Análisis Comparativo de Implementaciones HLS de Filtro Sobel en SoC

Roberto Millon[1] , Enzo Rucci[2] , and Emmanuel Frati[1]

[1] Departamento de Ciencias Básicas y Tecnológicas, UNdeC
Chilecito (5360), La Rioja, Argentina
{rmillon,fefrati}@undec.edu.ar
[2] III-LIDI, Facultad de Informática, UNLP – CeAs CICPBA.
La Plata (1900), Bs As, Argentina
erucci@lidi.info.unlp.edu.ar

Resumen El filtro Sobel se ha consolidado como una alternativa popular para detectar bordes en imágenes por su simpleza y baja sensibilidad al ruido. Como consecuencia, Sobel ha sido ampliamente estudiado, existiendo numerosas variantes que ofrecen distintos compromisos entre costo computacional y calidad de filtrado. En este trabajo se sintetizaron tres variantes del filtro Sobel en alto nivel sobre una plataforma SoC y se analizó consumo de recursos, tiempo de procesamiento y calidad de filtrado de cada una de ellas. Los resultados obtenidos muestran que la velocidad de cómputo no presenta variaciones significativas entre las tres variantes mientras que la demanda de recursos sí lo hace, aunque no constituye una restricción de diseño. Por último, en cuanto a la calidad del filtrado, si bien las imágenes producidas son similares, presentan diferencias en los bordes detectados.

Keywords: Detección de bordes · Sobel · SoC · Zybo · HLS

1. Introducción

Una imagen digital se puede definir como un conjunto de puntos o píxeles dispuestos en una matriz de dos dimensiones (x e y). El procesamiento digital de imágenes consiste en la aplicación de técnicas y algoritmos computacionales complejos sobre una imagen para extraer información de ella o mejorar sus características [15]. Entre estos algoritmos, se pueden mencionar los relacionados a detección de bordes o contornos, ampliamente utilizados como etapa previa a otros algoritmos de procesamiento [12]. Las técnicas de detección de bordes identifican zonas en la imagen donde los píxeles presentan cambios abruptos en intensidad o niveles de grises, permitiendo segmentar una imagen en regiones discontinuas [3]. Como resultado, se reduce significativamente la cantidad de datos en la imagen sin alterar sus propiedades estructurales [16].

Al momento de detectar contornos, no existe un único método. Los operadores Laplacianos (o de segundo orden) permiten detectar bordes en cualquier orientación pero tienen una alta tasa de detección de falsos positivos y son muy

ISBN 978-987-4417-90-9

sensibles al ruido. Por otro lado, se encuentran los algoritmos basados en gradientes (o de primer orden), que se pueden considerar menos sensibles al ruido respecto a los operadores Laplacianos. Sin embargo, no detectan bordes diagonales y generan contornos muy gruesos. Dentro de los algoritmos basados en gradiente, se destaca el operador Sobel por tener mejor inmunidad al ruido respecto a otras alternativas como el operador Roberts-Cross o el Prewitt [16]. Como consecuencia, el algoritmo Sobel ha sido ampliamente estudiado, existiendo numerosas variantes que ofrecen distintos compromisos entre costo computacional y calidad de filtrado.

Respecto a las plataformas hardware elegidas para procesamiento de imágenes, las FPGAs se han destacado por ser dispositivos de bajo consumo energético y alta productividad [6]. La disponibilidad de grandes bancos de memorias internas en estas arquitecturas permiten accesos en forma paralela para ejecutar funciones en pocos ciclos de reloj, a diferencia de otras tecnologías como los procesadores convencionales que operan en forma secuencial y requieren una gran cantidad de ciclos de reloj para resolver las tareas [5].

En este trabajo se presenta un análisis comparativo de tres variantes del filtro Sobel sintetizadas en lenguaje de alto nivel para procesar imágenes en una plataforma System-on-Chip (SoC) ZYBO. El análisis realizado considera uso de recursos, tiempo de procesamiento y calidad del filtrado de cada variante Sobel, y puede resultar útil al momento de tener que elegir una de ellas.

