llamado VCA_2 en el que se ha generado una serie de archivos .cpp y .h que corresponden al código fuente de C++.

Prueba de funcionamiento y adecuación del código C

Una vez que tenemos el código C generado por el *Embedded* de Matlab lo primero que hacemos es mirar lo que se ha generado para poder probar posteriormente su correcto funcionamiento. El formato por defecto de generación de *Embedded* produce un

documento .cpp y .h por cada función utilizada en el código fuente de Matlab. De esta forma se han generado los códigos correspondientes a las funciones matriciales: *abs, max*. Como podemos ver se utilizan menos funciones que en la primera versión gracias a la simplificación del código. Junto a estos archivos fuente se generan también archivos fuentes auxiliares que definen los tipos de datos utilizados en el código, y funciones de inicialización y finalización del algoritmo. Finalmente aparece la función principal que tiene el mismo nombre que la función de Matlab utilizada VCA_2.cpp y VCA_2.h. Abriendo el archivo de cabecera podemos ver cuál es el prototipo de función generada y cuáles y de qué tipo son los parámetros de entrada y salida generados para la utilización de la función. A continuación se presenta el prototipo de función generada.

```
void VCA_2(uint16_T eml_p,real_T eml_y_data[50000],int32_T eml_y_sizes[2],
int32_T eml_indice_data[20], int32_T eml_indice_sizes[2]);
```

Podemos ver como la estructura de los parámetros de entrada y salida ha cambiado respecto al prototipo generado en la primera versión. Se observa como el parámetro de entrada p es del tipo `uint16_T = unsigned short int` y también podemos observar como el parámetro de salida del índice pasa a ser un `int32_T` como consecuencia de la definición de la variable como entera dentro de la función.

Una vez visto el prototipo de la función de C, vamos a la función principal *main* para modificar la llamada a la función VCA respetando este formato. Por lo tanto creamos un nuevo proyecto de C++ en la IDE de programación Code::Block incluyendo los archivos fuentes generados y realizando una llamada a la función VCA_2 desde la función principal.

Se realizaron los mismos test de imágenes realizados para la primera versión pudiéndose constatar que se obtenía el mismo resultado tanto en para Matlab como para el código generado de C.

Generación del lenguaje de descripción hardware a partir del código C

Una vez que se comprueba que el código C generado por *Embedded* Matlab es correcto se procede a utilizar la herramienta de sintetizado hardware *Catapult* C. El proceso fue muy similar al descrito en la primera versión, solamente destacar que el número de funciones matemáticas utilizadas en este algoritmo VCA fue muy inferior al original utilizando

solamente la función *fabs*. Los tiempos de ejecución para la generación del código de descripción hardware fue elevado, igual que la primera versión del algoritmo, de la misma forma que en la síntesis mediante la utilidad Precision necesitó de cerca de 6 horas. Para la síntesis de esta versión del VCA se ha seleccionado la Altera Stratix III, al igual que en la primera versión. Se pretende utilizar este mismo dispositivo en todos los sintetizados para poder comparar los resultados obtenidos.

4.1.3. ***Versión 3. Algoritmo VCA mejorado que hace uso del tipo de punto fijo de Matlab Fixed-Point***

En la tercera versión del sintetizado del algoritmo VCA se han necesitado realizar los siguientes pasos: adecuación del código Matlab, generación del código C a partir del código Matlab mediante la *Toolbox Embedded*, prueba de funcionamiento y adecuación del código C, generación del lenguaje de descripción hardware a partir del código C haciendo uso del programa *Catapult* C. A continuación se describen estos paso con mayor detalle.

Adecuación del código Matlab

Para continuar con el proceso de posibles mejoras del algoritmo inicial vamos a utilizar la *Toolbox* de Matlab para punto fijo llamada *Fixed-Point*. En las primeras pruebas realizadas con esta *Toolbox* detectamos que no todas las utilidades de *Fixed-Point* están soportadas para la *Toolbox* de *Embedded* Matlab. El tamaño de las variables de punto fijo ha de ser prefijadas no pudiendo utilizar la funcionalidad de gestión automática proporcionada por Matlab. Además se ha de definir para cada variable el tamaño de la ALU para la suma y el producto indicando el formato de suma y producto. Los formatos compatibles para *Embedded* son únicamente los que se parecen más al comportamiento en el uso del entero del lenguaje C/C++. Por lo tanto la definición de una variable *Fixed-Point* queda mucho más compleja para la utilización de *Embedded*. A continuación se muestra la inicialización de una variable *Fixed-Point* genérica de Matlab y la inicialización de la misma variable con compatibilidad con la *Toolbox* de *Embedded*.

```
Pi_ = fi(3.1416);
Pi_ = fi(3.1416,1,64,32'SumMode','KeepLSB','ProductMode','KeepLSB', …
'ProductWordLength',128,'ProductFractionLength',64,'SumWordLength', …
128,'SumFractionLength',64);
```

Los parámetros de la segunda inicialización tienen el siguiente significado:

- 3.1416: Es el valor numérico que inicializa el *Fixed-Point*
- 1: Indica que la variable tiene signo
- 64: Indica el tamaño total de la variable en bits
- 32: Indica el tamaño correspondiente a la parte decimal
- `'SumMode','KeepLSB'`: Modo de suma igual a KeepLSB
- `'ProductMode','KeepLSB'`: Modo de multiplicación igual a KeepLSB
- `'ProductWordLength',128`: Tamaño del producto igual a 128
- `'ProductFractionLength',64`: Tamaño parte decimal del producto igual a 64
- `'SumWordLength', 128`: Tamaño de la suma igual a 128
- `'SumFractionLength',64`: Tamaño de la parte decimal de la suma igual a 64

En la Figura 22 se presentan la función VCA mejorada que hace uso del tipo de datos de punto fijo de Matlab y es compatible con el *Toolbox Embedded* de Matlab.

```
function indice = VCA_3_fix( p, y ) %#eml
    assert(p <= 50);
    indice = zeros(1,p,'int32');
    wl = 96;
    fl = 24;
    pwl = 128;
    pfl = 64;
    swl = 128;
    sfl = 64;
    T = numerictype(1,wl,fl);
    A  = fi(zeros(p,p),1,wl,fl,'SumMode','KeepLSB', 'ProductMode', 'KeepLSB','ProductWordLength',…
          pwl, 'ProductFractionLength', pfl, 'SumWordLength', swl, 'SumFractionLength', sfl);
    u  = fi(zeros(p,p),1,wl,fl,'SumMode','KeepLSB', 'ProductMode', 'KeepLSB', 'ProductWordLength',…
          pwl, 'ProductFractionLength', pfl, 'SumWordLength', swl, 'SumFractionLength', sfl);
    c2 = fi(zeros(1,p),1,wl,fl,'SumMode','KeepLSB', 'ProductMode', 'KeepLSB', 'ProductWordLength',…
         pwl, 'ProductFractionLength', pfl, 'SumWordLength', swl, 'SumFractionLength', sfl);
    f  = fi(zeros(1,p),1,wl,fl,'SumMode','KeepLSB', 'ProductMode', 'KeepLSB', 'ProductWordLength',…
         pwl, 'ProductFractionLength', pfl, 'SumWordLength', swl, 'SumFractionLength', sfl);
    A(p,1) = fi(1,1,wl,fl,'SumMode','KeepLSB', 'ProductMode', 'KeepLSB', 'ProductWordLength',…
          pwl, 'ProductFractionLength', pfl, 'SumWordLength', swl, 'SumFractionLength', sfl);
    for i=1:p
          w = fi(ones(p,1),1,wl,fl,'SumMode','KeepLSB', 'ProductMode', 'KeepLSB', 'ProductWordLength',…
                pwl, 'ProductFractionLength', pfl, 'SumWordLength', swl, 'SumFractionLength', sfl);
          if (i == 1)
              f = w;
              f(p) = fi(0,1,wl,fl,'SumMode','KeepLSB', 'ProductMode', 'KeepLSB', 'ProductWordLength',…
                  pwl, 'ProductFractionLength', pfl, 'SumWordLength', swl, 'SumFractionLength', sfl);
          elseif (i == 2)
              f(p) = fi(0,1,wl,fl,'SumMode','KeepLSB', 'ProductMode', 'KeepLSB', 'ProductWordLength',…
                   pwl, 'ProductFractionLength', pfl, 'SumWordLength', swl, 'SumFractionLength', sfl);
              for ii=1:p-1
                 f(ii) = fi(1,1,wl,fl,'SumMode','KeepLSB','ProductMode','KeepLSB', 'ProductWordLength',…
                   pwl, 'ProductFractionLength', pfl, 'SumWordLength', swl, 'SumFractionLength', sfl);
                 f(p) = f(p) - divide(T,A(ii,1),A(p,1));
              end
              u(:,1) = A(:,1);
              c2_1 = fi(0,1,wl,fl,'SumMode','KeepLSB', 'ProductMode', 'KeepLSB', 'ProductWordLength',…
                  pwl, 'ProductFractionLength', pfl, 'SumWordLength', swl, 'SumFractionLength', sfl);
              c2_2 = fi(0,1,wl,fl,'SumMode','KeepLSB', 'ProductMode', 'KeepLSB', 'ProductWordLength',…
                  pwl, 'ProductFractionLength', pfl, 'SumWordLength', swl, 'SumFractionLength', sfl);

              for ii=1:p
                  c2_1 = fi(c2_1+u(ii,1),1,wl,fl,'SumMode','KeepLSB','ProductMode','KeepLSB',…
                           'ProductWordLength', pwl, 'ProductFractionLength', pfl, 'SumWordLength',…
                           swl, 'SumFractionLength', sfl);
                  c2_2 = fi(c2_2+(u(ii,1)*u(ii,1)),1,wl,fl,'SumMode','KeepLSB','ProductMode','KeepLSB',…
```

```
                    'ProductWordLength',pwl,'ProductFractionLength',pfl,'SumWordLength',swl,…
                    'SumFractionLength', sfl);
        end
        c2(i-1) = divide(T, c2_1, c2_2);
    else  %%%% i > 2
        c1 = fi(zeros(1,i-2),1,wl,fl,'SumMode','KeepLSB','ProductMode','KeepLSB',…
            'ProductWordLength',pwl,'ProductFractionLength',pfl,'SumWordLength',swl,…
            'SumFractionLength', sfl);
        c1_1 = fi(zeros(1,i-2),1,wl,fl,'SumMode','KeepLSB', 'ProductMode', 'KeepLSB',…
            'ProductWordLength',pwl,'ProductFractionLength',pfl,'SumWordLength',swl,…
            'SumFractionLength', sfl);
        c1_2 = fi(zeros(1,i-2),1,wl,fl,'SumMode','KeepLSB','ProductMode','KeepLSB',…
            'ProductWordLength',pwl,'ProductFractionLength',pfl,'SumWordLength',swl,…
            'SumFractionLength', sfl);
        for ii=1:i-2
           for jj=1:p
               c1_1(ii) = fi((c1_1(ii)+(u(jj,ii)*A(jj,i-1))),1,wl,fl,'SumMode','KeepLSB',…
                    'ProductMode', 'KeepLSB', 'ProductWordLength', pwl, 'ProductFractionLength',…
                    pfl, 'SumWordLength', swl, 'SumFractionLength', sfl);
               c1_2(ii) = fi((c1_2(ii)+(u(jj,ii)*u(jj,ii))),1,wl,fl,'SumMode','KeepLSB',…
                    'ProductMode', 'KeepLSB', 'ProductWordLength', pwl, 'ProductFractionLength',…
                    pfl, 'SumWordLength', swl, 'SumFractionLength', sfl);
           end
        end
        for ii=1:i-2
           c1(ii) = divide(T, c1_1(ii), c1_2(ii));
        end
        u_sum = fi(zeros(p,1),1,wl,fl,'SumMode','KeepLSB', 'ProductMode', 'KeepLSB',…
                    'ProductWordLength', pwl, 'ProductFractionLength', pfl, 'SumWordLength', swl,…
                    'SumFractionLength', sfl);
        for ii=1:i-2
           for jj=1:p
               u_sum(jj) = u_sum(jj) + ((c1(1,ii)*u(jj,ii)));
           end
        end
        for jj=1:p
           u(jj,i-1) = A(jj,i-1)-u_sum(jj);
        end
        c2_1 = fi(0,1,wl,fl,'SumMode','KeepLSB', 'ProductMode', 'KeepLSB', 'ProductWordLength',…
                    pwl,'ProductFractionLength',pfl,'SumWordLength',swl,'SumFractionLength',sfl);
        c2_2 = fi(0,1,wl,fl,'SumMode','KeepLSB', 'ProductMode', 'KeepLSB', 'ProductWordLength',…
                    pwl,'ProductFractionLength',pfl,'SumWordLength',swl,'SumFractionLength',sfl);
        for ii=1:p
           c2_1 = fi((c2_1+u(ii,i-1)),1,wl,fl,'SumMode','KeepLSB', 'ProductMode', 'KeepLSB',…
                    'ProductWordLength', pwl, 'ProductFractionLength', pfl, 'SumWordLength',…
                    swl, 'SumFractionLength', sfl);
           c2_2 = fi((c2_2+(u(ii,i-1)*u(ii,i-1))),1,wl,fl,'SumMode','KeepLSB', 'ProductMode',…
                    'KeepLSB', 'ProductWordLength', pwl, 'ProductFractionLength', pfl,…
                    'SumWordLength', swl, 'SumFractionLength', sfl);
        end
        c2(i-1) = divide(T,c2_1,c2_2);
        f_sum = fi(zeros(p,1),1,wl,fl,'SumMode','KeepLSB', 'ProductMode', 'KeepLSB',…
                    'ProductWordLength', pwl, 'ProductFractionLength', pfl, 'SumWordLength', swl,…
                    'SumFractionLength', sfl);
        for ii=1:i-1
           for jj=1:p
               f_sum(jj)=fi(f_sum(jj)+c2(1,ii)*u(jj,ii),1,wl,fl,'SumMode','KeepLSB','ProductMode'…
                           ,'KeepLSB','ProductWordLength',pwl,'ProductFractionLength',pfl,…
                           'SumWordLength', swl, 'SumFractionLength', sfl);
           end
        end
        for jj=1:p
           f(jj)=fi(w(jj)-f_sum(jj),1,wl,fl,'SumMode','KeepLSB', 'ProductMode', 'KeepLSB',…
                    'ProductWordLength', pwl, 'ProductFractionLength', pfl, 'SumWordLength',…
                    swl, 'SumFractionLength', sfl);
        end
    end
    [nf nc] = size(y);
    v = fi(zeros(1,nc),1,wl,fl,'SumMode','KeepLSB','ProductMode','KeepLSB','ProductWordLength',…
            pwl, 'ProductFractionLength', pfl, 'SumWordLength', swl, 'SumFractionLength', sfl);
    for ii=1:nc
       for jj=1:p
           fi_y = fi(int32(y(jj,ii)),1,wl,fl,'SumMode','KeepLSB', 'ProductMode', 'KeepLSB',…
                    'ProductWordLength', pwl, 'ProductFractionLength', pfl, 'SumWordLength',…
                    swl, 'SumFractionLength', sfl);
           v(ii) = fi((v(ii)+(f(jj)*fi_y)),1,wl,fl,'SumMode','KeepLSB','ProductMode','KeepLSB',…
                    'ProductWordLength', pwl, 'ProductFractionLength', pfl, 'SumWordLength', swl,
                    'SumFractionLength', sfl);
       end
    end
    [v_max indice(i)] = max(abs(v));
    A(:,i) = fi(int32(y(:,indice(i))),1,wl,fl,'SumMode','KeepLSB', 'ProductMode', 'KeepLSB',…
                    'ProductWordLength', pwl, 'ProductFractionLength', pfl, 'SumWordLength', swl,
```

```
                        'SumFractionLength', sfl);
    end
end
```

Figura 22. Función VCA_3_fix que utiliza aritmética en punto fijo del *Toolbox Fixed-Point*

Podemos observar como el uso del *Fixed-Point* en el código compatible para *Embedded* Matlab dificulta de forma importante su comprensión. Sin embargo los parámetros de entrada y salida que la función utiliza no requieren del uso del tipo *Fixed-Point*.

