FPGA Based Implementation of ImageZero Compression Algorithm

L. Leiva, M. Vázquez, M. Tosini, O. Goñi, and J. Noguera

***Abstract*—The main goal of information compression is to achieve the greatest reduction possible of the volume of data. This action affects both storage and transmission. However, the most used algorithms have considerable latency and are not practical for real-world applications. In the image compression field, the situation is more complex because the resolutions are constantly increasing. Nowadays, it is common to deal with uncompressed images that exceed 60 MB. Dedicated compressors present strong restrictions of time, space and power so a thorough design and development should be made. In this article, an architecture of *ImageZero* compression lossless algorithm is presented. The algorithm was selected for its low latency both in encoding and decoding. The novelty of this work is the hardware implementation for this algorithm, based on FPGA. The synthesis results demonstrate that an increase in input image resolution does not affect the compression speedup. Also, the throughput for this proposal is greatest than other hardware implementations of image compressors.**

***Index Terms*—Image Compression, ImageZero, FPGA.**

I. Introducción

La compresión de imágenes comenzó a usarse desde hace ya muchos años con la popularización de dispositivos capaces de captar y almacenar imágenes y videos. Los algoritmos de compresión de imágenes generan una representación más compacta de la imagen con el menor tamaño posible sin perder información esencial. Los métodos de compresión pueden clasificarse en: sin pérdida o reversibles (*lossless compression*) y con pérdida o irreversibles (*lossy compression*) [1]. La alternativa con pérdida es usada cuando la compresión (y posterior descompresión) de la imagen se realiza para ser percibida por el ojo humano y no es necesaria gran calidad de imagen. Técnicamente, estos algoritmos no mantienen la integridad de la información por lo que los datos descomprimidos son aproximados a los originales. La mayor ventaja del procesamiento con pérdida es la alta tasa de compresión lograda (a costa de pérdida de información que el ojo humano igualmente no percibirá). Los algoritmos de compresión de imágenes sin pérdida son usados habitualmente para transmitir o almacenar imágenes sin pérdida de información de pixeles o de color. Específicamente, mantienen la integridad de la información puesto que los datos descomprimidos coinciden con los datos originales. Al contrario de los algoritmos con pérdida estos últimos presentan bajos porcentajes de compresión, ya que no pueden descartar ninguna información de la imagen original como lo hacen los primeros.

La mayoría de los algoritmos de compresión sin pérdida se emplean en medicina, para el almacenamiento de imágenes con calidad igual a como fueron captadas por los instrumentos radiológicos [2]; almacenamiento y transmisión de imágenes capturadas por vehículos aéreos (satélites, globos atmosféricos) [3], [4] y aplicaciones de monitoreo remoto [5], como la vigilancia de incendios forestales [6].

Los métodos de compresión sin pérdida se dividen a su vez en estadísticos -o de codificación basada en entropía- y basados en diccionario, y en ambos casos pueden basarse en a) modelos estáticos, usan un modelo universal que no cambia; b) semiestáticos, arman el modelo basados en los datos de entrada; y c) dinámicos o adaptativos que actualizan el modelo al leer cada nuevo dato. Los modelos difieren principalmente en: el número de pasadas necesarias por los datos para la compresión, y la complejidad y costo computacional de los métodos.

Los tipos de modelos de compresión principales son los métodos basados en entropía, como Shannon, Huffman (estático y dinámico), codificación aritmética; y los métodos basados en diccionario, como la familia de algoritmos LZ (Lempel-Ziv)[7]: LZ77, LZ78, pkzip, ARJ, winrar, PNG, Run-Length Encoder (RLE), etc.

En el *benchmark* [8] se evalúan diferentes compresores usando un conjunto de imágenes común. Los resultados señalan que el algoritmo *ImageZero* (IZ) [9] posee una tasa de compresión competitiva frente a otras propuestas y un rendimiento superior, tanto en la compresión como en la descompresión, superando a PNG [10] o JPEG-LS [11][12].

El algoritmo IZ pertenece al conjunto de compresores de imágenes sin pérdida, basados en entropía y de tipo estático con un modelo de tipo Huffman con distribución de probabilidades fija (una tabla de probabilidades independiente de los datos de entrada). Está diseñado para ser utilizado en fotos naturales en espacio de colores RGB de 24 bits. El algoritmo utiliza un predictor de 3 píxeles, y los datos son codificados teniendo en cuenta un valor de contexto único, utilizado para codificar los píxeles a partir de las tablas estáticas mencionadas.

En general los algoritmos de compresión de datos, imágenes y videos trabajan en software sobre procesadores de uso general y como *codecs* de programas específicos que usan o generan información comprimida. Además de la compresión de espacio, en muchos casos se requiere que la información sea generada o transmitida en tiempo real para poder satisfacer las demandas de las aplicaciones que la usan [3], [4], [6]. Cuando

L. Leiva, M. Vázquez, M. Tosini, O. Goñi, and J. Noguera are with the LabSET/INTIA, Department of Computers and Systems, UNICEN, Tandil, 7000 Argentina e-mail: (see http://www.labset.exa.unicen.edu.ar/).

