UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

Grado en Ingeniería de Tecnologías y Servicios de Telecomunicación

TRABAJO FIN DE GRADO

ALGORITMO DE COMPRESIÓN DE BAJA LATENCIA EN FPGA USANDO SDSOC

Ángel López García-Arias
Tutor: Gustavo Sutter Capristo

JULIO 2017

ALGORITMO DE COMPRESIÓN DE BAJA LATENCIA EN FPGA USANDO SDSOC

AUTOR: Ángel López García-Arias
TUTOR: Gustavo Sutter Capristo

High Performance Computing and Networking (HPCN-UAM)
http://www.hpcn-uam.es/
Dpto. de Tecnología Electrónica y de las Comunicaciones
Escuela Politécnica Superior
Universidad Autónoma de Madrid
Julio de 2017

Resumen

Tras dos décadas de existencia, la síntesis de alto nivel (HLS) sigue sin cruzar la brecha de la adopción tecnológica. Sin embargo, un boom de tecnologías computacionalmente costosas, como la visión artificial o el aprendizaje automático, está haciendo presión para que el proceso de desarrollo de hardware se acelere. Para esto, es necesario elevar el nivel de abstracción, y fabricantes como *Xilinx* están lanzando nuevas herramientas con este objetivo.

La nueva herramienta de diseño de sistemas empotrados *SDSoC* de *Xilinx* permite crear coprocesadores hardware para los procesadores ARM presentes en las familias de dispositivos *Zynq* y *Zynq UltraScale+*, generando sistemas empotrados híbridos CPU- FPGA para la aceleración de algoritmos. *SDSoC* va un paso más allá de *Vivado HLS*, ofreciendo un flujo de diseño similar al desarrollo de software que permite la generación de este tipo de sistemas a partir de descripciones de software

Por otro lado, el creciente mercado de drones se está viendo obligado a hacer uso de tecnologías de vídeo analógico para la conducción remota basada en vídeo, debido a que la latencia de los codificadores de video digitales imposibilita la conducción a altas velocidades.

En este Trabajo de Fin de Grado se evalúa el potencial de la herramienta *SDSoC* para el diseño de aceleradores hardware a partir de especificaciones C/C++. Concretamente se analizará su uso para la implementación de aplicaciones de procesamiento de vídeo. Se estudiará la posibilidad de implementar un sistema de compresión de imagen con una latencia lo suficientemente baja como para poder superar la frecuencia de refresco de los sistemas PAL/NTSC. Para ello se tratará de sintetizar un diseño basado en un nuevo algoritmo de compresión con pérdidas, LHE (*Logarithmical Hop Encoding*), que ofrece un costo computacional reducido al no operar en el dominio de la frecuencia.

Partiendo de una implementación ineficiente del algoritmo LHE en Python, se ha realizado una traducción a lenguaje C, una serie de optimizaciones, y finalmente una serie de transformaciones hasta lograr con *SDSoC* un diseño sintetizable, implementable y acelerado. El diseño se ha realizado para la plataforma *ZynqBerry*, basada en un *Zynq XC7Z010*.

Palabras clave

SDSoC, ZynqBerry, LHE, Compresión de imagen, Baja Latencia, Aceleración de algoritmos, Sistemas embebidos, Sistemas híbridos, Zynq, ARM, Síntesis de alto nivel, Automatización de diseño electrónico, FPGA, Xilinx, Vivado HLS, Preprocesamiento de imagen, drones, OpenCV.

Abstract

After two decades of existence, High Level Synthesis (HLS) has still not crossed the chasm of technology adoption. However, a boom of computationally expensive technologies, such as computer vision or machine learning, is putting pressure on hardware designers to improve the productivity. To achieve this, it is necessary to raise the abstraction level, and manufacturers like *Xilinx* are releasing new tools oriented to this goal.

Xilinx's new embedded system design tool *SDSoC* allows the design of hardware coprocessors for the ARM cores found in *Zynq* and *Zynq UltraScale+* SoC families, generating hybrid CPU – FPGA embedded systems for algorithm acceleration. *SDSoC* goes a step further than *Vivado HLS*, offering a design workflow similar to the one of software development that allows the generation of these systems from software descriptions.

Besides, the growing market of video-enabled drones is having to rely on old-school analog systems for video based remote operation, since the latency of digital video codecs do not allow high-speed piloting. Among many other implications, this means a huge waste of bandwidth.

This Graduation Project will try to evaluate the potential of *SDSoC* for the design of hardware accelerators from C/C++ specifications, specifically for the implementation of video processing applications. The feasibility of implementing an image compression system with a latency low enough to surpass the frame rate of PAL/NTSC technologies and its implications will be studied. This will be explored through the synthesis of a design based on the novel lossy compression algorithm LHE (Logarithmical Hop Encoding), which promises reduced computational cost by not operating on the frequency domain.

Starting from an inefficient implementation of the LHE algorithm in Python, a translation to C language will be performed. After multiple optimizations, several refactorizations will be applied to the code towards achieving via *SDSoC* a synthesizable, implementable and accelerated system. This design has been done for the *Zynq XC7Z010* based platform *ZynqBerry*.

Keywords

SDSoC, ZynqBerry, LHE, Image compression, Low latency, Algorithm acceleration, Embedded systems, Hibrid systems, Zynq, ARM, High Level Synthesis, Electronic Design Automation, FPGA, Xilinx, Vivado HLS, Image preprocessing, Video-enabled drones, OpenCV.

Agradecimientos

A mi tutor, Gustavo Sutter Capristo, por darme la oportunidad de trabajar con él y su apoyo en este y otros proyectos.

A Mario Ruiz Noguera y Tobías Alonso Pugliese, por su imprescindible ayuda y experiencia durante el desarrollo de este trabajo.

A mi amigo Eduardo Ramos Rodríguez, por llevarme en coche a la biblioteca durante años.

A mis padres, por su apoyo durante la carrera.

Muchísimas gracias.

ÍNDICE DE CONTENIDOS

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

ÍNDICE DE ECUACIONES

1 INTRODUCCIÓN

1.1 MOTIVACIÓN

A diferencia de la evolución que ha experimentado el desarrollo de software, los desarrolladores de hardware han sido tradicionalmente reacios a elevar el nivel de abstracción en el que trabajan, razón por la cual el desarrollo de hardware sigue siendo a día de hoy un proceso costoso en tiempo y recursos. Para abordar este problema ha surgido la síntesis de alto nivel (High Level Synthesis), pero tras dos décadas de existencia sigue sin experimentar una adopción mayoritaria. En cualquier otro ámbito se consideraría que esta tecnología no ha conseguido cruzar la brecha de la adopción tecnológica y que, por tanto, ha fracasado, pero a la vez que existe este escepticismo hacia la automatización del diseño, sigue creciendo inevitablemente una necesidad por la optimización de la productividad del proceso de diseño y la reducción del Time To Market (TTM). Esta tensión está tratando de ser solucionada por fabricantes como Xilinx, que no deja de lanzar al mercado nuevo software de diseño que trata de elevar el nivel de abstracción, incluyendo la herramienta Vivado HLS, con la que ha cosechado considerable éxito. Este Trabajo de Fin de Grado trata de hacer una aportación al proceso de adopción de metodologías de diseño de hardware más avanzadas, analizando la herramienta de diseño de sistemas empotrados SDSoC de Xilinx, que va incluso un paso más allá del HLS.

Este Trabajo de Fin de Grado se ve además inspirado por el proyecto *Racing Drones* [6] , en el que participa el laboratorio *High Performance Computing and Networking* de la *Universidad Autónoma de Madrid* (*HPCN-UAM*) en consorcio con *ALCATEL-LUCENT ESPAÑA S.A., INNOVATI SERVICIOS TECNOLÓGICOS S.L.* y la *Universidad Politécnica de Madrid*. El objetivo de este proyecto es la creación de un videojuego online multijugador basado en la conducción remota a través de vídeo FPV (First Person View) de drones terrestres de carreras reales. Actualmente, para la operación basada en vídeo de drones, tanto aéreos como terrestres, se están utilizando sistemas de vídeo analógico como PAL o NTSC debido a la imposibilidad de operar máquinas a alta velocidad con las latencias que supone la utilización de sistemas de vídeo digitales. Entre los muchos inconvenientes que supone utilizar estas tecnologías, que en el ámbito de la televisión bien podrían considerarse obsoletas, se incluyen el excesivo uso de ancho de banda y la distorsión y ruido que este supone. Uno de los objetivos principales del proyecto *Racing Drones* es mejorar las capacidades de tiempo real de un nuevo algoritmo de compresión de baja latencia, conocido como *Logarithmical Hop Encoding* (*LHE*), lo suficiente como para poder implantar en los drones un sistema de vídeo completamente digital centrado en este. *LHE* es un algoritmo de compresión multimedia con pérdidas presentado en el año 2015 que promete batir los tiempos de compresión de los algoritmos tradicionales basados en la operación frecuencial mediante una ventaja computacional obtenida al operar en el dominio espacial.

Este Trabajo de Fin de Grado tomará la parte del proyecto *Racing Drones* que busca la aceleración del algoritmo *LHE* como diseño objetivo para sintetizar con la herramienta *SDSoC*, tratando de servir, a la vez, como prueba de concepto del uso de herramientas de diseño de hardware situadas en un nivel de abstracción superior, y como prueba de concepto del uso del algoritmo LHE como elemento principal de un codificador de vídeo de baja latencia para el pilotaje manual de drones basado en vídeo FPV.

1.2 OBJETIVOS

El objetivo principal de este TFG es llegar a sintetizar e implementar un diseño empotrado híbrido CPU-FPGA que implemente una versión básica del algoritmo LHE a través del uso de la herramienta *SDSoC* y lenguajes de programación (no de descripción de hardware). Dado que la herramienta *SDSoC* es muy nueva y poco documentada, y que esta será una de las primeras implementaciones del algoritmo LHE, no se espera conseguir un sistema acelerado, pero si se perseguirá el estudio de las implicaciones en el proceso de diseño que conlleva utilizar esta nueva herramienta y los problemas del algoritmo LHE para su implementación en hardware.

El primer objetivo consistirá en el estudio de la descripción del algoritmo y la traducción de una implementación funcional y estable en lenguaje Python recibida de los desarrolladores de LHE a C, un lenguaje que sí puede ser utilizado para HLS.

El segundo objetivo será la refactorización del código traducido y la inclusión de directivas de compilación hardware, con el fin de llegar a un diseño sintetizable e implementable. Este objetivo incluye el requisito previo de familiarización con la herramienta *SDSoC*, para lo que será necesario estudiar su documentación y la realización de tutoriales y experimentos básicos.

El tercer objetivo estará en el análisis del sistema sintetizado y las complicaciones existentes que hayan impedido su aceleración mediante hardware, sentando las bases para un futuro trabajo que logre implementar un sistema con una latencia de compresión de imagen reducida.

Este es, por tanto, un trabajo de carácter multidisciplinar para el que serán necesarios conocimientos de desarrollo de software, arquitectura y desarrollo de sistemas digitales, procesamiento de imagen, y, en menor medida, conocimientos de redes, sistemas de transmisión de señales multimedia y arquitecturas Linux.

1.3 ORGANIZACIÓN DE LA MEMORIA

La memoria consta de los siguientes capítulos:

- Capítulo 1: Se introduce la motivación y los objetivos planteados para este trabajo.
- Capítulo 2: Estado del arte. Se introducen la herramienta y plataforma utilizadas.
- Capítulo 3: Breve estudio de la versión básica del algoritmo LHE. Proceso de traducción de la implementación en Python a una versión en lenguaje C, y la optimización de esta.
- Capítulo 4: Modificaciones más relevantes a las que ha sido necesario someter el código para conseguir un diseño sintetizable, implementable y acelerado.
- Capítulo 5: Medidas de rendimiento del sistema obtenido y comparaciones con otros resultados.
- Capítulo 6: Conclusiones de este trabajo y líneas de trabajo futuro.
- Capítulo 7: Bibliografía.
- Anexo A: Se anexa la versión final del código fuente desarrollado con todas las refactorizaciones y directivas que se han necesitado para que *SDSoC* sintetice adecuadamente el sistema híbrido deseado.
- Anexo B: Se ha realizado un breve estudio sobre la compatibilidad de los módulos de cámara digital de *Raspberry-Pi* con *ZynqBerry*.

2 ESTADO DEL ARTE

2.1 HERRAMIENTA UTILIZADA: SDSOC

2.1.1 Descripción

La herramienta utilizada en este proyecto para la síntesis del diseño desarrollado es el software *SDSoC*, concretamente la versión *SDSoC 2016.2*, corriendo sobre el sistema operativo *Windows 10 Educational*.

SDSoC es un IDE (Integrated Design Environment) basado en Eclipse para la implementación de sistemas empotrados híbridos CPU – FPGA utilizando la familia de plataformas MPSoC (MultiProcessor System-on-Chip) *Zynq-7000* y *Zynq UtraScale+* desarrollado por *Xilinx* [14] [13] . Proporciona un entorno de desarrollo de aplicaciones C/C++ para sistemas embebidos destinado a desarrolladores software. *SDSoC* incluye un compilador de sistemas C/C++ que supone la capa superior de un complejo toolchain que incluye la compilación cruzada de aplicaciones para procesadores ARM, la síntesis de lógica programable y la generación automática de las redes de movimiento de datos que comunican CPU y FPGA.

El compilador de *SDSoC* (`SDSCC/SDS++`) se encarga de analizar el código y las directivas SDS incluidas en el, y a partir de esto infiere unas directivas de *Vivado HLS*. *SDSoC* crea un proyecto de *Vivado HLS* con estas directivas, y este infiere mediante HLS (High Level Synthesis) el IP (Intelectual Property) del acelerador que será incluido a su vez en un proyecto de Vivado. Este proyecto de Vivado genera, a partir de este IP y de la especificación de la plataforma, el diseño del sistema híbrido. De *SDSoC* se obtiene finalmente una imagen de boot con el firmware, el sistema operativo y el ejecutable de la aplicación. El firmware contiene las instrucciones necesarias para que el boot loader (*U-Boot*) encuentre el sistema operativo (*Petalinux*) en la tarjeta SD, y el bitstream para programar la FPGA. *SDSoC* ofrece además una serie de herramientas entre las que destacan las destinadas a la medición y análisis del rendimiento de los sistemas sintetizados. Además integra un útil módulo para establecer conexiones FTP con la plataforma y poder leer y escribir datos en ella de forma sencilla.

2.1.2 Metodología de diseño

A continuación se describe la metodología de diseño de sistemas híbridos con *SDSoC*, que se propone como una metodología similar a la de desarrollo de software, y por tanto como accesible a diseñadores sin experiencia en desarrollo de hardware, y que constituye una característica fundamental de la herramienta. Esta metodología, que es la que se ha seguido durante el desarrollo de este proyecto, se representa también en la Figura 2.1.1.

Esta metodología parte de un diseño software funcional, que puede someterse a una compilación cruzada dentro de la herramienta y ser ejecutado en la plataforma como comprobación.

El primer paso hacia la optimización del sistema consiste en analizar la aplicación que se quiere acelerar y encontrar las partes con un mayor costo computacional, es decir, las funciones candidatas a ser aceleradas mediante hardware. Estas funciones han de ser aisladas para su conversión a hardware, y han de ser escritas según las directivas de programación de *SDSoC* [13] y las directivas de programación de *Vivado HLS* [11] . Una vez aisladas y escritas de la manera adecuada, estas funciones pueden marcarse para conversión a hardware. Entonces puede utilizarse el compilador de sistemas de *SDSoC* para generar el diseño híbrido y comprobar si efectivamente se ha conseguido un speedup fructuoso.

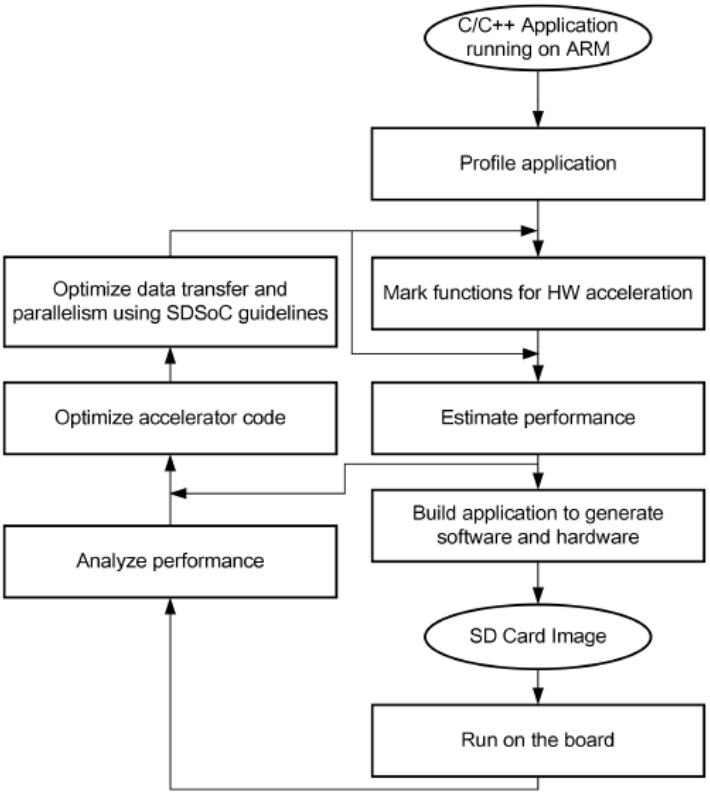

Figura 2.1.1 "*SDSoC Environment Flow*" [14]

Debido a que la generación del diseño completo es un proceso que consume grandes cantidades de tiempo (más de media hora para el diseño objeto de este proyecto), *SDSoC* proporciona una opción que permite estimar el rendimiento del sistema (Estimate Performance en la Figura 2.1.1), arrojando una medida aproximada del speedup del tiempo de ejecución del programa en el sistema acelerado sobre el necesitado ejecutándolo exclusivamente en el procesador. De esta opción se hará un extensivo uso en el capítulo 4, Optimizaciones orientadas a implementación hardware, donde se muestran los resultados de esta estimación para ir mostrando los resultados de cada optimización implantada.

Una vez conseguida la aceleración deseada, se procede a generar el diseño del sistema en su completitud y se configura la plataforma, donde se ejecuta finalmente el programa acelerado.