Generación del código C a partir del código Matlab mediante la *Toolbox Embedded*

Siguiendo el proceso indicado para la primera versión se procedió a realizar la generación de código C a partir del código fuente del VCA mejorado con *Fixed-Point*. Los pasos de configuración del *Embedded* fueron similares a la primera versión, los cambios realizados fueron el nombre de la función VCA_3_fix.m y por consiguiente el nombre de la salida VCA_3_fix. Además para eliminar el uso del *double* en el código de la función se utiliza una entrada entera de 8 bits sin signo *uint8* que es el formato de la mayoría de las imágenes. A continuación se muestra el comando para la generación del código C del *Toolbox Embedded* Matlab.

```
>> emlc -s rtwcfg -d VCA_3_fix VCA_3_fix.m -eg {'uint16',
      emlcoder.egs('uint8', [5 10000])} -report
```

Una vez ejecutado el comando se obtiene un resultado positivo al cabo de unos pocos segundos de ejecución. Como resultado se obtiene un directorio llamado VCA_3_fix en el que se ha generado una serie de archivos .cpp y .h que corresponden al código fuente de C++.

Prueba de funcionamiento y adecuación del código C

Una vez que tenemos el código C generado por el *Embedded* de Matlab lo primero que hacemos es comprobar lo que se ha generado para poder probar posteriormente su correcto funcionamiento. El formato por defecto de generación de *Embedded* produce un documento .cpp y .h por cada función utilizada en el código fuente de Matlab. Abriendo el archivo de cabecera podemos ver cuál es el prototipo de la función generada y cuáles y de qué tipo son los parámetros de entrada y salida generados para la utilización de la función. A continuación se presenta el prototipo de función generada.

```
void VCA_3_fix(uint16_T eml_p, char_T eml_y_data[50000], int32_T
eml_y_sizes[2], int32_T eml_indice_data[20], int32_T eml_indice_sizes[2]);
```

Podemos observar como el prototipo de la función es similar a la de la versión anterior puesto que solo ha cambiado el tipo de la variable `eml_y_data` que es un entero de 8 bits *char_T*. Los tipos *Fixed-Point* generados se utilizan de forma interna en la función siendo una estructura que almacena una serie de valores enteros gestionados mediante aritmética de punto fijo.

```
/* Type Definitions */
typedef struct {
    uint32_T chunks[4];
} int128m_T;
typedef struct {
    uint32_T chunks[5];
} int160m_T;
typedef struct {
    uint32_T chunks[2];
} int64m_T;
typedef struct {
    uint32_T chunks[3];
} int96m_T;
```

Podemos observar las estructuras de C/C++ que almacenan los valores de punto fijo teniendo diferentes tamaños dependiendo de las necesidades del algoritmo.

Una vez visto el prototipo de la función de C, vamos a la función principal *main* para modificar la llamada a la función VCA respetando este formato. Por lo tanto creamos un nuevo proyecto de C++ en la IDE de programación Code::Block incluyendo los archivos fuentes generados y realizando una llamada a la función VCA_3_fix desde la función principal.

Se realizaron múltiples pruebas con diferentes imágenes generadas desde Matlab y se pudo constatar que se obtenía el mismo resultado únicamente variando el orden de los *Endmembers* en alguna ocasión.

Generación del lenguaje de descripción hardware a partir del código C

Una vez que se comprueba que el código C generado por *Embedded* Matlab es correcto se procede a utilizar la herramienta de sintetizado hardware *Catapult* C. El proceso fue muy similar al descrito en versiones anteriores. Los tiempos de ejecución para la generación del

código de descripción hardware fue mucho más elevado que el tiempo para la síntesis del algoritmo en coma flotante, el total del tiempo necesario para la finalización del proceso superó los 2 días. Finalmente se obtuvo los valores del sintetizado hardware para la FPGA Altera Stratix III necesarios para la comparación de las diferentes versiones.

4.1.4. Versión 4. Algoritmo VCA mejorado que hace uso del punto fijo para C++

En la cuarta versión del sintetizado del algoritmo VCA se han necesitado realizar los siguientes pasos: adecuación del código Matlab, generación del código C a partir del código Matlab mediante la *Toolbox Embedded*, prueba de funcionamiento y adecuación del código C, generación del lenguaje de descripción hardware a partir del código C haciendo uso del programa *Catapult* C. A continuación se describen estos paso con mayor detalle.

Adecuación del código Matlab

Como alternativa del uso de la *Toolbox* de punto fijo de Matlab se ha utilizado una librería de punto fijo para C++ que permite reemplazar el tipo de coma flotante por el punto fijo desde el código C++. Por lo tanto la adecuación del código Matlab no contempla el uso del *Fixed-Point* utilizando el tipo *double* exactamente de la misma manera que el código de la versión dos.

Generación del código C a partir del código Matlab mediante la *Toolbox Embedded*

Siguiendo el proceso indicado para la primera versión se procedió a realizar la generación de código C a partir del código fuente del VCA. Los pasos de configuración del *Embedded* fueron muy similares a los de la versión anterior de *Fixed-Point*, los únicos cambios realizados fueron el nombre de la función VCA_4_fixcpp.m y por consiguiente el nombre de la salida VCA_4_fixcpp, así como la entrada en formato *uint8*. A continuación se muestra el comando para la generación del código C del *Toolbox Embedded* Matlab.

```
>> emlc -s rtwcfg -d VCA_4_fixcpp VCA_4_fixcpp.m -eg {'uint16',
            emlcoder.egs('uint8', [5 10000])} -report
```

Una vez ejecutado el comando se obtiene un resultado positivo al cabo de unos pocos segundos de ejecución. Como resultado se obtiene un directorio llamado VCA_4_fixcpp en el que se ha generado una serie de archivos .cpp y .h que corresponden al código fuente de C++.

Prueba de funcionamiento y adecuación del código C

Una vez que tenemos el código C generado por el *Embedded* de Matlab lo primero que hacemos es mirar lo que se ha generado para poder probar posteriormente su correcto funcionamiento. El formato por defecto de generación de *Embedded* produce un documento .cpp y .h por cada función utilizada en el código fuente de Matlab. Abriendo el archivo de cabecera podemos ver cuál es el prototipo de la función generada y cuáles y de qué tipo son los parámetros de entrada y salida generados para la utilización de la función. A continuación se presenta el prototipo de función generada.

```
void VCA_4_fixcpp(uint16_T eml_p, char_T eml_y_data[50000], int32_T
eml_y_sizes[2], int32_T eml_indice_data[20], int32_T eml_indice_sizes[2]);
```

Para hacer uso de la librería de *Fixed-Point* de C++ es necesario incluir la fuente de la clase *fixed*.h y *fixed*.cpp. Una vez hecho esto se ha de redefinir el tipo `real_T` que hasta el momento corresponde a un *double* por la clase *fixed*. La redefinición de este tipo se puede realizar de una forma muy sencilla gracias que todo el código depende de la definición de los tipos generados en el archivo rtwtypes.h. Una vez modificados topos los tipos de coma flotante del archivo por la clase *fixed* logramos que el código resultante deje de usar coma flotante para hacer uso de la aritmética de punto fijo. En la figura 23 se muestra el resultado de la redefinición de los tipos.

```
/*=======================================================================*
 * Generic type definitions: real_T, time_T, boolean_T, int_T, uint_T,   *
 *                           ulong_T, char_T and byte_T.                 *
 *=======================================================================*/
// typedef double real_T
typedef fixed real_T;
typedef unsigned char boolean_T;
typedef int int_T;
typedef unsigned uint_T;
typedef unsigned long ulong_T;
// typedef char char_T
typedef unsigned char char_T;
typedef char_T byte_T;
typedef signed char int8_T;
typedef unsigned char uint8_T;
typedef short int16_T;
typedef unsigned short uint16_T;
typedef int int32_T;
typedef unsigned int uint32_T;
// typedef float real32_T
typedef fixed real32_T;
// typedef double real64_T
typedef fixed real64_T;
```

Figura 23. Archivo fuente con los tipos definidos por el *Embedded* C de Matlab

Una vez modificados los tipos se procedió a compilar el proyecto. La compilación produjo múltiples errores relacionados por la utilización de las clausulas *union* que pertenece al lenguaje C y no permite utilizar clases de C++. Para solucionar este problema se cambió la clausula *union* por la de *struct* de C++ que realiza la misma función y realiza el mismo tipo de acceso a las variables. A continuación se presenta un ejemplo de la modificación.

```
//union {
struct {
   LittleEndianIEEEDouble bitVal;
   real_T fltVal;
} tmpVal;
```

Para lograr el correcto funcionamiento se eliminaron las inicializaciones en coma flotante de las variables por inicializaciones en *Fixed-Point*. Dado que no se permite la asignación directa de los valores en coma flotante en la clase *fixed* se fueron eliminando los errores indicados por el compilador. A continuación se indica un ejemplo.

```
// real_T val = 1.0;
real_T val = fixed(1);
```

Realizadas estas modificaciones se volvió a compilar el proyecto de C++ obteniendo una compilación correcta.

Se realizaron los test con las imágenes utilizadas para testear el funcionamiento de las versiones anteriores, pudiéndose constatar que se obtenían los mismos resultados tanto en Matlab como en el código C generado por *Embedded*. Se observó un problema en el uso de la librería y es que se producían *overflow* dado que la clase *fixed* hacía uso de 64 bits para el almacenamiento de la variable, por lo que se tuvo que regular a mano el número de bits dedicados a la parte decimal para evitar el *overflow* y no perder la precisión necesaria para que el algoritmo diera un resultado correcto. El número de bits utilizados en la parte decimal dependió del número de *Endmembers* puesto que a mayor número de *Endmembers* los productos matriciales de mayor tamaño generaban mayores valores con lo que se tuvo que aumentar los bits de la parte entera. Dependiendo de este parámetro el número de bits dedicados a la parte decimal se situó entre 16 y 24 bits dejando los restantes bits de la variable de 64 bits a la parte entera.

Generación del lenguaje de descripción hardware a partir del código C

Una vez que se comprueba que el código C generado por *Embedded* Matlab es correcto se procede a utilizar la herramienta de sintetizado hardware *Catapult* C. El proceso fue muy similar al descrito en versiones anteriores. Los tiempos de ejecución para la generación del código de descripción hardware fue mucho más elevado que el tiempo para la síntesis del algoritmo en coma flotante pero muy inferior al utilizado en el *Fixed-Point* de Matlab, el total del tiempo necesario para la finalización del proceso supero las 12 horas. Finalmente se obtuvo los valores del sintetizado hardware para la FPGA Altera Stratix III necesarios para la comparación de las diferentes versiones.

4.1.5. Versión 5. Algoritmo VCA mejorado que hace uso de aritmética entera

En la quinta y última versión propuesta del sintetizado del algoritmo VCA se han necesitado realizar los siguientes pasos: adecuación del código Matlab, generación del código C a partir del código Matlab mediante la *Toolbox Embedded*, prueba de funcionamiento y adecuación del código C, generación del lenguaje de descripción hardware a partir del código C haciendo uso del programa *Catapult* C. A continuación se describen estos paso con mayor detalle.

Adecuación del código Matlab

Como alternativa del punto fijo se ha implementado el algoritmo haciendo uso de la aritmética entera. Para poder trabajar correctamente con aritmética entera es necesario utilizar enteros de 64 bits para evitar en la medida de lo posible el *overflow*, sin embargo Matlab no proporciona este tipo de dato sino una clase que permite el almacenamiento de enteros de 64 bits pero no permite la utilización de operaciones, por lo que no se ha hecho uso de esta clase. Como no podemos utilizar el entero largo en Matlab se ha optado por utilizar el tipo de datos *double* para redefinir en C++ el tipo *real_T* como *long long int*.

La aritmética entera realizada en el algoritmo ha intentado simular en parte el cálculo del punto fijo, dejando una parte de los bits de la variable a la parte entera y el resto a la parte decimal. Para ello se realizan multiplicaciones y divisiones por una constante de conversión que es un número base dos por lo que estas conversiones se realizan mediante el desplazamiento binario de los datos. En la Figura 24 se muestra la función acondicionada del VCA para el uso en *Embedded* Matlab.

```
function indice = VCA_5_entero( p, y ) %#eml
    assert(p <= 50);
    deci = double(256);
    indice = zeros(1,p,'int32');
    A = zeros(p,p,'double');
    A(p,1) = double(1);
    u = zeros(p,p,'double');
    c2 = zeros(1,p,'double');
    f  = zeros(1,p,'double');
    for i=1:p
         % Parte uno
         w = ones(p,1,'double')*deci;
         if (i == 1)
           f = w;
           f(p) = 0;
         elseif (i == 2)
            f = [ones(p-1,1,'double')*deci;-(deci*sum(A(1:p-1,1))/A(p,1))];
            u(:,1) = A(:,1);
            c2_1 = double(0);
            c2_2 = double(0);
            for ii=1:p
                c2_1 = c2_1 + (u(ii,1));
                c2_2 = c2_2 + ((u(ii,1)*u(ii,1))/deci);
            end
            c2(i-1) = (deci*c2_1)/c2_2;
         else  %%%% i > 2
            c1   = zeros(1,i-2,'double');
            c1_1 = zeros(1,i-2,'double');
            c1_2 = zeros(1,i-2,'double');
            for ii=1:i-2
               for jj=1:p
                  c1_1(ii) = c1_1(ii) + ((u(jj,ii)*A(jj,i-1))/deci);
                  c1_2(ii) = c1_2(ii) + ((u(jj,ii)*u(jj,ii))/deci);
               end
            end
            for ii=1:i-2
               c1(ii) =  (deci*c1_1(ii))/c1_2(ii);
            end
            u_sum = zeros(p,1,'double');
            for ii=1:i-2
               for jj=1:p
                  u_sum(jj) = u_sum(jj) + ((c1(1,ii)*u(jj,ii)/deci));
               end
            end
            for jj=1:p
               u(jj,i-1) = A(jj,i-1) - u_sum(jj);
            end
            c2_1 = double(0);
            c2_2 = double(0);
            for ii=1:p
               c2_1 = c2_1 + u(ii,i-1);
               c2_2 = c2_2 + ((u(ii,i-1)*u(ii,i-1))/deci);
            end
            c2(i-1) = (deci*c2_1)/c2_2;
            f_sum = zeros(p,1,'double');
            for ii=1:i-1
               for jj=1:p
                  f_sum(jj) = f_sum(jj) + ((c2(1,ii)*u(jj,ii)/deci));
               end
            end
            for jj=1:p
               f(jj) = w(jj) - f_sum(jj);
            end
         end
         [nf nc] = size(y);
         v = zeros(1,nc,'double');
         for ii=1:nc
            for jj=1:p
                v(ii) = v(ii) + (f(jj)*double(y(jj,ii)));
            end
         end
         [v_max indice(i)] = max(abs(v));
         A(:,i) = double(y(:,indice(i)))*deci;
     end
end
```

Figura 24. Algoritmo VCA programado en Matlab con aritmética entera

Podemos observar cómo se mantienen las variables en tipo *double* mientras que se realiza la aritmética entera haciendo uso de la variable *deci* que permite simular el funcionamiento del punto fijo.