L. Leiva and M. Vázquez are with of Computers Engineering, UNTREF, Buenos Aires, Argentina.

se considera la implementación de algoritmos sin pérdida para satisfacer la demanda de velocidad de procesamiento, se deben tener en cuenta: a) minimización del número de pasos de proceso y reducción del uso de memoria y consumo; b) la habilidad del algoritmo de operar en una pasada; y c) la habilidad del algoritmo de operar con dispositivos hardware determinados. En estos casos el uso de soluciones hardware es efectivo, especialmente considerando el uso hardware programable, y ha aportado la base en los últimos años para la implementación de soluciones de alta velocidad.

A diferencia de las CPUs y las GPUs, los recursos de un FPGA pueden ser configurados para crear *pipelines* de instrucciones específicos para el problema a resolver. Si bien operan a frecuencias de reloj más bajas y tienen picos de rendimiento inferiores, permiten adaptar el hardware para cada aplicación particular. En este sentido, el uso de FPGAs no siempre representa la mejor opción en términos de rendimiento, pero generalmente son la mejor alternativa en cuanto a eficiencia energética [13].

En este trabajo se aborda la implementación de una arquitectura para FPGA del algoritmo IZ. La principal contribución del trabajo, según conocimiento de los autores, es que es la primera implementación de este algoritmo en hardware. Asimismo, los resultados obtenidos demuestran que la arquitectura presentada supera la velocidad de compresión de otras arquitecturas de compresores de imágenes desarrolladas para FPGAs. Se establece como límite sólo la compresión de datos (o *encoder*). La sección 2 detalla algunos trabajos relacionados de implementaciones de compresores de imágenes sobre hardware específico, y en la sección 3 se realiza una descripción del algoritmo IZ detallando las ventajas consideradas para su selección. En la sección 4 se presenta en detalle la arquitectura, y en la sección 5 los resultados experimentales de síntesis. Finalmente, se exponen las conclusiones y líneas futuras de trabajo en la sección 6.

II. Trabajos Previos Relacionados

En los últimos años varios autores han propuesto diversas soluciones efectivas de algoritmos tradicionales de compresión de imágenes y alternativas de los mismos con resultados aceptables implementados en tecnologías de hardware, tanto FPGA como ASIC.

En el trabajo presentado por Rusyn et al. [5] se hace un extenso análisis de métodos conocidos implementados en FPGA y comparando sus ventajas y desventajas según parámetros de tasa de compresión, velocidad de procesamiento, consumo y otros. En la presentación se analizan varios métodos clásicos tales como RLE, Huffman, felics, LZW, LZMA, entre otros. Las variantes de compresión se prueban en tiempo real sobre dispositivos Spartan 6 y Virtex 6 de Xilinx con buenos resultados en velocidad para algunos de los algoritmos.

El trabajo [14] propone una variante específica del algoritmo *Fast, Efficient, Lossless Image Compression System* (FELICS) llamada VLSI-oriented FELICS, que tiene como ventaja la no dependencia de datos. El trabajo presentado implementa el algoritmo en una arquitectura en dos *pipelines* paralelos de cuatro etapas que clasifican dos píxeles consecutivos en muestras pares e impares respectivamente. La aplicación de destino trabaja con datos en Full-HD de 1080p y RGB de 24 bits. El dispositivo se implementó como un ASIC en tecnología de 0.13 micras de 12700 compuertas equivalentes, con una frecuencia de operación de 273 MHz.

En el trabajo [15] se describe una solución en FPGA de una variante del método de Lempel Ziv Welch (LZW) para compresión sin pérdida de imágenes. Este método usa tablas para almacenar cadenas de caracteres asociados a códigos. La clave del método LZW reside en que es posible crear sobre la marcha, de manera automática y en una única pasada un diccionario de cadenas que se encuentren dentro del texto a comprimir mientras al mismo tiempo se procede a su codificación. Dicho diccionario no es transmitido con el texto comprimido, puesto que el descompresor puede reconstruirlo usando la misma lógica con que lo hace el compresor y, si está codificado correctamente, tendrá exactamente las mismas cadenas que el diccionario del compresor. El método se implementó en un CPLD Altera Cyclone II.

Wei Cui [16] también aborda la implementación de un algoritmo a partir del método LZW, en donde se expresa que la desventaja principal del método LZW es el tiempo para crear y realizar búsquedas en el diccionario. Incluso algunas alternativas, como DLZW (Dynamic LZW) usan jerarquías de diccionarios con incrementos sucesivos de los anchos de palabra y búsquedas por *hashing* con tiempos de compresión variables y prohibitivos para aplicaciones de tiempo real. A fin de solucionar esa desventaja, el autor propone una variante que divide el diccionario principal en varios diccionarios de igual tamaño con la posibilidad de operar en paralelo en todos ellos. De esta manera es posible implementar una arquitectura de hardware paralela que incrementa sensiblemente la velocidad de compresión. En particular, propone la división del diccionario en sub diccionarios y su implementación en paralelo. La arquitectura final se implementó en un FPGA Virtex 4 de Xilinx.