2.1.3 Red de movimiento de datos (*Data Motion Network*)

Dentro de los muchos factores que afectan al rendimiento de un sistema, en el caso de un sistema híbrido, la red de movimiento de datos es un elemento determinante. Aun consiguiendo un acelerador óptimo, si la transferencia de datos entre este, el procesador y la memoria no es el adecuado, globalmente el rendimiento del sistema podrá ser incluso peor que el del que no hace uso del acelerador. Es por tanto crucial para el diseñador conocer los mecanismos de comunicación entre acelerador y PS (Processing System), y saber tanto elegir las configuraciones óptimas como la forma de guiar al compilador de sistemas a inferir la generación de estas.

La red de movimiento de datos la forman tres elementos: los puertos del sistema de memoria del PS, la interfaz hardware en el acelerador, y los *data movers* que establecen la comunicación entre ambos [13] . En la Figura 2.1.2 se representa el esquema de comunicación básico para el caso concreto de un *Zynq XC7Z010*, que es el que incorpora la plataforma utilizada *ZynqBerry*, que se introducirá más adelante.

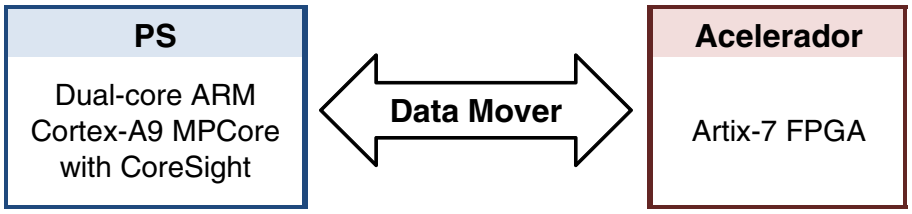

Figura 2.1.2 Esquema básico de la red de movimiento de datos, para el caso de un *Zynq XC7Z010*.

Los procesadores ARM A9 de la familia Zynq-7000 tienen dos niveles de caché y una memoria DDR. *SDSoC* puede crear en la lógica programable bloques DMA (Direct Memory Access) que permitan acceder a datos del procesador directamente a través de los puertos de interfaz del sistema. Existen tres tipos de puertos de interfaz del sistema: ACP (Accelerator Coherence Port), que permite al hardware acceder directamente a la caché del procesador de forma coherente; HPx (High Performance ports, donde x es un número de 0 a 3), que permite acceso directo a la memoria DDR del procesador a través de interfaces FIFO asíncronas (AFI); y puertos GPx (General-Purpose IO, donde x es 0 o 1), que permiten al procesador manipular registros físicos. *SDSoC* es capaz de tomar decisiones sobre qué tipo de interfaces conviene utilizar para cada caso de forma automatizada, pero para obtener un sistema optimizado es el diseñador el que debe analizar la aplicación y guiar al compilador a través de directivas a la implementación de las interfaces de datos más adecuadas.

En el acelerador se genera una interfaz de datos para cada dato de entrada/salida (esto es, para cada argumento y para el valor de retorno de la función hardware). Cada interfaz es generada dependiendo del tipo de dato. Para los escalares se genera una interfaz sencilla que pase o saque los datos del acelerador. El caso de los arrays es más complejo: se puede inferir una interfaz RAM o una interfaz stream. La interfaz RAM permite al acelerador realizar accesos aleatorios, pero requiere movimientos de datos mayores y consume un número de recursos elevado (especialmente de BRAM), mientras que las interfaces stream suponen la limitación de que los accesos sean estrictamente secuenciales, pero requieren de menos recursos. La configuración de una u otra de estas interfaces es uno de los problemas principales que se tratan en este proyecto (4.1.7). En el caso de clases o estructuras existen dos estrategias. La primera consiste en "deshacerlas" en sus componentes y transferir estos como escalares, mientras que la segunda consiste en empacar y alinear sus componentes, convirtiéndolos en datos únicos de mayor tamaño. Estos métodos también se abordaran en pruebas avanzadas de optimización (4.1.9).

El tipo de data mover generado por *SDSoC* depende de nuevo de los tipos de datos. Para los escalares siempre se utiliza un data mover de tipo `AXI_LITE`. Para los arrays, dependiendo de sus atributos de memoria y tamaño, se pueden generar data movers de tipos `AXI_DMA_SIMPLE`, `AXI_DMA_SG`, `AXI_DMA_2D`, `AXI_FIFO`, `AXI_M` o `AXI_LITE`. El uso de uno u otro tipo dependerá del tamaño de las transferencias, de la contigüidad en memoria de los datos y de su coherencia de caché. Los requerimientos de algunos de estos tipos se recogen en la Tabla 2.1.1, aunque sus características no se cubren en profundidad en la documentación de *SDSoC*. Las estructuras y clases, al ser transformados a escalares, se trataran como estos.

Data Mover	Physical Memory Contiguity	Data Size (bytes)
AXIDMA_SG	Either	> 300
AXIDMA_Simple	Contiguous	< 8M
AXIDMA_2D	Contiguous	< 8M
AXI_FIFO	Non-contiguous	< 300

Tabla 2.1.1 "Data Mover Selection" [13] .

2.1.4 Pautas de programación y directivas de compilación de sistema

Como se ha mencionado anteriormente, existen unas pautas de programación, recogidas en [13] y [11] para la escritura del código correspondiente a funciones destinadas a hardware, esto es, pautas de programación para HLS. Estas pautas, lejos de ser triviales, constituyen un extenso conjunto de normas y suponen un proceso de reeducación para el programador de C/C++, siendo necesario deshacerse de muchos prácticas tradicionalmente consideradas buenos hábitos en el mundo del desarrollo de software que sin embargo son desaconsejadas o incluso resultan en diseños imposibles cuando son aplicadas al desarrollo de hardware (HLS). Esto supone un contraste con la idea de que *SDSoC* pueda ser utilizado por desarrolladores software, y más adelante se verá la dificultad de construir un código sintetizable.

Además de la necesidad de seguir dichas pautas, *SDSoC* hace uso de unas directivas de compilación propias (`#pragma SDS …`) de un nivel de abstracción superior, y por tanto distintas, a las directivas HLS utilizadas en *Vivado HLS* (`#pragma HLS …`). El compilador de *SDSoC* se encarga de analizar el código y utilizar las directivas SDS proporcionadas por el diseñador para inferir unas directivas HLS, más precisas (de menor nivel de abstracción). No obstante, es posible, e incluso recomendado en la documentación de la herramienta, la inclusión de directivas HLS directamente en el código, directivas que *SDSoC* tomará aun cuando infiera que otras son mejores.

Estas pautas y directivas forman, junto a la lógica general de programación, el conjunto de recursos que tiene el diseñador para conseguir que la herramienta infiera un sistema eficiente, lo que las hace de vital importancia. Es por ello que su utilización en este proyecto será detallada a lo largo de esta memoria.

2.2 PLATAFORMA UTILIZADA: ZYNQBERRY

La plataforma elegida para el desarrollo de este proyecto es el SoM (System on Module) *ZynqBerry,* o también *TE0726* (Figura 2.2.1), fabricado por *Trenz Electronic* [10] . Está basado en la popular plataforma de código cerrado *Raspberri-Pi Model 2*, con sus mismos conectores periféricos, pero sustituye su procesador por un SoC *Zynq XC7Z010* de *Xilinx*, que, por incluir a su vez un procesador ARM, permite que la plataforma pueda funcionar de la misma manera.

La revisión 3 utilizada en este proyecto (*TE0726-03*), cuyo diagrama de bloques se muestra en la Figura 2.2.2, incorpora:

- DDRL3L SDRAM 512 MB
- Memoria flash QSPI de 16 MB
- 4 puertos USB 2.0
- 1 conector RJ45 para Ethernet de 10/100 Mbit
- Lector de tarjetas Micro SD donde se cargan los archivos de configuración.
- Conector HDMI
- Conector DSI
- Conector CSI-2 para cámara
- Jack de audio estéreo de 3.5 mm
- Bus HAT de 16 I/Os
- Micro-USB para
  > Alimentación de la placa
  > USB UART
  > JTAG y debug de la FPGA

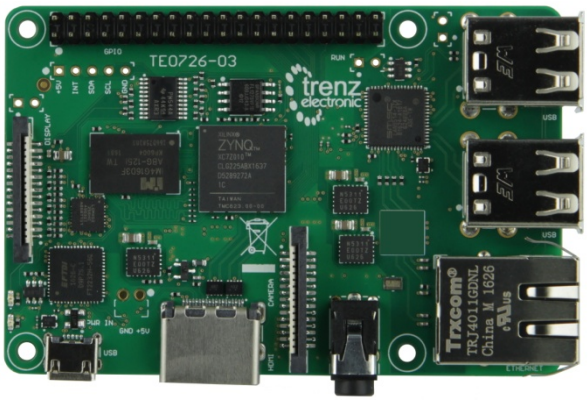

Figura 2.2.1 Fotografía de la plataforma *ZynqBerry* [9] .

El SoC *Zynq XC7Z010* incorpora principalmente un procesador *Dual-core ARM Cortex-A9 MPCore with CoreSight* y una FPGA *Artix-7* de *Xilinx*. Sus otros recursos y especificaciones más detalladas pueden encontrarse en [16] .

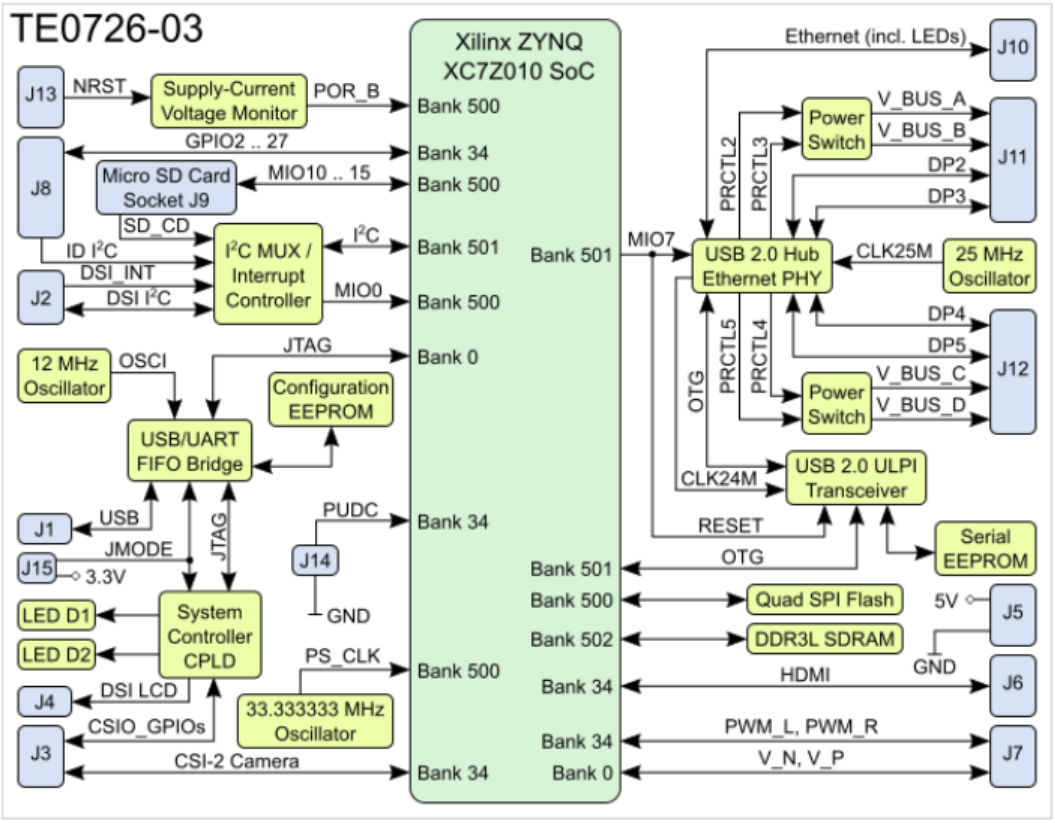

Figura 2.2.2 Diagrama de bloques de la plataforma *ZynqBerry* [10] .

El principal motivo de elegir esta plataforma es la facilidad de añadirle periféricos que proporciona el hecho de que esté basada en Raspberri-Pi. Concretamente, fue de especial interés en el planteamiento original de este trabajo la posibilidad de utilizar el módulo de cámara de vídeo Raspberri-Pi Camera, tema que se trata en el anexo B. Además, tiene un tamaño mucho menor a otras placas soportadas por SDSoC, como la Zedboard (85 x 56 mm vs 160 x 135 mm).

Cabe destacar que aunque la plataforma salió a la venta a comienzos del año 2017, momento en el que comenzó a ser utilizada en este proyecto, la mayor parte de la documentación sobre esta no vio la luz hasta mediados del mismo año, por lo que se careció de mucha de la información ahora disponible en [10] durante el desarrollo de este proyecto.

3 ALGORITMO DE COMPRESIÓN DE BAJA LATENCIA LHE

3.1 DESCRIPCIÓN

El algoritmo LHE (Logarithmical Hop Encoding) [4] es un algoritmo de compresión multimedia con pérdidas que opera en el dominio espacial (no en el frecuencial), basado en la percepción del ojo humano y de bajo coste computacional, dirigido a constituir el núcleo de un codificador de imagen de baja latencia.

El funcionamiento de LHE está basado en la Ley de Weber-Fechner, según la cual para notar un cambio lineal en la percepción es necesario un cambio exponencial en el estímulo físico que la produce. Aplicando esta ley psicofísica al ojo humano, se puede concluir que dentro del espectro continuo y lineal de valores con los que codificamos imágenes solamente habrá un pequeño subconjunto de valores que supongan cambios logarítmicos perceptibles por el sistema visual humano. Este algoritmo se aprovecha de esta conclusión para hacer una cuantificación, utilizando solamente los valores correspondientes a cambios perceptibles y por tanto logrando una compresión con pérdidas.

A continuación se describe el funcionamiento de una versión básica del algoritmo: *LHE básico*.

En primer lugar se calcula el color de fondo (la ***predicción***, o *hop nulo*), promediando los valores del píxel izquierdo y el superior-izquierdo. Existen cuatro casos especiales: la predicción del primer píxel es igual a su valor; la predicción de los píxeles de la primera línea es igual al valor de su píxel izquierdo; la predicción de los píxeles de la primera columna es igual al valor de su píxel superior; y la predicción de los píxeles de la última columna es igual a la media entre los valores de los píxeles izquierdo y superior. Estos cinco casos de cálculo de predicción se muestran en la Figura 3.1.1, donde los píxeles marcados con *a* y *b* son los utilizados para la predicción de cada píxel adyacente coloreado en azul.

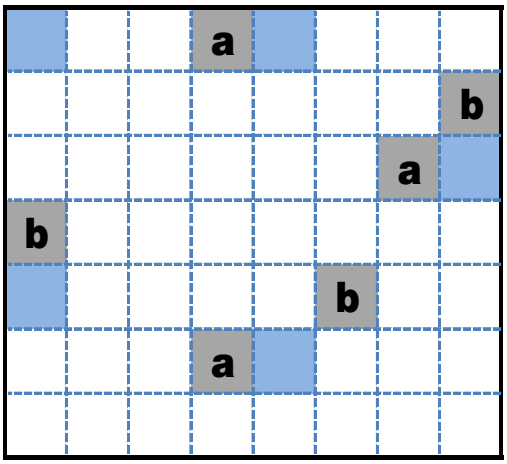

Figura 3.1.1 Cálculo de predicción según la posición relativa del píxel.

La diferencia entre la predicción calculada y el valor real será cuantificada a una serie de ***hops*** (saltos logarítmicos) definidos en referencia a la predicción. Se utilizarán 4 hops positivos, 4 hops negativos y un hop nulo (el valor de predicción), y se almacenará el identificador del hop correspondiente, por lo que los 256 grados de libertad originales se reducen a 9 símbolos consiguiendo así una primera compresión. El ***primer hop*** (h_1) es el salto mínimo capaz de detectar el sistema visual humano, y su valor será modificado dinámicamente durante el procesamiento de la imagen según el comportamiento de acomodación al brillo y el umbral de detección del ojo humano. El resto de saltos se definen según la Ec. 3.1.1.

$$h_{-1} = -h_1$$
$$h_2 = r \cdot h_1 \qquad h_{-2} = r \cdot h_{-1}$$
$$h_3 = r \cdot h_2 \qquad h_{-3} = r \cdot h_{-2}$$
$$h_4 = r \cdot h_3 \qquad h_{-4} = r \cdot h_{-3}$$

Ec. 3.1.1 Valor de los hops.

La ***razón*** *r* ha de ser lo suficientemente pequeña como para disponer de hops cercanos para zonas suaves y a la vez lo suficientemente grande para disponer de hops grandes que codifiquen zonas de cambios abruptos. Se ha definido una razón óptima $r_{opt} = 2.5$, pero ha de cuidarse que los hops no se salgan del rango de valores $(0 : 255)$.

La actualización del valor del primer hop se ha diseñado teniendo en cuenta dos factores: el umbral de detección y la adaptación al brillo del ojo humano. El ojo humano es capaz de detectar cambios en la luminancia a partir de un 2% de variación. Por ello, la diferencia mínima entre el hop nulo y el primer hop ha de ser de un 2%. Sin embargo, este umbral se incrementa cuando la luminosidad es muy alta o muy baja. Debido a esta imprecisión del ojo en zonas alejadas del brillo medio, deberá modificarse dinámicamente el valor del primer hop al llegar a estas zonas extremas para disponer de saltos más grandes que puedan ser detectados por el ojo. Se ha elegido 4 como valor mínimo para h_1, un valor por encima del umbral mínimo de detección (2%) para zonas de luminosidad media (~150). Como valor máximo se ha elegido 10 a forma de compromiso entre precisión y compactación de datos.

De esta forma, se van a codificar los valores de cada canal según 9 símbolos que dependen de la predicción y del primer hop, resultando en una tabla de tres dimensiones. Para obtener el símbolo de un valor (partiendo de que se trata del espacio de color YUV) primero se habrá de calcular la predicción. Después, según esta y el valor de h_1 se habrá de hallar en la tabla el hop más cercano a la diferencia entre la predicción y el valor a codificar, y, finalmente, se actualizará el valor de h_1.

