Implementación de un algoritmo de Estimación de Movimiento para FPGAs utilizando OpenCL

Manuel de Castro[1], Roberto R. Osorio [2], Yuri Torres[1]
Arturo González-Escribano[1] y Diego R. Llanos[1]

Resumen— **La estimación de movimiento es una de las principales tareas detrás de cualquier codificador de vídeo. Es una tarea computacionalmente costosa, por lo que habitualmente se suele delegar a hardware específico o reconfigurable, como FPGAs. A lo largo de los años se han desarrollado múltiples implementaciones del algoritmo para FPGAs, utilizando principalmente lenguajes de descripción de hardware como Verilog o VHDL. Como la programación en estos lenguajes es compleja, es deseable la utilización de lenguajes de más alto nivel para desarrollar aplicaciones para FPGAs.**

En este trabajo presentamos una implementación paralela del algoritmo de estimación de movimiento por *block-matching* utilizando OpenCL para FPGAs. Nuestra propuesta procesa fotogramas Full HD completos dentro de la FPGA de forma eficiente. La propuesta ha sido desarrollada con la FPGA Intel Stratix 10 en mente, por lo que mostramos la utilización de recursos de nuestra propuesta cuando se sintetiza en una FPGA de este modelo. También mostramos una comparación de rendimiento entre nuestra propuesta y varias implementaciones CPU con distinto nivel de optimización y capacidades vectoriales. Este trabajo también busca evaluar OpenCL como herramienta para el desarrollo de aplicaciones FPGA, en términos de expresividad.

Palabras clave— **Codificación de Vídeo, Estimación de Movimiento, FPGA, OpenCL**

I. Introducción

LOS codificadores de vídeo avanzados se valen de la Estimación de Movimiento y Compensación de Movimiento para obtener altas relaciones de compresión. El movimiento es comúnmente estimado a través del *block matching* (coincidencia de bloques), que divide un fotograma dado en bloques de píxeles, y para cada uno de ellos trata de encontrar el bloque más parecido en uno o más fotogramas previamente codificados. De esta forma, grandes bloques de píxeles pueden ser codificados como vectores de movimiento, que son referencias espaciales a los bloques más parecidos en otro fotograma.

Block-matching es una tarea computacionalmente intensiva que habitualmente se implementa a través de hardware específico, incluyendo ASICs (*application specific integrated circuits*), FPGAs (*field programmable gate arrays*), hardware específico en GPUs (*graphic processing units*), y coprocesadores multimedia en procesadores de propósito general, como CPUs.

Por otra parte, una gran número de técnicas y heurísticas han sido propuestas para reducir la carga computacional de este algoritmo, aunque la búsqueda completa (*full search*) obtiene los mejores resultados. En ocasiones los codificadores de vídeo permiten seleccionar el nivel de precisión, de tal forma que el usuario puede priorizar tiempo de codificación o relación de compresión. Esto tiene sentido, ya que el codificador puede trabajar tanto en contextos de codificación en tiempo real, como en aplicaciones *offline* en las que el contenido es codificado solo una vez, pero transmitido, almacenado y reproducido una gran cantidad de veces.

En nuestra investigación, tratamos la aceleración de la Estimación de Movimiento utilizando FPGAs. Dada la gran cantidad de implementaciones posibles, tratamos de reducir el coste de desarrollo utilizando OpenCL, un lenguaje de alto nivel derivado de C. En este artículo detallamos la implementación más directa, búsqueda completa. Utilizando OpenCL, somos capaces de describir y sintetizar una implementación completamente paralela de alto rendimiento.

El principal objetivo de este trabajo es evaluar la expresividad de OpenCL como lenguaje de diseño para Estimación de Movimiento por *block-matching* y otras aplicaciones similares, a partir de la evaluación de la calidad de la implementación y comparándola con implementaciones optimizadas a mano.

El resto del artículo se estructura como sigue. La sección II explora el trabajo previo de implementación de algoritmos de Estimación de Movimiento en FPGAs. La sección III describe el algoritmo de Estimación de Movimiento y sus diversas variantes. La sección IV describe OpenCL, tanto a nivel general como a nivel de herramienta de desarrollo para aplicaciones HPC en FPGAs. La sección V detalla el proceso de desarrollo y las características de nuestra implementación del algoritmo. La sección VI evalúa tanto nuestra propuesta como OpenCL como herramienta de desarrollo de aplicaciones FPGA. Finalmente, la sección VII detalla las conclusiones de este trabajo, así como el trabajo futuro.