Otro método de diccionario usado para compresión de datos sin pérdida es LZMA (Lempel-Ziv-Markov). El trabajo [17] presenta una implementación del mismo en FPGA acompañada de una buena descripción de la familia de algoritmos LZ con breves descripciones de sus alternativas. El algoritmo se implementa en un FPGA Spartan 3E.

La variante JPEG-LS de JPEG se presenta en [12], el cual consta de dos etapas bien distinguidas: modelado y codificación. Esta variante se basa en el algoritmo LOCO-I que utiliza predicción, modelado y codificación de restos basado en el contexto. Para la codificación se utilizan los códigos de Golomb-Rice.

JPEG-LS se clasifica como *lossless/near-lossless*, sin embargo los autores proponen una variante removiendo las partes *near-lossless* del algoritmo, reduciendo su complejidad algorítmica y, posteriormente, lograr un menor consumo de área y energía. El diseño, realizado en VHDL, consta de un pipeline de seis etapas lo que le permite tener una demora en el camino crítico aceptable. Luego de su validación, se sintetizó para tecnologías ASIC TSMC 0.18 micras, UMC 0.18 micras y TSMC 0.09 micras Sage-X Artisan.

El trabajo [18] propone la realización en hardware de una

TABLA I
COMPARACIÓN DE ALGORITMOS DE COMPRESIÓN RESPECTO A TASAS Y TIEMPOS

Algoritmo	Tasa de compresión	Compresión (MB/s)	Descompresión (MB/s)
bzip2 -1	56.3 %	7	16
PNG	42.5%	4	49
JPEG-LS	37.4%	33	38
WebP	35.6 %	0.5	32
IZ	41.3 %	158	154

versión del algoritmo DEFLATE, a través de una variante de los métodos de Lempel-Ziv, utilizando tablas de Huffman para la codificación de símbolos a partir de información estadística. El algoritmo se implementó sobre un FPGA Altera Stratix V.

III. ALGORITMO DE COMPRESIÓN IMAGEZERO

Una de las principales ventajas del algoritmo de compresión IZ es que no posee ramificaciones dentro de su código, lo que lo hace altamente competitivo respecto al rendimiento temporal frente a otras soluciones. Los resultados demuestran que puede ser dos veces más rápido que PNG al descomprimir y mas de 20 veces más rápido al comprimir. Además su tasa de compresión alcanza relaciones de compresión cercanas o mejores a PNG [19] para fotos naturales, a veces incluso mejores que JPEG-LS para fotos de muy alta calidad. El algoritmo de compresión JPEG-LS [20] posee mejor rendimiento que PNG respecto a la velocidad de compresión, e incluso puede superar hasta en un 15% la tasa de compresión, pero sus principales desventajas son la velocidad descompresión, y el modelado de contexto que se encuentra patentado por Hewlett Packard.

Si bien JPEG es el estándar de facto para la compresión de imágenes, y es uno de los métodos más rápidos disponibles, su compresión es con pérdida. Por tanto, no es aplicable en situaciones que requieran una reconstrucción exacta de la imagen original.

La Tabla I presenta los resultados obtenidos en el *benchmarck* [8] para un conjunto de 214 imágenes conformando un volumen de información total de 3.46 GB. Si bien se analizan los resultados de 82 compresores sin pérdida diferentes, en la tabla se detallan sólo los más difundidos o competitivos. En consecuencia, se demuestra que el algoritmo *ImageZero* posee una tasa de compresión competitiva frente a otras propuestas y un rendimiento superior, tanto en la compresión como en la descompresión.

IV. ARQUITECTURA PROPUESTA

El algoritmo de compresión IZ comienza con la escritura del encabezado del archivo resultante. El encabezado se compone de la concatenación de los siguientes campos:

- *b*: número de bits necesarios para almacenar alto y ancho de la imagen, codificado en valor binario de 4 bits.
- *h*: alto de la imagen - 1, codificado en valor binario de *b* bits.
- *w*: ancho de la imagen - 1 , codificado en valor binario de *b* bits.

El valor b es calculado a partir de los valores de las dimensiones de la imagen. Es decir, suponiendo que la imagen de entrada posee una dimensión de 3264×2448, entonces $h = 2447_{10} = 1001\ 1000\ 1111_2$ y $w = 3263_{10} = 1100\ 1011\ 1111_2$; por tanto $b = 12_{10} = 1100_2$, y el encabezado del archivo será: $1100\ 1001\ 1000\ 1111\ 1100\ 1011\ 1111_2$.

Asumiendo que la arquitectura del compresor propuesta será implementada en una plataforma preestablecida (con resoluciones de imágenes de entrada fijas), se considera que los valores de resolución de la imagen serán constantes, y por ende también el encabezado será constante.

IZ realiza compresión de la información visual capturada en una imagen de $w \times h$ píxeles, representados en 24 bits, que definen los componentes de color *Red*, *Green* y *Blue* (R,G,B), utilizando 8 en bits en cada uno de ellos. El algoritmo está basado en un filtro de ventana (de dimensión 2×2) que es aplicado sobre toda la imagen. Para cada píxel $p[i,j]$ (denominado como *pix*), el algoritmo predice el valor del mismo (*pp*) a partir de los valores de $p[i-1,j]$ (como y), $p[i,j-1]$ (como x), y $p[i-1,j-1]$ (como xy), con i en $\{0,1,..,h-1\}$ y j en $\{0,1,..,w-1\}$ (Fig. 1).