Debido a que en las imágenes suele haber grandes zonas de cambios suaves, es decir, estadísticamente priman variaciones suaves [4] , se aprovecha esta distribución estadística no uniforme de los hops para conseguir aun más compactación aplicando codificación Huffman a los identificadores de los hops. Debido a esta codificación, que asigna códigos más cortos a cambios más suaves, de haber dos hops a la misma distancia del error de predicción se elegirá el hop menor, al que corresponde un código más corto.

En la Figura 3.1.2 se presenta el diagrama de bloques del compresor LHE. Este cálculo se efectúa por separado para cada canal (Y, U y V).

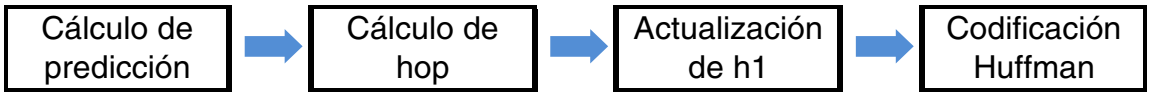

Figura 3.1.2 Diagrama de bloques del algoritmo LHE.

LHE hace uso del espacio de color YUV debido a que el ojo humano es menos sensible a los cambios de color (U, V) que a los cambios de brillo (Y), lo que permite aplicar un esquema de submuestreo de crominancia (U, V) para comprimir aún más la información de la imagen sin provocar en esta un impacto cualitativo. Los esquemas de submuestreo contemplados por el algoritmo son los modelos YUV420 y YUV 422, los más utilizados por otros estándares de compresión como JPEG.

Hasta aquí se ha descrito la versión básica del algoritmo (*LHE básico*), que es con la que se trabajará en este proyecto. Esta versión tiene limitaciones importantes, como la imposibilidad

de elegir el ratio de compresión. No obstante, esta versión del algoritmo es capaz de obtener resultados de compresión superiores a los de JPEG e inferiores a los de JPEG2000 en tiempos de ejecución menores a los de ambos. Se definen además una versión avanzada del algoritmo (*LHE avanzado*) y una versión primitiva de un códec de vídeo (*LHE vídeo*), que no se cubren en este proyecto.

3.2 TRADUCCIÓN A LENGUAJE C

Dado que *SDSoC* se ofrece como una herramienta para convertir software a un diseño hardware de una manera lo suficientemente automática para que pueda ser utilizada por un desarrollador software sin experiencia en hardware, se ha elegido en este proyecto partir de la implementación de los desarrolladores originales en vez de escribir una implementación desde cero a partir de la descripción del algoritmo.

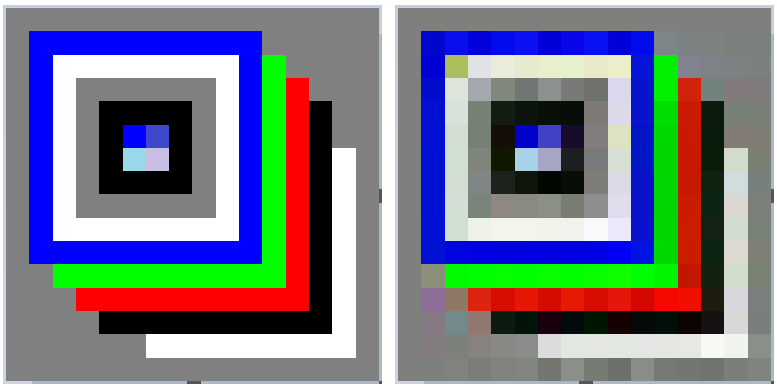

Figura 3.2.1 Imagen "*prueba3*" original y descomprimida.

Figura 3.2.2 Imagen "*lena*" original y descomprimida.

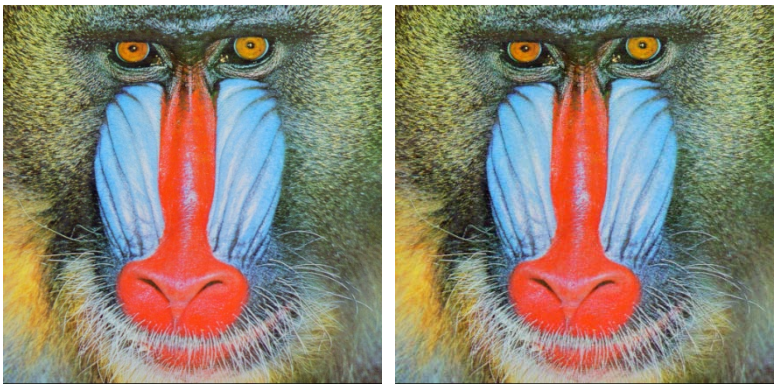

Figura 3.2.3 Imagen "*baboon*" original y descomprimida.

De los desarrolladores originales del algoritmo se ha recibido una implementación estable de las funciones básicas del algoritmo escrita en lenguaje Python. Se ha probado este código comprimiendo y descomprimiendo una serie de imágenes, cuyos originales y versiones descomprimidas se muestran en la página anterior. La Figura 3.2.1 (*"prueba3"*) es una imagen de 16 x 16 píxeles creada por el autor de este proyecto con el fin de disponer de una imagen sencilla con un diseño que resulta especialmente complicado al algoritmo LHE. Las imágenes Figura 3.2.2 y Figura 3.2.3 (las clásicas *"lena"* y *"baboon"*) son fotografías de 512 x 512 píxeles obtenidas de la base de datos de imágenes *USC-SIPI* [7] .

Aunque este código original implementa algunas estrategias para reducir el tiempo de ejecución, dado que es un algoritmo pensado para tener una baja latencia, este código ha sido escrito principalmente teniendo como objetivo la posibilidad de explorar distintas configuraciones y alteraciones del algoritmo LHE, priorizando así la flexibilidad del código sobre su rendimiento. Es por ello que se observan tiempos de compresión muy elevados incluso para software (Tabla 3.2.1 [1]).

	Imagen			
	prueba3	*lena*	*baboon*	*Media*
Tasa de compresión	2,2645	4,9077	3,2428	3,47167
Tiempo de compresión (s)	6,4070	12,6420	13,2640	-

Tabla 3.2.1 Tasas y tiempos de compresión obtenidos para imágenes de prueba en Python.

Puesto que los lenguajes de programación soportados por *SDSoC* son C y C++, se ha tenido que hacer una traducción de este código Python a lenguaje C. Además, se ha hecho una serie de primeras optimizaciones, descritas más adelante, con el fin de obtener un mejor rendimiento del programa, ya que no es posible plantearse sintetizar en hardware un diseño que ni siquiera es mínimamente eficiente en software. La parte del algoritmo traducida es la de compresión, que es la que se desea acelerar por hardware en este proyecto, mientras que el decodificador utilizado ha sido implementado por otro autor [1] ,por lo que no se profundiza en su descripción en esta memoria.

3.2.1 Primeras optimizaciones

Una de las primeras preocupaciones en relación con el rendimiento al traducir el código Python ha sido su exhaustivo uso de arrays sobredimensionados, de múltiples dimensiones y tamaños considerables.

Para el cálculo de los hops se necesita una tabla con todos los posibles saltos según las posibles predicciones y primeros hops. Para esta tabla, en el código original se utilizan dos arrays de cuatro dimensiones que son generadas al comienzo de cada ejecución. Esto tiene utilidad a la hora de explorar el diseño del algoritmo, probando distintas configuraciones y parámetros, pero de cara a la productividad suponen una pesada carga innecesaria. Este cálculo inicial ha sido sustituido por un solo array estático precalculado de tres dimensiones, que posteriormente será sintetizado como una memoria ROM. El uso de esta tabla con valores precalculados se sugiere en [4] .

Para el cálculo de las predicciones se necesita conocer el resultado de cuantificación de píxeles anteriores, para lo cual es necesario guardar estos valores durante la ejecución del programa. En el código original se guarda el valor de cuantificación de todos y cada uno de los píxeles, haciendo uso de un array con un tamaño igual al de la imagen original. Debido a que

[1] En la Tabla 3.2.2 no se calcula la media de tiempos de compresión por tratarse de imágenes de tamaño diferente.

los píxeles cuyos valores de cuantificación se necesitan leer son el superior (o el superior derecho) y el izquierdo al píxel que se está procesando, en el peor de los casos se accede a un píxel un número de posiciones atrás igual al número de píxeles en una línea de la imagen (Figura 3.2.4). Por ello, se ha sustituido el array original por un sistema de listas FIFO de longitud fija igual al ancho de la imagen.

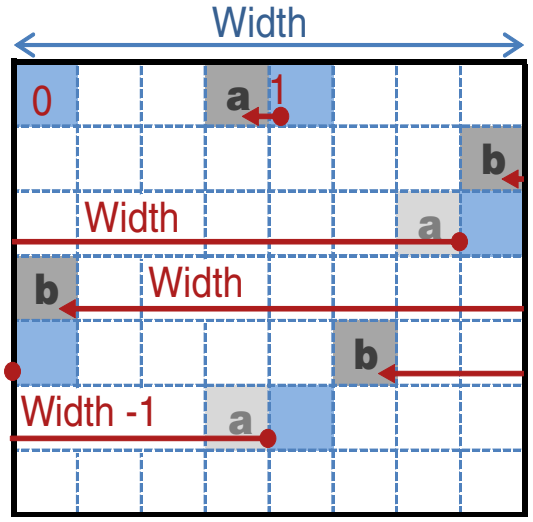

Figura 3.2.4 Longitud máxima de caché de resultados necesaria para cada caso de predicción.

Por otro lado, se ha modificado el orden de procesamiento de imagen del código original. En el código Python se hace primero una conversión YUV de la imagen completa para pasar después a codificar el canal Y por completo, seguido de los canales U y V, para finalmente escribir en el archivo de salida la completitud de la imagen codificada. Esto supone recorrer múltiples veces la imagen completa, incrementando el número de accesos a memoria e impidiendo la paralelización del procesamiento. Para solventar esa situación se ha reescrito el código de forma que el procesado completo, desde la conversión YUV a la escritura en el archivo, se haga píxel a píxel, siendo suficiente con recorrer la imagen una sola vez y permitiendo tanto la paralelización que se hará en el cálculo de hops de los canales como la inserción del pipeline, descritos en el capítulo siguiente.

Por último, se ha eliminado una funcionalidad del código original con el fin de simplificar el desarrollo de este proyecto. Esta funcionalidad es la de configuración del esquema de submuestreo de crominancia, pudiendo elegir entre los esquemas 4:4:4 (sin submuestreo), 4:2:2 (componentes de crominancia muestreados por un factor horizontal 2) y 4:2:0 (componentes de crominancia muestreados por un factor horizontal 2 y un factor vertical 2), siendo este último el esquema utilizado por algoritmos de compresión como JPEG. En esta refactorización se ha dejado como fijo el esquema de muestreo 4:4:4, o lo que es lo mismo, se ha eliminado la capacidad de realizar submuestreo. El submuestreo de crominancia tiene una gran relevancia en el algoritmo LHE, especialmente en su versión avanzada *LHE avanzado*, donde se ha introducido una nueva estrategia de submuestreo (*downsampling elástico*). En una versión siguiente de este sistema se deberá devolver esta funcionalidad al algoritmo, y por tanto deberá también actualizarse la definición del formato del archivo de salida añadiendo un campo que indique al decodificador el esquema utilizado en compresión (3.3). Por tanto, puede decirse que el sistema desarrollado en este proyecto arrojará unos resultados pesimistas respecto a los que se obtendrían con una implementación completa del algoritmo.

Se representa a continuación (Figura 3.2.5) el diagrama de bloques del código desarrollado.

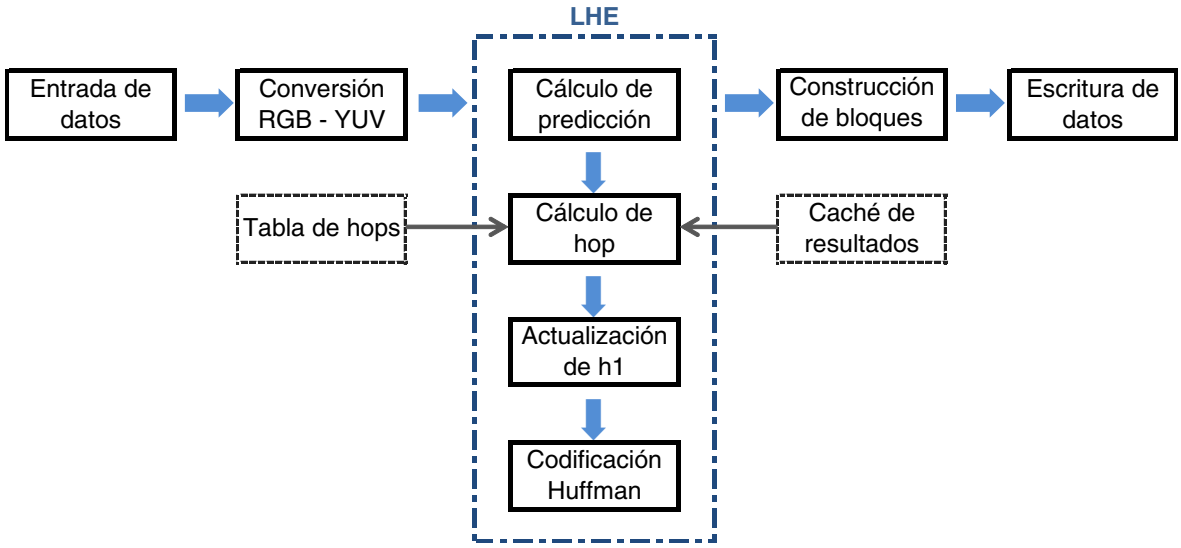

Figura 3.2.5 Diagrama de bloques del programa.

3.2.2 Librería stb_image

El código original hace uso de la librería de procesamiento de imagen para Python `PIL` (*Python Imaging Library*). En vez de traducir a C esta librería por completo, se ha optado por sustituirla por otra librería de procesamiento de imagen escrita para C. La librería escogida ha sido `stb_image.h`, una librería de dominio público escrita en C en un solo archivo de cabecera por Sean T. Barrett [2] . Dado que es una librería principalmente pensada para el desarrollo de videojuegos indie, y no para el tratamiento de imágenes, no es una librería especialmente rápida. Sin embargo es una librería de uso bastante extendido, de dominio público, sencilla y con soporte a los principales formatos de imagen, razones por las que se ha elegido como sustituta. Más adelante, de cara a conseguir un sistema más eficiente, se planteó originalmente sustituir esta librería por la librería *OpenCV*, pero como se explicará en la sección *OpenCV*, esto no ha sido finalmente posible.

La parte de esta librería que se está utilizando es la función `stbi_load`, que se encarga de leer una imagen, cuya ruta se le pasa como argumento, y de devolver sus dimensiones y sus datos RGB en un array unidimensional. Esta función llama a su vez a las funciones necesarias según el formato de la imagen a leer.

3.3 DEFINICIÓN DEL FORMATO DE ARCHIVO .LHE

Entre el desarrollador del decodificador LHE [1] y el autor de este proyecto, con permiso del desarrollador original del algoritmo, se ha acordado una definición del formato de archivo `.lhe`, que son los que genera el codificador desarrollado con los datos de imagen comprimidos.

En la Figura 3.3.1 se representan los campos de los archivos. Los primeros dos campos son el ancho y el alto en píxeles de la imagen, respectivamente, en enteros de 16 bits Little endian cada uno. Los siguientes 3 campos, de 8 bits cada uno, son los valores Y, U y V, en ese orden y sin codificar, del primer píxel de la imagen. A partir de este primer píxel se aplicarán los saltos para obtener el resto de píxeles. A continuación se escriben los saltos, codificados según la codificación Huffman presentada en la Tabla 3.3.2, de todos los demás píxeles, leídos de izquierda a derecha y de arriba abajo.

WIDTH	HEIGHT	Y	U	V
16 bits Little endian	16 bits Little endian	8 bits	8 bits	8 bits

HUFFMAN						
V HOP	U HOP	Y HOP	V HOP	U HOP	Y HOP	…
Little endian						

	HUFFMAN					PADDING
…	U HOP	Y HOP	V HOP	U HOP	Y HOP	0
	Little endian					

Figura 3.3.1 Campos del formato .lhe

Los códigos Huffman son escritos de forma binaria, por lo que puede ser que sea necesario un rellenado (padding) al final del archivo para completar un byte. Dado que el decodificador conoce el número de píxeles de la imagen al leer los primeros campos del archivo, una vez haya decodificado el último píxel dejará de leer y el rellenado nunca será procesado, por lo que interesa que este campo tenga la longitud mínima imprescindible de cara a compresión de archivos, y podría ignorarse de cara a transmisión. En este proyecto se ha realizado el rellenado con ceros, pero, por la misma razón anterior, podría ser rellenado con cualquier otro valor.

Utilizar una codificación Huffman dinámica proporcionaría una optimización en la compresión, pero incrementaría la latencia y supondría tener que transmitir el diccionario al decodificador antes de transmitir la imagen comprimida, por lo que se ha optado por una codificación Huffman estática. Para determinar los códigos (*Huffman code*) para cada salto (*Hop*) mostrados en la Tabla 3.3.2, se han utilizado las probabilidades estadísticas (*Probability %*) obtenidas en [4] . La columna *Hop Number* muestra el valor con el que se ha representado cada *hop* dentro del código desarrollado.

Hop	*Hop Number*	*Huffman code*	*Probability (%)*
0	4	1'	42,5
1	5	000'	18,83333333
-1	3	100'	16,16666667
2	6	110'	8
-2	2	0010'	7,166666667
3	7	01010'	3,233333333
-3	1	111010'	3
4	8	1011010'	0,7333333333
-4	0	0011010'	0,3666666667

Tabla 3.3.2 Codificación Huffman utilizada

Como se adelantó en 3.2.1, una futura versión de esta definición habrá de incluir un campo adicional que indique al decodificador el esquema de submuestreo de crominancia utilizado en compresión.