II. Trabajo relacionado

La Estimación de Movimiento por emparejamiento de bloques ha sido sujeto de numerosa investigación, desde algoritmos y arquitecturas básicas [1, 2], hasta

[1]Departamento de Informática, Universidad de Valladolid e-mail: {yuri.torres|arturo|diego}@infor.uva.es, mdcc.prof@gmail.com
[2]Departamento de Enxeñaría de Computadores, CITIC, Universidade da Coruña, España, e-mail: roberto.osorio@udc.es

implementaciones más sofisticadas en un esfuerzo por reducir la carga computacional [3–5] reduciendo la precisión de la solución.

Las implementaciones hardware presentan un gran interés, particularmente las que utilizan FPGAs como plataforma reconfigurable para implementar aceleradores. Sin embargo, los costes de desarrollo son generalmente muy elevados. Es por ello que el uso de lenguajes de alto nivel para describir arquitecturas es deseable. OpenCL es la opción más atractiva, debido a su amplia adopción, portabilidad entre plataformas, y reducción de complejidad de desarrollo en comparación con VHDL o Verilog.

Aunque existen trabajos previos que tratan implementaciones de *block-matching* utilizando OpenCL, todas ellas se enfocan en CPUs o GPUs [6–9]. Incluso las implementaciones del algoritmo en OpenCL de Intel no incluyen una para FPGAs, por lo que el lenguaje de síntesis de alto nivel de Xilinx es probablemente el lenguaje de más alto nivel para el que se han publicado arquitecturas del algoritmo de *block-matching*.

III. Estimación de Movimiento

En esta sección describimos el algoritmo de Estimación de Movimiento, ampliamente utilizado en los codificadores de vídeo.

La Estimación de Movimiento (ME) es un proceso por el cual se determinan los vectores de movimiento que describen los cambios ocurridos entre dos imágenes, normalmente fotogramas de un mismo vídeo. La Estimación de Movimiento tiene aplicación en diversas áreas tecnológicas, como seguimiento de objetos, interacción persona-computadora, interpolación temporal, filtrado espacio-temporal y compresión de vídeo. En este trabajo discutimos la Estimación de Movimiento en el marco de la compresión de vídeo, aunque el algoritmo utilizado pueda aplicarse a otros contextos.

La Estimación de Movimiento se puede realizar mediante métodos directos o indirectos. Los métodos directos trabajan sobre los píxeles del vídeo directamente, mientras que los métodos indirectos intentan detectar elementos en los fotogramas primero, para evitar falsos positivos de movimiento. El uso de los métodos directos está más extendido, destacando los métodos de correlación de fase, los derivados del cómputo del gradiente espacio-temporal (como el de flujo óptico y algoritmos de píxeles recursivos), y *block matching* (coincidencia de bloques). El método de *block matching* probablemente sea el más utilizado debido a su elevado grado de intuitividad, y es utilizado por la gran mayoría de algoritmos y estándares de compresión de vídeo existentes.

La importancia de ME en estándares de codificación de vídeo es tal que lleva utilizándose desde los orígenes de las mismas. Dentro de la codificación de vídeo, ME es la parte principal de la fase de predicción entre fotogramas. Es la causa de la mayor parte de las ganancias en compresión de cualquier estándar de vídeo, incluyendo el estándar más utilizado, AVC, y su sucesor, HEVC. Por tanto, ME, concretamente en su versión *block matching*, es uno de los principales causantes de la revolución multimedia que estamos viviendo, sino el que más. Sin embargo, también se trata de un algoritmo computacionalmente costoso, suponiendo entre el 62 % y el 94 % del tiempo de codificación en el estándar HEVC [10].

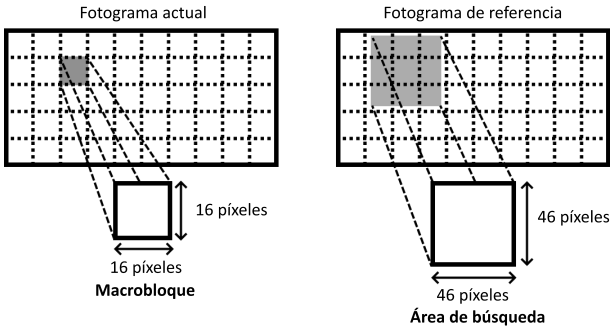

Fig. 1: División de un fotograma en macrobloques, y área de búsqueda correspondiente a un macrobloque dado, según el algoritmo de Estimación de Movimiento por *block-matching*.