La predicción del valor para cada píxel de la imagen se determina de acuerdo a la ecuación 1, considerando las situaciones especiales en los bordes de la imagen (Fig. 2).

$$pp_{[i,j]} = \begin{cases} 0 & \text{, si } i=0 \text{ y } j=0 \\ x & \text{, si } i=0 \text{ y } j\neq 0 \\ y & \text{, si } i\neq 0 \text{ y } j=0 \\ predict(x,y,xy) & \text{, si } i\neq 0 \text{ y } j\neq 0 \end{cases} \quad (1)$$

En el algoritmo, la función $predict(x,y,xy)$ ofrece varias alternativas:

- *Predictor3med*
- *Predictor3alpha*
- *Predictor3plane*
- *Predictor3avgplane*
- *Predictor2avg*

Si bien se consideraron todas las alternativas presentadas, la implementación del diseño se basó en *Predictor3avgplane* descrita en la librería según la ecuación 2.

$$predict(x,y,xy) = \frac{3x+3y-2xy+2}{4} \quad (2)$$

El algoritmo luego continúa con la substracción del valor correspondiente entre $pix[i,j]$ y $pp[i,j]$, dando lugar a $o\_sub[i,j]$.

$$o\_sub[i,j] = pix[i,j] - pp[i,j] \quad (3)$$

La Fig. 3 detalla la arquitectura propuesta para el cálculo de la predicción de una de las componentes de color. En cada ciclo de reloj, cada píxel de la imagen $pix[i,j]$ (con i en $0,1,..,w-1$ y j en $0,1,..,h-1$) ingresa al circuito mediante la señal pix_in. La entrada pix_in posee 24 bits, correspondiéndose con los valores R, G y B del píxel. Se observa que para procesar el píxel actual, se deben tener almacenados los $w+1$ píxeles anteriores. Nótese que para

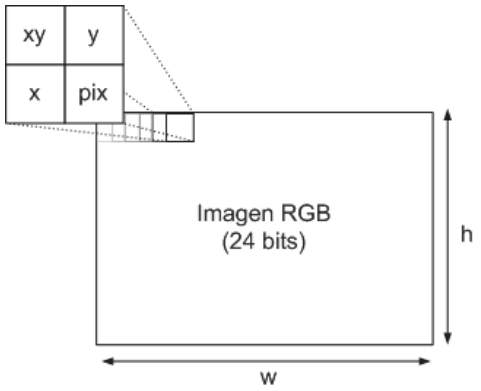


Fig. 1. Entrada del algoritmo ImageZero.

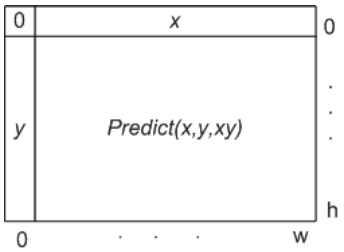


Fig. 2. Cálculo de predictor (*pp*).

$i > 0$ y $j > 0$, cuando $mode = 3$, el registro $reg[1]$ se corresponde con el píxel actual $pix[i, j]$, $reg[2]$ se corresponde con $pix[i, j-1]$, $reg[w+1]$ con $pix[i-1, j]$ y $reg[w+2]$ con $pix[i-1, j-1]$. En el caso que $i = j = 0$ ($mode = 0$), solo se requiere $reg[0]$; en el caso que $j > 0$ e $i = 0$ ($mode = 1$), se requiere $reg[0]$ y $reg[1]$; y en el caso que $j = 0$ e $i > 0$ ($mode = 2$), se requiere $reg[0]$ y $reg[w + 1]$.

El componente *Process_Pixel* se encuentra replicado en la arquitectura final del algoritmo (Fig. 4), cada uno realizando la predicción sobre una componente de color particular. El algoritmo continúa aplicando una (*Forward Transform*) según la ecuación 4, considerando las componentes de color del resultado de la diferencia entre el valor del píxel y su predictor.

$$\begin{aligned} o\_sub_R[i,j] &= o\_sub_R[i,j] - o\_sub_G[i,j] \\ o\_sub_B[i,j] &= o\_sub_B[i,j] - o\_sub_G[i,j] \end{aligned} \quad (4)$$

Tanto los valores resultantes de la transformación, como el valor de $o\_sub_G[i, j]$ son llevados a una representación natural (sin signo), que además de formar parte de la salida codificada, es utilizada para la actualización del valor del contexto. El valor de contexto es el número de bits necesarios para codificar la mayor diferencia de componente entre el píxel predicho y el píxel real. Cuanto más alto sea el valor de contexto, más ruidoso será el píxel (asumiendo que el ruido es impredecible). Es decir, siendo *nl* el número de bits mínimo necesario para almacenar cualquiera de los componentes $o\_sub[i, j]$, el valor de contexto se actualiza de acuerdo a la ecuación 5 y su implementación puede observarse en el registro *reg_cx*.