Generación del código C a partir del código Matlab mediante la *Toolbox Embedded*

Siguiendo el proceso indicado para la primera versión se procedió a realizar la generación de código C a partir del código fuente del VCA. Los pasos de configuración del *Embedded* fueron idénticos que en la primera versión, los cambios realizados fueron el nombre de la función VCA_5_entero.m y por consiguiente el nombre de la salida VCA_5_entero. Además se va a introducir los valores de la matriz de entrada con tipo entero de 8 bits, dado que se va a trabajar únicamente con datos enteros, haremos la suposición de que como la mayoría de los datos de imágenes se introducen en el rango 0-255. A continuación se muestra el comando para la generación del código C del *Toolbox Embedded* Matlab.

```
>> emlc -s rtwcfg -d VCA_5_entero VCA_5_entero.m -eg {'uint16',
            emlcoder.egs('uint8', [5 10000])} -report
```

Una vez ejecutado el comando se obtiene un resultado positivo al cabo de unos pocos segundos de ejecución. Como resultado se obtiene un directorio llamado VCA_5_entero en el que se ha generado una serie de archivos .cpp y .h que corresponden al código fuente de C++.

Prueba de funcionamiento y adecuación del código C

Una vez que tenemos el código C generado por el *Embedded* de Matlab lo primero que hacemos es comprobar lo que se ha generado para poder probar posteriormente su correcto funcionamiento. El formato por defecto de generación de *Embedded* produce un documento .cpp y .h por cada función utilizada en el código fuente de Matlab. Abriendo el archivo de cabecera podemos ver cuál es el prototipo de la función generada y cuáles y de qué tipo son los parámetros de entrada y salida generados para la utilización de la función. A continuación se presenta el prototipo de función generada.

```
void VCA_5_fixcpp(uint16_T eml_p, char_T eml_y_data[50000], int32_T
eml_y_sizes[2], int32_T eml_indice_data[20], int32_T eml_indice_sizes[2]);
```

Para hacer uso de la aritmética entera se ha de redefinir el tipo *real_T* con el tipo de *long long int* con esto se logra utilizar el tipo de entero de 64 bits que nos permitirá eliminar parte del riesgo de *overflow*. Si comprobamos el tipo asignado al entero de 8 bits sin signo vemos que Matlab lo ha hecho corresponder con el *char_T*, este tipo definido en el archivo rtwtypes.h es erróneo puesto que corresponde con el tipo de C++ char (entero de 8 bits con signo). Este hecho producirá un erro en la lectura de los datos de entrada, para solucionarlo se ha de redefinir el tipo como *unsigned char*. Una vez modificados topos los tipos de coma flotante del archivo y eliminado el error del tipo *char_T* podemos continuar con la compilación correcta del código. En la figura 25 se muestra el resultado de la redefinición de los tipos.

```
/*=======================================================================*
 * Generic type definitions: real_T, time_T, boolean_T, int_T, uint_T,   *
 *                           ulong_T, char_T and byte_T.                 *
 *=======================================================================*/

// typedef double real_T
typedef long long int real_T;
typedef unsigned char boolean_T;
typedef int int_T;
typedef unsigned uint_T;
typedef unsigned long ulong_T;
// typedef char char_T
typedef unsigned char char_T;
typedef char_T byte_T;
typedef signed char int8_T;
typedef unsigned char uint8_T;
typedef short int16_T;
typedef unsigned short uint16_T;
typedef int int32_T;
typedef unsigned int uint32_T;
// typedef float real32_T
typedef long long int real32_T;
// typedef double real64_T
typedef long long int real64_T;
```

Figura 25. Archivo fuente donde se definen los tipos de datos generados por *Embedded*

Se observó un problema en el uso de la aritmética entera al producirse *overflow* dado que se hace uso de un entero de 64 bits para el almacenamiento de variables de gran tamaño, por lo que se tuvo que regular el parámetro *deci* que indicaba el número de bits dedicados a la parte decimal. De esta forma se logró evitar el *overflow* y sin perder la precisión necesaria en el algoritmo.

Generación del lenguaje de descripción hardware a partir del código C

Una vez que se comprueba que el código C generado por *Embedded* Matlab es correcto se procede a utilizar la herramienta de sintetizado hardware *Catapult* C. El proceso fue muy

similar al descrito en versiones anteriores. Los tiempos de ejecución para la generación del código de descripción hardware fue similar al tiempo para la síntesis del algoritmo en coma flotante, el total del tiempo necesario para la finalización del proceso supero las 6 horas. Finalmente se obtuvo los valores del sintetizado hardware para la FPGA Altera Stratix III necesarios para la comparación de las diferentes versiones.

4.2. Metodología para la síntesis hardware de algoritmos escritos en Matlab

La metodología que se propone en este TFM para la síntesis hardware de algoritmos escritos en Matlab procede de la experiencia obtenida de las diferentes pruebas y versiones generadas del algoritmo VCA. Esta metodología, aún siendo generada para el algoritmo VCA pretende ser una referencia inicial para la síntesis de algoritmos de cálculos matriciales programados en Matlab.

Para realizar esta metodología se ha de partir de unas presunciones de partida sobre el funcionamiento de las FPGAs y de que tipos de datos son más adecuados para el cómputo en este tipo de dispositivo. Conociendo el funcionamiento genérico de las FPGAs que carecen de unidades de cálculo para coma flotante o *FPU,* y sin tener en cuenta la presencia de posibles módulos especializados para realizar sumas o productos en coma flotante implementados por las diferentes marcas de FPGAs, se parte de la presunción que el cálculo de los algoritmos en aritmética entera o en punto fijo, que sigue siendo aritmética entera, son mucho más eficientes que el cálculo en coma flotante. Dado que el cálculo en punto fijo es aritmética entera que utiliza ciertas funciones de desplazamiento binario para adecuar los datos según las operaciones realizadas, asumimos que el cálculo de la aritmética entera es más eficiente que el cálculo en punto fijo. Como resumen partimos con la presunción de que el uso de la aritmética entera va a proporcionar los mejores resultados tanto en frecuencia como en el uso de recursos hardware, seguido del uso del punto fijo y finalmente el cálculo con coma flotante. Por lo tanto, la metodología va a favorecer, en la medida de lo posible, la aritmética entera, cuando no sea posible la aritmética de punto fijo, y como última opción, el cálculo en coma flotante.

Para tener presente los pasos necesarios en la implementación del algoritmo VCA en las diferentes versiones y tipos se presenta en la tabla 1 con los pasos realizados.

Tabla 1. Proceso de sintetizado hardware para las diferentes versiones del algoritmo VCA

	Acondicionado código Matlab	*Embedded* Matlab	Acondicionado código C++	*Catapult* C / Precision
Versión 1. Código original	Uso cabecera `%#eml` Inicialización estática de variables	Configuración `rtwcfg` Parámetros entrada y salida tipo *double*	Eliminación del math.h Generación del math_fun.h Implementación de las funciones: fabs, sqrt, floor, pow para *double*	Inserción de los archivos .cpp del VCA Selección de FPGA (Stratix, 400 MHz) Generación código VHDL/Verilog y sintetizado
Versión 2. Código mejorado	Uso cabecera `%#eml` Inicialización estática de variables Uso del tipo entero en las variables necesarias Simplificación del algoritmo	Configuración `rtwcfg` Parámetros entrada p *uint16,* matriz de entrada *double* y salida tipo *int32*	Eliminación del math.h Generación del math_fun.h Implementación de la función fabs para *double*	Inserción de los archivos .cpp del VCA Selección de FPGA (Stratix, 400 MHz) Generación código VHDL/Verilog y sintetizado
Versión 3. Código mejorado + *Fixed-Point* Matlab	Uso cabecera `%#eml` Inicialización estática de variables Uso tipo entero Uso tipo *Fixed-Point* **COMPATIBLE** con *Embedded* Simplificación del algoritmo	Configuración `rtwcfg` Parámetros entrada p *uint16,* matriz de entrada *uint8* y salida tipo *int32*	Eliminación del math.h	Inserción de los archivos .cpp del VCA Selección de FPGA (Stratix, 400 MHz) Generación código VHDL/Verilog y sintetizado
Versión 4. Código mejorado + *Fixed* C++	Uso cabecera `%#eml` Inicialización estática de variables Uso de tipo entero Simplificación del algoritmo	Configuración `rtwcfg` Parámetros entrada p *uint16*, matriz de entrada *uint8* y salida tipo *int32*	Incluir la clase *fixed* de C++ redefiniendo el tipo *real_T* como *fixed* Eliminación del math.h Generación del math_fun.h Implementación de la función llabs, para *long long int*	Inserción de los archivos .cpp del VCA Selección de FPGA (Stratix, 400 MHz) Generación código VHDL/Verilog y sintetizado
Versión 5. Código mejorado + aritmética entera	Uso cabecera `%#eml` Inicialización estática de variables Uso de aritmética entera en las variables *double* que van a ser redefinidas en C++ Simplificación del algoritmo	Configuración `rtwcfg` Parámetros entrada p *uint16, matriz de entrada uint8* y salida tipo *int32*	Redefinir el tipo *real_T* como *long long int* Eliminación del math.h Generación del math_fun.h Implementación de la función llabs, para *long long int*	Inserción de los archivos .cpp del VCA Selección de FPGA (Stratix, 400 MHz) Generación código VHDL/Verilog y sintetizado

- La inicialización estática de las variables se refiere a que una vez definida la variable no puede ser cambiado su tamaño en tiempo de ejecución. Esta es una práctica habitual en Matlab pero no es compatible en C/C++ por lo que *Embedded* no lo permite.
- El *Fixed-Point* compatible con *Embedded* es aquel que define de forma estática el tamaño de la variable de punto fijo así como la parte destinada para la parte entera y el decimal. Además ha de definir el modo de suma y producto indicando sus tamaños.

Como podemos observar de la tabla 1, se pueden obtener unas pautas en los procesos de sintetizado de las diferentes versiones del VCA, esto nos permitirá generar una metodología adecuada.

4.2.1. ***Pros y contras del uso de las soluciones propuestas para la síntesis del algoritmo VCA***

Para continuar con el estudio de la metodología hemos de describir los pros y los contras de la utilización de cada una de las soluciones propuestas en las versiones del VCA, dado que según la presunción inicial el uso de la aritmética entera sería la más adecuada.

El uso de la aritmética entera es adecuado en los algoritmos matemáticos que no hacen uso de divisiones u otras funciones que requieren de la precisión de los números reales como las funciones trigonométricas. Para ciertos algoritmos se puede seguir haciendo uso de este tipo de aritmética aún usando divisiones y multiplicaciones de números reales cuando la exactitud no tiene una importancia capital. Para lograr el uso de la aritmética entera en este contexto se implementa el funcionamiento del punto fijo de una manera simplificada realizando las conversiones necesarias mediante productos y divisiones enteras. Esto nos permite implementar muchos algoritmos que solo utilizan sumas, restas, multiplicaciones y divisiones de variables. Además, aunque no siempre se puede implementar todo el algoritmo en aritmética entera se puede utilizar en partes del código que sí son adecuados. La aritmética entera tiene como problema el tamaño máximo de la variable, aún utilizando enteros de 64 bits el tamaño máximo almacenable en ocasiones no es suficiente, agravándose aún más cuando se utiliza el punto fijo simplificado. Por lo tanto se pueden producir *overflow* que producirían errores en el cálculo del algoritmo. Por este motivo se ha de utilizar la aritmética entera en los algoritmos que no generen *overflow* por causa del cálculo con múltiples multiplicaciones. Esta restricción es mucho más importante en el caso del punto fijo pues además de almacenar la parte entera en la variable almacena en forma de entero la parte decimal.

Para el caso del uso de la aritmética de punto fijo de la clase *fixed* de C++, la situación es muy parecida a la de la aritmética entera. La utilización del potencial de C++ a la hora de proporcionar operadores de todo tipo para las sumas, restas, productos, divisiones, comparaciones, etc. permite la utilización del punto fijo en códigos más complejos donde la aritmética entera no puede llegar. Gracias al uso de C++ se logra que el código generado por *Embedded* no tenga que ser modificado para la utilización del punto fijo. Las restricciones de uso son idénticas que en el caso anterior, puesto que el valor en punto fijo se almacena en una variable entera de 64 bits, por lo que existe el riesgo de producirse *overflow* o falta de resolución de la parte decimal. Por lo tanto, esta solución que es casi

tan buena desde el punto de vista de ejecución que la aritmética entera, no se ha de utilizar en algoritmos que puedan generar desbordamientos, siendo indicado en algoritmos en donde no se pueda utilizar la aritmética entera y en donde no se pueda utilizar el *Fixed-Point* de Matlab por problemas en la compatibilidad con las funciones realizadas.

Para el caso de la aritmética en punto flotante de *Fixed-Point* de Matlab se logran mejores resultados en algoritmos con valores numéricos elevados puesto que el *Fixed-Point* permite el uso de variables de hasta 128 bits. Por lo tanto la posibilidad de *overflow* se minimiza. En cambio el código C generado tiene una gran complejidad y realiza una programación a la defensiva (comprobando en cada operación posibles desbordamientos) que provoca que sea mucho más ineficiente que las soluciones anteriores. Además el uso del *Fixed-Point* de Matlab tiene la limitación de que muchas de las funciones matriciales y matemáticas compatibles en *Embedded* no lo son para el uso del *Fixed-Point* por lo que su uso se restringe de forma significativa.

Finalmente si todas las soluciones anteriores no pueden ser aplicadas al algoritmo se procede a utilizar la coma flotante para la obtención del sintetizado hardware. Se presupone que ha de ser el resultado menos satisfactorio pero es la que menos restricciones tiene en su uso. *Embedded,* y en general Matlab tiene la máxima compatibilidad para este tipo de datos, además no hay problemas de desbordamiento ni de ningún otro tipo.

*4.2.2. **Metodología propuesta***

A continuación se presenta la metodología propuesta teniendo en cuenta las presunciones iniciales, los pros y los contras detectados en la implementación de las diferentes versiones del algoritmo VCA, y de las pautas obtenidas en la implementación.

Paso 1. Estudio del algoritmo, detección de las funciones matemáticas utilizadas, simplificación de las funciones que no son fundamentales para el funcionamiento como las inicializaciones *random*, o la normalización de resultados. Una vez estudiado el algoritmo hemos de saber si se puede utilizar la aritmética entera, si es así pasamos al paso 2, de lo contrario pasamos al paso 3.