Para realizar ME, los fotogramas del vídeo se dividen en bloques pequeños, denominados macrobloques (*macroblocks*). Los macrobloques habitualmente son de tamaño 16×16 píxeles, aunque su tamaño puede ser variable en los estándares de codificación más avanzados. ME es aplicado sobre los macrobloques del fotograma, tratando de encontrar para cada macrobloque el macrobloque más parecido de entre una lista de candidatos de un fotograma de referencia. Al encontrarse, no será necesario representar en el fichero de vídeo el macrobloque actual, solo el vector de movimiento con respecto al macrobloque al que se asemeja, el índice del fotograma de referencia, y el error de predicción. Por tanto, la cantidad de información que se debe almacenar en el fichero de vídeo se ve reducida de forma considerable.

El área de búsqueda de ME suele estar restringida, conformando una pequeña región que incluye los alrededores del macrobloque correspondiente al que se desea encontrar, pero en el fotograma de referencia. Esta técnica reduce considerablemente la complejidad computacional con respecto a buscar en el fotograma entero, sin comprometer excesivamente los resultados de la aplicación del algoritmo. Esto se debe a que, por lo general, se espera que los macrobloques se desplacen a posiciones cercanas entre fotogramas consecutivos; es decir, el macrobloque más parecido generalmente se encontrará cerca de su análogo en el fotograma de referencia. La figura 2 muestra la división de un fotograma en macrobloques y un ejemplo de elección de área de búsqueda para un macrobloque dado.

Existen múltiples variantes del algoritmo *block matching* de ME. La búsqueda completa (*Full Search*) computa todos los macrobloques candidatos, encontrando siempre la opción óptima. Otras búsquedas basadas en heurísticas reducen la cantidad de candidatos comparados para acelerar el cómputo, obteniendo una solución subóptima. Estas pue-

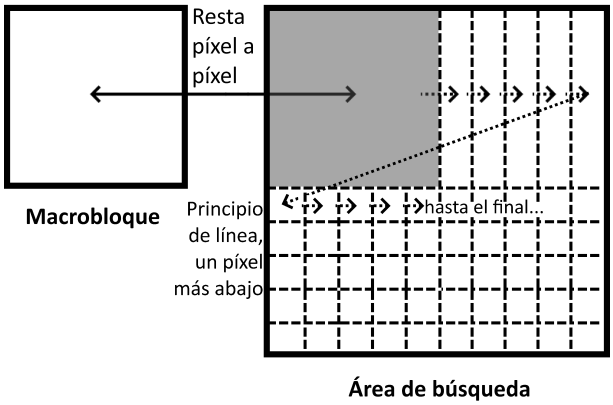

Fig. 2: Cálculo de similitud entre un macrobloque y todos los macrobloques candidatos del área de búsqueda. El valor de similitud es la suma de los valores absolutos de las 256 restas computadas (a menor valor, mayor similitud).

den ser la búsqueda en diamante (*Diamond Search*), la búsqueda en hexágono (*Hexagon Search*) y la búsqueda en zona de prueba (*Test Zone Search*).

Para encontrar el macrobloque más parecido al actual, se utilizan criterios de similitud. Ejemplos de estos criterios son la suma de diferencias absolutas (SAD) y la suma del cuadrado del error (SSE). SAD es la opción usada más comúnmente en estos contextos, debido a su baja complejidad computacional, pese a que la SSE sea más precisa. Su cómputo se realiza según la ecuación 1, donde w y h representan la anchura y altura del macrobloque, respectivamente, *Ref* el macrobloque candidato, y *Act* el macrobloque actual.

$$SAD = \sum_{j=0}^{w-1} \sum_{i=0}^{j-1} |Ref_{i,j} - Act_{i,j}| \qquad (1)$$

La figura **??** muestra cómo se calcularía la similitud con los múltiples candidatos del área de búsqueda siguiendo el algoritmo de búsqueda completa.

Se puede observar que el cómputo de SAD para cada macrobloque candidato es independiente. Dada la gran cantidad de bloques candidatos que deben procesarse por macrobloque, y la gran cantidad de macrobloques que hay que procesar por fotograma, ME ofrece grandes oportunidades de optimización mediante implementaciones paralelas.