$$context = context \times 16 + nl \quad (5)$$

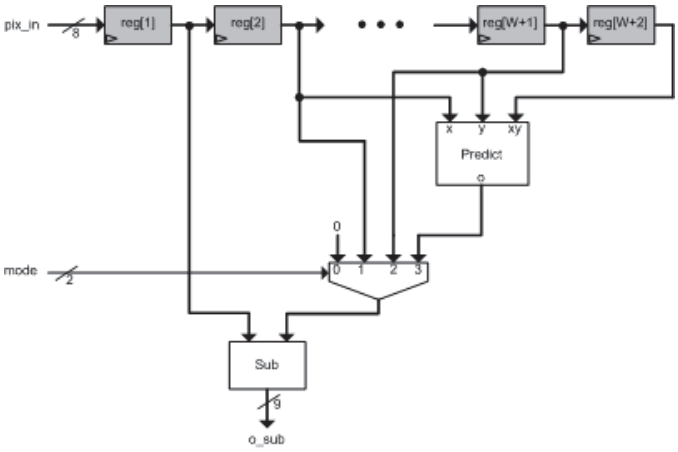


Fig. 3. Circuito *Process_Pixel*.

Por ejemplo, al predecir (R', G', B') = (123,45,67), con un píxel real (R, G, B) = (120, 50, 60), entonces las diferencias son (-3, 5, -7). La conversión del valor a natural se realiza complementando (en el caso que el valor sea negativo) y moviendo el bit de signo al LSB, lo que generaría (5, 10, 13). El mayor valor de la tripla requiere de 4 bits para su representación (en este caso, el valor es 4). La codificación de los píxeles es realizada según el valor de contexto anterior (*nl*), asumiendo que los píxeles con alta presencia de ruido, se encuentren rodeados de otros píxeles con ruido.

El algoritmo IZ utiliza códigos de Huffman canónicos fijos, considerando hasta 6 bits de profundidad. Estos valores son almacenados en tablas estáticas que contienen los valores de estos códigos (*dbits*), y dimensión en bit de los mismos (*dcount*). La entrada de la tabla se corresponde con el valor almacenado en *reg_cx*. Estas tablas son implementadas en la arquitectura como ROMs con valores constantes, los cuales pueden ser optimizados por el sintetizador de acuerdo a los parámetros de síntesis seleccionados, y se presenta en la arquitectura propuesta como el componente *Tables*.

El componente *Control* genera las señales de control que se corresponden con la habilitación de los registros (*en_regs*) y la señal *mode*, la cual vale 0 cuando se procesa el primer píxel, vale 1 cuando se procesa un píxel de la primer fila, vale 2 cuando se procesa un píxel de la primer columna, y vale 3 en el resto de los casos.

Debido a que la arquitectura propuesta se adapta de manera directa a la segmentación, se dividió el *datapath* en tres etapas de *pipelining* con el objetivo de aumentar la frecuencia de reloj. De este modo, se aprecia que la latencia de la propuesta en términos de ciclo de reloj, para una imagen de dimensión $h \times w$, es $L = h \times w + 2$.

La salida de la arquitectura se corresponde con la concatenación de los *dcount*[*reg_cx*] bits obtenidos de la tabla *dbits*[*reg_cx*], con los *nl* bits de $o\_sub_G[i, j]$, $o\_sub_R[i, j]$ y $o\_sub_B[i, j]$, tal como se presenta en la Fig. 5. Esta funcionalidad se encuentra implementada en el componente *Gen_Out* de la arquitectura. La implementación de este componente encapsula la información de salida en bloques de 32 bits en la medida que se obtienen. El comportamiento de *Gen_Out* se

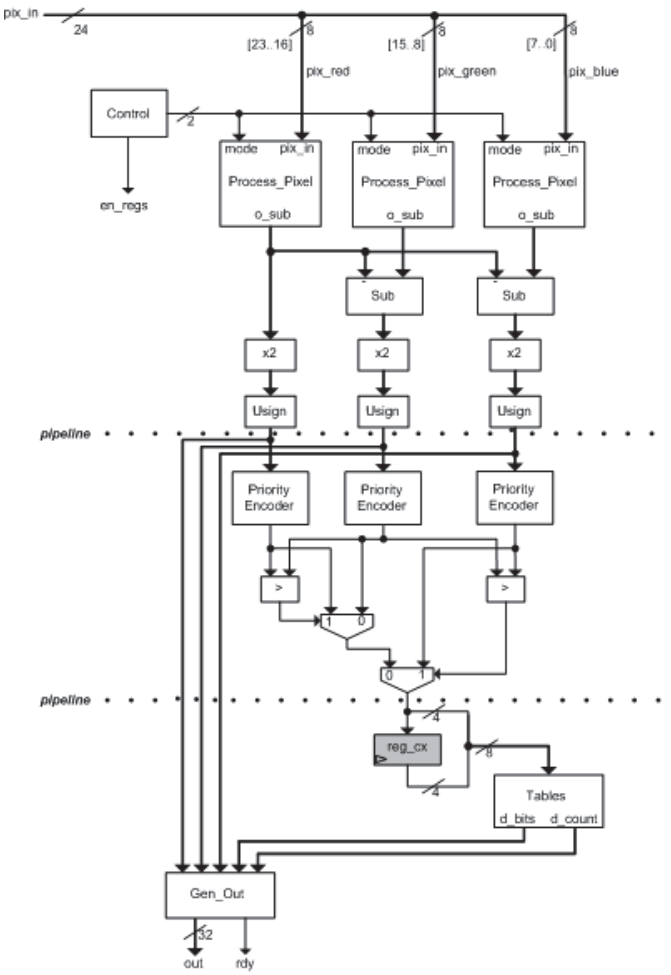


Fig. 4. Arquitectura de *Encoder* IZ propuesta.