Paso 2. En el paso dos se realiza el proceso de sintetizado del algoritmo para la aritmética entera, en donde se realizará el acondicionamiento del código Matlab haciendo uso de la aritmética entera, se procede a la generación del código C++ mediante el *Embedded*

Matlab, y se acondiciona el código C++ para el uso del *Catapult* C con lo que se ha de implementar las funciones necesarias en el archivo math_fun. Finalmente se comprueba el correcto funcionamiento del algoritmo en C++ para detectar posibles errores por sobrecargas. Si se detectan estos errores la utilización de la aritmética entera no será adecuada para el algoritmo por lo que se procederá a utilizar el punto fijo. Por lo tanto se pasará al paso número 3. Si el funcionamiento es correcto se procederá al sintetizado hardware mediante el *Catapult* C y la herramienta Precision.

Paso 3. En el paso tres se realiza el proceso de sintetizado del algoritmo para la aritmética de punto fijo de C++. Para ello se realizará el acondicionamiento del código Matlab, se procederá a la generación del código C++ mediante el *Embedded* Matlab, se acondicionará el código C++ para el uso de la librería de punto fijo *fixed* y la implementación de las funciones necesarias en el archivo math_fun. Finalmente se comprobará el correcto funcionamiento del algoritmo en C++ para detectar posibles errores por sobrecargas o por falta de resolución de la parte decimal. Si se detectan estos errores la utilización de la aritmética de punto fijo de C++ no será adecuada para el algoritmo por lo que se procederá a estudiar alternativas en el punto 4. Si el funcionamiento es correcto se procederá al sintetizado hardware mediante el *Catapult* C y la herramienta Precision.

Paso 4. En este punto se procede a realizar el estudio de la compatibilidad de las funciones matemáticas y matriciales del algoritmo con el *Toolbox* de *Fixed-Point* de Matlab. Si todas las funciones son soportadas por *Fixed-Point* se pasa al paso 5, de lo contrario pasamos al paso 6.

Paso 5. En el paso cinco se realiza el proceso de sintetizado del algoritmo para aritmética de punto fijo de Matlab *Fixed-Point*. Se ha de seleccionar el tamaño de la variable punto fijo para almacenar la parte entera y la parte decimal. A continuación se realiza el acondicionamiento del código Matlab con *Fixed-Point*, se genera el código C++ mediante el *Embedded*, se procede a la adecuación del código C++ para su uso en Catuapult y se comprueba que no se producen errores por sobrecarga o por falta de precisión en el *Fixed-Point*. Si ocurre este problema se regresa a la primera parte del proceso aumentando el tamaño de la variable *Fixed-Point* o variando el tamaño de la parte decimal hasta que se elimine el error. Finalmente se procederá al sintetizado hardware mediante el *Catapult* C y la herramienta Precision.

Paso 6. En el paso seis se realiza el proceso de sintetizado del algoritmo para la aritmética de coma flotante. Se ha de acondicionar el código Matlab al uso del *Embedded*, se procederá a generar el código C++, acondicionándolo para su uso en *Catapult* con lo que se implementarán las funciones necesarias en el archivo math_fun. Finalmente se comprueba el buen funcionamiento del código y se procederá al sintetizado hardware mediante el *Catapult* C y la herramienta Precision.

Paso 7. En este último paso se genera el lenguaje de descripción hardware y el posterior sintetizado de este código. El proceso es común para todas las soluciones del algoritmo y sólo variará la cantidad de tiempo necesaria según la complejidad de cada código C++ introducido.

Para finalizar el estudio de la metodología propuesta en este TFM para la síntesis hardware de algoritmos matriciales escritos en Matlab se presenta en la Figura 26 el diagrama de flujo de los pasos a seguir en la metodología propuesta.

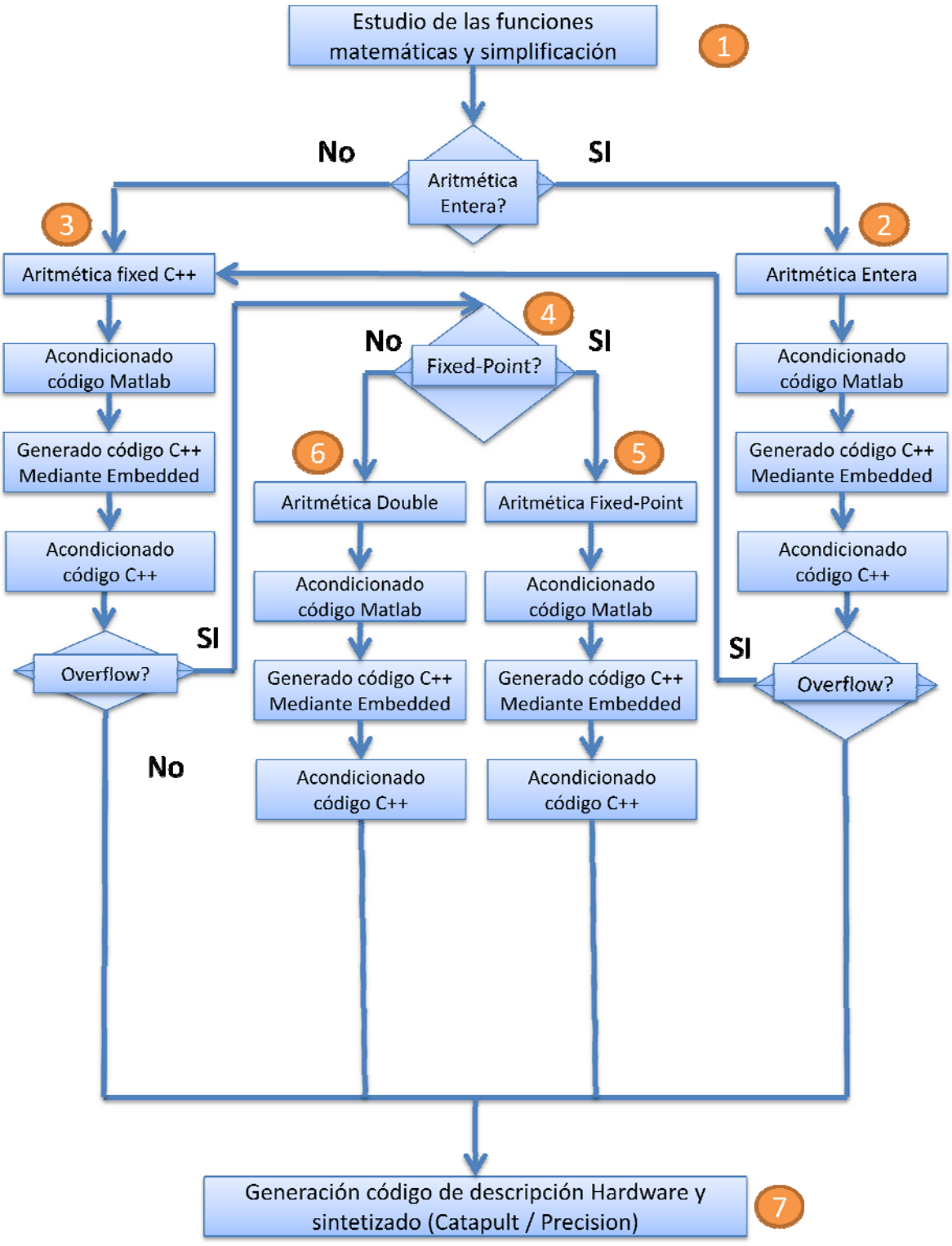


Figura 26. Metodología para la síntesis hardware de funciones escritas en Matlab

5. Resultados obtenidos

En el presente capítulo se van a describir los resultados obtenidos para las diferentes versiones del algoritmo VCA. Se van a presentar resultados del funcionamiento del algoritmo, tanto los tiempos de ejecución del VCA en los lenguajes de alto nivel, así como la bondad de los resultados del cálculo de los *Endmembers* para la implementación realizada en Matlab y C++. Además se van a describir los resultados obtenidos en la síntesis de las versiones de VCA para los recursos hardware utilizados en la síntesis y la frecuencia máxima a la que pueden ejecutarse estos algoritmos.

Teniendo en cuenta que se van a comparar versiones del algoritmo que realizan cálculo entero con otras versiones cuyas entradas son en punto fijo se va a unificar el formato de entrada a entero de 8 bits sin signo para todos los algoritmos, logrando de esta forma almacenar las demos en formato de imagen (tiff) para posteriormente utilizarlas en los algoritmos de C++ permitiéndonos comparar los resultados obtenidos con los de Matlab.

5.1. Testeo de los algoritmos mediante el uso de la demo_vca

Para realizar la medida de la bondad del funcionamiento de los algoritmos implementados en Matlab y posteriormente generados en C++ mediante la *Toolbox Embedded* se va a hacer uso del código de prueba denominado *demo_vca* proporcionado por los autores del algoritmo del VCA original. Entre las opciones de test que proporciona la demo vamos a utilizar dos para validar nuestros resultados. La primera (demo 2) genera una imagen sintética a partir de firmas espectrales definidas mediante la utilización de una distribución de *dirichlet*, con estos datos sintéticos se conocen cuales son exactamente los *Endmembers* que se han de buscar. A esta información se le introduce cierto ruido que típicamente mantiene un SNR igual a 30 dB. Posteriormente se ejecuta el algoritmo VCA para obtener los *Endmembers* comparándose los resultados obtenidos con los generados de forma sintética. Se obtiene un error numérico E_sid que mide el parecido de las firmas espectrales de los *Endmembers* y otros dos angulares que indican el error cometido en el cálculo de abundancias. La segunda (demo 3) utiliza una imagen hiperespectral del sensor AVIRIS llamada *Cuprite* que es ampliamente utilizada para pruebas de este tipo al conocerse la composición del suelo y por lo tanto se puede comparar los *Endmembers* calculados con los datos obtenidos por mediciones in-situ.

El uso de la *demo_vca* permite visualizar de una forma muy intuitiva los resultados obtenidos por el VCA, para le demo 2 se pueden calcular los errores de la firma espectral y de las abundancias gracias a que se conocen los resultados a priori al ser datos sintéticos. La demo permite visualizar la proyección de puntos en un plano generando un perímetro mediante la unión de los *Endmembers* mediante rectas. En la demo 3 se pueden ver las firmas espectrales de los *Endmembers* obtenidos generando imágenes de abundancia. En la Figura 27 se muestra la proyección de valores de los píxeles y los *Endmembers* verdaderos y calculados para el plano que forma el canal 50 y 150. Podemos ver como los *Endmembers* calculados se ajustan en gran medida a los verdaderos.

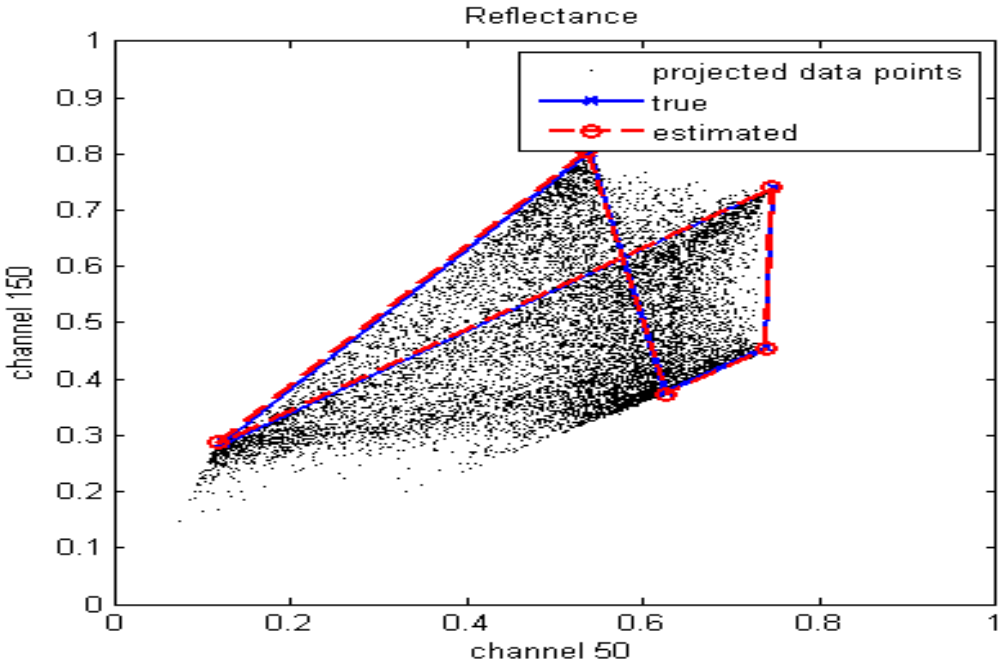


Figura 27. Cálculo de los *Endmembers* y proyección en el plano de los canales 50 y 150

En la Figura 28 se muestran las firmas espectrales de los *Endmembers* verdaderos y de los *Endmembers* calculados por el algoritmo VCA original, siendo los resultados muy similares para el resto de las versiones. También podemos ver las abundancias calculadas a partir de la imagen sintética. Los valores de firma espectral obtenidos de *Endmembers* son muy parecidos a los originales.

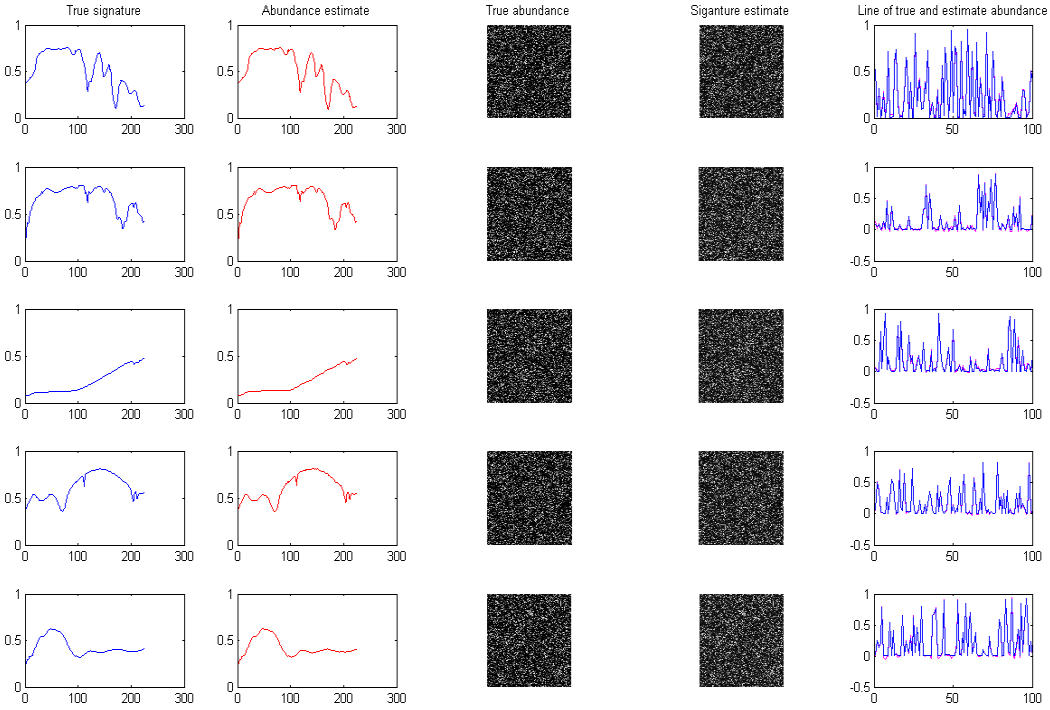


Figura 28. Cálculo de las firmas espectrales y las abundancias de los *Endmembers* calculados

En la Figura 29 se muestra las firmas espectrales calculadas para los 19 *Endmembers* de la imagen *Cuprite*. En la figura 30 se muestran las imágenes con las abundancias de cada uno de los *Endmembers* calculados.