IV. OpenCL para FPGAs de Intel

En esta sección se explica brevemente el modelo de programación para sistemas heterogéneos OpenCL. Comenzamos describiendo las características principales del mismo, y posteriormente las características y particularidades de su sistema de soporte para FPGAs de Intel.

A. Características principales de OpenCL

OpenCL [11] es un estándar abierto para la programación paralela de sistemas heterogéneos. Su objetivo principal es el de permitir al usuario escribir programas paralelos que sean portables entre distintos tipos de sistemas computacionales como CPUs,

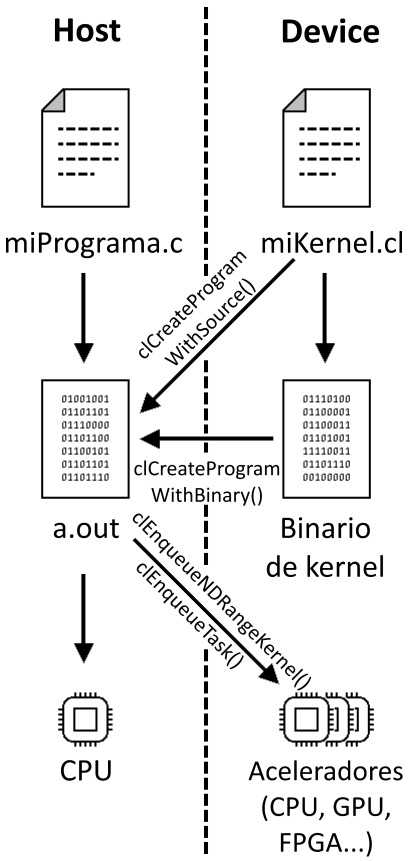

Fig. 3: Distintos componentes ejecutables de una aplicación OpenCL e interacciones entre ellos.

GPUs, y otro tipo de aceleradores, de forma que el mismo código fuente sea ejecutable en todos ellos y se pueda explotar al máximo el rendimiento de todos los recursos hardware de los sistemas heterogéneos.

El modelo de programación de OpenCL diferencia entre *host* (anfitrión) y *devices*, tanto a nivel de aplicación como a nivel de recursos del sistema. El sistema *host* es la CPU que ejecuta el código principal del programa cuando este es invocado, mientras que los *devices* son los dispositivos de cómputo del sistema, pudiendo ser CPUs, GPUs, FPGAs, y otros aceleradores. El código ejecutado por los *devices* se denomina *kernels*, y se escribe utilizando lenguaje OpenCL C (un dialecto de C99). El código *host* de la aplicación se encarga de coordinar los *devices* y configurar la invocación de los *kernels*, y se escribe comúnmente en C o C++, aunque también puede escribirse en otros lenguajes como Python. La figura 3 ilustra las interacciones entre la aplicación *host* y los kernels del *device* en la ejecución de programas OpenCL.

OpenCL permite dos formas de compilar los *kernels*: *online* y *offline*. La compilación *online* es un mecanismo de compilación JIT (*just in time*) que aumenta la portabilidad de los *kernels* al retrasar su compilación hasta el momento de ejecución del programa, antes de ser invocados. La compilación *offline* es equivalente a la compilación habitual de programas C o C++, compilando los *kernels* a ficheros binarios que son cargados por el código *host*.

La ejecución de *kernels* en dispositivos puede ser

de dos formas: *NDRange*, y de *task* (tarea). La forma habitual de ejecutar *kernels* es *NDRange*, que crea múltiples hilos para explotar paralelismo de datos dentro del *kernel*. Los *kernels* de tipo *task* solo se ejecutan en un hilo en el *device*.

B. OpenCL para FPGAs de Intel

Intel ofrece el software *Intel FPGA SDK for OpenCL* (al que nos referiremos como "OpenCL para FPGAs de Intel" a partir de ahora) dentro de su conjunto de herramientas *oneAPI*. Este permite la compilación de *kernels* de OpenCL para FPGAs de Intel, ajustándose al estándar OpenCL 1.0 de agosto de 2009. La compilación de *kernels* de OpenCL para FPGAs es un proceso que suele tardar varias horas en finalizar, por lo que OpenCL para FPGAs de Intel solo soporta el modo de compilación *offline*. Además, debido a los altos tiempos de compilación, OpenCL para FPGAs de Intel soporta un modo de compilación y ejecución en emulación de FPGAs para facilitar el desarrollo y la depuración de software para FPGAs de Intel. De esta forma, los *kernels* de OpenCL para FPGAs se pueden compilar para emulación, tardando el proceso pocos segundos, y ejecutar en CPU para verificar su corrección.