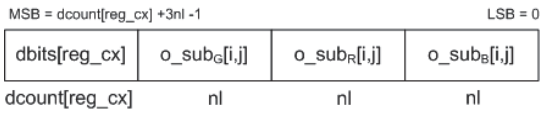


Fig. 5. Salida de la arquitectura.

describe en la Fig. 6.

V. Resultados Experimentales

La arquitectura propuesta para la compresión de imágenes *ImageZero*, se describió en VHDL [21] de manera portable e independiente de los fabricantes y/o dispositivos FPGAs utilizados para la implementación final. Los códigos se encuentran disponibles en [22]. La solución fue validada mediante el uso de *testbenches*. Para ello, se contrastaron los resultados de ejecutar el *encoder* de la librería C++ con la arquitectura para diferentes valores de entrada generados aleatoriamente.

Para la implementación se utilizaron dispositivos de Xilinx Virtex UltraScale+ con speed grade 3 [23] (Tabla II). Las implementaciones se realizaron con imágenes de diferentes dimensiones: 256x256, 512x512, 1024x1024. La síntesis e implementación se realizaron con el XST (Xilinx Synthesis Technology) [24] y Xilinx Vivado v2018.1 [25]. Esto se llevó a cabo dejando los parámetros de síntesis por defecto.

$m \leftarrow 3 \times nl + dcount$
$reg\_out \leftarrow reg\_out_{[63-m:0]} \& dbits_{[count-1:0]} \&$
$\text{s\_r}_{[nl-1:0]} \& s\_g_{[nl-1:0]} \& s\_b_{[nl-1:0]}$
if $reg\_val \geq 32$ **then**
  $rdy \leftarrow 1$
  $out \leftarrow reg\_out_{reg\_val-1:reg\_val-32}$
  $reg\_val \leftarrow reg\_val - 32$
else
  $rdy \leftarrow 0$
  $out \leftarrow reg\_out_{31:0}$
end if

Fig. 6. Algoritmo *Gen_Out*

TABLA II
Tiempo de cálculo y ocupación de LUTs del encoder ImageZero en Virtex 7 con speed grade 3s

h×w	Tiempo ciclo (ns)	Ciclos (L)	LUTs lógica	LUTs memoria
256x256	2.94	65538	1079	192
512x512	2.94	262146	1082	400
1024x1024	3.15	1048578	1112	832

TABLA III
Velocidad de compresión

h×w	Tiempo ciclo (ns)	MB/s
256x256	2.94	1020
512x512	2.94	1020
1024x1024	3.15	951

Apréciese que el tiempo de ciclo no presenta variaciones significativas en las implementaciones, debido a que la ruta más larga no se encuentra relacionada a la cantidad de píxeles que posee la imagen. Por otro lado, en lo que respecta a la ocupación en área, la ocupación de LUTs utilizados para lógica es similar a todas las configuraciones. En el caso de la ocupación de LUTs utilizadas como memorias, se incrementa en la medida que aumenta el tamaño de la imagen, más precisamente cuando aumenta el tamaño de la fila (parámetro w). Este incremento se produce porque el componente $Process\_Pixel$, debe registrar los $w+1$ píxeles anteriores para poder procesar correctamente el píxel corriente.

Se obtuvo que el módulo *Gen_Out* posee una amplia ocupación de LUTs comparado con el diseño total. Este módulo ocupa 674 LUTs utilizadas como lógica, aproximadamente entre el 61% y el 62.5%, de la ocupación total de la propuesta. Esto se debe a que posee la implementación de módulos basados en costosos *barrel-shifters*.

La Tabla III detalla la velocidad de compresión de la implementación propuesta en número de bytes por segundo. El cálculo fue realizado considerando a los tiempos de ciclos obtenidos, y que el algoritmo se aplica sobre imágenes de entrada con una representación de 3 bytes por píxel (1 byte por cada componente de color).

Se destaca que los resultados de velocidad de compresión superan a otras implementaciones, considerando que la arqui-

tectura propuesta es capaz de procesar 1020 MB/s (o 340 Mpx por segundo), mientras que [12] alcanza los 265 Mpx/s, e incluso superando a [14] el cual presenta una implementación sobre ASIC que con capacidad de comprimir entre 200 y 275 MB/s. Si bien la tecnología de las implementaciones difiere, se resalta que los resultados son superadores.