3.4 CONCLUSIONES

Las mejoras anteriormente descritas han sido introducidas tratando de respetar la forma en que los autores originales implementaron su algoritmo. Tras varias revisiones del código se ha logrado un software con un rendimiento fructuoso, pero este código sigue sin ser sintetizable por *SDSoC*, por lo que serán necesarias una serie de refactorizaciones orientadas a la implementación en hardware descritas en el capítulo siguiente. Por la cantidad de

modificaciones que ha necesitado el código original, podría concluirse que aunque *SDSoC* se promete como una herramienta para desarrolladores software, no lo es tanto. De cara al desarrollador de hardware, se tienen dudas sobre si partir de un código software para posteriormente optimizarlo ofrece ventajas de productividad sobre escribir desde cero un código orientado a hardware a partir de la descripción del algoritmo, especialmente si se tiene en cuenta que existen formas de implementar este algoritmo de forma más eficiente, una de las cuales se describirá en el apartado 6.2.3.

Sin alterar el funcionamiento del algoritmo ni, por tanto, las tasas de compresión de la Tabla 3.2.1, **se ha conseguido reducir en tres órdenes de magnitud el tiempo de compresión** en esta traducción a lenguaje C (Tabla 3.4.1).

	Imagen		
Tiempo de compresión (s)	*prueba3*	*lena*	*baboon*
Python	6,4070	12,6420	13,2640
C	< 0,001	0,0215	0,0305

Tabla 3.4.1 Comparación de tiempos de compresión obtenidos en software.

4 OPTIMIZACIONES ORIENTADAS A IMPLEMENTACIÓN HARDWARE

4.1 REFACTORIZACIÓN DE CÓDIGO

La refactorización de código es el proceso consistente en la reestructuración de un código fuente existente sin cambiar su funcionalidad. Dado que las estrategias para optimización de código destinado a microprocesadores no siempre son las más adecuadas para un código destinado a ser sintetizado como hardware específico, será necesario refactorizar el código C (resultante de la traducción descrita en el capítulo anterior) para obtener el mejor rendimiento posible del sistema a sintetizar sin alterar su funcionamiento. Dado que se trata de un sistema híbrido procesador-FPGA, no solo bastará con refactorizar las partes que se quieran sintetizar en hardware sino que será necesario tener en cuenta las transacciones de datos que habrá entre el procesador ARM y el hardware (2.1.3).

En este capítulo se describen las modificaciones y refactorizaciones realizadas para obtener un diseño hardware lo más óptimo posible, y además las directivas SDS y HLS introducidas para guiar al compilador tanto para este fin como para la inserción de los data movers más apropiados para este sistema.

El diagrama de bloques del sistema que se desea conseguir se presenta en la Figura 4.1.1.

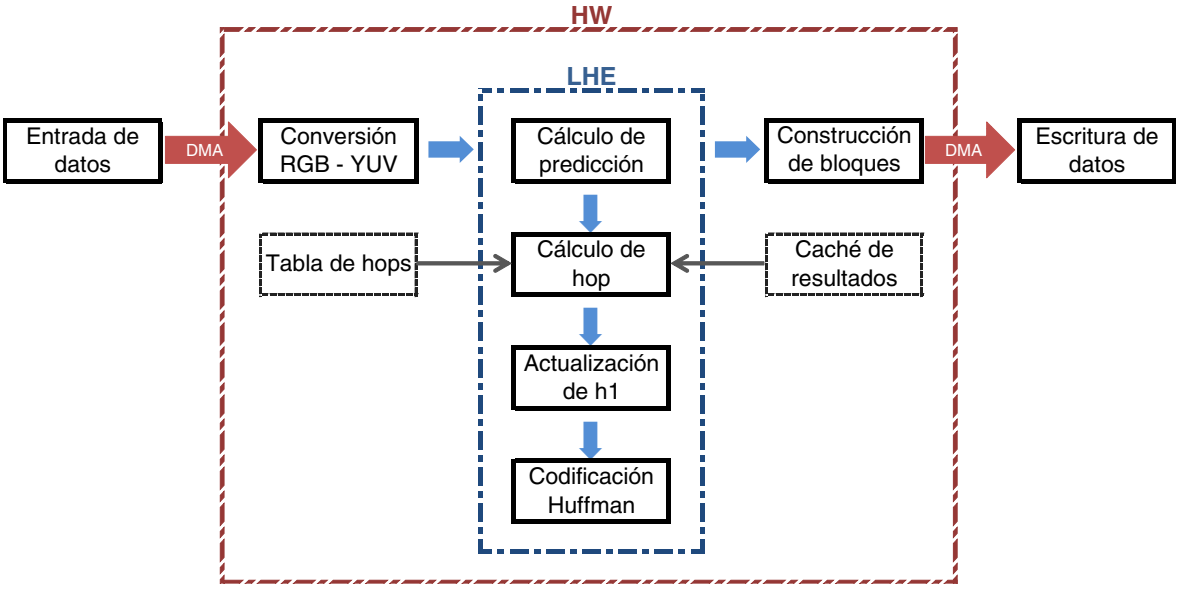

Figura 4.1.1 Diagrama de bloques del programa con conversión hardware.

4.1.1 Conversión de cachés de resultados a listas circulares

Aunque el uso de listas enlazadas para guardar los valores cuantificados de la última línea procesada supone una optimización respecto a guardar los valores de toda la imagen en un array, está lejos de ser una solución óptima. Para la generación del acelerador se ha decidido sustituir esta estrategia por el uso de unas sencillas listas circulares, implementadas como arrays unidimensionales de longitud arbitraria que son recorridos de forma circular con un puntero de lectura (el que lee los valores de los pixeles superiores) y un puntero de escritura (el que escribe los valores de los píxeles según se van procesando). Para el uso del valor del píxel anterior se ha incluido una variable estática en la función que recuerda el resultado de la llamada anterior.

Dado que, para el cálculo de predicciones, en el primer píxel de cada línea se usa el superior (el primero de la línea anterior), y en el segundo se pasa a usar su superior derecho (el tercero

de la línea anterior), se puede deducir que el valor cuantificado del segundo píxel de cada línea solo se utiliza en el cálculo de la predicción del tercer píxel de esa línea, por lo que es suficiente con leerlo de la variable que guarda el resultado anterior y nunca van a ser leídos de las listas circulares, por lo que no hace falta incluirlos estas (Figura 4.1.2).

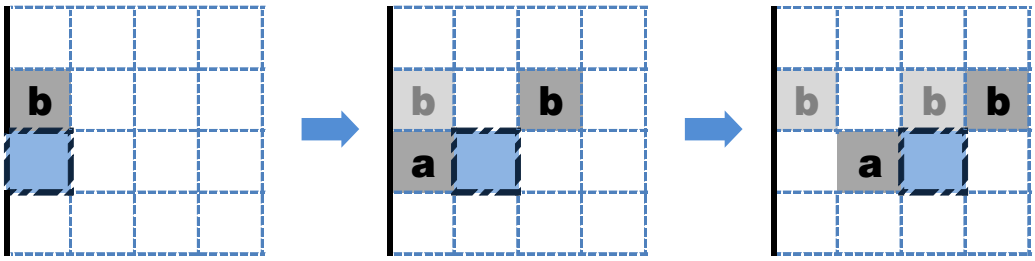

Figura 4.1.2 Nunca se lee el segundo píxel de cada línea en la caché de resultados.

De esta manera, además, no hace falta describir un caso particular para que el puntero de lectura pase del primer al tercer píxel, dado que se encuentran guardados de forma contigua en las listas. Además, dado que para las predicciones de la primera línea solo se necesita el valor del píxel anterior, el puntero de lectura no empieza a actualizarse hasta la segunda línea.

4.1.2 Primera síntesis

Tras la modificación descrita en la sección anterior, se ha reducido y simplificado lo suficiente el diseño como para lograr una primera síntesis marcando para conversión a hardware las funciones de conversión a YUV y de cálculo de hops. Como se puede ver en la Figura 4.1.3, el sistema híbrido logrado tarda aproximadamente el doble de tiempo que el procesador por sí solo. Además se observa un uso de recursos muy elevado, llegando a superar el 100% de LUTs disponibles en la plataforma (Figura 4.1.4).

Performance estimates for 'main' function

SW-only (Measured cycles)	582497614
HW accelerated (Estimated cycles)	1104289522
Estimated speedup	0,53

Figura 4.1.3 Deterioro del rendimiento del algoritmo en hardware.

Resource utilization estimates for hardware accelerators

Resource	Used	Total	% Utilization
DSP	53	80	66,25
BRAM	13	60	21,67
LUT	22687	17600	128,9
FF	17126	35200	48,65

Figura 4.1.4 Uso excesivo de recursos en primera síntesis.

Esta necesidad de recursos, totalmente excesiva para el sistema que se desea sintetizar, ha de minimizarse antes de plantearse el poder conseguir una aceleración.

Se ha de destacar que para llegar a esta primera síntesis han sido necesarias múltiples revisiones hasta llegar a un código muy optimizado, un estudio profundo de las pautas de programación dictadas por *Xilinx* [13], y numerosas pruebas de ensayo y error para familiarizarse con el escasamente documentado compilador de *SDSoC*. Y, aun habiendo logrado un diseño sintetizable en hardware, no ha sido posible marcar para conversión todas

las funciones que se deseaba, no se ha logrado una aceleración, ni siquiera un diseño implementable por exceso de tamaño. Lejos de ser una herramienta dirigida a diseñadores sin experiencia en hardware, podría decirse que el uso de *SDSoC* necesita de una curva de aprendizaje y de experimentación.

4.1.3 Problema de la conversión YUV

Del exagerado uso de DSPs, se ha deducido que el problema estaba en el uso de cálculo con coma flotante. En el código original se utiliza una fórmula para la conversión de espacios de color RGB a YUV que pondera los valores RGB con unos pesos con valores decimales (Ec. 4.1.1). En hardware esto se traduce a la inserción de bloques de cálculo con coma flotante, que requieren una cantidad elevada de recursos, en especial de DSPs.

```
Y = (0.299 * R + 0.587 * G + 0.114 * B);
U = (12 - 0.168736 * R -0.331364 * G + 0.5 * B);
V = (128 +0.5 * R -0.418688 * G -0.081312 * B);
```

Ec. 4.1.1 RGB a YUV con coma flotante.

Para solventarlo, se ha modificado la fórmula utilizando promedios con pesos enteros y divisiones por enteros (Ec. 4.1.2). Esto se infiere en hardware como operaciones sencillas y desplazamientos, que requieren un uso de recursos mucho menor.

```
Y = ((77 * R + 150 * G + 29 * B) >> 8)&0xFF;
U = 128 + (((- 43 * R - 85 * G + 127 * B) >> 8)&0xFF);
V = 128 + (((127 * R - 107 * G - 21 * B) >> 8)&0xFF);
```

Ec. 4.1.2 RGB a YUV sin coma flotante.

En la Figura 4.1.5 se muestra cómo baja el requerimiento de recursos, especialmente el de DSPs y LUTs, al aplicar esta modificación. Sin embargo, el uso de LUTs sigue siendo excesivo y, al ser cercano al 80%, puede que el sistema sea inimplementable por la dificultad que supondría en el proceso de placement.

Resource utilization estimates for hardware accelerators

Resource	Used	Total	% Utilization
DSP	7	80	8,75
BRAM	13	60	21,67
LUT	13872	17600	78,82
FF	12773	35200	36,29

Figura 4.1.5 Reducción de recursos por modificación del cálculo YUV.

Aunque con esta modificación se ha llegado a la versión finalmente utilizada de esta función, esta función sigue estando lejos de lo óptimo y supone un cuello de botella principal para la aceleración del algoritmo, como se estudiará más adelante en este capítulo.

4.1.4 Problema del uso de la operación módulo

Para determinar el origen del uso excesivo de LUTs, se ha acudido al informe de síntesis que genera *Vivado HLS*, un informe mucho más rico que los que ofrece actualmente *SDSoC*. A partir de este se ha podido averiguar que el origen de este exceso de requerimiento de LUTs eran las funciones que calculan los hops, y que dentro de estas el causante era la inclusión de módulos `srem` (`signed remainder`), utilizados para el cálculo de operaciones módulo (Figura 4.1.6).

Instance	Module	BRAM_18K	DSP48E	FF	LUT
getPixelHops_srem_32ns_12ns_32_36_U0	getPixelHops_srem_32ns_12ns_32_36	0	0	1610	1610
getPixelHops_srem_32ns_12ns_32_36_U1	getPixelHops_srem_32ns_12ns_32_36	0	0	1610	1610
Total		2	0	0	3220	3220

Figura 4.1.6 Utilización de recursos de la operación módulo

Para recorrer las cachés de resultados de cuantificación de píxeles anteriores, se utilizan un puntero de lectura y uno de escritura que recorren un array secuencialmente, en forma de lista circular. Para que los punteros vuelvan al comienzo del array al llegar su final, se aplicó originalmente la función módulo a la posición del array a la que deben apuntar.

```
first_p = &cache[f_p % CACHE_LENGTH];
last_p = &cache[l_p % CACHE_LENGTH];
//donde f_p  l_p son enteros que se incrementan en cada llamada y
//CACHE_LENGTH es la longitud del array
```

Al sustituir estas operaciones por una reinicio de posición de vector al llegar al extremo del array, el uso de recursos se redujo considerablemente (Figura 4.1.7), llegando a un diseño implementable.

```
if(f_p == WIDTH)
   f_p = 0;
if(l_p == WIDTH)
   l_p = 0;
first_p = &cache[f_p];
last_p = &cache[l_p];
```

Resource utilization estimates for hardware accelerators

Resource	Used	Total	% Utilization
DSP	7	80	8,75
BRAM	13	60	21,67
LUT	5142	17600	29,22
FF	3623	35200	10,29

Figura 4.1.7 Reducción de uso de recursos

Cabe destacar el hecho de que para diagnosticar este problema fue necesario acudir al informe de *Vivado HLS*, pues con los de *SDSoC* no muestran ese nivel de detalle y no habría sido sencillo diagnosticar el causante del problema.

4.1.5 Introducción de pipeline

Una vez solucionado el problema de utilización de recursos, se ha trabajado para conseguir incrementar el rendimiento del acelerador. Dado que para procesar un píxel se necesitan conocer los resultados del procesamiento de píxeles anteriores, es decir, dado que hay dependencia entre píxeles, este algoritmo no es apto para un diseño en el que se procesen varios píxeles en paralelo. Se ha optado por tanto por implementar un diseño en pipeline con tres etapas: conversión del pixel a YUV, cálculo de sus hops, codificación Huffman de sus hops y escritura. En la Figura 4.1.8 se representa el esquema del pipeline que se desea hacer inferir al compilador.

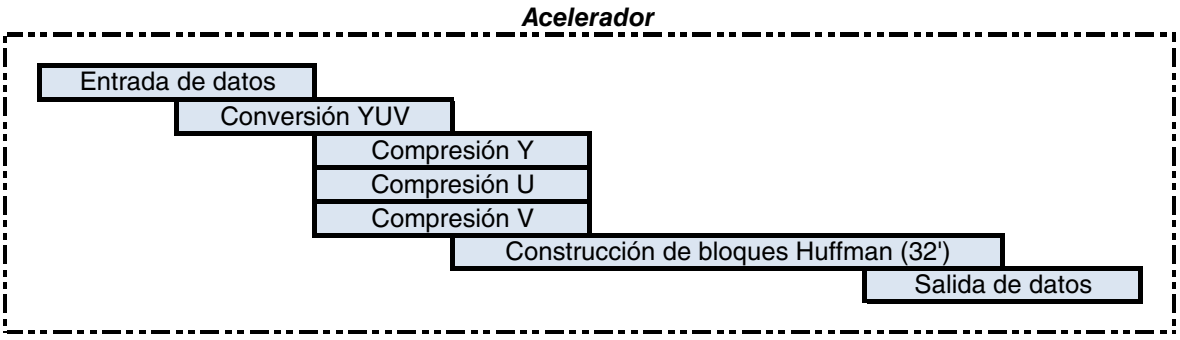

Figura 4.1.8 Esquema del pipeline deseado.

Originalmente se marcaron estas funciones para conversión a hardware de forma individual, corriendo el bucle principal que recorre la imagen y llama a estas funciones en el procesador. Para insertar un pipeline es necesario que este bucle esté en una función hardware, por lo que ha sido necesaria una refactorización del código, creando una función hardware que contiene el bucle principal y llama a las tres etapas del pipeline. Dado que al marcar una función para conversión a hardware se convierten también todas las funciones a la que esta llama, será suficiente con convertir esta función top level.

Dentro de esta nueva función top level, se introduce el pipeline escribiendo la directiva

```
#pragma HLS PIPELINE II=1
```

después de la declaración del nivel de bucle en el que se quiere introducir el pipeline. *SDSoC* intentará sintetizar un pipeline con un initiation interval igual al solicitado en la directiva (`II=1`), y si no se logra se implementará el del valor conseguido más cercano [13]. Se ha añadido además la directiva

```
#pragma HLS INLINE
```

en las funciones más simples (la de conversión a YUV y las de cálculo de hops), de forma que sean integradas de manera óptima en el pipeline [13].

4.1.6 Limitación de dimensiones de imagen

Para poder sintetizar el bucle principal en hardware, es necesario fijar el ancho y alto de la imagen de entrada, dado que se necesita conocer el número de iteraciones en tiempo de compilación. Esto reduce notablemente la flexibilidad del sistema, dado que el acelerador implementado solo podrá procesar imágenes de un único tamaño. No obstante, dado que el planteamiento original es que las imágenes de entrada sean las capturadas por una cámara en un dron, y por tanto tengan todas las mismas dimensiones, esto no debería suponer un problema. Para el resto de este proyecto se ha fijado un ancho de 1280 píxeles y un alto de 720 píxeles, dimensiones correspondientes al formato de **resolución 720P**.

Fijar las dimensiones de la imagen de entrada implica que pueden realizarse optimizaciones adicionales en otras funciones con dependencias con las dimensiones de la imagen de entrada, a costo de reducir su flexibilidad.

Para el cálculo de hops es necesario conocer la posición dentro de la imagen del píxel que se está procesando. Las dimensiones de la imagen y la posición del píxel a calcular se han calculado de manera interna en la función de cálculo de hops a partir de unas banderas *fin de línea* y *fin de archivo,* y del número de llamadas a esta función. Conociendo las dimensiones de la imagen en tiempo de compilación, es suficiente con el número de llamadas, pudiendo prescindir de las banderas y minimizando así tanto la transferencia de datos necesarios para llamar a estas funciones como los cálculos internos.