Durante la compilación de un *kernel* para FPGAs, el compilador de Intel genera un informe con información sobre las características del binario a generar. Entre ellas se encuentran estimaciones de la frecuencia operacional máxima del *kernel*, la cantidad de recursos de la FPGA utilizados, informes sobre el rendimiento de los bucles presentes en el código (en términos de intervalos de iniciación y latencias), uso de la jerarquía de memoria, y esquemáticos de los bloques hardware implementados.

Los *kernels* para FPGAs de Intel pueden ser tanto *NDRange* como *task*. Sin embargo, Intel recomienda programar los *kernels* para FPGAs como *kernels* de tipo *task* siempre que sea posible. Esto se debe a que, generalmente, ese tipo de *kernels* se adaptan mejor a la arquitectura de *pipeline* que presentan las FPGAs, pudiendo explotar eficientemente su paralelismo a nivel de instrucción. Los kernels de tipo *NDRange* se adaptan mejor a aceleradores vectoriales como las GPUs.

V. Implementación de la propuesta

En esta sección describimos nuestra implementación del algoritmo de Estimación de Movimiento como un *kernel* de OpenCL para FPGAs de Intel. Primero describimos el proceso de desarrollo de la propuesta y las versiones preliminares desarrolladas, y posteriormente las características de la versión final.

A. Proceso de desarrollo y versiones preliminares

Uno de los objetivos de este trabajo es el de evaluar OpenCL como herramienta de desarrollo de aplicaciones para FPGAs, especialmente en términos de capacidad expresiva. Es por ello que la aproximación que tomamos para desarrollar nuestra propuesta fue incremental, observando y analizando el resultado de la compilación (mediante el informe generado por el compilador) antes de añadir más características e incrementar la complejidad del *kernel*. De esta forma, pudimos analizar en más detalle el comportamiento del compilador de OpenCL para FPGAs, y observar cómo influían en el diseño los distintos parámetros de caracterización del kernel.

La propuesta desde un principio fue desarrollada para ser ejecutada utilizando una FPGA Intel Stratix 10 [12], modelo adecuado para la ejecución de aplicaciones utilizadas en centros de datos, computación en la nube, y otros contextos donde se requiere gran potencia de cómputo. Este tipo de FPGAs se conectan al sistema *host* a partir del puerto PCIe. La elección de esta FPGA influenció el proceso de desarrollo, dado que contábamos con que la cantidad de recursos de la FPGA no iba a suponer una limitación para la implementación del algoritmo.

Partimos de una versión de referencia para CPU accesible en [13], que portamos como *kernel* de tipo tarea de OpenCL, delegando tan solo el cálculo del SAD de cada macrobloque a la FPGA. El *kernel* recibía los macrobloques actual y un candidato como parámetros, calculaba su SAD, y lo retornaba al *host*. El tamaño de los macrobloques con los que trabaja nuestra propuesta es de 16×16,

Esta primera versión sirvió como toma de contacto con las herramientas y entornos de desarrollo de Intel. Sobre ella probamos distintos parámetros de configuración y caracterización que ofrece Intel en forma de *pragmas* (directivas de precompilador de C), para controlar el modo en el que se sintetizan ciertos elementos del *kernel*. Por ejemplo, existen *pragmas* para indicar que un bucle no debe segmentarse automáticamente, el intervalo de iniciación deseado para la segmentación de un bucle, y que un bucle debe ser desenrollado parcial o totalmente, entre otros.

Tras verificar que el compilador de Intel no tiene ningún problema en sintetizar un *kernel* tan simple como el de cálculo de SAD, decidimos implementar la operación de Estimación de Movimiento completa para un macrobloque dado, utilizando búsqueda completa. Esta versión recibía el macrobloque actual y el área de búsqueda completa como parámetros, computaba todos los SADs posibles, y retornaba su suma acumulada al *host*. El tamaño del área de búsqueda con el que trabaja nuestra propuesta es de 46×46; es decir, un área de búsqueda completa contiene un total de 900 macrobloques distintos, parcialmente solapados entre sí. El correcto funcionamiento de esta versión fue verificado utilizando datos sintéticos.

Finalmente, tras comprobar que este último *kernel* también era sintetizado correctamente a una implementación hardware razonable, procedimos a implementar un *kernel* que realizase el procesamiento de un fotograma completo en la FGPA. Tal *kernel* se describe a continuación.