VI. Conclusiones y Trabajo Futuro

En este trabajo se propone la implementación eficiente del algoritmo IZ basada en FPGA, la cuál es novedosa al no encontrarse otras implementaciones de este algoritmo en la literatura. El diseño se realizó en VHDL, y es portable a cualquier dispositivo programable. Las implementaciones fueron validadas contrastando los resultados respecto a la versión original software.

La solución planteada se ajusta a las velocidades de transmisión que exigen las aplicaciones actualmente. Si bien las características del algoritmo permiten implementarse en otras tecnologías (como microcontroladores), la arquitectura ofrece simpleza respecto a otras alternativas, pudiéndo aplicarse en sistemas embebidos, como por ejemplo equipamiento médico y nanosatélites.

Asimismo, se ha explotado la capacidad de paralelización y *pipelining* de la arquitectura FPGA, manteniendo la fiabilidad y tasa de compresión del algoritmo original. En este sentido, aplicar la técnica de *pipelining* al circuito permitió diminuir considerablemente los tiempos de ciclos, sin perjudicar en área, y ofreciendo una velocidad de compresión superior a otras alternativas.

Como línea de trabajo futuro se propone realizar el análisis de consumo de la arquitectura. Por otra parte, el área del circuito puede ser reducida, mediante la utilización de memorias FIFO/BRAM contenidas en el dispositivo destino. Se pretende incorporar esta mejora a fin de disminuir la cantidad LUTs que, en esta versión, son utilizadas como memorias.

Agradecimientos

Este trabajo fue parcialmente financiado por la SeCAT de UNICEN (Código de Proyecto 03/C287) y la Secretaría de Investigación de la Universidad Nacional de Tres de Febrero. El trabajo además se enmarca dentro del proyecto ”Optimización de consumo en comunicación de información satelital mediante el uso de FPGAs”, aprobado por la Secretaría de Potíticas Universitarias en su convocatoria ”Universidades Agregando Valor 2018”, realizado en conjunto a Satellogic Research.

Referencias

[1] M. A. Joshi, M. S. Raval, Y. H. Dandawate, K. R. Joshi, and S. P. Metkar, *Image and video compression: Fundamentals, Techniques, and Applications*. Chapman and Hall/CRC, 2014.

[2] M. F. Ukrit, A. Umamageswari, and G. Suresh, “A survey on lossless compression for medical images,” *International Journal of Computer Applications*, vol. 31, no. 8, pp. 47–50, 2011.

[3] Y. Deigant, V. Akshat, H. Raunak, P. Pranjal, and J. Avi, “A proposed method for lossless image compression in nano-satellite systems,” in *2017 IEEE Aerospace Conference*. IEEE, 2017, pp. 1–11.

[4] B. Huang, *Satellite data compression*. Springer Science & Business Media, 2011.

[5] B. Rusyn, O. Lutsyk, Y. Lysak, A. Lukenyuk, and L. Pohreliuk, “Lossless image compression in the remote sensing applications,” in *2016 IEEE First International Conference on Data Stream Mining & Processing (DSMP)*. IEEE, 2016, pp. 195–198.

[6] B. U. Töreyin, “Smoke detection in compressed video,” in *Applications of Digital Image Processing XLI*, vol. 10752. International Society for Optics and Photonics, 2018, p. 1075232.

[7] T. A. Welch, “Technique for high-performance data compression,” *Computer*, no. 52, 1984.

[8] A. Rhatushnyak. (2018, Jun.) Lossless photo compression benchmark. [Online]. Available: http://qlic.altervista.org/

[9] C. Feck. (2013, Nov.) Fast lossless color image compression library). [Online]. Available: https://github.com/cfeck/imagezero

[10] G. Roelofs and R. Koman, *PNG: the definitive guide*. O’Reilly & Associates, Inc., 1999.

[11] S. D. Rane and G. Sapiro, “Evaluation of jpeg-ls, the new lossless and controlled-lossy still image compression standard, for compression of high-resolution elevation data,” *IEEE Transactions on Geoscience and Remote sensing*, vol. 39, no. 10, pp. 2298–2306, 2001.

[12] M. Papadonikolakis, V. Pantazis, and A. P. Kakarountas, “Efficient high-performance asic implementation of jpeg-ls encoder,” in *2007 Design, Automation & Test in Europe Conference & Exhibition*. IEEE, 2007, pp. 1–6.

[13] D. Castells-Rufas, A. Saa-Garriga, and J. Carrabina, “Energy efficiency of many-soft-core processors,” *arXiv preprint arXiv:1601.07133*, 2016.

[14] T.-H. Tsai, Y.-H. Lee, and Y.-Y. Lee, “Design and analysis of high-throughput lossless image compression engine using vlsi-oriented felics algorithm,” *IEEE transactions on very large scale integration (VLSI) systems*, vol. 18, no. 1, pp. 39–52, 2009.

[15] M. Almelkar and S. Gandhe, “Implementation of lossless image compression using fpga,” *International Journal of Emerging Technology and Advanced Engineering*, vol. 4, pp. 2250–2459, 2014.

[16] W. Cui, “New lzw data compression algorithm and its fpga implementation,” in *Proc. 26th Picture Coding Symposium (PCS 2007)*, 2007.