También necesitaba de la bandera *fin de archivo* la función que construye los bloques de escritura para saber cuándo hacer el rellenado del último bloque. Dado que esta función será llamada una vez por píxel, y se conoce la cantidad total de píxeles de la imagen en tiempo de compilación, es de nuevo suficiente con llevar la cuenta del número de llamadas para saber cuándo se está procesando el último píxel, pudiendo así prescindir totalmente de banderas.

Con la introducción del pipeline y la limitación de dimensiones de entrada se consigue una drástica mejora en aceleración en comparación con la primera implementación. En la Figura 4.1.9 se observa cómo el número de ciclos necesarios para la ejecución del programa se ha reducido prácticamente a la octava parte, y en la Figura 4.1.10 se muestra el **speedup de 4.37** obtenido para la función principal.

Performance estimates for 'main' function

SW-only (Measured cycles)	386354386
HW accelerated (Estimated cycles)	138181753
Estimated speedup	2,8

Figura 4.1.9 Aumento del rendimiento del programa por introducción de pipeline.

Performance estimates for 'encode_image in LHE_encoder_AL ...

SW-only (Measured cycles)	321831220
HW accelerated (Estimated cycles)	73658587
Estimated speedup	4,37

Figura 4.1.10 Aumento del rendimiento de la función principal por introducción de pipeline.

No obstante, la introducción del pipeline supone también un aumento del uso de recursos, como se observa en la Figura 4.1.11. Esto es debido a que, mientras que previamente se recorría el bucle en software y se pasaban píxeles solitarios a las funciones hardware, ahora se pasan los arrays completos de entrada y salida de datos a la función hardware top level. Por ello, se hace necesaria una lógica adicional en los data movers para permitir el acceso a arrays.

Resource utilization estimates for hardware accelerators

Resource	Used	Total	% Utilization
DSP	7	80	8,75
BRAM	15	60	25
LUT	6339	17600	36,02
FF	4206	35200	11,95

Figura 4.1.11 Aumento del uso de recursos por introducción de lógica de data movers.

4.1.7 Inclusión de funciones SDS para reserva de memoria contigua

Con el fin de simplificar los data movers creados entre el procesador y hardware, se puede guiar al compilador para que use Simple-DMA en vez de scatter-gather DMA situando los datos en memoria físicamente contigua [13]. Para reservar memoria físicamente contigua se sustituyen las funciones `malloc` y `free` por las funciones de la librería `sds_lib.h` de *SDSoC* `sds_alloc` y `sds_free`, respectivamente. Dado que la librería de procesamiento de imagen utiliza internamente la función `malloc` para reservar memoria para el array que devuelve con los datos de la imagen, es necesario recolocar dicho array en memoria físicamente contigua antes de pasarlo a la FPGA. Esto se resuelve sin necesidad de modificar

la librería copiando los datos del array devuelto por esta a otro array reservado con la función `sds_alloc` utilizando la función `memcpy`. Para indicar al compilador que los datos están situados en memoria física contigua, se añade la directiva

```
#pragma SDS data mem_attribute (arg: PHYSICAL_CONTIGUOUS)
```

donde `arg` es el argumento al que se quiere aplicar la directiva [13]. Es además necesario indicar al compilador el tamaño que tienen los arrays apuntados por los punteros que se pasan como argumentos. Esto se consigue incluyendo por cada array una directiva

```
#pragma SDS data copy(arg[0:size])
```

donde `arg` es el nombre del argumento y `size` el tamaño del array apuntado [13].

Se puede optimizar aún más el acceso a datos si se indica que estos van a ser accedidos de forma secuencial, de modo que no se sintetice la lógica adicional de comunicación necesaria para accesos aleatorios. Esto se consigue ajustando el patrón de acceso de datos a modo secuencial mediante la directiva

```
#pragma SDS data access_pattern(arg:SEQUENTIAL)
```

donde `arg` es el argumento al que se quiere aplicar la directiva. De esta manera se consigue una interfaz de datos en forma de stream en vez de una interfaz RAM [13].

Con la inferencia de unos data movers más eficientes se logra una reducción del tamaño del diseño (Figura 4.1.12).

Resource utilization estimates for hardware accelerators

Resource	Used	Total	% Utilization
DSP	7	80	8,75
BRAM	13	60	21,67
LUT	4545	17600	25,82
FF	2411	35200	6,85

Figura 4.1.12 Reducción de uso de recursos por uso de streaming de datos.

Se puede comprobar en la pestaña de directivas de *Vivado HLS* que, efectivamente, *SDSoC* ha inferido unos puertos de entrada y salida de datos FIFO y ha incluido las directivas necesarias en el proyecto de *Vivado HLS* (Figura 4.1.13).

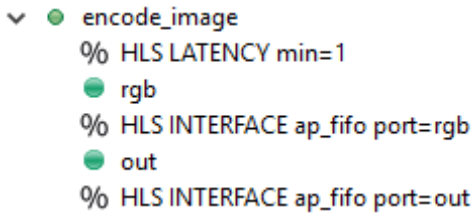

Figura 4.1.13 Inferencia de puertos FIFO

4.1.8 Paralelización de lectura de tabla de saltos

La forma original en la que se ha implementado el cálculo de hops, mediante bucles que van buscando el salto más adecuado, se sintetiza en un hardware muy lento debido a la dependencia entre iteraciones de dichos bucles que impiden su desenrollamiento y paralelización.

Para tratar de acelerar este cálculo sin cambiar completamente la implementación del algoritmo, se ha tratado de reducir el número de accesos a memoria que se realizan. En vez de realizar un acceso al hop necesario en cada iteración del bucle, se cargan en un array auxiliar los 8 posibles saltos, además del salto nulo, al principio de la llamada. Para que este array auxiliar se rellene en un solo ciclo, se ha escrito un bucle que rellena cada posición del array y se ha desenrollado utilizando la directiva

```
#pragma HLS unroll
```

Luego, para leer esas 8 lecturas puedan realizarse en paralelo en un solo ciclo, se ha realizado un particionado del array de hops, de forma que cada uno de los 8 posibles saltos se encuentra en una memoria separada. Para hacer este particionado se ha introducido la directiva

```
#pragma HLS ARRAY_PARTITION variable=caches_t factor=8 dim=3
```

con la que se indica que se ha de particionar en 8 la dimensión 3 (la que contiene los 8 saltos) del array `caches_t [13]`.

Adicionalmente, dado que a estas memorias accederán las funciones de los tres canales en paralelo, se ha indicado al compilador que las memorias sean ROM de doble puerto para reducir el uso de recursos mediante la directiva

```
#pragma HLS RESOURCE variable=caches_t core=ROM_2P_BRAM
```

En el manual de usuario de *SDSoC* no se cubre el uso de este tipo de directivas avanzadas de HLS, pero su uso permite guiar al compilador con una mayor precisión y obtener un acelerador más optimizado.

Dentro de los bucles, para obtener el hop más cercano se calcula en cada iteración la diferencia entre el valor a codificar con uno de los posibles saltos leídos. Todos esos cálculos también se han sacado del bucle, calculando y guardando todas las posibles diferencias necesarias en un segundo array auxiliar al principio de la llamada, también de forma paralela mediante desenrrollamiento. De esta manera en cada iteración no hay más que leer datos de estos arrays auxiliares, y se consiguen unas aceleraciones mucho más atractivas, llegando a **una estimación de speedup de 16.39** para la función principal (Figura 4.1.14, Figura 4.1.15) sin un coste de recursos adicional (Figura 4.1.16).

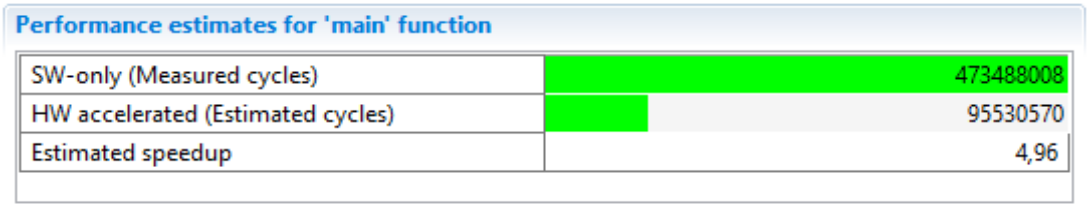

Figura 4.1.14 Rendimiento global del programa.

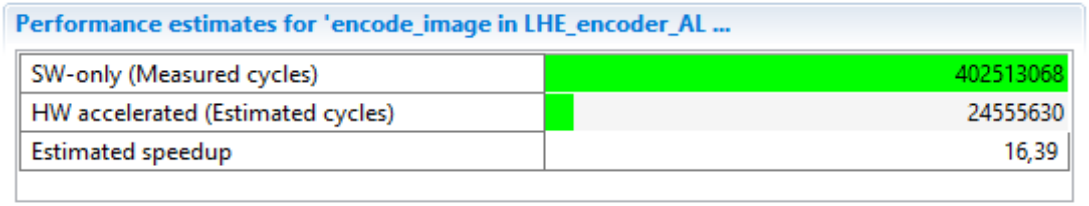

Figura 4.1.15 Rendimiento de la función marcada para conversión hardware.

Resource utilization estimates for hardware accelerators

Resource	Used	Total	% Utilization
DSP	7	80	8,75
BRAM	13	60	21,67
LUT	4954	17600	28,15
FF	1811	35200	5,14

Figura 4.1.16 Utilización de recursos.

Además de conseguir acelerar el cálculo de los hops, al haber reducido el tiempo necesario para cada iteración se consigue minimizar la diferencia de tiempos necesarios para el peor y el mejor caso.

4.1.9 Paralelización de transferencia de canales

Al comprobar en el reporte de *Vivado HLS* del IP creado la forma en que se han inferido los canales de transmisión de datos, se advierte que se están realizando tres transferencias (una por cada canal) de datos de pixel por llamada (Figura 4.1.17).

	Operation\Control Step	C0	C1	C2	C3	C4	C5
11	col_mid2(select)						
12	rgb_read(read)						
13	tmp_138(|)						
14	tmp_146(icmp)						
15	col_3(+)						
16	rgb_read_1(read)						
17	tmp_134(*)						
18	rgb_read_2(read)						
19	tmp_130(*)						

Figura 4.1.17 Triple transferencia de datos por píxel.

Para tratar de reducir las transferencias de datos, transmitiendo los tres canales del píxel en una sola transacción, se ha hecho un empacamiento a 4 bytes de la estructura Pixel utilizada y un casting del array que contiene los datos de la imagen.

```
typedef struct __attribute__ ((packed, aligned(4))) Pixel{
   uint8_t value1; //R or Y or its hop
   uint8_t value2; //G or U or its hop
   uint8_t value3; //B or V or its hop
}Pixel;
```

Con esto se ha logrado efectivamente que efectivamente se haga una sola lectura por píxel (Figura 4.1.18), pero no se observa una aceleración dado que las tres lecturas anteriores se diluían en el pipeline durante la ejecución de la función que convierte RGB a YUV que es donde se encuentra el cuello de botella de la latencia del bucle principal (Figura 4.1.19).

Hasta el momento de la escritura de esta memoria no se ha encontrado la forma de escribir la función de conversión a YUV de modo que se sintetice en menos ciclos, pero sí se ha encontrado un IP de *Xilinx* que realiza esta operación en un solo ciclo [12], por lo que debería ser posible optimizarla.

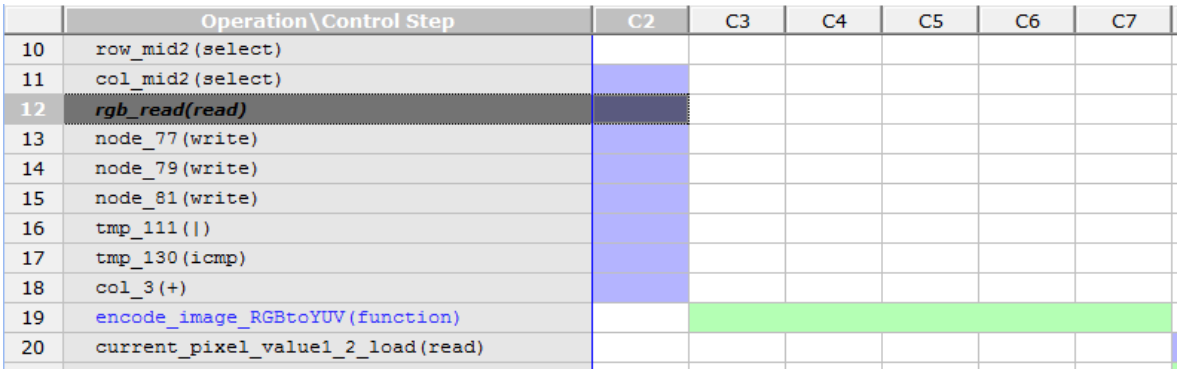

Figura 4.1.18 Única lectura de píxel.

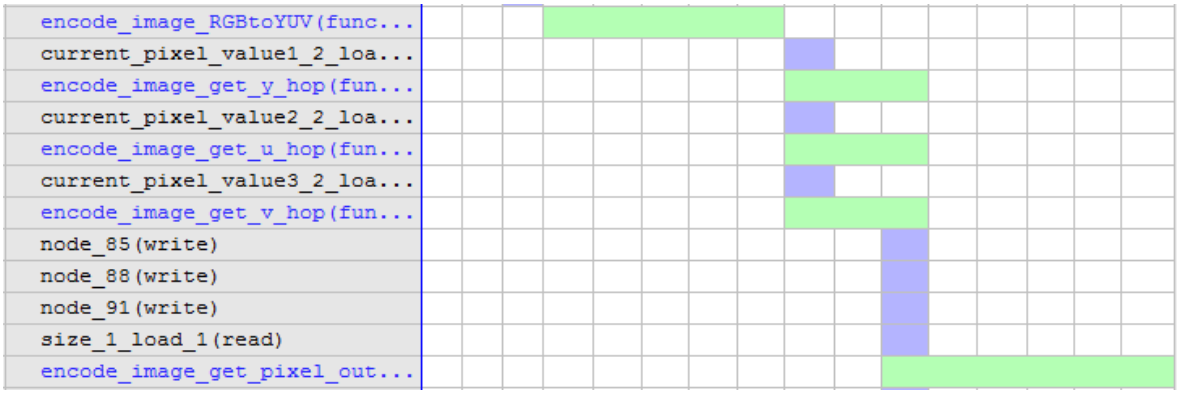

Figura 4.1.19 Cuello de botella en el pipeline.

4.1.10 Prueba de instanciación múltiple

En el cálculo de hops hay múltiples variables que deben ser guardadas entre iteraciones y que son independientes para cada canal, que han sido implementadas como variables estáticas. Para mantener esta independencia sin complicar el código, se han creado tres copias idénticas con distintos nombres de la función que calcula los hops (una por cada canal).

En hardware podría resolverse este problema con un código más sencillo haciendo tres instanciaciones independientes de la misma función. Se ha intentado hacer una instanciación por canal de una única función de cálculo de hops mediante la directiva de *SDSoC* `#pragma async (id)`, que permite hacer llamadas a funciones hardware de forma asíncrona y esperar sus resultados con la directiva `#pragma wait (id)`. Según la el manual de usuario de *SDSoC*, estas directivas se pueden utilizar para guiar al compilador a crear múltiples instanciaciones de la función [13].

```
#pragma SDS async(1)
   yuv->value1 = get_y_hop(yuv->value1);
#pragma SDS async(2)
   yuv->value2 = get_y_hop(yuv->value2);
#pragma SDS async(3)
   yuv->value3 = get_y_hop(yuv->value3);
#pragma SDS wait(1)
#pragma SDS wait(2)
#pragma SDS wait(3)
```

Sin embargo, al probar a utilizarlas se observa una caída en la estimación de aceleración (Figura 4.1.20), por lo que se intuye que se está infiriendo algo diferente. Al consultar el manual de usuario de versiones posteriores de *SDSoC* se observa que se ha dejado de recomendar el uso de estas directivas para paralelización, y que ha pasado a recomendarse la

directiva `#pragma SDS resource (id)` (que no es reconocida por el compilador de *SDSoC* 2016.2), por lo que es posible que esta funcionalidad haya sido arreglada en versiones posteriores, o bien que la recomendación de estas directivas para este fin fuesen una errata.

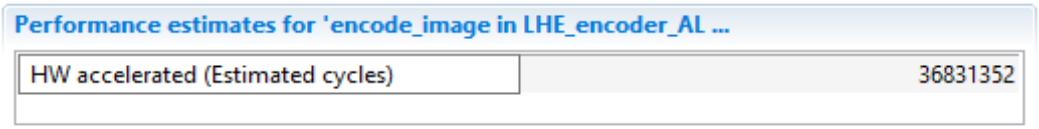

Figura 4.1.20 Desaceleración en instanciación múltiple.

Debido a la escasa documentación, no se ha podido profundizar más en esta prueba y finalmente se han mantenido las tres copias de la función. Una posible solución de cara a mejorar la legibilidad reduciendo esta replicación de código sería mantener una sola copia genérica y utilizar tres funciones envoltorio que contuviesen las variables estáticas de cada canal, aunque existen dudas sobre la forma en que *SDSoC* traduciría tal estrategia a hardware.

4.2 CONCLUSIONES

Dado que las pruebas descritas en 4.1.9 y 4.1.10 no han mejorado el rendimiento del sistema, el diseño finalmente implementado es el descrito hasta 4.1.8. Dado que la estimación de aceleración que hace *SDSoC* se calcula mediante la comparación de un mismo diseño en software y hardware, puede resultar una medida engañosa. Lo correcto será comparar los menores tiempos obtenidos para el diseño software más rápido (4.1.6) y el diseño hardware más rápido (4.1.8), de la siguiente manera (Figura 4.2.1):

Performance estimates for 'main' function	
SW-only (Measured cycles)	386354386
HW accelerated (Estimated cycles)	95530570
Estimated speedup	4,04

Performance estimates for 'encode_image' function	
SW-only (Measured cycles)	321831220
HW accelerated (Estimated cycles)	24555630
Estimated speedup	13,11

Figura 4.2.1 Estimación final de aceleración

Al haber fijado el tamaño de las imágenes de entrada a 1280 x 720 (4.1.6), las dimensiones correspondientes al formato de resolución 720P, las imágenes utilizadas para las pruebas de compresión y descompresión y medidas de tiempo y rendimiento dejan de ser las presentadas en el capítulo 3 para ser fotografías de estas dimensiones. Se ha utilizado una imagen con todos sus píxeles en color blanco (*blank*), que será la más sencilla de comprimir para el algoritmo, y tres fotografías que, en orden ascendente de complejidad, son: *sensor* (Figura 4.2.2), *komainu* (Figura 4.2.3) y *kannon* (Figura 4.2.4), todas ellas tomadas por el autor de este trabajo.