El resto del documento se organiza de la forma siguiente. En la Sección 2 se presentan los antecedentes del presente trabajo mientras que en la Sección 3 se describe la implementación propuesta. A continuación, en la Sección 4 se presentan los resultados experimentales y, finalmente, la Sección 5 resume las conclusiones y posibles trabajos futuros.

2. Antecedentes

2.1. Filtro de Detección de Bordes Sobel

El filtro Sobel es un método de detección de contornos que se aplica a imágenes en escala de grises [12] y que detecta sus componentes de alta frecuencia mediante la derivada primera. Un componente de alta frecuencia se genera por cambios abruptos de intensidad en píxeles contiguos de la imagen, los cuales se corresponden con un borde. El vector gradiente detecta esos cambios por medio de las derivadas parciales en los ejes horizontal y vertical. La Ecuación 1 define al vector gradiente por sus componentes horizontal G_x y vertical G_y [5].

$$\nabla f = \begin{pmatrix} G_x \\ G_y \end{pmatrix} \tag{1}$$

El módulo o magnitud del vector gradiente se denomina fuerza de borde y expresa la tasa de cambio por unidad de distancia. De este modo, una mayor variación de intensidad en píxeles vecinos genera un aumento en la magnitud del gradiente que se computa como un borde. La magnitud del gradiente se define por la Ecuación 2.

ISBN 978-987-4417-90-9

$$|\nabla f| = \sqrt{{G_x}^2 + {G_y}^2} \tag{2}$$

Las implementaciones del filtro Sobel emplean dos máscaras de convolución M_x y M_y para obtener las componentes horizontal y vertical del vector gradiente, respectivamente. Las máscaras de convolución se muestran en la Fig. 1.

$$M_x = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \qquad M_y = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

Figura 1: Máscaras de convolución Sobel

Desde un punto de vista matemático, las componentes del gradiente se obtienen de la multiplicación de cada máscara con la imagen. El proceso se realiza de izquierda a derecha y de arriba hacia abajo hasta recorrer la imagen completa. En la Fig. 2 se puede observar el proceso de convolución entre una imagen y una máscara del filtro Sobel.

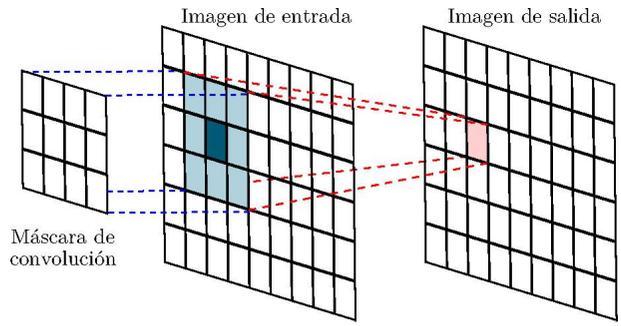

Figura 2: Aplicación del filtro Sobel a una imagen

Variantes de Sobel. Existen distintas alternativas del filtro Sobel que tienen por objetivo variar el costo computacional y/o la calidad de imagen obtenida. Un método popular para disminuir la carga computacional consiste en aproximar la magnitud del gradiente por la suma de los valores absolutos de sus componentes, evitando así recurrir a operaciones en punto flotante. Mediante esta alternativa, varios trabajos han logrado reducir el tiempo de procesamiento [5, 14, 18]. En la Ecuación 3 se presenta la fórmula de aproximación para el cálculo de la magnitud del vector gradiente.

$$|\nabla f| = |G_x| + |G_y| \tag{3}$$

Por otra parte, también existen implementaciones arbitrarias de Sobel, como la usada por el editor de imágenes GIMP [3] o la librería de visión por computadora OpenCV [4]. GIMP computa la magnitud del gradiente empleando la fórmula

[3] GNU Image Manipulation Program (GIMP). `https://www.gimp.org/`
[4] Open Source Computer Vision Library (OpenCV). `https://https://opencv.org/`

ISBN 978-987-4417-90-9

convencional (expresada en la Ecuación 2) pero además dividiendo su resultado por un número real fijo (5.66). Por su parte, OpenCV emplea un enfoque similar al de GIMP, sólo que usando la formula de aproximación (expresada en la Ecuación 3) y dividiendo por un número entero fijo (2).