B. Características de la versión final de la propuesta

Hemos desarrollado una implementación para FPGAs del algoritmo de búsqueda completa de Estimación de Movimiento por *block-matching* utilizando OpenCL para FPGAs de Intel. La versión desarrollada procesa un fotograma Full HD ($1\,920 \times 1\,080$ píxeles) completamente dentro de la FPGA, sin intervención del *host*. El tamaño de los macrobloques del algoritmo es de 16×16 píxeles, y el área de búsqueda es de 46×46 píxeles. Los fotogramas se amplían en altura 8 píxeles (a $1\,920 \times 1\,088$ píxeles), replicando los píxeles de la última línea en las líneas adicionales, para que cada fotograma sea perfectamente divisible a macrobloques exactos. El número de macrobloques procesados por la FPGA es de 8 160 por fotograma, lo que resulta en 7 344 000 vectores de movimiento computados por fotograma, o 1,88 mil millones de operaciones de SAD.

El kernel recibe por parámetros dos punteros a dos fotogramas, uno sobre el que aplicar ME y uno de referencia. Se trabaja exclusivamente con la componente de luminancia de los fotogramas (equivalente a trabajar con los fotogramas transformados a escala de grises), y cada píxel es representado con 8 bits (1 byte). También recibe tres punteros donde almacenar los resultados de la Estimación de Movimiento a retornar al *host*: el SAD mínimo encontrado para cada macrobloque, la coordenada x del vector de movimiento correspondiente a ese SAD, y la coordenada y del vector de movimiento correspondiente a ese SAD.

Nuestra propuesta optimiza el uso de memoria de la FPGA al utilizar registros o memoria RAM interna para almacenar el macrobloque actual y el área de búsqueda. Dichos arrays se declaran como memoria local de OpenCL, que el compilador es capaz de implementar como RAM interna (*embedded SRAM* o bloques *M10K*), o en caso de ser muy pequeña, incluso como registros (bloques de memoria denominados *MLAB*). El macrobloque actual, cuyo tamaño es de 256 bytes, se implementa utilizando 128 MLABs, lo que permite la lectura concurrente de sus 256 píxeles. El área de búsqueda, cuyo tamaño es de 2 116 bytes, se implementa utilizando 2 RAMs internas, replicadas 256 veces, para un total de 512 RAMs internas. Como las memorias que implementan el área de búsqueda están replicadas 256 veces, se permite una lectura concurrente de 256 de sus píxeles. Esto implica que nuestra propuesta efectúa el cómputo de las 256 operaciones de SAD necesarias para comparar un macrobloque candidato de forma completamente paralela.

El flujo de operaciones del kernel es el que sigue. El *kernel* itera sobre los macrobloques del fotograma actual ejecutando las siguientes operaciones: (1) Se carga el macrobloque desde fotograma actual en memoria global a memoria local (implementada como registros MLAB). (2) Se carga el área de búsqueda desde el fotograma de referencia en memoria global a memoria local (implementada con bloques RAM interna). (3) Se itera sobre los 900 macrobloques candidatos, computando su SAD con el macrobloque actual, y almacenando en los vectores de resultados el SAD mínimo junto con su vector de movimiento correspondiente. (4) Comienza la siguiente iteración (paso (1)), con el siguiente macrobloque.

Los bordes y esquinas de los fotogramas suponen un problema en el sentido de que el área de búsqueda no puede expandirse completamente en todas las direcciones. Es necesario realizar ajustes en el algoritmo para evitar accesos fuera de los límites de memoria de los fotogramas. En este trabajo hemos desarrollado dos alternativas. La primera alternativa trabaja con fotogramas preprocesados para tener bordes extendidos 30 píxeles en cada dirección, de manera similar a como trabajan algunos estándares de codificación de vídeo como AVC. De esta forma, no es necesario añadir ninguna lógica compleja para lidiar con los bordes. Tan solo hace falta añadir un offset de 15 píxeles en cada dirección a las coordenadas de todos los macrobloques. La segunda alternativa añade lógica de control que ajusta dinámicamente el tamaño del área de búsqueda. En esta versión de la propuesta, para los macrobloques que se localizan cercanos a los bordes del fotograma se computan

	ALMs	**REG**	**MLAB**	**RAM**	**DSP**
Sistema completo	244 913 (26 %)	421 377 (11 %)	990 (1 %)	1 187 (10 %)	0 (0 %)
Sistema de kernels	50 536 (5 %)	137 020 (4 %)	990 (1 %)	756 (6 %)	0 (0 %)
Lógica del kernel de ME (estimado)	20 922.5 (2 %)	72 446 (2 %)	1 440 (2 %)	663 (6 %)	0 (0 %)
Disponibles	933 120	3 732 480	93 312	11 721	5 760

Tabla I: Utilización de recursos de la versión de nuestra propuesta que trabaja con fotogramas con bordes extendidos, según reporta el informe producido por el compilador de Intel (*aoc*).