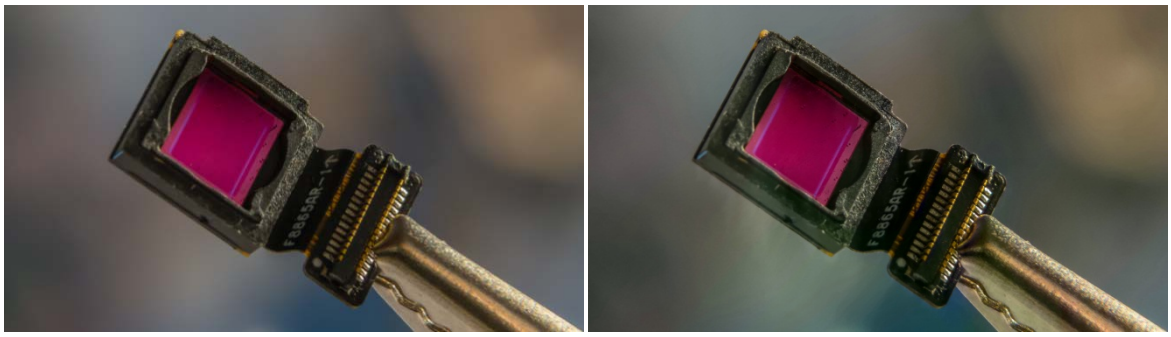

Figura 4.2.2 Imagen "*sensor*" original y descomprimida.

Figura 4.2.3 Imagen "*komainu*" original y descomprimida.

Figura 4.2.4 Imagen "*kannon*" original y descomprimida.

Este orden ascendente de complejidad de las imágenes, que se puede observar en las tasas de compresión obtenidas para cada una de ellas (Tabla 4.2.2) y que tiene que ver con las componentes de alta frecuencia de las imágenes, será importante más adelante para comparar el intervalo de confianza de distintitas implementaciones del algoritmo.

	Imagen				
	blank	*sensor*	*komainu*	*kannon*	*Media*
Tasa de compresión	7,99999	6,79734	5,99571	4,49243	6,32137

Tabla 4.2.2 Tasa de compresión obtenida para imágenes de resolución 720P.

En las pruebas de compresión se observan algunos errores de color en las imágenes decodificadas, especialmente en *kannon* (Figura 4.2.4). Esto debe tratarse de una inconsistencia entre codificador y decodificador, que deberá ser solucionada en el futuro desarrollo del sistema.

5 RENDIMIENTO

5.1 MEDICIÓN Y COMPARACIÓN DE TIEMPOS DE COMPRESIÓN

Para medir los tiempos de compresión de las distintas implementaciones y plataformas descritas en esta memoria, se han utilizado funciones de cronometraje (función `time.time` en Python y función `clock` en C). Para cada implementación se han realizado múltiples pruebas de compresión de las imágenes presentadas en la sección 4.2, tomando la media de tiempo de compresión de cada imagen y la media total de cada plataforma (Tabla 5.1.1).

	Imagen				
Tiempo de compresión (s)	*blank*	*sensor*	*komainu*	*kannon*	*Media*
Python	27,4783	27,7690	28,4660	29,1963	28,2274
Intel i5 6300-HQ @ 2,30GHz	0,0363	0,0413	0,0470	0,0573	0,0455
Dual-core ARM Cortex-A9	0,5504	0,6031	0,6194	0,6455	0,6046
Zynq XC7Z010	0,0484	0,0484	0,0485	0,0485	0,0485

Tabla 5.1.1 Tiempos de compresión y medias para cada plataforma e imagen de prueba.

Como se adelantó en la sección 3.4, los tiempos de la traducción a C respecto del código original en Python muestran **una mejora tres órdenes de magnitud**. Las compresiones de las implementaciones en ambos lenguajes han sido ejecutadas en un procesador Intel i5 6300-HQ @ 2,30GHz, bajo el sistema operativo *Windows 10 Educational*.

Al ejecutar el código mejorado en C en la plataforma *ZynqBerry* exclusivamente en software (plataforma *Dual-core ARM Cortex-A9* en la Tabla 5.1.1) los tiempos se ralentizan un orden de magnitud respecto a los obtenidos en el *Intel*, debido a la gran diferencia de capacidad de proceso entre ambos (el procesador embebido en el *Zynq XC7Z010* es un *Dual-core ARM Cortex-A9 MPCore* [16] . Esta diferencia es la que habrá de eliminar el acelerador hardware del sistema híbrido diseñado (plataforma *Zynq XC7Z010* en la Tabla 5.1.1), y puede comprobarse que se ha conseguido. Ambos diseños en *ZynqBerry* corren bajo la distribución de *Linux* de *Xilinx*, *Petalinux*.

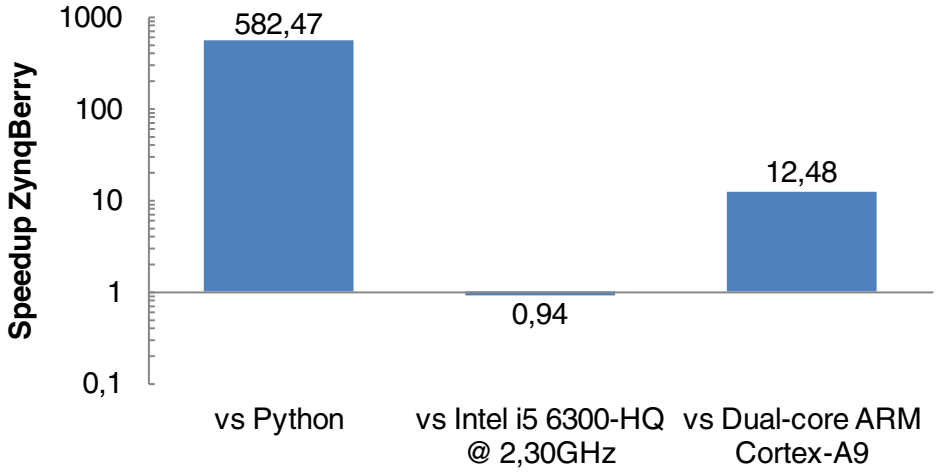

Figura 5.1.1 Speedup del diseño híbrido respecto a los otros diseños; representación semilogarítmica.

En la Figura 5.1.1 se muestran las aceleraciones del sistema híbrido respecto de las demás plataformas: **un speedup de 582.47** respecto al código original en Python, **un speedup de 0.94** respecto a la traducción en C corriendo en *Intel*, y **un speedup de 12,48** en comparación

con el software corriendo en la misma plataforma, una medida muy cercana a la que se calculó en la Figura 4.2.1 en base a las estimaciones de rendimiento de *SDSoC*.

Aunque el sistema híbrido diseñado tiene una media de tiempos de compresión mayor a la del procesador *Intel* utilizado para las imágenes de prueba que se han utilizado, como se adelantó en la sección 4.1.8, al incluir paralelización en la función de cálculo de hops, reduciendo el número de ciclos utilizados para accesos a memoria, la diferencia de tiempos de compresión entre imágenes sencillas y complejas en el sistema híbrido es minúscula. En la Figura 5.1.2 se comparan los tiempos de compresión para cada imagen y las medias del sistema híbrido (*ZynqBerry*) y el microprocesador, donde se puede apreciar que la dispersión de tiempos es mucho mayor en el *Intel* y que **para imágenes más complejas es más eficiente el sistema diseñado**.

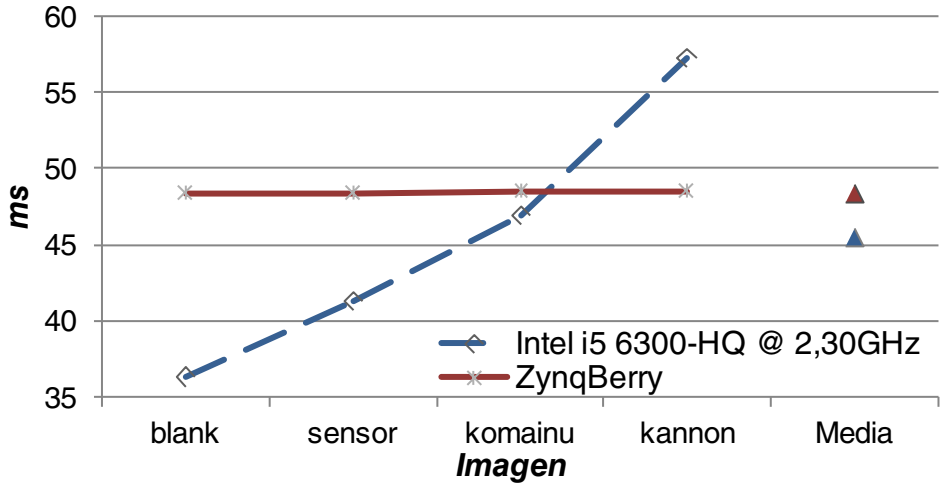

Figura 5.1.2 Comparación de dispersiones de tiempo de compresión Intel – *ZynqBerry*.

Este **mejorado intervalo de confianza** permitirá que el empaquetamiento y transmisión de la información se pueda realizar en intervalos regulares, resultando en un sistema con un comportamiento mucho más predecible.

Modificando el sistema actual incluyendo la lógica necesaria para acceder a distintos fotogramas, sería posible utilizar el acelerador sintetizado para el procesamiento de vídeo. Dado que se ha calculado una media de tiempo de compresión por imagen de 0,0485 segundos (Tabla 5.1.1), se calcula que **se podrán procesar 20 fps**.

A parte del uso de *timers*, *SDSoC* proporciona una serie de herramientas adicionales para la comprobación de sistemas. En primer lugar, ofrece la posibilidad de introducir módulos tanto en software como en hardware que, durante el tiempo de ejecución de la aplicación, envían trazas a través del puerto serie y de la conexión Ethernet a *SDSoC*, que las recoge y muestra en un gráfico temporal permitiendo al diseñador comprobar qué procesos se están realizando en qué instantes y encontrar cuellos de botella. Además, también ofrece la posibilidad de insertar un *AXI Performance Monitor* (*APM*), que registra la actividad de los data movers, reuniendo valiosas estadísticas sobre el funcionamiento de la red de movimiento de datos. Por desgracia, aunque sí se han podido probar estas herramientas en diseños más sencillos y tutoriales, *SDSoC* ha sido incapaz de insertar ninguno de estos módulos en el diseño de este proyecto, resultando en todos los casos en accesos ilegales a memoria. Se ha tratado de portar el diseño a una versión más moderna de la herramienta (2016.4), con la esperanza de que estas funcionalidades hubiesen sido reparadas, pero la falta de retrocompatibilidad lo ha imposibilitado. Es por ello que las únicas formas de validar el diseño han sido las estimaciones de aceleración de *SDSoC*, los informes de síntesis de *Vivado HLS*, y las iniciativas del autor a nivel programación (`timers` y `printf`).

5.2 OPENCV

Como se adelantó en la subsección Librería stb_image, originalmente se quiso hacer este diseño haciendo uso de la librería para procesamiento de imagen *OpenCV*. Esto no ha sido finalmente posible por la dificultad que supone el uso de librerías dinámicas en diseños de *SDSoC*. Tanto en versiones anteriores de la herramienta (*SDSoC* 2015.4) como en versiones de la herramienta relacionada *SDK*, se soporta de manera nativa el uso de la librería *OpenCV*, encontrándose las versiones precompiladas de la librería tanto para arquitectura x86/x86-64 como para procesadores ARM en sus directorios de instalación. Sin embargo en versiones más recientes, y en concreto en la utilizada para este proyecto (2016.2), este soporte ha desaparecido.

El problema principal en su inclusión manual radica en la dificultad y escasa documentación del procedimiento necesario para la utilización de librerías dinámicas en *SDSoC*. Para poder hacer uso de ellas, estas se tienen que encontrar incluidas dentro de archivo de especificación de plataforma de *SDSoC* para la que se está compilando el diseño, para lo cual es necesario regenerar este, un proceso complejo que se documenta en una guía separada [15] . Tratándose de una plataforma de código cerrado, es posible que esta regeneración sea imposible de realizar sin obtener archivos e información adicionales por parte del fabricante, de quien se ha obtenido el archivo de especificación de plataforma que se ha utilizado ya generado.

6 CONCLUSIONES Y FUTURO TRABAJO

6.1 CONCLUSIONES

Los objetivos que se marcaron en el primer capítulo (1.2) fueron los siguientes:

1. Análisis de LHE y traducción del código Python a lenguaje C.
2. Estudio de *SDSoC* y modificación del código para conseguir un sistema sintetizable.
3. Análisis de las limitaciones del código para su aceleración en hardware.

El primer objetivo, cubierto en el capítulo 3, no solo se ha logrado, habiendo conseguido una traducción a C funcional del algoritmo básico, sino que se ha ido más allá y se ha optimizado la aplicación, obteniendo **tiempos de compresión tres órdenes menores** en la traducción C que en la versión original en Python (Tabla 3.4.1).

El segundo objetivo, como se ha descrito en el capítulo 4, también ha sido completado, consiguiendo un diseño sintetizable e implementable, y aún más allá, **se ha conseguido un diseño con una aceleración considerable**, como se muestra en el capítulo 5. Es por ello que el tercer objetivo no se ha cumplido como tal, puesto que no se ha realizado un análisis de limitaciones de cara a una implementación futura, sino que se ha completado de forma integrada en el proceso del segundo objetivo, no solo analizando las limitaciones sino yendo un paso más allá y además solucionándolas.

Podría concluirse que el resultado del proyecto ha excedido los objetivos planteados en su comienzo, y que *SDSoC* ha superado las expectativas originales, permitiendo a un diseñador inexperto (el autor) llegar a un diseño acelerado. No obstante, se ha de tener en cuenta que ha habido otros factores que tampoco fueron previstos en el planteamiento inicial. En este se propuso seguir la metodología de diseño ofrecida por *SDSoC*, una metodología orientada a desarrolladores software no expertos en el uso de HLS. Sin embargo el estudio que se ha hecho sobre HLS, tanto a nivel de pautas de programación como de directivas, ha excedido el propuesto por la documentación de *SDSoC*, e incluso se han utilizado herramientas más avanzadas como son las de *Vivado HLS*. Esto ha sido posible gracias a la experiencia compartida por los estudiantes del laboratorio *HPCN-UAM*, de la oportunidad de participar en un seminario sobre herramientas de *Xilinx* que cubrió el uso de *Vivado HLS* [3] , y de la oportunidad de participar en una reunión del proyecto *Racing Drones*.

6.1.1 Sobre LHE

Si se utilizase en un codificador de vídeo el compresor LHE desarrollado, y este fuese el elemento de la cadena de proceso más lento, se calcula que **podrían transmitirse 20 fps en calidad 720P** (más atrás). Esto es una tasa de refresco menor a la que ofrecen las tecnologías de vídeo analógico que se están utilizando en drones (25~30 fps para PAL/NTSC), por lo que **no supone una mejora**. Sin embargo, ha de recordarse que: el algoritmo implementado es una versión básica; que no está siguiendo las estrategias de procesamiento de vídeo de LHE; que no se está utilizando ningún esquema de submuestreo de crominancia; que no se está aplicando ninguna estrategia de paralelización de píxeles; que existen soluciones para la conversión RGB a YUV [12] que tardan menos ciclos; que se conocen métodos de implementación del cálculo de hops más eficientes que el utilizado (más adelante). Por estos motivos, **puede esperarse que una siguiente implementación sobrepase las tasas de refresco de los sistemas analógicos** y llegue a tener una latencia lo suficientemente pequeña para permitir la conducción remota de drones a alta velocidad, o, por lo menos, no suponer la latencia limitante en la cadena de transmisión.

Por ello, se cree que este trabajo ha conseguido aportar una prueba de concepto positiva para la utilización del algoritmo LHE en sistemas de vídeo digital de baja latencia.

6.1.2 Sobre SDSoC

No son pocos los problemas encontrados a lo largo del desarrollo de este trabajo con la herramienta *SDSoC*.

En primer lugar, se han experimentado distintas inestabilidades por parte de la herramienta, como diversos bugs entre los que se incluye la confusión de perfiles de ejecución/debug de los distintos proyectos dentro del workspace, o inconsistencias en los resultados de compilación. También se han encontrado herramientas que no han funcionado debidamente (5.1), siendo necesario acudir a las herramientas de *Vivado HLS* en múltiples ocasiones. Ha sido imposible, además, hacer funcionar correctamente el programa en un sistema operativo distinto de *Windows*. Se ha probado en *Ubuntu 14.04*, *Ubuntu 16.04* y *CentOS 7*, con resultados muy negativos tanto por falta de soporte como por falta de documentación.

Las de instalación no son las únicas carencias en la documentación de *SDSoC*. A pesar de la gran importancia que tiene el diseño de la red de movimiento de datos en el proceso de diseño de sistemas híbridos, la información que se ofrece sobre los distintos tipos de puertos, interfaces y data movers que puede generar *SDSoC* es escasa y poco detallada.

Las descripciones de las plataformas de *SDSoC* conforman una gran parte del trabajo de síntesis que hace el compilador para llegar a generar el diseño de un sistema completo. Entre otras cosas, contienen las librerías que se pueden utilizar, el sistema operativo que se monta en la plataforma, y el sistema de archivos. A pesar de tener un papel tan central, se les ha dado poco protagonismo tanto en la documentación como en la herramienta. En próximas actualizaciones será necesario que se puedan modificar con más facilidad y libertad estas plataformas, dado que las posibilidades de manipulación del sistema operativo y de utilización de librerías dinámicas son de gran importancia para el diseño de sistemas Linux empotrados, especialmente para el desarrollador software acostumbrado a disponer de mayor libertad.