2.2. Síntesis de Alto Nivel en FPGAs

Hasta principios de siglo, el modelo de programación en FPGA se centraba en descripciones hardware a nivel de transferencia entre registros (RTL). Las implementaciones en RTL requieren un gran conocimiento de la arquitectura de esta tecnología, además de elevados tiempos de desarrollo y pruebas. Esto limitaba el uso de las FPGAs al no adecuarse a los acotados tiempos de mercado [1].

Numerosas propuestas académicas y comerciales [11], entre ellas las herramientas de síntesis de alto nivel (HLS) de Xilinx, han elevado el nivel de abstracción en el diseño de circuitos al utilizar lenguajes como C, C++, OpenCL, entre otros. Esto permite acelerar los tiempos de desarrollo y reducir las diferencias en los modelos de programación entre procesadores y FPGAs. No obstante, resulta necesario el uso de directivas de optimización específicas para lograr buenas prestaciones en los diseños HLS [17].

Vivado HLS es la herramienta de Xilinx para síntesis de alto nivel en FPGA, la cual transforma un diseño escrito en C, C++ o System C en una implementación RTL [2]. Los diseños RTL son exportadas como bloques de propiedad intelectual (IP) para ser utilizados por otras herramientas de Xilinx. Vivado Design Suite (VDS) emplea un entorno gráfico para utilizar los bloques IP.

2.3. Estado del Arte

En el ámbito de las FPGAs, existen numerosas implementaciones HLS del operador Sobel. En [10] se analizan distintas técnicas de optimización para síntesis de alto nivel, eligiendo al algoritmo Sobel como caso de estudio. Al igual que el trabajo anterior, en [7] se usa a Sobel como caso de estudio para comparar y evaluar metodologías de diseño en FPGAs. A diferencia de los anteriores, en [13] se analiza el consumo energético de una implementación Sobel en una plataforma SoC. Por último, varios trabajos han desarrollado soluciones del filtro Sobel para detectar bordes en tiempo real usando diferentes tecnologías y plataformas [4, 8, 19–21]. Sin embargo, hasta donde llega el conocimiento de los autores, no existen trabajos que realicen un análisis comparativo de distintas variantes de este operador en cuanto a costo computacional y calidad del filtrado.

3. Implementación

Para el análisis comparativo en el presente trabajo se desarrollaron tres versiones del operador Sobel, las cuales toman como base la implementación [9]. Todas las versiones tienen en común dos estructuras de memoria denominadas *buffer de línea* y *ventana deslizante* [20]. Los buffers de línea son arreglos

ISBN 978-987-4417-90-9

unidimensionales que almacenan filas completas de la imagen, con el objeto de mantener el contexto necesario para procesar cada píxel. Estas estructuras de memoria permiten procesar una imagen completa sin necesidad de almacenar todos sus píxeles a la vez, constituyendo un ahorro considerable de memoria. El operador Sobel utiliza tres buffers de línea, ya que se requiere el mismo número de filas al mismo tiempo para el filtrado.

Por su parte, la ventana deslizante es un arreglo bidimensional que contiene el conjunto de píxeles a los cuales se le aplicará las máscaras de convolución. La ventana deslizante se desplaza de izquierda a derecha por los buffers de línea y tiene dimensión 3×3. En la Fig. 3 se puede observar la aplicación de la convolución en Sobel, destacándose en color amarillo los buffer de línea y en color rojo la ventana deslizante.