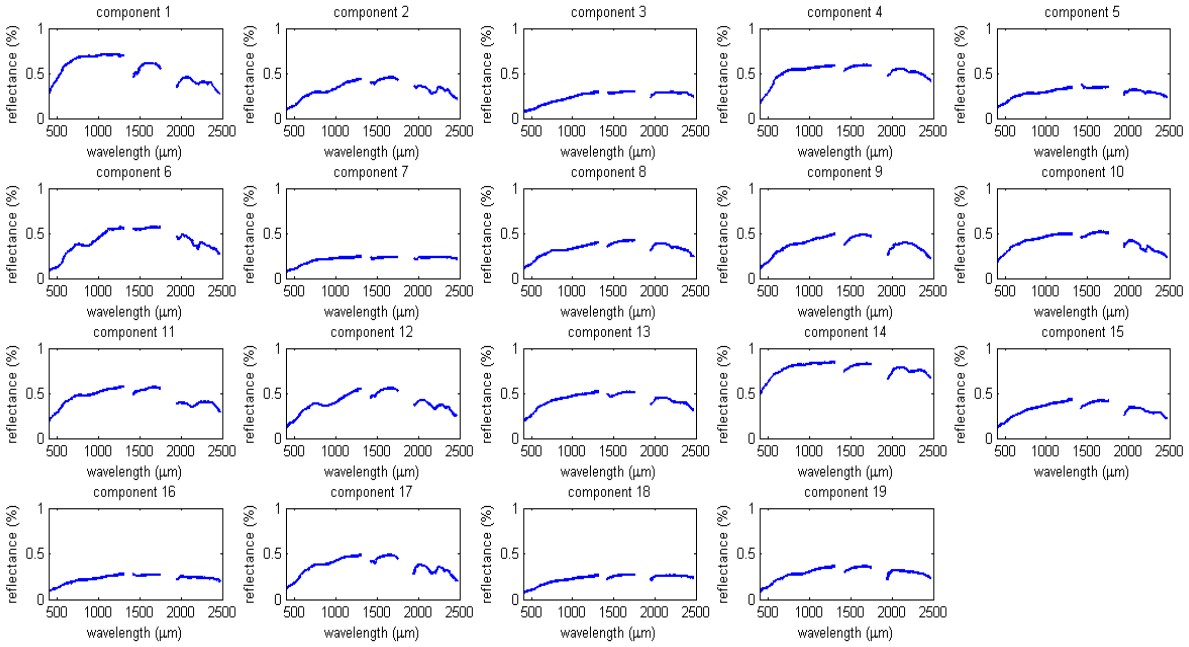


Figura 29. Firmas espectrales de los 19 *Endmembers* calculados de la imagen Cuprite

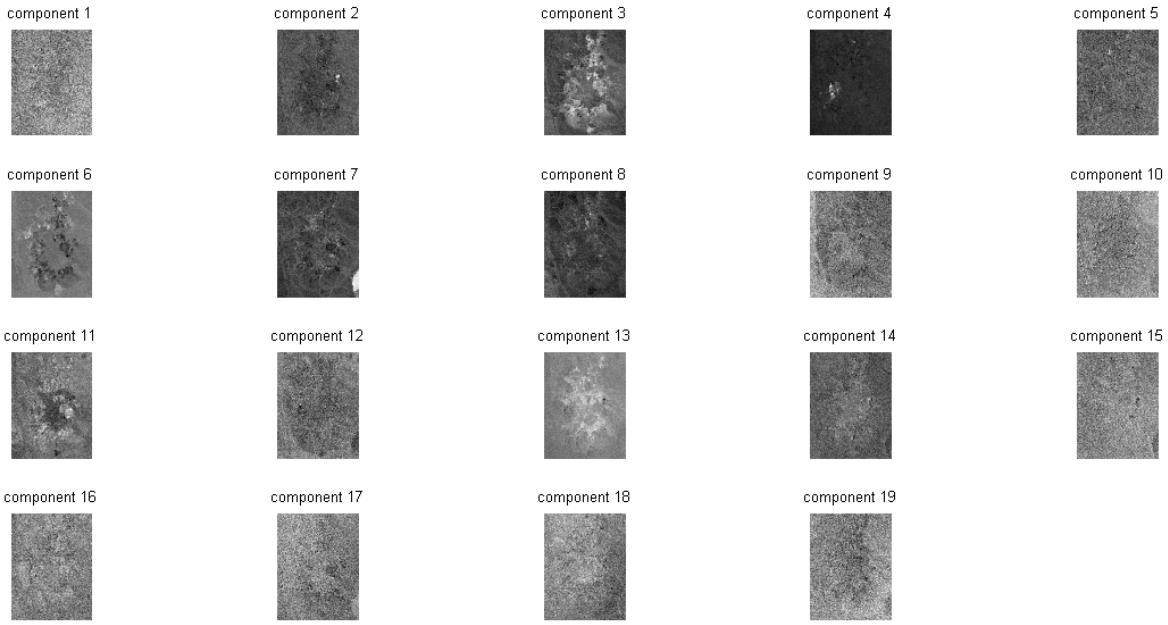


Figura 30. Abundancias de los 19 elementos o *Endmembers* calculados para la imagen Cuprite

5.2. Cálculo de los tiempos de ejecución y los errores de las diferentes versiones del algoritmo VCA

A continuación en la Tabla 2 se muestran los resultados de las mediciones de la demo para una imagen sintéticas de 10000 píxeles y para cálculos de 5, 10 y 15 *Endmembers*. También se incluyen los resultados de la imagen *Cuprite* (250x191) para el cálculo de 19 *Endmembers*.

La columna versión indica la versión del algoritmo a la que le corresponden los datos de la fila. La columna código indica el tipo de código fuente que ha generado el resultado (Matlab o C++). La columna índices indica los valores de índices de los *Endmembers* obtenidos para cada test de 5, 10, 15 *Endmembers*. La columna tiempo indica el tiempo de ejecución de cada test, obtenido de la media de 10 ejecuciones. La columna Errores indica los valores de los errores obtenidos para el parecido de la firma espectral y para el parecido de las abundancias.

Tabla 2. Resultados de la ejecución de las versiones del algoritmo VCA en la demo_VCA

Versión	Código	Índices	Tiempo (10 rep)	Errores Sid / Teta / Beta
Demo 2. *Endmembers* = 5, Líneas = 100 y Columnas = 100				
VCA_1	Matlab	2103 3313 3883 2169 8030	0,0010	0.00185 / 1.22º / 3.7239º
	C++	2103 3313 3883 2169 8030	0,0007	0.00185 / 1.22º / 3.7239º
VCA_2	Matlab	2103 4910 3600 3883 8030	0,0186	0.00185 / 1.22º / 3.7267º
	C++	2103 4910 3600 3883 8030	0,0007	0.00185 / 1.22º / 3.7267º
VCA_3_fix	Matlab	2103 4910 3600 3883 8030	586,32	0.00185 / 1.22º / 3.7196º
	C++	2103 4910 3600 3883 8030	0,042912	0.00185 / 1.22º / 3.7196º
VCA_4_fixcpp	C++	2103 4910 3600 **1165** **2971**	0,00016	**0.0518 / 8.61º / 11.4927º**
VCA_5_entera	C++	2103 4910 3600 3883 8030	0,00268	0.00185 / 1.22º / 3.7267º
Demo 2. *Endmembers* = 10, Líneas = 100 y Columnas = 100				
VCA_1	Matlab	9289 5467 2085 3063 9373 7972 8469 964 3658 9955	0,0027	0.000213/ 0.768º /4.797º
	C++	9289 5467 2085 3063 9373 7972 8469 964 3658 9955	0,00259	0.000213/ 0.768º / 4.797º
VCA_2	Matlab	9289 1737 5467 3060 9373 7972 8469 1590 3658 9955	0,0653	0.00118 / 1.17º / 4.799º
	C++	9289 1737 5467 3060 9373 7972 8469 1590 3658 9955	0,00229	0.00118 / 1.17º / 4.799º
VCA_3_fix	Matlab	9289 1737 5467 3060 9373 7972 8469 1590 3658 9955	2345,9	0.00118 / 1.17º / 4.802º
	C++	9289 1737 5467 3060 9373 7972 8469 1590 3658 9955	0,151598	0.00118 / 1.17º / 4.802º
VCA_4_fixcpp	C++	9289 1737 5467 3060 9373 7972 8469 1590 3658 9955	0,00038	0.00118 / 1.17º / 4.802º
VCA_5_entera	C++	9289 1737 5467 3060 9373 7972 8469 1590 3658 9955	0,00875	0.00118 / 1.17º / 4.802º
Demo 2, *Endmembers* = 15, Líneas = 100 y Columnas = 100				
VCA_1	Matlab	8508 4782 701 6408 8690 218 3128 95 2597 5865 3496 5543 1386 406 8774	0,0048	0.000473/ 0.792º / 4.8321º
	C++	8508 4782 701 6408 8690 218 3128 95 2597 5865 3496 5543 1386 406 8774	0,00585	0.000473 /0.792º /4.8321º
VCA_2	Matlab	8508 2302 2656 1388 6408 218 3128 95 2597 5865 3496 5543 1386 8988 2995	0,1302	0.000519/ 0.834º /4.8317º
	C++	8508 2302 2656 1388 6408 218 3128 95 2597 5865 3496 5543 1386 8988	0,00683	0.000519 / 0.834º /4.8317º

		2995		
VCA_3_fix	Matlab	8508 2302 2656 1388 6408 218 3128 95 2597 5865 3496 5543 1386 8988 2995	5354,1	0.000519/ 0.834º /4.8312º
	C++	8508 2302 2656 1388 6408 218 3128 95 2597 5865 3496 5543 1386 8988 2995	0,35413	0.000519/ 0.834º /4.8312º
VCA_4_fixcpp	C++	8508 2302 2656 1388 6408 218 3128 95 2597 5865 3496 5543 1386 8988 3364	0,00117	0.000519/0.835º /4.8322º
VCA_5_entera	C++	8508 2302 2656 1388 6408 218 3128 95 2597 5865 3496 5543 1386 8988 2995	0,01885	0.000519 /0.834º /4.8317º
Demo 3. *Endmembers* = 19, Líneas = 250 y Columnas = 191				
VCA_1	Matlab	10876 35611 16192 13152 19305 9928 46970 24823 1775 30019 35135 9709 22424 34109 44198 43258 9774 23791 37156	0,0294	---
	C++	10876 35611 16192 13152 19305 9928 46970 24823 1775 30019 35135 9709 22424 34109 44198 43258 9774 23791 37156	0,0332	---
				---
VCA_2	Matlab	10876 32654 15150 13152 19305 34611 47722 46505 16333 26232 1775 25323 30260 13614 33611 23791 9774 5288 45015	0,9566	---
	C++	10876 32654 15150 13152 19305 34611 47722 46505 16333 26232 1775 25323 30260 13614 33611 23791 9774 5288 45015	0,04056	---
				---
VCA_3_fix	Matlab	10876 32654 15150 13152 19305 34611 47722 46505 16333 26232 1775 25323 30260 13614 33611 23791 9774 5288 34611	40631,3	---
	C++	10876 32654 15150 13152 19305 34611 47722 46505 16333 26232 1775 25323 30260 13614 33611 **19938 41501 7168 880**	2,922	---
VCA_4_fixcpp	C++	10876 32654 15150 13152 **47259 34861 19056 26805 45482 46505 21833 31768 22170 33859 29 27744 14645 9887 21674**	0,00642	---
				---
VCA_5_entera	C++	10876 32654 15150 13152 **47259** 34611 **19056** 46505 16333 26232 1775 25323 30260 13614 33611 **5020 35183 10660** 45015	0,138	---

Los datos en rojo indican valores erróneos que son atribuidos a desbordamientos por el uso de la aritmética entera o de punto fijo. Se puede observar que los errores cometidos para los ejemplos de 5, 10, 15 *Endmembers* son mínimos, siendo mucho más visible para la imagen *Cuprite* de 19 *Endmembers*. Esto es debido a que el algoritmo realiza cálculos matriciales con matrices cuyas dimensiones son iguales al número de *Endmembers*, por lo tanto los productos de matrices de gran tamaño pueden producir desbordamientos con mayor facilidad que las matrices pequeñas.

5.2.1. ***Medidas de los tiempos de ejecución de los algoritmos desarrollados***

Vasados en estos resultados se van a mostrar comparativas entre la velocidad de ejecución de las diferentes versiones tanto en Matlab como en C++.

En la Figura 31 se muestra la comparativa entre el tiempo de ejecución del algoritmo original en Matlab y C++ para las imágenes sintéticas de prueba de 100x100 píxeles y 5, 10, 15 *Endmembers* junto a la imagen *Cuprite* de 250x191 píxeles y 19 *Endmembers*. Se puede observar como los resultados son muy semejantes, siendo contra pronóstico más rápida la ejecución del algoritmo en Matlab que en C++, esto se debe a que parte de la ejecución del algoritmo en Matlab se realiza mediante una librería que es ejecutada en código máquina en lugar de ser interpretada como el resto del código de Matlab. Esta función (SVD) es la parte más importante del algoritmo computacionalmente hablando, ya que se utiliza para calcular la pseudo-inversa de la matriz.

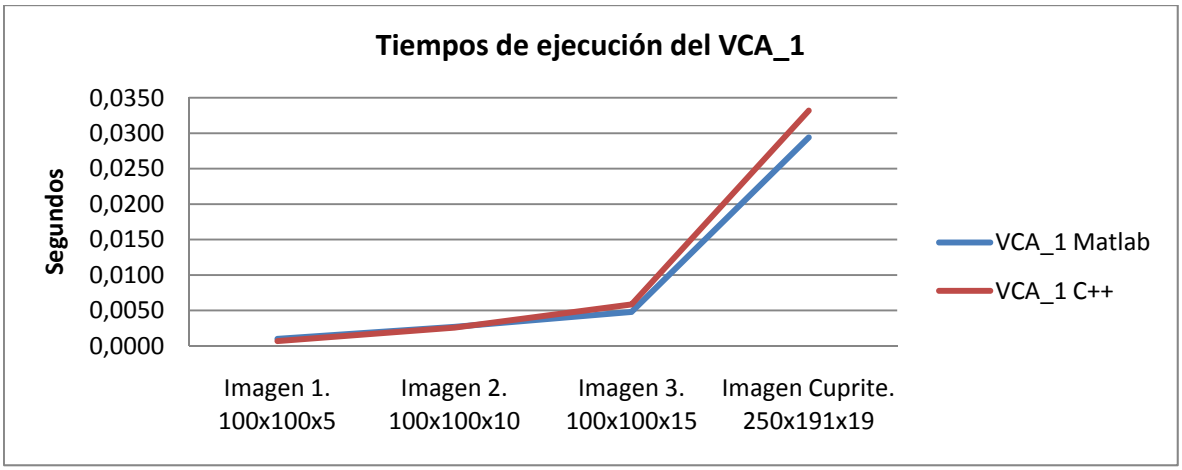


Figura 31. Diferencia entre el tiempo de ejecución del algoritmo original en Matlab y C++

En la Figura 32 podemos observar como en la versión 2 del algoritmo VCA la ejecución de la versión en C++ es mucho más rápida que la versión en Matlab. En esta ocasión todo el código de Matlab es interpretado por lo que se ven grandes diferencias de tiempo de ejecución.