	ALMs	**REG**	**MLAB**	**RAM**	**DSP**
Sistema completo	247 311 (27 %)	433 503 (12 %)	783 (1 %)	1 198 (10 %)	3 (0 %)
Sistema de kernels	52 468.9 (6 %)	149 222 (4 %)	783 (1 %)	767 (7 %)	5 (0 %)
Lógica del kernel de ME (estimado)	24 122 (3 %)	92 840 (2 %)	1 294 (1 %)	678 (6 %)	2.5 (0 %)
Disponibles	933 120	3 732 480	93 312	11 721	5 760

Tabla II: Utilización de recursos de la versión de nuestra propuesta que añade lógica de detección de bordes, según reporta el informe producido por el compilador de Intel (*aoc*).

menos vectores de movimiento posibles que para los macrobloques interiores, ya que su área de búsqueda presenta menos de los 900 candidatos habituales. Esto podría resultar en un aumento en el rendimiento de la implementación, a costa de los recursos necesarios para implementar la lógica de control adicional.

VI. Evaluación de la propuesta

En esta sección se realiza una evaluación de nuestra propuesta, primero en términos de recursos utilizados de la FPGA, y posteriormente en términos de rendimiento, tanto a nivel de frecuencia de operación como en comparación con otras implementaciones. También se realiza una breve evaluación de OpenCL como herramienta para el desarrollo de aplicaciones FPGA.

A. Recursos utilizados

La propuesta ha sido compilada para FPGAs Stratix 10 de Intel, un modelo de FPGA que cuenta con una gran cantidad de recursos ya que está enfocado a contextos de computación intensiva. El kernel se ha compilado utilizando *aoc*, el compilador OpenCL *offline* de Intel para FPGA. Por debajo, *aoc* utiliza *Quartus* para generar el bitstream para programar la FPGA. Los informes de compilación reportan que la versión de *Quartus* utilizada es la 19.2.0 Build 57 Pro, y que la versión de SYCL (que incluye OpenCL para FPGAs de Intel) utilizada es 2022.2.0 Build 133.4.

Las tablas I, II muestran respectivamente los recursos utilizados, en términos absolutos y relativos, por la versión de la propuesta que trabaja con fotogramas con bordes extendidos y por la versión de la propuesta que añade lógica para procesar los macrobloques cercanos a los bordes del fotograma. Las tablas muestran en las dos primeras filas los recursos reales utilizados por la implementación para (1) el sistema completo incluyendo la lógica asociada a la API de OpenCL (2) el sistema de *kernels*, que incluye el propio *kernel*, además de la interconexión y ROM de descripción del sistema necesarios para ejecutar cualquier *kernel* compilado. La tercera fila muestra una estimación de los recursos utilizados tan solo para la lógica asociada al *kernel* desarrollado. Los informes producidos por el compilador de Intel no muestran los recursos reales utilizados exclusivamente para cada *kernel* particular, por lo que tenemos que utilizar los datos que se estiman en un paso intermedio de compilación, antes de comenzar la compilación a bitstream de FPGA. Al contener la tercera fila datos estimados y no reales, existen discrepancias con las dos filas anteriores. Por otra parte, la tabla II muestra que el sistema completo utiliza 3 DSPs, y que a la vez el (sub)sistema de *kernels* utiliza 5. No tenemos una explicación para esta discrepancia, más que pueda ser un error por parte del compilador de Intel a la hora de generar el informe.

Se puede observar que la cantidad de recursos utilizados por nuestra propuesta es relativamente pequeña. El sistema completo, incluyendo la lógica necesaria para implementar las APIs de OpenCL ocupa aproximadamente la cuarta parte del área de la FPGA. Para el *kernel* en sí mismo, la utilización de recursos no llega al 5 % en todos los apartados menos las RAMs, que alcanzan el 6 %. Estos resultados ilustran la gran cantidad de recursos disponibles en una FPGA Intel Stratix 10. En términos absolutos, la utilización de recursos por nuestra propuesta es considerable, especialmente si se compara con implementaciones de bajo nivel para FPGAs embedidas o en sistemas-en-chip (SoCs).