Las diferencias entre las variantes del operador Sobel se encuentran en la fórmula para computar la magnitud del gradiente. Se distinguen tres versiones:

1. *RMS:* utiliza la fórmula convencional definida en la Ecuación 2.
2. *ABS:* utiliza la fórmula aproximada definida en la Ecuación 3.
3. *GIMP:* utiliza la fórmula convencional definida en la Ecuación 2 pero dividiendo su resultado por un número real fijo (5.66).

Independientemente de la variante Sobel utilizada, es necesario incorporar otro bloque al sistema que permita realizar pruebas reales en la plataforma SoC ZYBO. El sistema completo está formado por un bloque DMA que accede a la memoria microSD para la lectura de imágenes en formato BMP y envía el flujo de píxeles al núcleo de procesamiento de imagen. Luego del filtrado, el flujo de píxeles es devuelto al DMA para su posterior almacenamiento en memoria microSD. La configuración y habilitación de cada modulo del sistema es realizada por el procesador. Todos los módulos fueron integrados usando la herramienta Vivado 2019.1. En la Fig. 4 se observa un diagrama en bloques del sistema completo.

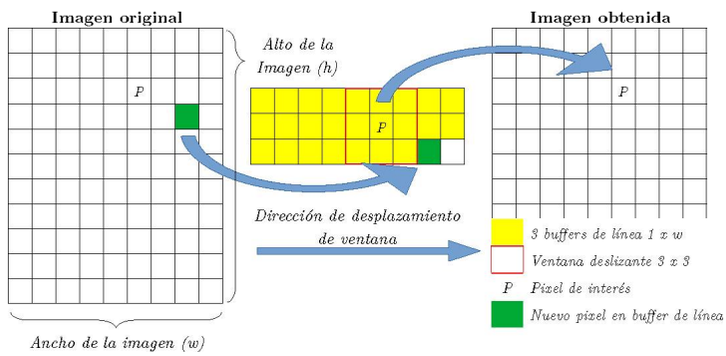

Figura 3: Aplicación de convolución

ISBN 978-987-4417-90-9

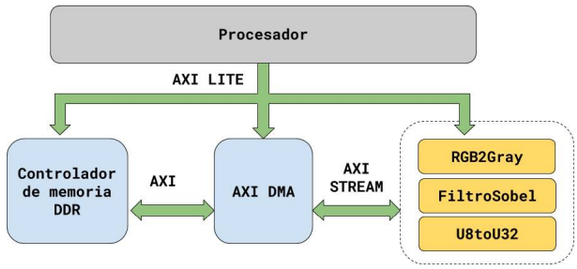

Figura 4: Sistema implementado

El núcleo de procesamiento está formado por tres bloques sintetizados y validados en alto nivel con la herramienta Vivado HLS versión 2019.1 de Xilinx. El primer bloque (RGB2Gray) convierte imágenes a color en escala de grises al promediar el valor de los tres canales (RGB) en cada píxel. El segundo bloque (FiltroSobel) es el responsable de aplicar el operador Sobel para detectar contornos en alguna de sus tres variantes. Por último, se incorpora un bloque adicional (U8toU32) que concatena cuatro caracteres de 8 bits con el fin de sincronizar las comunicaciones con el DMA (ya que opera con palabras de 32 bits).

Resulta importante mencionar que se utilizaron las directivas de optimización para sintetizar cada bloque del núcleo de procesamiento. En particular, se usó `HLS INTERFACE` para emplear los estándares AXI Stream y AXI Lite en las comunicaciones de las imágenes y configuración de parámetros. También se utilizó `HLS ARRAY PARTITION` para mapear arreglos en múltiples recursos de memoria y permitir accesos concurrentes (buffers de línea, ventana deslizante y máscaras). Por último, se incluyó la directiva `HLS PIPELINE` que permite aumentar la productividad al ejecutar en paralelo funciones y operaciones en bucles [1].