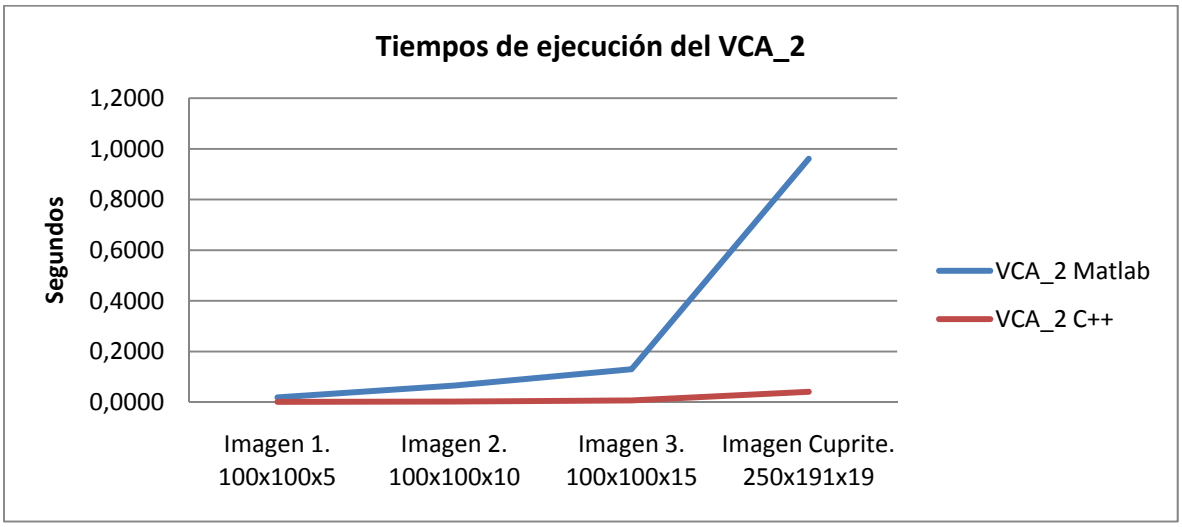


Figura 32. Diferencia entre el tiempo de ejecución de la versión 2 del algoritmo en Matlab y C++

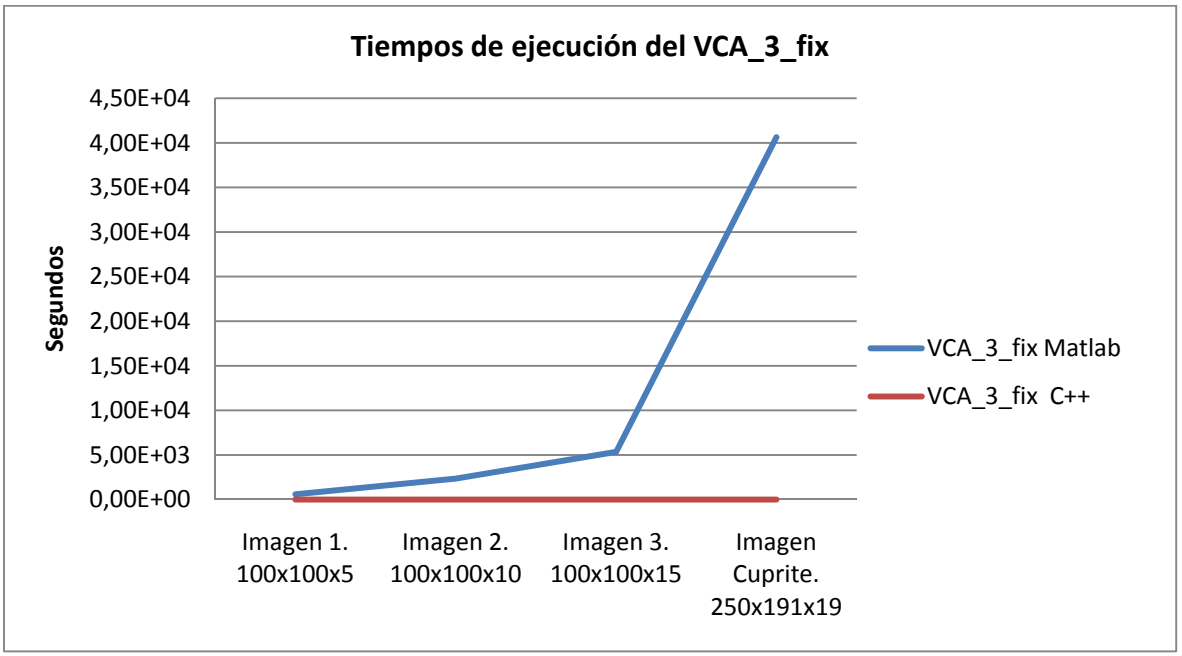


Figura 33. Diferencia entre el tiempo de ejecución de la versión 3 que hace uso de *Fixed-Point* en Matlab y C++

En la Figura 33 se puede ver como en el caso del uso del *Fixed-Point* de Matlab las diferencias se extreman siendo necesarias más de 11 horas para realizar el cálculo más complejo en Matlab mientras que en el caso de C++ este cálculo se realizó en unos 3 segundos.

Para los casos de la implementación del algoritmo en aritmética entera y en punto fijo de C++ podemos ver como los tiempos de ejecución son muy bajos (Figura 34).

En la comparativa de las implementaciones de C++ vemos como los tiempos son mínimos tanto en las versiones en coma flotante como en aritmética entera y punto fijo de C++, siendo más elevados los tiempos de la implementación del *Fixed-Point* de Matlab.

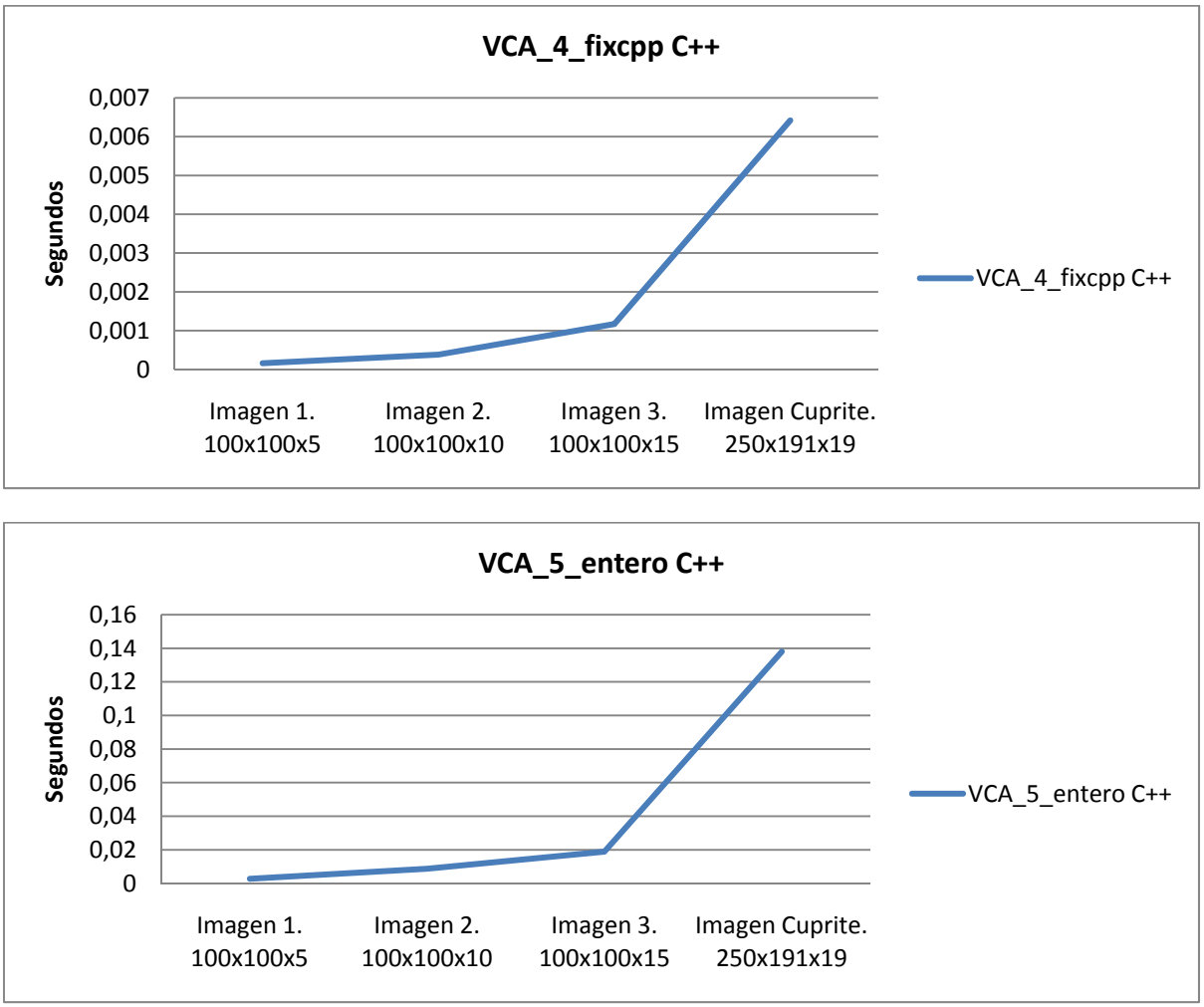


Figura 34. Tiempos de ejecución de la versión entera del algoritmo VCA (abajo) y de la versión en punto fijo de la clase *fixed* (arriba)

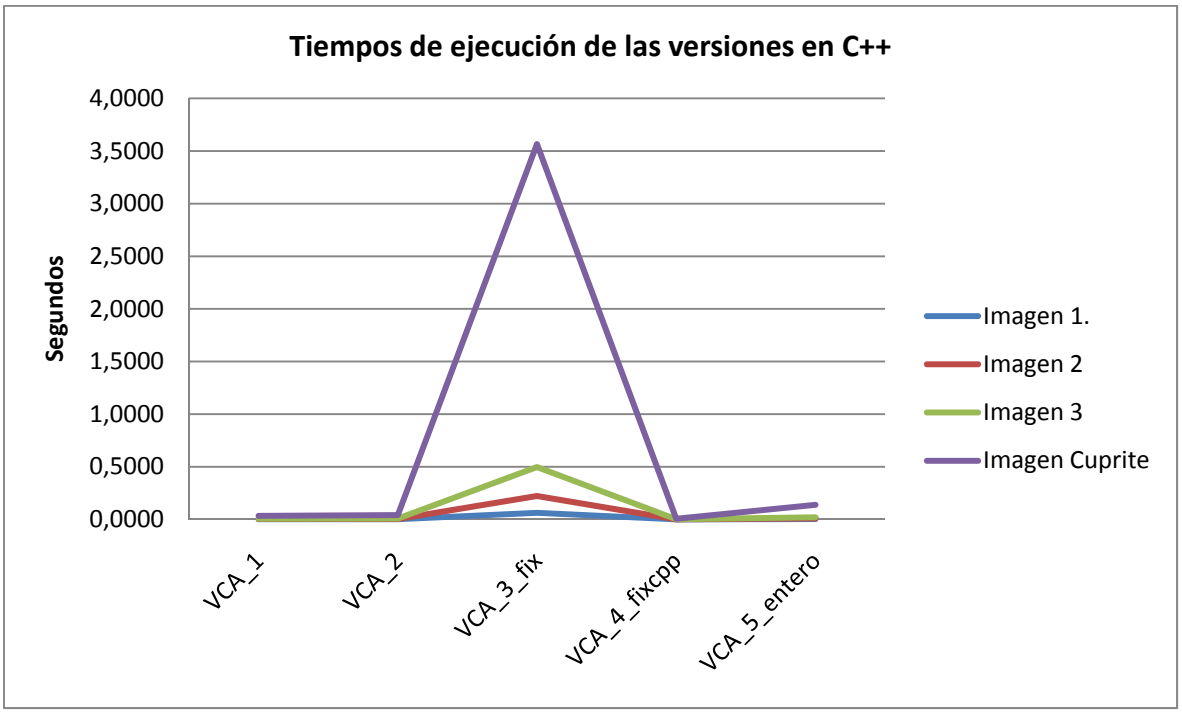


Figura 35. Comparativa de los tiempos de ejecución de las implementaciones del algoritmo VCA en C++

En la figura 35 se muestran los tiempos de ejecución de las cinco versiones generadas en C++. Se puede observar como el uso de coma flotante en los ordenadores permiten obtener gran rendimiento sobre los algoritmos en aritmética entera y punto flotante. Para la síntesis posterior de estos algoritmos el uso de punto fijo ha de mostrarse beneficioso en comparación del uso de la coma flotante.

*5.2.2. **Medidas de error las versiones desarrolladas***

Para realizar las medidas de error de las versiones del algoritmo VCA implementadas se van a utilizar los datos obtenidos de la demo 2, en donde se utilizan imágenes sintéticas a partir de firmas espectrales conocidas, con lo que se conocen a priori los valores de los *Endmembers*. Se ha introducido cierto ruido a los valores de la imagen (SNR = 30 dB) para simular de una forma más real los datos obtenidos por un sensor hiperespectral.

Los ángulos de medida de error permiten conocer el parecido entre las firmas espectrales de los *Endmembers* calculados y de los reales (SID), permite conocer la diferencia entre la fase de las firmas espectrales de los *Endmembers* (Teta), y permite conocer la diferencia entre las abundancias de los *Endmembers* obtenidos (Beta).

A continuación en la Figura 36 se muestra el valor de divergencia de la información espectral de los *Endmembers* (SID) obtenida entre los *Endmembers* obtenidos de las diferentes versiones del algoritmo VCA y los valores reales de los *Endmembers*.

Se puede observar como los errores obtenidos han sido muy bajos, muy similares a los errores obtenidos por el algoritmo VCA original, y se mantienen en todas las implementaciones menos para el caso del ejemplo 100x100x5 en el punto fijo de la clase *fixed*. Para este ejemplo en particular se produce un desbordamiento en el cálculo entero que produce un error importante.

En la Figura 37 se muestra los errores obtenidos para el cálculo Teta mientras que en la Figura 38 se muestra el error obtenido en el ángulo Beta.