Se echan en falta funcionalidades de las que sí dispone *Vivado HLS*, como la generación de directivas guiada por menús, la posibilidad de crear múltiples configuraciones de directivas para un mismo proyecto (*solutions*) y el detalle en los informes de síntesis. Sería además muy productiva la posibilidad de regenerar solo el IP del acelerador (la parte que se genera en *Vivado HLS*) al modificar un proyecto en *SDSoC* y volver a compilar, dado que el proceso completo de compilación de *SDSoC* consume grandes cantidades de tiempo.

Durante el tiempo transcurrido en el desarrollo de este trabajo, *Xilinx* ha lanzado cuatro versiones diferentes de la herramienta (2016.1, 2016.2, 2016.4, 2017.1). Estas actualizaciones no han tenido un impacto positivo en el desarrollo del proyecto, ya que la total falta de retrocompatibilidad imposibilita portar los proyectos a una versión más actualizada. Incluso, ha tenido un efecto negativo, dado que la documentación no se ha actualizado con la misma frecuencia, y muchos tutoriales en la documentación muestran pantallas y opciones que no se corresponden con la versión utilizada.

En general, el uso de *SDSoC* ha requerido tiempo para una curva de aprendizaje tanto sobre la herramienta como del método correcto de escritura de código, tiempo para la resolución de problemas relacionados con errores del propio software y de su falta de soporte, y tiempo para la experimentación y aprendizaje por ensayo y error de los múltiples aspectos no documentados. Este tiempo necesitado ha levantado la duda sobre la supuesta ventaja de productividad ofrecida por *SDSoC*, y sobre si habría sido más rápido implementar el algoritmo desde cero partiendo de su descripción y utilizando herramientas de un nivel más bajo (pero con un mayor conocimiento y experiencia de desarrollo de hardware). Esta pregunta, que

invita a una comparación objetiva de tiempos de desarrollo y calidad de resultados entre distintas herramientas, habrá de ser abordada en futuros trabajos.

Sin embargo, y a pesar de todos estos problemas, ***SDSoC* funciona**. Se ha conseguido sintetizar e implementar un complejo sistema híbrido consistente en una distribución Linux, un código software compilado de forma cruzada para ARM, un acelerador en FPGA, y una red de movimiento de datos entre el acelerador y el procesador, por parte de un diseñador novel y en el tiempo de un semestre. Esto difícilmente habría sido posible con la utilización de otro tipo de herramientas. *SDSoC* está lejos de ser una herramienta completamente automatizada al alcance de un desarrollador software sin conocimientos de hardware, y su uso ha supuesto para el autor un intenso aprendizaje, no solo sobre la propia herramienta y otras de *Xilinx*, sino también de multitud de técnicas de optimización de software y hardware, de diseño de sistemas y de aceleración de algoritmos en general. Por ello, aunque en la actualidad se encuentran en una fase muy inicial en su desarrollo, el futuro de herramientas como *SDSoC* se ofrece prometedor.

Prometedor para el desarrollador de software, para el prototipado rápido y exploración de arquitecturas. Por el contrario, el uso de un lenguaje de programación, como es C/C++, para el desarrollo de hardware se presenta un tanto incómodo para el diseñador con experiencia en lenguajes de descripción de hardware (HDL) como VHDL. Al tratarse de un lenguaje concebido para otra cosa, en muchas ocasiones, y a pesar de conocer las pautas dictadas por *Xilinx*, se duda sobre la forma adecuada de escribir código para llegar al diseño deseado. Muchos conceptos de programación, como el de los punteros, se vuelven confusos cuando el código que se está escribiendo es destinado a ser inferido en un diseño hardware. También se echan en falta, por ejemplo, tipos de entero de longitud variable a nivel bit, y flexibilidad para realizar operaciones con buses de bits. En definitiva, para el desarrollador de hardware la metodología de diseño planteada por *SDSoC* resulta imprecisa, y se echan en falta librerías e instrucciones de mayor precisión y orientadas a hardware que acerquen C/C++ a un HDL, como las que ofrece *Vivado HLS*.

6.2 *FUTURO TRABAJO*

6.2.3 Modificaciones del código y del algoritmo

Una posible mejora que se podría añadir sin demasiada dificultad al sistema implementado es la de submuestreo de crominancia, que, como se relata en la subsección 3.2.1, fue eliminado al hacer la traducción a lenguaje C. La inclusión de esta funcionalidad, a parte de la evidente mejora en la compresión, supondría una aceleración adicional. Esta aceleración se produciría en las funciones convertidas a hardware que no han sido paralelizadas por canales (la de conversión a YUV y la de construcción de bloques codificados), y en la reducción de transacciones de datos tanto a la entrada como a la salida del acelerador. Reinsertar esta funcionalidad requerirá, como se comenta en la Definición del formato de archivo .lhe, de la inclusión de un nuevo campo en los archivos comprimidos que indique al decodificador el esquema de submuestreo utilizado en compresión.

Dado que tanto la compresión como la decodificación con este algoritmo de un píxel dependen de píxeles anteriores, la pérdida de un paquete o un error durante la transmisión de la imagen comprimida podría tener unas consecuencias fatales. Por ello, se propone enviar un píxel sin comprimir, de la misma forma que el primero de cada imagen, a intervalos de cierta frecuencia para tratar de minimizar este problema. La forma de indicar cuándo se transmite un píxel no codificado, la frecuencia de estas transmisiones y si su carácter es estático o dinámico, habrán de estudiarse en trabajos futuros.

El principal cuello de botella del diseño obtenido se encuentra en la función de conversión de espacios de color. Como se comenta en 4.1.9, existen implementaciones de este cálculo que consiguen realizar la conversión en tan solo dos ciclos, mientras que este diseño está tardando cinco. Se ha de encontrar la forma de guiar al compilador o de reescribir esta función de forma que se infiera un diseño paralelizado.

Aunque se estima que añadiendo el submuestreo de crominancia y optimizando la función de conversión de espacio de color se podría llegar a una tasa de compresión superior a 30 fps, se propone una modificación que permitiría reducir el tiempo de compresión obtenido. Dado que la utilización de recursos del acelerador (Figura 4.1.16) es lo suficientemente pequeño como para permitir la instanciación de múltiples aceleradores, una posible aceleración es la que se obtendría al dividir las imágenes de entrada en un número de franjas igual al de los aceleradores instanciados y comprimir dichas franjas en paralelo. Esta estrategia se está utilizando implementada por *Texas Instruments* para compresión de video de baja latencia en drones pilotados por FPV [8] (donde se refieren a estas franjas como *slices*).

Como se comentó en la sección 3.4, existen dudas sobre si habría sido más eficiente escribir el código desde cero, partiendo de la descripción del algoritmo. Existe una forma más rápida de efectuar al cálculo de hops que la que se implementó en el código original Python y que se ha respetado en este proyecto. Esta consiste en utilizar rangos de cuantificación en vez de los valores absolutos de cuantificación. Esto es, en vez de comparar con todos los posibles valores de cuantificación hasta encontrar el que tiene un error más pequeño con el valor a cuantificar, utilizar una tabla precalculada que contenga los puntos medios entre estos valores (la frontera de sus rangos) y sencillamente comparar con estos, calculando directamente a qué rango corresponde el dato de entrada.

Por otro lado, en futuros trabajos se habrán de implementar las funciones más avanzadas del algoritmo, la versión que se conoce como LHE avanzado [4] . También se propone en la publicación original del algoritmo una arquitectura que permite paralelización en el procesamiento de píxeles a través de una lectura en diagonal [5] , que potencialmente podría acelerar el sistema enormemente. No obstante, por los resultados de requerimiento de recursos obtenidos en este proyecto, se prevé que el tamaño de tal diseño sería demasiado grande. Concretamente, para una imagen 720P se necesitarían 720 aceleradores similares al presentado en este proyecto, que de tener también un tamaño similar, supondrían que el diseño total llegase a necesitar un 3700.8% del recurso menos utilizado (FF). Aún con una FPGA de mayor tamaño, no parece que esta sea una arquitectura viable, aunque sí podría adaptarse para procesar N líneas, donde N es el número de aceleradores que caben en la FPGA.

Por último, si se quiere utilizar finalmente este algoritmo para la compresión de vídeo, será necesario estudiar la descripción de la versión para vídeo de LHE [4] , puesto que la que se ha implementado en este proyecto es la versión para imagen estática.

Aunque es un aspecto que no se ha planteado durante el desarrollo de este trabajo, sería de vital importancia realizar un estudio sobre el consumo de energía de los sistemas sintetizados por *SDSoC*, y su relación speedup/consumo. Si el objetivo de estos sistemas de compresión de imagen es ser implementados en drones, para los que se trata de montar baterías del menor tamaño posible y obtener la mayor autonomía posible, en especial en el caso de los drones de carreras, es fundamental que la ventaja de latencia que proporcione este sistema no suponga un costo excesivo de consumo energético.

6.2.4 Utilización de otras herramientas

Durante el desarrollo de este proyecto han salido al mercado cuatro versiones diferentes de la herramienta *SDSoC* (2016.1, 2016.2, 2016.4, 2017.1). Dada la alta frecuencia con la que se han lanzado nuevas versiones, es posible que muchos de los errores, carencias, falta de soporte y escasez de documentación encontrados en la versión utilizada (2016.2) hayan sido solucionados en versiones posteriores. Sería deseable probar a realizar este diseño con estas versiones modernas. Durante el desarrollo de este proyecto se intentó actualizar la herramienta a la versión siguiente (2016.4), sin embargo esto no fue posible debido a la falta de retrocompatibilidad entre versiones, tanto a nivel de proyecto como a nivel de sintaxis de directivas.

Sería además interesante, para poner la supuesta ventaja de productividad ofrecida por *SDSoC* en perspectiva, implementar este diseño con otras herramientas de síntesis de hardware. Especialmente la implementación en *Vivado HLS*, de cuyas directivas de compilación e informes de síntesis se ha hecho uso en este proyecto. También, durante este año 2017 *Xilinx* ha presentado un conjunto de recursos dirigido específicamente al diseño de hardware para visión artificial y aprendizaje automático: *reVISION Stack* [17] , herramienta con la que sería atractivo experimentar.

Principalmente, en trabajos futuros se habrá de encontrar una herramienta que permita y facilite el uso de librerías dinámicas, pues se podría conseguir un diseño de mayor optimización utilizando librerías especializadas de procesamiento de imagen (OpenCV).

7 REFERENCIAS

[1] Alonso Pugliese, T. (2017). *LHE_decode*. HPCN-UAM.

[2] Barret, S. (n.d.). *stb_image.h*. [online] GitHub. Available at: https://github.com/nothings/stb/blob/master/stb_image.h.

[3] Electra Training (2017). *Seminario Xilinx SDx: SDSoC, SDNet, SDAccel, HLS, IPI, QUEMU*.

[4] García Aranda, J. (2015). *Nuevo algoritmo de compresión multimedia: codificación por saltos logarítmicos*.

[5] García Aranda, J., Cao Cueto, M., Navarro Salmerón, J., González Casquete, M. and González Vidal, F. (2015). Logarithmical hopping encoding: a low computational complexity algorithm for image compression. IET Image Processing, 9(8), pp.643-651.

[6] Hpcn-uam.es. (2017). *Racing Drones (Programa Retos-Colaboración)*. [online] Available at: http://www.hpcn-uam.es/project/racing-drones/.

[7] Sipi.usc.edu. (n.d.). *SIPI Image Database*. [online] Available at: http://sipi.usc.edu/database.

[8] Texas Instruments (2016). *Low-latency design considerations for video-enabled drones*.

[9] Trenz Electronic (2017). *TE0726 Resources - Public Docs - Trenz Electronic Wiki*. [online] Available at: https://wiki.trenz-electronic.de/display/PD/TE0726+Resources.

[10] Trenz Electronic (2017). *TE0726 TRM*.

[11] Xilinx (2014). *Vivado Design Suite User Guide: High-Level Synthesis (UG902)*.

[12] Xilinx (2015). *RGB to YCrCb Color-Space Converter v7.1: LogiCORE IP Product Guide (PG013)*.

[13] Xilinx (2016). *SDSoC Environment User Guide (UG1027)*.

[14] Xilinx (2016). *SDSoC Environment User Guide: An Introduction to the SDSoC Environment (UG1028)*.

[15] Xilinx (2016). *SDSoC Environment User Guide: Platforms and Libraries (UG1146)*.

[16] Xilinx (2017). *Zynq-7000 All Programmable SoC Data Sheet: Overview (DS190)*.

[17] Xilinx.com. (2017). *reVISION Zone | Machine Learning | Computer Vision*. [online] Available at: https://www.xilinx.com/products/design-tools/embedded-vision-zone.html.

8 GLOSARIO

ACP	Accelerator Coherence Port
AFI	Asinchronous FIFO Interface
APM	AXI Performance Monitor
CPU	Central Processing Unit
DMA	Direct Memory Access
DSP	Digital Signal Processor
FF	Flip-Flop
FPGA	Field Programmable Gate Array
fps	frames per second
FPV	First-Person View
GPx	General-Purpose IO, donde x es 0 o 1
HDL	Hardware Description Language
HLS	High Level Synthesis
HPx	High Performance ports, donde x es un número de 0 a 3
LHE	Logarithmical Hop Encoding
LUT	Look Up Table
MPSoC	MultiProcessor System-on-Chip
PS	Processing System
SDSoC	Software Defined System-on-Chip
SoM	System on Module

ANEXOS

A. CÓDIGO FUENTE DEL DISEÑO HÍBRIDO FINAL

A continuación se adjunta el código fuente del diseño híbrido final (la versión refactorizada para *SDSoC*). No se adjunta el código de la librería `stb_image.h`, que puede consultarse en [2] . Además, se ha omitido en esta memoria la declaración de la tabla de hops, que se extiende más de 2300 líneas. Esta puede generarse siguiendo la descripción del algoritmo en [4] . Se han omitido también las declaraciones de las funciones de cálculo de hops de los canales U y V, por ser copias idénticas de la función para el canal Y. El código se forma, a parte de la librería `stb_image.h`, de tres ficheros: `main.c`, `LHE_encoder_AL.h` y `LHE_encoder_AL.c`.

main.c

```c
#include "LHE_encoder_AL.h"

//AL stands for Angel Lopez

int main(int argc, char** argv) {
    if(!LHE_encode(argv[1], argv[2])){
       printf("ERROR: Could not encode image.\n");
       return 0;
    }
    return 1;
}
```

LHE_encoder_AL.h

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <time.h>
#include <stdint.h>
#include <sds_lib.h>
#include "stb_image.h"

#ifndef SRC_LHE_ENCODER_AL_H_
#define SRC_LHE_ENCODER_AL_H_
#endif /* SRC_LHE_ENCODER_AL_H_ */

typedef struct Pixel{
    uint8_t value1; //R or Y or its hop or its value (for first pixel)
    uint8_t value2; //G or U or its hop or its value (for first pixel)
    uint8_t value3; //B or V or its hop or its value (for first pixel)
}Pixel;

#define WIDTH 1280
#define HEIGHT 720

/*TOP LEVEL FUNCTION*/
uint8_t LHE_encode(char const * input_file, char const * output_file);

/*READING*/
uint8_t read_image(char const * input_file, uint8_t ** rgb, int * width,
    int * height);
```

```
/*ENCODING*/
uint32_t encode_image(uint8_t rgb[(WIDTH*HEIGHT*3)], uint32_t * out, Pixel
    * first_pixel);
void RGBtoYUV(Pixel * rgb);
void get_pixel_hops(Pixel * yuv);
uint8_t get_y_hop(uint8_t input);
uint8_t get_u_hop(uint8_t input);
uint8_t get_v_hop(uint8_t input);
void get_pixel_output(Pixel current_pixel_hops, uint32_t * out, uint32_t *
    size);

/*WRITING*/
uint8_t write_output(char const * output_file, uint32_t * out, uint32_t
    size, Pixel first_pixel);
```

LHE_encoder_AL.c

```
#include "LHE_encoder_AL.h"

#ifndef STB_IMAGE_IMPLEMENTATION
    #define STB_IMAGE_IMPLEMENTATION
    #include "stb_image.h"

    #ifndef STBI_ASSERT
        #define STBI_ASSERT(x)
    #endif
#endif

//NOTA: Se ha omitido la declaración de la tabla de hops en esta memoria
    por su enorme extensión.