4. Resultados Experimentales

4.1. Diseño experimental

Las pruebas se realizaron sobre una plataforma ZYBO compuesta por un SoC ZYNQ-7000 de Xilinx. El SoC está integrado por un procesador de doble núcleo ARM Cortex-A9 y una FPGA XC7Z010-1-CLG400C. El sistema completo se diseñó en un entorno gráfico dentro del software VDS. Para la implementación de cada variante, se reemplazó únicamente el bloque IP del kernel Sobel por alguna de sus alternativas (RMS, ABS o GIMP). En cada una de ellas se registró el uso de recursos informado por la herramienta VDS.

Para realizar las pruebas se desarrolló una aplicación *test* con la herramienta XSDK 2019.1 de Xilinx. Además, se seleccionaron tres imágenes de repositorios públicos para realizar las pruebas en cada variante Sobel: *Peppers* de 512×512[5], *Kodim21* de 768×512[6] y *Fox* de 1280×853 [7]. En las pruebas realizadas, se midieron los tiempos de ejecución mediante la librería `xtime_l.h`, informando el promedio de realizar 10 repeticiones de cada una de ellas.

[5] https://www.hlevkin.com/06testimages.htm
[6] https://github.com/lemire/kodakimagecollection
[7] https://pixabay.com/es/photos/fuchs-mundo-animal-animales-salvajes-5303221/

ISBN 978-987-4417-90-9

4.2. Resultados del filtrado

En la Fig. 5 se muestran las imágenes originales que fueron seleccionadas para las pruebas junto a las imágenes producidas por las distintas implementaciones desarrolladas. Como se puede apreciar, las tres variantes consiguen detectar los bordes relevantes de cada imagen. Sin embargo, al menos para este conjunto de imágenes, RMS y ABS producen resultados similares mientras que en GIMP los bordes detectados son más débiles.

4.3. Resultados de uso de recursos y rendimiento

La Tabla 1 presenta el uso de recursos y los tiempos de procesamiento obtenidos para las tres variantes Sobel desarrolladas. Los valores de las columnas