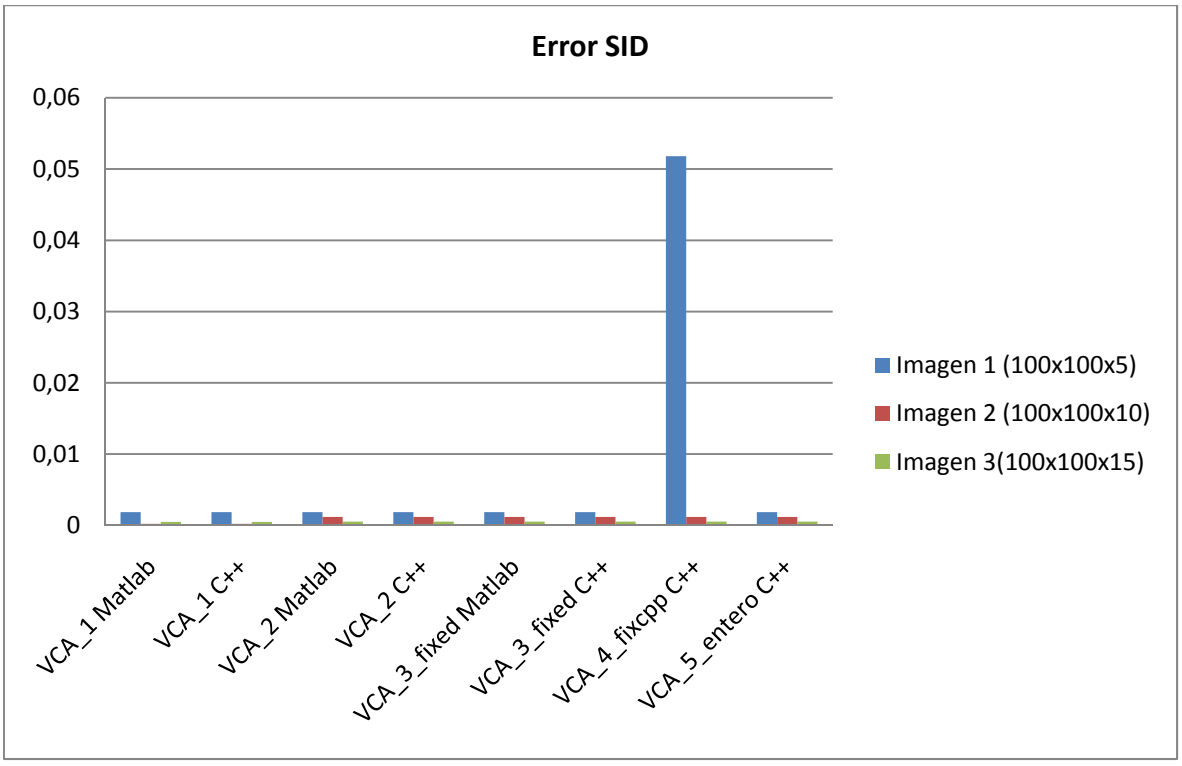


Figura 36. Error SID en el cálculo de los *Endmembers*

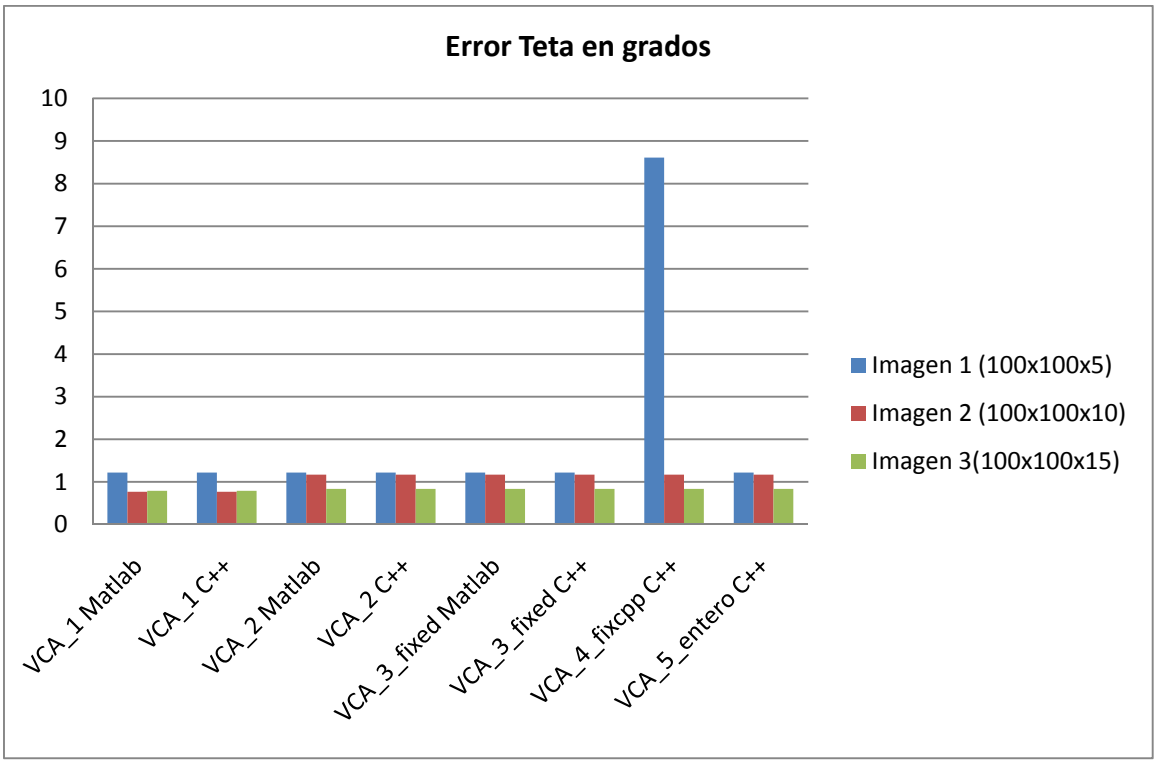


Figura 37. Error del ángulo Teta

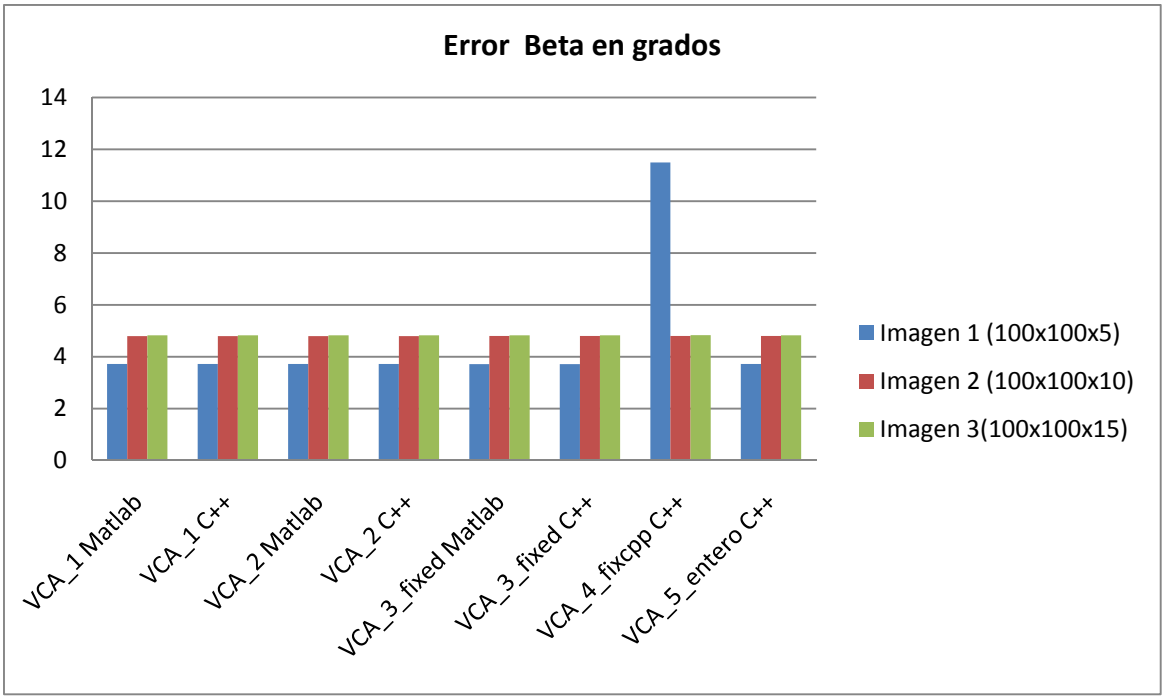


Figura 38. Error ángulo Beta

Podemos observar como los errores de los ángulos Teta y Beta son bajos para todas las implementaciones al igual que los obtenidos en el VCA original, solamente destacar el aumento del error cometido por el VCA de punto fijo en C++ para el cálculo de la imagen de 5 *Endmembers*.

5.3. Resultados del sintetizado de las diferentes versiones del algoritmo VCA

Para realizar el sintetizado hardware y facilitar la generación de un código con variables de tamaños manejables que permita un sintetizado más rápido se ha optado por sintetizar los algoritmos para entradas de 36x36 píxeles con un número máximo de *Endmembers* igual a 5. Se ha seleccionado una FPGA Altera Stratix III para la síntesis hardware del algoritmo VCA. A continuación se van a exponer los resultados sobre la frecuencia máxima de utilización de la FPGA y de los recursos hardware utilizado. En la tabla 3 se muestran los resultados obtenidos para las diferentes versiones del algoritmo.

Tabla 3. Frecuencias máximas y áreas obtenidas en la síntesis de las diferentes versiones del VCA

Versión Algoritmo	Mínimo Periodo y Frecuencia	Ciclos de latencia	Tiempo de ejecución	Área utilizada
VCA Original	179,85 us /5,56 MHz	39.850.048	7167 ms	IOs *** LUTs *** Registers *** Memory Bits *** DSP block 18-bit elems *
VCA Mejorada	5,598 us / 178,635 MHz	230	0,00128 ms	IOs 370 LUTs 36101 Registers 18881 Memory Bits 87552 DSP block 18-bit elems 9
VCA *Fixed-Point*	5,804 us / 172,295 MHz	1.574.746	9,139824 ms	IOs 370 LUTs 114324 Registers 65952 Memory Bits 179904 DSP block 18-bit elems 10
VCA *fixed* C++	9.738 us / 102.690 MHz	168	0,00163 ms	IOs 370 LUTs 70983 Registers 37701 Memory Bits 175104 DSP block 18-bit elems 13
VCA aritmética entera	11,084 us / 90,220 MHz	393.534	4,361937	IOs 370 LUTs 70511 Registers 39058 Memory Bits 177799 DSP block 18-bit elems 25

Tabla 4. Datos de rendimiento obtenidos del sintetizado del *Catapult* C

	Latency Cycles	Freq MHz	Run Time (ms)	Total Area
VCA Original	39850048	5,560000	7167,274820	333.668
VCA Mejorada	230	178,635000	0,001288	46.214
VCA Fixed-Point	1.574.746	172,295000	9,139824	144.875
VCA *fixed* C++	168	102,690000	0,00163	105.420
VCA aritmética entera	393.534	90,220000	4,361937	92.827

En la tabla 4 se muestran los datos de rendimiento obtenidos de la síntesis hardware de las diferentes versiones de VCA. La primera columna indica el número de ciclos de reloj necesarios para obtener una salida de datos desde que se introdujo la entrada. La segunda columna representa la frecuencia máxima de utilización del dispositivo. La tercera columna representa el tiempo de ejecución teniendo en cuenta la latencia y la frecuencia de funcionamiento del dispositivo. La última columna representa el área utilizada en la síntesis de los algoritmos. En la figura 39 se muestran las frecuencias máximas de funcionamiento de cada síntesis de la versión.

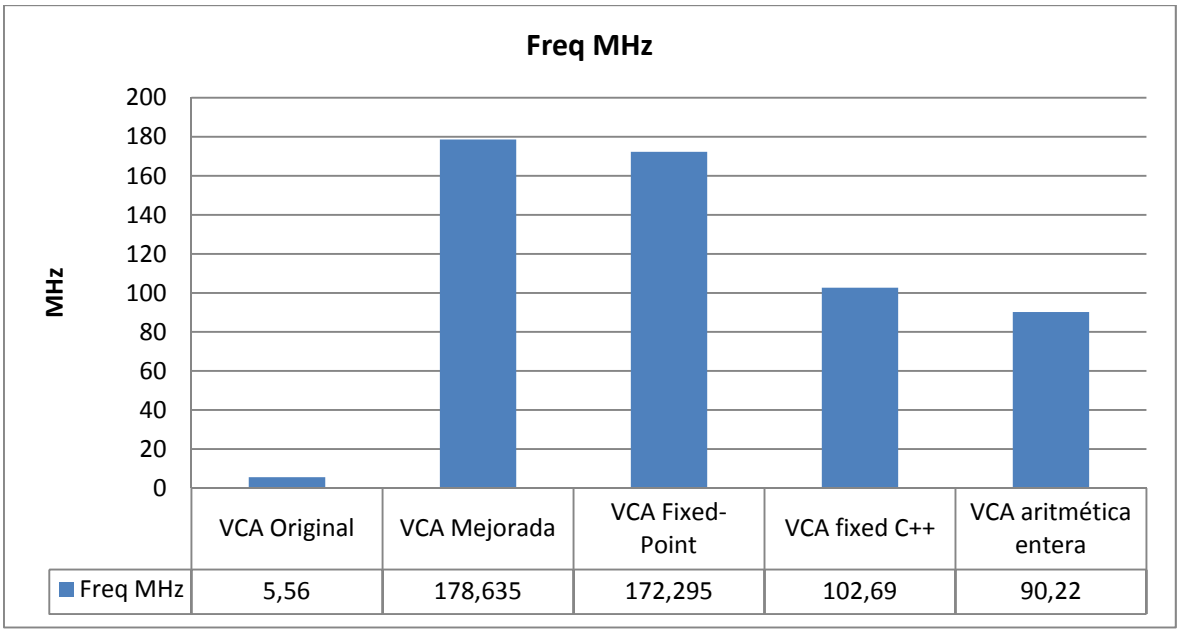


Figura 39. Frecuencia máxima de funcionamiento obtenida para cada versión del VCA

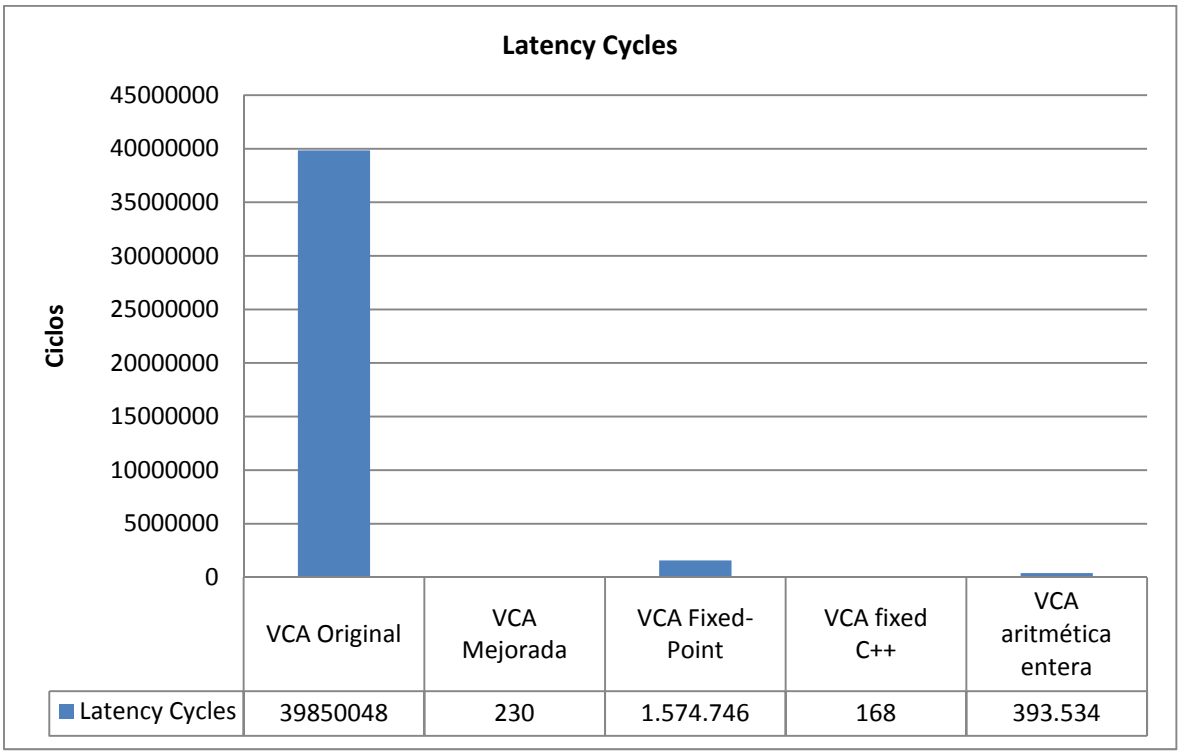


Figura 40. Latencia de las implementaciones hardware de los algoritmos

En la Figura 40 se muestra la latencia de cada versión del algoritmo VCA sintetizado. En la figura 41 se muestra el área generada por la síntesis hardware de las diferentes versiones.