//Warning: Huge declaration (over 2300 lines)
static const uint8_t caches_t[256][7][8] = {
    //|h-4|h-3|h-2|h-1|h1 |h2 |h3 |h4 |
    {                                           // prediction: 0
      {  0,  0,  0,  0,  4, 10, 25, 62}, // h1= 4
      {  0,  0,  0,  0,  5, 12, 31, 78}, // h1= 5
      {  0,  0,  0,  0,  6, 15, 37, 93}, // h1= 6
      {  0,  0,  0,  0,  7, 17, 43,109}, // h1= 7
      {  0,  0,  0,  0,  8, 20, 50,125}, // h1= 8
      {  0,  0,  0,  0,  9, 22, 56,140}, // h1= 9
      {  0,  0,  0,  0, 10, 25, 62,156}  // h1= 10
    },
    //              (...)
    {                                           // prediction: 255
      {193,230,245,251,255,255,255,255}, // h1= 4
      {177,224,243,250,255,255,255,255}, // h1= 5
      {162,218,240,249,255,255,255,255}, // h1= 6
      {146,212,238,248,255,255,255,255}, // h1= 7
      {130,205,235,247,255,255,255,255}, // h1= 8
      {115,199,233,246,255,255,255,255}, // h1= 9
      { 99,193,230,245,255,255,255,255}  // h1= 10
    }
};
```

```
uint8_t LHE_encode(char const * input_file, char const * output_file){
/* Takes an input image and outputs the .lhe file of the compressed image
 * -Arguments:
 *     input_file: path/name of input image
 *     output_file: path/name of output .lhe file
 * -Returns:
 *     0: failure
 *     1: success
 *  -Target: SW
 */
    uint8_t * rgb;
    uint8_t * rgb_sds;
    uint32_t * out;
    int width, height;
    uint32_t size;
    Pixel first_pixel;

    //load RGB values of every pixel
    if(!read_image(input_file, &rgb, &width, &height)){
        printf ("ERROR: Could not load image.\n");
        return 0;
    }

    //check fixed dimensions for input image
    if(width != WIDTH || height != HEIGHT){
        printf ("ERROR: Image with unsupported dimensions.\n");
        return 0;
    }

    //auxiliary array for contiguous memory allocation
    rgb_sds = (uint8_t*) sds_alloc((WIDTH * HEIGHT * 3) *
    sizeof(uint8_t)); //more than enough memory for worst case scenario
    if(rgb_sds == NULL){
        printf ("ERROR: Memory allocation unsuccessful.\n");
        return 0;
    }

    //copy the image data into contiguous memory
    memcpy (rgb_sds, rgb, (WIDTH * HEIGHT * 3));

    //contiguous memory allocation for output data
    out = (uint32_t*) sds_alloc((2 * sizeof(uint16_t))+((WIDTH * HEIGHT *
    3) * sizeof(uint8_t)));
    if(out == NULL){
        printf ("ERROR: Memory allocation unsuccessful.\n");
        return 0;
    }

    //print info.
    printf("Input Image:\t%s\n",input_file);
    printf("\tWidth: %d\tHeight: %d\n",width, height);
    printf("Output File:\t%s\n",output_file);

    clock_t start = clock(), diff; //for performance measurement

    //  IMAGE COMPRESSION  //
    size = encode_image(rgb_sds, out, &first_pixel);

    //Performance measurement
    diff = clock() - start;
    double sec = (double)diff / (double)CLOCKS_PER_SEC;
```

```
    printf("\nTime taken: %f seconds (%d ticks, %d ticks per sec)\n", sec,
    diff, CLOCKS_PER_SEC); //debug

    if(!write_output(output_file, out, size, first_pixel)){
        printf ("ERROR: Could not write output file.\n");
        return 0;
    }

    free(rgb);
    sds_free(rgb_sds);
    sds_free(out);

    printf("\nOK\n"); //debug

    return 1;
}
uint8_t read_image(char const * input_file, uint8_t ** rgb, int * width,
    int * height){
/* Takes an input image and outputs its RGB values in a 1D array (R G B R
 *     G B ...). Also gets the width and height.
 * -Arguments:
 *     input_file: path/name of input image
 *     rgb: 1D array containing the RGB values of all the pixels of the
 *         image (R G B R G B ...)
 *     width: width of image in pixels
 *     height: height of image in pixels
 * -Returns:
 *     0: failure
 *     1: success
 * -Target: SW
 */
    int  comps;

    *rgb = stbi_load(input_file, width, height, &comps, 3); //3 for RGB (4
    also reads alpha)
    if (*rgb == NULL){
        return 0;
    }

    return 1;
}
```

```
#pragma SDS data copy(rgb[0:(WIDTH*HEIGHT*3)])
#pragma SDS data copy(out[0:((WIDTH*HEIGHT*3)/4)])
#pragma SDS data mem_attribute (rgb:PHYSICAL_CONTIGUOUS)
#pragma SDS data mem_attribute (out:PHYSICAL_CONTIGUOUS)
#pragma SDS data access_pattern(rgb:SEQUENTIAL)
#pragma SDS data access_pattern(out:SEQUENTIAL)
uint32_t encode_image(uint8_t rgb[(WIDTH*HEIGHT*3)], uint32_t * out, Pixel
    * first_pixel){
/* Performs LHE algorithm. Main HW function!
 * -Arguments:
 *     rgb: 1D array containing the RGB values of all the pixels of the
 *         image (R G B R G B ...)
 *     out: 1D array where 32 bit output blocks are written
 *     first_pixel: values of first pixel, not coded
 * -Returns:
 *     size: Number of 32 bit blocks in out.
 * -Target: HW
 */
   uint32_t size = 0;
   uint16_t row, col;
   Pixel current_pixel;

   for(row = 0; row < HEIGHT; row++){
      for(col = 0; col < WIDTH; col++){
      #pragma HLS PIPELINE II=1

         //load current pixel
         current_pixel.value1=rgb[((row*WIDTH)+col)*3];     //R
         current_pixel.value2=rgb[((row*WIDTH)+col)*3 + 1]; //G
         current_pixel.value3=rgb[((row*WIDTH)+col)*3 + 2]; //B

         //convert RGV to YUV
         RGBtoYUV(&current_pixel);

         //get the pixel's hops
         get_pixel_hops(&current_pixel);

         //output pixel
         if(row == 0 && col == 0){
            //first pixel is not encoded
            first_pixel->value1 = current_pixel.value1; //Y
            first_pixel->value2 = current_pixel.value2; //U
            first_pixel->value3 = current_pixel.value3; //V
         }else{
            //get the output coded in Huffman
            get_pixel_output(current_pixel, out, &size);
         }
      }//col
   }//row

   return size;
}
```

```
void RGBtoYUV(Pixel * rgb){
/* Converts one pixel's RGB values to YUV values (without using float
 *     logic!)
 * -Arguments:
 *     rgb: 1D array containing the RGB values of all the pixels of the
 *     image (R G B R G B ...)
 *         the output YUV values are overwritten in this array
 * -Target: HW
 */

    Pixel aux;

    aux.value1 = ((77 * rgb->value1 + 150 * rgb->value2 + 29 * rgb-
    >value3) >> 8)&0xFF;           //Y
    aux.value2 = 128 + (((- 43 * rgb->value1 - 85 * rgb->value2 + 127 *
    rgb->value3) >> 8)&0xFF); //U
    aux.value3 = 128 + (((127 * rgb->value1 - 107 * rgb->value2 - 21 *
    rgb->value3) >> 8)&0xFF);  //V

    rgb->value1 = aux.value1; //Y
    rgb->value2 = aux.value2; //U
    rgb->value3 = aux.value3; //V
}

void get_pixel_hops(Pixel * yuv){
/* Takes the YUV values of a pixel and calculates the hop number for each
 *     channel (Y, U and V).
 * -Arguments:
 *     yuv: 1D array containing the YUV values of one pixel.
 *         The output hop numbers are overwritten in this structure.
 * -Target: HW
 */

    yuv->value1 = get_y_hop(yuv->value1); //Y hop
    yuv->value2 = get_u_hop(yuv->value2); //U hop
    yuv->value3 = get_v_hop(yuv->value3); //Vhop
}

uint8_t get_y_hop(uint8_t input){
#pragma HLS RESOURCE variable=caches_t core=ROM_2P_BRAM
#pragma HLS ARRAY_PARTITION variable=caches_t factor=8 dim=3
/* Takes the Y values of a pixel and calculates its hop number.
 * -Arguments:
 *     input: Y value of a pixel.
 * -Returns:
 *     hop_number: identifier of the Y hop calculated
 * -Target: HW
 */
    //statics
    static uint16_t col = 0, row = 0; //current pixel position, calculated
    within this function
    static uint8_t cache[WIDTH]; //circular list with quantized value of
    last (width) pixels
    static uint8_t * first_p; //pointer to first element of cache (read
    pointer)
    static uint8_t * last_p; //pointer to last element of cache (write
    pointer)
    static uint16_t f_p = 0, l_p = 0; //pointer position in cache
    static uint8_t hop1 = 7; //= start_hop1
    static bool last_small_hop = false;
    static bool small_hop = false;
```

```
    static uint8_t previous;
    //consts
    const uint8_t min_hop1 = 4;
    const uint8_t max_hop1 = 10;
    const uint8_t start_hop1 = 7; //= (max_hop1+min_hop1)/2; //start in
    the center of the interval

    uint8_t hop_number = 4; // Pre-selected hop -> 4 is null hop
    uint8_t hop0 = 0; // Predicted luminance signal
    int j;

    // Initial error values
    int emin = 256; // Current minimum prediction error
    int e2 = 0; // Computed error for each hop
    uint8_t finbuc = 0; // We can optimize the code below with this
    //Parallelization
    uint8_t aux_line[9];
    int error[9];

    //circular list pointer logic
    if(f_p == WIDTH)
        f_p = 0;
    if(l_p == WIDTH)
        l_p = 0;
    first_p = &cache[f_p];
    last_p = &cache[l_p];


    if(col != 0 || row != 0){ //if not first pixel
        last_small_hop = small_hop;

    // HOP0 PREDICTION //
        // If we are not in a border, we need the previous pixel and the
    upper-right one.
        //if (row > 0 && col > 0 && col != width - 1 && col != width){
        if (row > 0 && col > 0){
            hop0 = (*first_p+previous)/2;

        // If we are in the beginning of a row, we reset Hop1
        }else if (col == 0 && row > 0){
            hop0 = *first_p;
            last_small_hop = false;
            hop1 = start_hop1;

        // If we are in the first row, the hop (from 0 to 256) will be the
    result of the previous pixel
        }else if (row == 0 && col > 0){
            hop0 = previous;
        }

      // HOPS COMPUTATION  //
        //all hops read in parallel
        for(j = 0; j < 9; j++){
        #pragma HLS unroll
            if (j < 4)
                aux_line[j] = caches_t[hop0][(hop1-4)][j];
            else if(j == 4)
                aux_line[j] = hop0;
            else
                aux_line[j] = caches_t[hop0][(hop1-4)][j-1];
        }
```

```
        //all diferences calculated in parallel
        for(j = 0; j < 9; j++){
        #pragma HLS unroll
            if(input >= hop0)
                error[j] = input - aux_line[j];
            else
                error[j] = aux_line[j] - input;
        }

        if(input == hop0){ //if null hop, no need for further calculation
            hop_number = 4;
            previous = input;
        }else{
            emin = 256;
            e2 = 0;
            finbuc=0;

          // POSITIVE hops computation
            if(input >= hop0){
                for(j = 4; j < 9; j++){
                    //Prediction error
                    e2 = error[j];

                    if(e2 < 0){
                        e2 = - e2;
                        finbuc = 1; //If error is negative, we got the hop
                    }
                    if(e2 < emin){
                        previous = aux_line[j];
                        hop_number = j; // Hop assignment
                        emin = e2;
                        if(finbuc == 1) //This avoids useless iterations
                            break;
                    }else
                        break;
                }

          // NEGATIVE hops computation.
            }else{
                for(j = 4; j > -1; j--){
                    e2 = error[j];
                    if(e2 < 0){
                        e2 = - e2;
                        finbuc = 1;
                    }
                    if(e2 < emin){
                        previous = aux_line[j];
                        hop_number = j;
                        emin = e2;
                        if(finbuc == 1)
                            break;
                    }else
                        break;
                }
            }
        }

        // Save quantized value to cache
        *last_p = previous;
```

```
        //h1 adaptation
        small_hop = false;
        if (hop_number <= 5 && hop_number >= 3)
            small_hop = true;
        else
            small_hop = false;

        if (small_hop && last_small_hop){
            hop1 = hop1-1;
            if (hop1 < min_hop1)
                hop1 = min_hop1;
        }else{
            hop1 = max_hop1;
        }

        if(col != 1){ //second pixel of each row is not used, so is
    overwritten by third
            l_p++; //update pointer to last
        }
        //Update pointer to first
        //  pointer to first should not be updated on first pixel, nor on
    second to last of each row
        //  and it is not needed on first row
        if(row != 0 && (col != WIDTH - 2))
            f_p++;
    }else{ //if first pixel
        previous = input;
        *last_p = input; //first pixel's hops are always null hop, so
    result = input
        l_p++; //always update pointer to last
        hop_number = input;
    }

    col++; //increment column position
    if(col == WIDTH){
        row++; //increment row position
        col = 0; //back to first pixel in row

        if(row == HEIGHT){ //reset statics
            col = 0;
            row = 0;
            f_p = 0;
            l_p = 0;
            hop1 = 7;
            small_hop = false;
            last_small_hop = false;
        }
    }

    return hop_number;
}


uint8_t get_u_hop(uint8_t input){
    //NOTA: Se ha omitido la declaración de esta función en esta memoria
    por ser una copia idéntica de la función get_y_hop.
}
uint8_t get_v_hop(uint8_t input){
    //NOTA: Se ha omitido la declaración de esta función en esta memoria
    por ser una copia idéntica de la función get_y_hop.
}
```

```
#pragma SDS data copy(out[0:((WIDTH*HEIGHT*3)/4)])
#pragma SDS data copy(size[0:1])
void get_pixel_output(Pixel current_pixel_hops, uint32_t * out, uint32_t *
    size){
/* Builds and outputs memory blocks joining the Huffman codes of the hop
    numbers passed.
 * An output block is NOT written every time the function is called.
 * After outputting the last pixel, the last memory block (if not empty)
    is padded and output.
 * -Arguments:
 *     current_pixel_hops: hop numbers of the three channels of a pixel.
 *     out: 1D array where memory blocks are written.
 *     size: number of blocks written in out.
 * -Target: HW
 */

    //Huffman codes for each hop
    static const uint8_t huff_table[]={0x1A, 0x3A, 0x02, 0x04, 0x01, 0x00,
    0x06, 0x0A, 0x5A};
    //Length of each code
    static const int huff_length[]={7,6,4,3,1,3,3,5,7};

    /*****************************************************************
     *                        HOPS' HUFFMAN CODES                    *
     *****************************************************************
     * Hop    Hop Number       Huffman code     Probability (%)        *
     *   0           4         1'               42,5                   *
     *   1           5         000'             18,83333333            *
     *  -1           3         001'             16,16666667            *
     *   2           6         011'             8                      *
     *  -2           2         0100'            7,166666667            *
     *   3           7         01010'           3,233333333            *
     *  -3           1         010111'             3                   *
     *   4           8         0101101'         0,7333333333           *
     *  -4           0         0101100'         0,3666666667           *
     *****************************************************************/

    static uint64_t memory1 = 0; //buffer
    static uint8_t bit_counter = 0; //number of bits in memory1
    static uint32_t countdown = (WIDTH * HEIGHT) - 1; //when countdown ==
    0 -> pad last block
    static uint32_t size_count = 0; //number of blocks output

    memory1 |= ((uint64_t)huff_table[current_pixel_hops.value1]) <<
    (bit_counter); //add to the memory block
    bit_counter += huff_length[current_pixel_hops.value1]; //add to the
    count

    memory1 |= ((uint64_t)huff_table[current_pixel_hops.value2]) <<
    (bit_counter); //add to the memory block
    bit_counter += huff_length[current_pixel_hops.value2]; //add to the
    count

    memory1 |= ((uint64_t)huff_table[current_pixel_hops.value3]) <<
    (bit_counter); //add to the memory block
    bit_counter += huff_length[current_pixel_hops.value3]; //add to the
    count

    if(bit_counter >= 32){ //if there is a complete block
        out[size_count] = (uint32_t)memory1; //output it
        size_count++;
```

```
        memory1 >>= 32; //shift buffer
        bit_counter -= 32;
    }
    countdown--; //countdown pixels until end of image

    if(!countdown){ //if all pixels have been coded
        if(bit_counter){ //and there are bits in the buffer
            out[size_count] = (uint32_t)memory1; //pad & write last block
            size_count++;
        }
        //reset statics
        memory1 = 0;
        bit_counter = 0;
        *size = size_count;
    }
}

uint8_t write_output(char const * output_file, uint32_t * out, uint32_t
    size, Pixel first_pixel){
/* Writes to an output file the output data of get_pixel_output.
 * Called once after image is compressed.
 * -Arguments:
 *     output_file: path/name of output .lhe file
 *     out: 1D array where memory blocks are written.
 *     size: number of blocks written in out.
 *     first_pixel: uncompressed values of first pixel
 * -Returns:
 *     0: failure
 *     1: success
 *  -Target: SW
 */
    FILE * f_out;
    uint16_t width = WIDTH;
    uint16_t height = HEIGHT;

    //open the file binary write mode
    f_out = fopen(output_file, "wb");
    if (f_out == NULL){
        return 0;
    }

    //Write image size
    fwrite(&width, sizeof(uint16_t), 1, f_out);
    fwrite(&height, sizeof(uint16_t), 1, f_out);

    //write first pixel YUV values
    fputc(first_pixel.value1, f_out); //Y
    fputc(first_pixel.value2, f_out); //U
    fputc(first_pixel.value3, f_out); //V

    //write encoded hops
    fwrite(out, sizeof(uint32_t), size, f_out);

    //close file
    if (fclose(f_out) != 0){
        return 0;
    }
    return 1;
}
```

B. COMPATIBILIDAD DE ZYNQBERRY CON RASPBERRY-PI CAMERA

Con el objetivo de poder realizar una demostración del sistema diseñado en este trabajo, comprimiendo imágenes capturadas directamente en la plataforma *ZynqBerry* a través de una cámara y mostrando en una pantalla, a través de su puerto HDMI, el resultado de la descompresión (corriendo el decodificador en el procesador exclusivamente), se estudió la posibilidad de utilizar el módulo de cámara digital para *Raspberri-Pi: Raspberry-Pi Camera Module.* Esta iniciativa se vio además motivada por el hecho de que el fabricante *Trenz Electronic* proporciona una aplicación de ejemplo similar, que captura imágenes a través de este módulo y las muestra por pantalla (sin someterlas a ningún procesado). Al estar *ZynqBerry* basada en *Raspberri-Pi Model 2,* sus módulos son físicamente compatibles.

Se decidió adquirir originalmente la última revisión del módulo: *Raspberri-Pi Camera Module V2.1*, que dispone de una cámara de 8 megapíxeles mientras que la primera revisión dispone de solo 5, pero no se consiguió hacer funcionar. El motivo es que la cámara que incorpora esta revisión es una *Sony IMX219*, mientras que la que incorpora el módulo original es una *OmniVision OV5647*. La aplicación de ejemplo de *Trenz Electronic* funciona con un código que controla el funcionamiento de la cámara escribiendo directamente en los registros de esta a través de los puertos GPIO correspondientes. Al tratarse de un modelo y fabricante diferente de cámara, tanto los registros, como las instrucciones, como los puertos GPIO son diferentes, por lo que la cámara de SONY resulta incompatible con esta aplicación. Se trató de modificar dicho código para poder utilizar esta cámara, pero toda la información técnica sobre este modelo está bajo un acuerdo de confidencialidad (*Non Disclosure Agreement*) y no puede ser consultada. La distribución de Linux de *Raspberri-Pi*, *Raspbian*, dispone de unas librerías que son capaces de interactuar con esta cámara, por lo que podrían portarse dichas librerías al *Petalinux* de *ZynqBerry* y manejar la cámara a través de estas. La complejidad actual de modificar las plataformas de *SDSoC* impidió que se pudiese aplicar esta solución, por lo que finalmente se montó la *OmniVision OV5647* de la revisión 1.3, cuya documentación es pública, que se pudo hacer funcionar sin problemas.