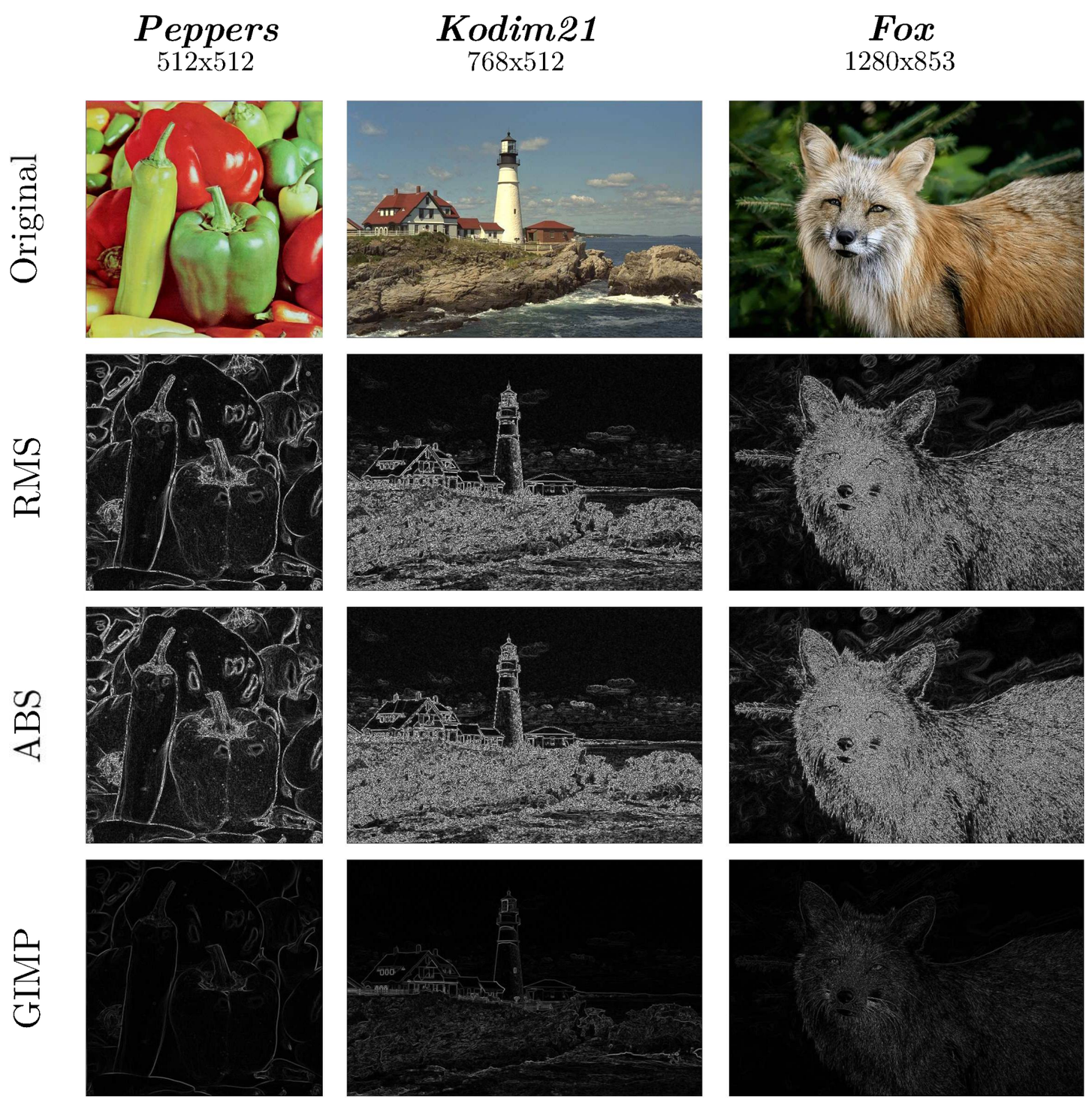

Figura 5: Resultados de aplicar variaciones del filtro Sobel a las imágenes de prueba.

ISBN 978-987-4417-90-9

Tabla 1: Uso de recursos y tiempo de procesamiento de las variantes de Sobel implementadas con optimizaciones

Variante	**Uso de Recursos (% del total)**					**Tiempos de procesamiento (ms)**					
						Peppers		Kodim21		Fox	
	S.LUTs	S.Registers	F7 Muxes	BRAM	DSPs	Sobel	Resto	Sobel	Resto	Sobel	Resto
RMS	16,3 %	9,1 %	< 0,1 %	1,6 %	2,5 %	40,96	293,98	40,97	388,59	68,26	984,65
ABS	4,4 %	2,4 %	< 0,1 %	1,6 %	0 %	40,96	293,98	40,96	388,59	68,26	986,88
GIMP	34,4 %	16,3 %	< 0,1 %	1,6 %	2,5 %	40,96	294,1	40,97	389,3	68,26	988,34

`S.LUTs`, `S.Registers`, `F7 Muxes`, `BRAM` y `DSPs` hacen referencia a los porcentajes de tablas de búsqueda, registros, multiplexores, bloques de memoria RAM y bloques DSPs utilizados, respectivamente.

En cuanto al uso de recursos, se puede notar que ninguna de las tres presenta un consumo excesivo de ellos. Aun así, ABS y GIMP se destacan como las opciones que menos y más recursos requieren, respectivamente. En particular, la versión GIMP demanda 7.8× más de LUTS y 6.8× más de Register en comparación a ABS. Por su parte, RMS se presenta como una versión intermedia, consumiendo 3.7×/3.8× más que ABS pero 0.47×/0.56× menos que GIMP en LUTS/Registers. No se observan diferencias significativas para F7 Muxes, BRAM y DSPs.

Se observa que el tiempo de procesamiento total aumenta a medida que se incrementa el tamaño de la imagen de entrada. Sin embargo, y contrariamente a lo esperado, se puede notar que todas las variantes de Sobel implementadas tuvieron tiempos de ejecución similares independientemente del método empleado para computar la magnitud del gradiente.