[17] A. AROHI and V. S. Kulkarni, “Fpga based implementation of data compression using dictionary based “ lzma ” algorithm,” 2014.

[18] J. Fowers, J.-Y. Kim, D. Burger, and S. Hauck, “A scalable high-bandwidth architecture for lossless compression on fpgas,” in *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2015, pp. 52–59.

[19] G. Roelofs and R. Koman, *PNG: the definitive guide*. O’Reilly & Associates, Inc., 1999.

[20] S. D. Rane and G. Sapiro, “Evaluation of jpeg-ls, the new lossless and controlled-lossy still image compression standard, for compression of high-resolution elevation data,” *IEEE Transactions on Geoscience and Remote sensing*, vol. 39, no. 10, pp. 2298–2306, 2001.

[21] “IEEE 1076-2008 - IEEE Standard VHDL Language Reference Manual ,” International Organization for Standardization, Geneva, CH, Standard, Jan. 2009.

[22] LabSET. (2018, Dec.) Vhdl implementation of imagezero encoder. [Online]. Available: https://github.com/LabSET-UNICEN/IZEncoder_VHDL

[23] Xilinx. (2018, Jan.) Ultrascale architecture and product data sheet: Overview. [Online]. Available: https://www.xilinx.com/

[24] ——. (2018, Jun.) Xst synthesis overview. [Online]. Available: https://www.xilinx.com/products/design-tools/xst..html

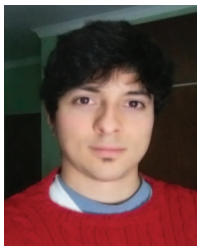
[25] ——. (2018, Jun.) Vivado design suite - hlx editions. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado.html

Lucas Leiva Es Ingeniero de Sistemas de la Universidad Nacional del Centro de la Provincia de Buenos Aires (UNICEN) desde 2007, y finalizó la Maestría en Ingeniería de Sistemas de UNICEN en 2010. Actualmente es profesor adjunto del Departamento de Computación y Sistemas de UNICEN, y del Departamento de Ingeniería en Computación de UNTREF. Es integrante del Laboratorio de Sistemas Embebidos Tandil (LabSET) dentro del Instituto INTIA de la Facultad de Ciencias Exactas de la UNICEN desde el año 2004. Hasta la actualidad participó de forma continua en proyectos de investigación y transferencia en temáticas relacionadas a sistemas embebidos y lógica programable. Sus principales áreas de intereés son: procesamiento de imágenes, diseño lógico, lógica programable y síntesis de alto nivel.

Martín Vázquez Es graduado de la carrera Ingeniería de Sistemas de la Universidad Nacional del Centro de la Provincia de Buenos Aires (UNICEN) en el año 2003. Obtuvo una Maestría en Ingeniería en Sistemas en el año 2011 y un Doctorado en Matemática Computacional e Industrial en el año 2018, en UNICEN. Actualmente se desempeña como Profesor Adjunto con dedicación exclusiva en UNICEN y en UNTREF. Integrante del Laboratorio de Sistemas Embebidos Tandil (LabSET) dentro del Instituto INTIA de la Facultad de Ciencias Exactas de la UNICEN. Ha participado de forma continua en proyectos de investigación y transferencia en temáticas relacionadas a sistemas embebidos, aritméticas de computadoras, diseño de sistemas digitales, desarrollo de plataformas Hardware/Software embebidas y diseño e implementación sobre dispositivos de lógica programable. Actualmente dicta cursos de posgrado y especialización referentes a estas temáticas.

Marcelo Tosini Recibió su título de Ingeniero de Sistemas de la Universidad Nacional del Centro de la Provincia de Buenos Aires (UNICEN) en 1991 y su Maestría en Ingeniería de Sistemas de la misma Universidad en 2002. Actualmente se encuentra realizando sus estudios de doctorado en aritméticas y hardware de computadoras. Integrante del Laboratorio de Sistemas Embebidos Tandil (LabSET) dentro del Instituto INTIA de la Facultad de Ciencias Exactas de la UNICEN. Es Profesor Adjunto en el Departamento de Computación y Sistemas en la UNICEN y Profesor Titular en la Universidad FASTA de Argentina. Sus intereses en investigación se enfocan a procesamiento de imágenes, aritmética en hardware, arquitectura de procesadores y lógica programable.

Oscar Goñi Es graduado de la carrera Ingeniería de Sistemas de la UNICEN (2009) y Doctor en Ciencias Informáticas de la Universidad Nacional de la Plata. Integrante del Laboratorio de Sistemas Embebidos Tandil (LabSET) dentro del Instituto INTIA de la Facultad de Ciencias Exactas de la UNICEN. Entre sus áreas de investigación se destacan verificación funcional de sistemas digitales, sistemas embebidos en aplicaciones críticas e internet de las cosas.



José Noguera Estudiante avanzado de la carrera de grado de Ingeniería de Sistemas de la UNICEN. Becario en programa Estímulo a las Vocaciones Científicas del Consejo Interuniversitario Nacional (EVC-CIN 2018). Auxiliar docente de UNICEN. Sus intereses de investigación actuales son el área de machine vision, machine learning, lógica programable y síntesis de alto nivel.