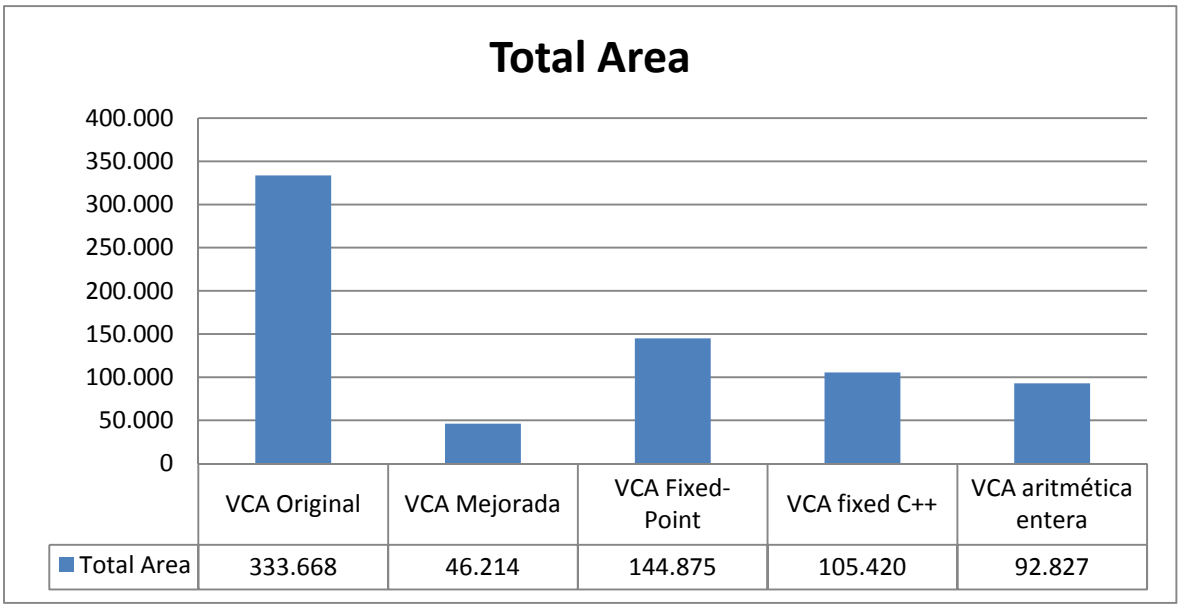


Figura 41. Área generada en la síntesis hardware de los algoritmos

Como se desprende de los resultados obtenidos la versión uno del algoritmo que ejecuta la pseudo-inversa de la matriz es la síntesis más complejo con la mayor área por lo que además genera una latencia muy importante en su ejecución y una muy pobre frecuencia. Por lo tanto el tiempo de ejecución tiene un valor exageradamente alto.

Los resultados del algoritmo modificado han sido sorprendentemente buenos, la frecuencia de funcionamiento y la latencia, gracias a una buena paralelización del hardware, son excelentes. La versión modificada del algoritmo mantenía el uso de los tipos de datos en coma flotante sin embargo las FPGAs carecen de una unidad FPU por lo que parece que el propio sintetizador del *Catapult* C ha realizado la conversión de coma flotante a punto fijo. Altera es capaz de sintetizar bloques de punto flotante (*Block-Floating-Point*, BFP) obteniendo buenos resultados de precisión flotante y de velocidad de ejecución [19]. Estos bloques son utilizados en implementaciones hardware de FFT/IFFT. Gracias a estos módulos vemos que la eficiencia del código generado es mejor que las soluciones propuestas de punto fijo de Matlab y de C++.

Los resultados obtenidos para el punto fijo de C++ son muy buenos con valores de latencia y frecuencia cercanos al del algoritmo modificado. Podemos observar que el tiempo de ejecución ha aumentado un poco. Respecto al área utilizada vemos como en esta ocasión este sintetizado duplica el área utilizada en la versión modificada.

Los resultados que se desprenden de la versión en aritmética entera son peores de lo esperado. La implementación de este algoritmo es muy similar al de punto fijo implementado en C++, puesto que se realizan las conversiones necesarias para las operaciones en punto fijo igual que la versión anterior. El problema puede venir del número de multiplicadores DSPs utilizados, parece que las conversiones para el punto fijo que son multiplicaciones en base dos, que han de ser implementadas por desplazamientos binarios, se han resuelto mediante el uso de estos módulos multiplicadores. Podemos ver que el uso de DSPs se ha disparado (25 módulos) respecto a los DSPs utilizados en la versión modificada del algoritmo (9 módulos). Este posible problema ha multiplicado por mil la latencia del algoritmo, disminuyendo a su vez la frecuencia de funcionamiento por lo que el resultado de tiempo de ejecución es visiblemente peor que las soluciones anteriores del VCA mejorado y el VCA con *Fixed-Point* de C++.

Los resultados obtenidos para el *Fixed-Point* de Matlab son pobres, la latencia del sintetizado es muy elevada similar al producido por la primera versión del algoritmo, a pesar de la alta frecuencia obtenida, por lo que el tiempo de ejecución es muy elevado.

Por lo tanto los resultados obtenidos son bastante prometedores en cuanto a la alta frecuencia obtenida y la baja latencia para las versiones del VCA mejorado y el VCA de punto fijo que hace uso de la librería de C++ *fixed*. Para el caso de la aritmética entera parece existir un error en su implementación que puede ser depurado. Para finalizar la aritmética en punto fijo de Matlab proporciona un sintetizado con una alta frecuencia que queda ensombrecida por su altísima latencia.

6. Conclusiones

El objetivo marcado en el TFM ha sido desarrollar una metodología para la síntesis hardware del algoritmo VCA implementado en Matlab. Para ello se han realizado diferentes versiones con la intención de mejorar los resultados en la implementación. La utilización de una metodología para la síntesis de algoritmos desarrollados en Matlab ha permitido mejorar el tiempo de diseño para la síntesis hardware en comparación con el tiempo de diseño obtenido mediante los lenguajes de descripción de hardware como Verilog o VHDL.

Los resultados de los algoritmos generados desde el punto de vista de funcionamiento han sido muy buenos, siendo validados gracias a la utilización de un entorno de validación desarrollado en Matlab.

Los resultados del sintetizado hardware obtenidos por el algoritmo han sido prometedores puesto que se han obtenido frecuencias de funcionamiento bastante altas para este tipo de dispositivos superiores a 178 MHz, valores cercanos a implementaciones ad-hoc en lenguajes de descripción RTL para este mismo algoritmo con parámetros semejantes. La

frecuencia de utilización en realizaciones ad-hoc de estos algoritmos es de unos 200–250 MHz [15].

Las presunciones iniciales con la que partimos no fueron completamente correctas, no se contó con la alta eficiencia de la herramienta *Catapult* C en sintetizar los tipos de datos en coma flotante. Siendo *Catapult* C el que mejor ha implementado el punto fijo de todas las versiones.

En cualquier caso se han sintetizado versiones del VCA que hacen uso de aritmética de punto fijo de una manera muy eficiente como la obtenida por la librería de C++ *fixed*. En general podemos decir que los resultados obtenidos mediante la metodología expuesta en el TFM han permitido obtener buenos resultados en un tiempo de implementación muy reducido.

La realización de la metodología para la síntesis a partir de código Matlab ha resultado ser más compleja de lo previsto. Si bien haciendo uso del código original se puede obtener su síntesis hardware de una manera meridianamente sencilla los resultados obtenidos son pobres. Sin embargo para mejorar estos resultados se ha de estudiar el algoritmo en alto nivel para lograr simplificarlo y adecuarlo lo máximo posible a los requerimientos del *Embedded* Matlab. La utilización de la metodología aplicada al algoritmo VCA ha permitido mejorar en gran medida los resultados.

Uno de los mayores problemas encontrados en el TFM ha sido los tiempos de sintetizado a partir del código C/C++ necesarios en la ejecución de la herramienta software *Catapult*/ Precision. Estos tiempos en algunos casos han sido superiores a 4 días.

Existen varias líneas futuras que quedan abiertas en este TFM. La primera y principal es la mejora y la estandarización de la metodología de sintetizado de código Matlab para todo tipo de algoritmos matemáticos y de tratamientos de imágenes.

Una segunda línea puede ser la prueba en diferentes dispositivos FPGAs del sintetizado del código VCA en Matlab, para identificar que dispositivos están mejor diseñados para la síntesis hardware a partir de lenguajes de alto nivel como Matlab.

La tercera línea puede ser continuar la síntesis hardware a partir de los algoritmos escritos en Matlab de toda la cadena de procesamiento de las imágenes hiperespectrales, realizando la metodología propuesta.

7. Bibliografía

Para finalizar se presenta la bibliografía utilizada para la realización del TFM.

1. **Hackwell JA, Warren DW, Bongiovi RP, Hansel SJ, Hayhurst TL, Mabry DJ, Sivjee MG, Skinner JW.** *LWIR/MWIR imaging hyperspectral sensor for airborne and ground-based remote sensing.* s.l. : Hackwell, JA (reprint author), AEROSP CORP, SPACE & ENVIRONM TECHNOL CTR, POB 92957, M2-251, LOS ANGELES, CA 90009 USA , 1996.

2. **Doxaran D, Froidefond JM, Lavender S, Castaing P.** *Spectral signature of highly turbid waters - Application with SPOT data to quantify suspended particulate matter concentrations.* Talence, France : ELSEVIER SCIENCE INC, 360 PARK AVE SOUTH, NEW YORK, NY 10010-1710 USA , 2002.

3. **ADAMS JB, SABOL DE, KAPOS V, ALMEIDA R, ROBERTS DA, SMITH MO, GILLESPIE AR.** *CLASSIFICATION OF MULTISPECTRAL IMAGES BASED ON FRACTIONS OF ENDMEMBERS - APPLICATION TO LAND-COVER CHANGE IN THE*

BRAZILIAN AMAZON. CAMBRIDGE, ENGLAND : ADAMS, JB (reprint author), UNIV WASHINGTON, DEPT GEOL SCI, SEATTLE, WA 98195 USA , 1995.

4. **Nascimento JMP, Dias JMB.** *Vertex component analysis: A fast algorithm to unmix hyperspectral data.* Lisbon, Portugal : IEEE-INST ELECTRICAL ELECTRONICS ENGINEERS INC, 445 HOES LANE, PISCATAWAY, NJ 08855 USA , 2005.

5. Mathworks. [En línea] [Citado el: 16 de 06 de 2011.] http://www.mathworks.com/products/matlab/.

6. *VHDL-AMS and Verilog-AMS as alternative hardware description languages for efficient modeling of multidiscipline systems.* **Pecheux F, Lallement C, Vachoux A.** Paris, France : IEEE-INST ELECTRICAL ELECTRONICS ENGINEERS INC, 445 HOES LANE, PISCATAWAY, NJ 08855 USA , 2005.

7. From MATLAB to Embedded C. [En línea] [Citado el: 16 de 06 de 2011.] http://www.mathworks.com/company/newsletters/news_notes/2009/matlab-embedded-c.html.

8. Fixed-Point Toolbox . [En línea] [Citado el: 16 de 06 de 2011.] http://www.mathworks.com/products/fixed/.

9. Libfixmath. [En línea] [Citado el: 16 de 06 de 2011.] http://code.google.com/p/libfixmath/.

10. Electronic System Level Design. [En línea] [Citado el: 16 de 06 de 2011.] http://www.mentor.com/esl/catapult/overview.

11. *Impact of Initialization on Design of Endmember Extraction Algorithms.* **Plaza, A., Chang, C.** 3397-3407, s.l. : IEEE Transactions on Geoscience and Remote Sensing, 2006.

12. *Pirex purity index-dased algorithms for endmember extraction from hyperspectral imagery.* **F. Chaudhry, C Wu, W. Liu, C. Chang, A. Plaza.** s.l. : Transworld Research Network, 2006, Vols. pp.30-62.

13. *A quantitative and comparative analysis of different implementations of N-FINDR: A fast endmember extraction algorithm.* **M. Zortea, A. Plaza.** pp. 787-791, s.l. : IEEE Geoscience and Remote Sensing Letters, 2009.

14. *Vertex component analysis: a fast algorithm to unmix hyperspectral data.* **J. Nascimento, J. Bioucas.** pp. 898-910, s.l. : IEEE Transactions on Geoscience and Remote Sensing, 2005.

15. **Pablo Hostrans Andaluz, Roberto Sarmiento Rodríguez, Sebastián López Suárez.** *Implementación hardware del algoritmo Vertex Component Analysis (VCA) para el procesamiento de imágenes hiperespectrales.* Las Palmas de G.C. : s.n., 2011.

16. Embedded Matlab Function Library Reference. [En línea] [Citado el: 1 de 6 de 2011.] http://www.kxcad.net/cae_MATLAB/toolbox/eml/ug/bq1h2z7-9.html.

17. **Erdelsky, Philip J.** A Fixed-Point Arithmetic Package. [En línea] [Citado el: 1 de 06 de 2011.] http://www.efgh.com/software/fixed.htm.

18. Catapult C Synthesis. [En línea] 1 de 06 de 2011. http://www.mentor.com/esl/catapult/overview.

19. FFT/IFFT Block Floating Point Scaling. [En línea] [Citado el: 1 de 06 de 2011.] http://www.altera.com/literature/an/an404.pdf.