Luego de profundizar en el análisis del código y sus resultados, se encontró a dos de las directivas de optimización como las responsables del tiempo de ejecución constante de las tres variantes del filtro para una determinada imagen. La combinación del solapamiento de las diferentes etapas de las operaciones (`HLS PIPELINE`) con el acceso concurrente a las estructuras de datos (`HLS ARRAY PARTITION`) conducen a que la diferencia de tiempo en el cómputo de la magnitud del gradiente no tenga peso suficiente como para impactar en el tiempo total de Sobel. Este aspecto se puede apreciar en la Tabla 2, la cual presenta los resultados de uso de recursos y tiempo de procesamiento para las tres variantes de Sobel desarrolladas, pero omitiendo en este caso el uso de las directivas de optimización mencionadas.

Se puede notar que las versiones no optimizadas sí reportan diferencias significativas en los tiempos de procesamiento (Sobel), mientras que el tiempo en el resto del sistema se mantiene estable. La versión ABS resulta ser la más eficiente en este aspecto, ejecutándose hasta 1.53× y 1.95× más rápido que RMS y GIMP, respectivamente. Por su parte, RMS se ubica en una posición intermedia, siendo hasta 1.27× más veloz que GIMP.

ISBN 978-987-4417-90-9

Tabla 2: Uso de recursos y tiempo de procesamiento de las variantes de Sobel implementadas sin optimizaciones

Variante	**Uso de Recursos (% del total)**					**Tiempos de procesamiento (ms)**					
						Peppers		Kodim21		Fox	
	S.LUTs	S.Registers	F7 Muxes	BRAM	DSPs	Sobel	Resto	Sobel	Resto	Sobel	Resto
RMS	15 %	8,5 %	0,1 %	3,3 %	6,3 %	320,19	293,96	461,34	387,42	1240,27	927,9
ABS	3,5 %	2 %	0,1 %	3,3 %	3,7 %	216,97	294,12	305,89	388,2	806,6	928,2
GIMP	33 %	15 %	0,1 %	3,3 %	6,3 %	400,19	295,27	581,81	389,01	1576,36	930,04

Omitir las directivas de optimización también impacta en el uso de recursos aunque no de forma significativa, ya que se mantienen las tendencias del caso optimizado (GIMP consume más recursos que el resto y ABS es la que menos demanda). Todas las versiones tienen una leve reducción en el consumo de LUTS y Register, pero un pequeño incremento en BRAM y DSPSs. No se observan cambios en Muxes.

5. Conclusiones y Trabajo Futuro

El operador Sobel resulta ser un método popular para la detección de bordes en imágenes, existiendo numerosas variantes que ofrecen distintos compromisos entre costo computacional y calidad de filtrado. En este trabajo se presentaron tres variantes de un filtro Sobel sintetizados en lenguaje de alto nivel sobre una plataforma SoC y se analizaron sus prestaciones considerando tiempo de procesamiento, uso de recursos y calidad del filtrado. Entre los resultados experimentales más destacados se pueden mencionar:

- Modificar la fórmula para computar la magnitud del vector gradiente no aumentó la velocidad de procesamiento en los diseños optimizados. Las 3 variantes presentaron tiempos constantes para cada imagen de prueba.
- Sí se encontraron diferencias significativas al utilizar una variante u otra en cuanto al uso de recursos. En ese aspecto, ABS y GIMP se destacaron como las opciones que menos y más recursos consumen. Más allá de eso, ninguna tiene un consumo que implique una restricción de diseño para este sistema.
- Las tres versiones produjeron imágenes de salida similares pero no idénticas para cada entrada, por lo que su elección dependerá del propósito de su uso.

Como trabajo futuro, se espera desarrollar las tres variantes del algoritmo en bajo nivel (HDL) y replicar el análisis realizado a fin de verificar la tendencia de los resultados obtenidos.

Referencias

1. Introduction to FPGA design with vivado high-level synthesis (UG998) `https://bit.ly/31ICddK`

2. Vivado Design Suite User Guide: High-Level Synthesis https://bit.ly/2PHyHuB
3. Acharya, T., Ray, A.K.: Image Processing - Principles and Applications. Wiley-Interscience, USA (2005)
4. Ben Amara, A., Pissaloux, E., Atri, M.: Sobel edge detection system design and integration on an FPGA based HD video streaming architecture. IEEE, Hammamet, Tunisia (Dec 2016). https://doi.org/10.1109/IDT.2016.7843033
5. Chaple, G., Daruwala, R.D.: Design of Sobel operator based image edge detection algorithm on FPGA. IEEE, Melmaruvathur, India (Apr 2014). https://doi.org/10.1109/ICCSP.2014.6949951
6. Daoud, L., Zydek, D., Selvaraj, H.: A Survey of High Level Synthesis Languages, Tools, and Compilers for Reconfigurable High Performance Computing. vol. 240. Cham (2014). https://doi.org/10.1007/978-3-319-01857-$7_4$7
7. Hong, N., Belleudy, C., Pham, T.: Performance and evaluation sobel edge detection on various methodologies . https://doi.org/10.12720/ijeee.2.1.15-20
8. Kowalczyk, M., Przewlocka, D., Kryjak, T.: Real-time implementation of contextual image processing operations for 4k video stream in zynq UltraScale+ MPSoC. https://doi.org/10.1109/DASIP.2018.8597105
9. Millon, R., Frati, E., Rucci, E.: Implementación de Filtro de Detección de Bordes Sobel en SoC usando Síntesis de Alto Nivel. In: Actas del Congreso Argentino de Sistemas Embebidos (CASE 2020). pp. 73–75 (2020)
10. Monson, J., Wirthlin, M., Hutchings, B.L.: Optimization techniques for a high level synthesis implementation of the sobel filter. https://doi.org/10.1109/ReConFig.2013.6732315, ISSN: 2325-6532
11. Nane, R., Sima, V.M., Pilato, C., Choi, J., Fort, B., Canis, A., Chen, Y.T., Hsiao, H., Brown, S., Ferrandi, F., Anderson, J., Bertels, K.: A survey and evaluation of FPGA high-level synthesis tools . https://doi.org/10.1109/TCAD.2015.2513673
12. Nausheen, N., Seal, A., Khanna, P., Halder, S.: A FPGA based implementation of Sobel edge detection (Feb 2018). https://doi.org/10.1016/j.micpro.2017.10.011
13. Nguyen, H.T., Belleudy, C., Pham, T.V.: Power evaluation of sobel filter on xilinx platform. https://doi.org/10.1109/FTFC.2014.6828607
14. Nosrat, A., S. Kavian, Y.: Hardware description of multi-directional fast sobel edge detection processor by VHDL for implementing on FPGA . https://doi.org/10.5120/7533-9872
15. Pavan Kumar, M.: Hardware Acceleration of Edge Detection Using HLS (2019), https://bit.ly/3i88uBW, Undergraduate thesis, California State University
16. Rashmi, Kumar, M., Saxena, R.: Algorithm and Technique on Various Edge Detection : A Survey (Jun 2013). https://doi.org/10.5121/sipij.2013.4306
17. Rupnow, K., Liang, Y., Li, Y., Chen, D.: A study of high-level synthesis: Promises and challenges. https://doi.org/10.1109/ASICON.2011.6157401
18. Sanduja, V., Patial, R.: Sobel edge detection using parallel architecture based on FPGA https://www.ijais.org/archives/volume3/number4/220-0515
19. Sikka, P., Asati, A.R., Shekhar, C.: High-speed and area-efficient sobel edge detector on fpga for artificial intelligence and machine learning applications . https://doi.org/10.1111/coin.12334
20. Vallina, F.M., Kohn, C., Joshi, P.: Zynq all programmable SoC sobel filter implementation using the vivado HLS tool https://bit.ly/3h6egD1
21. Zheng, Y.: The design of sobel edge extraction system on FPGA **11**. https://doi.org/10.1051/itmconf/20171108001