	Frecuencia
Versión con bordes extendidos	316.00 MHz
Versión con lógica de detección de bordes	308.00 MHz

Tabla III: Frecuencias de operación de las dos versiones del *kernel* de Estimación de Movimiento desarrolladas, según reporta el informe producido por el compilador de Intel (*aoc*)

B. Rendimiento

El informe producido por el compilador de Intel no reporta una estimación de los ciclos de reloj que tomaría la ejecución completa de un *kernel*, como sí que hacen otras herramientas de síntesis de alto nivel como las de Xilinx. Por tanto, no se puede presentar una estimación teórica del tiempo de ejecución de nuestra propuesta.

La tabla III muestra las frecuencias de operación reales para las dos versiones del *kernel* que hemos desarrollado, según reporta el informe generado por el compilador de Intel. Se puede observar cómo la versión que añade lógica de detección de bordes funciona a una frecuencia de operación ligeramente inferior que la versión que trabaja con fotogramas con bordes extendidos. Pese a que la versión que añade lógica de detección de bordes compute en total un menor número de SADs que la versión que trabaja con fotogramas con bordes extendidos, que trabaje a una frecuencia de reloj menor puede implicar que no obtenga un mayor rendimiento; especialmente considerando que el número de macrobloques beneficiados por la reducción de vectores de movimiento es una minoría.

Hemos desarrollado una experimentación comparativa de los dos *kernels* desarrollados con múltiples versiones CPU de referencia. En esta experimentación, ejecutamos Estimación de Movimiento sobre dos fotogramas en escala de grises (componente de luminancia) de un vídeo Full HD, de forma reiterada. Cada iteración, los punteros de cada fotograma se intercambian, efectivamente intercambiando el fotograma actual y el fotograma de referencia. La corrección de los resultados se realiza reportando la suma acumulada de todos los SADs mínimos hallados durante el procesado de cada fotograma. Como resultado de rendimiento, hemos medido el tiempo de ejecución de 500 repeticiones de la Estimación de Movimiento sobre los mismos dos fotogramas, simulando así el procesamiento de 1000 fotogramas de

vídeo.

Nuestra experimentación se ha llevado a cabo en la plataforma DevCloud de Intel. Esta plataforma permite trabajar con FPGAs de Intel, incluyendo modelos Arria 10 y Stratix 10, utilizando la suite de herramientas de oneAPI, que incluye OpenCL para FPGAs de Intel. En nuestro caso, las ejecuciones de las versiones FPGA se han realizado en una FPGA Stratix 10, y las ejecuciones de las versiones CPU en una CPU Xeon Platinum 8256.

Las distintas implementaciones del algoritmo ejecutadas y comparadas son las siguientes: (1) una versión secuencial, compilada con nivel de optimización -O2; (2) una versión que utiliza funciones vectoriales intrínsecas MMX (registros vectoriales de 8 bytes); (3) una versión que utiliza funciones vectoriales intrínsecas SSE (registros vectoriales de 16 bytes); (4) la misma versión secuencial, compilada con nivel de optimización -O3, que autovectoriza satisfactoriamente los bucles de cómputo utilizando registros e instrucciones SSE; (5) la versión FPGA que añade lógica de detección de bordes; y (6) la versión FPGA que trabaja con fotogramas con bordes extendidos. Los resultados obtenidos de la experimentación se presentan en la tabla IV. Se puede observar que las dos implementaciones FPGA desarrolladas obtienen un rendimiento similar a la versión CPU con máxima optimización (-O3, incluyendo autovectorización SSE); siendo esperable que el consumo energético de las versiones FPGA sea considerablemente menor. Ente las dos versiones FPGA desarrolladas, la que trabaja con fotogramas con bordes extendidos obtiene un mejor rendimiento, probablemente debido a que su frecuencia de trabajo es ligeramente superior.

C. *Evaluación de OpenCL como herramienta para el desarrollo para FPGAs*

Tras desarrollar la propuesta, consideramos que OpenCL es una opción interesante a la hora de desarrollar aplicaciones HPC para FPGAs, contando con gran cantidad de beneficios. Logra abstraer de forma eficiente los conceptos de electrónica de bajo nivel del programador, permitiendo escribir aplicaciones para FPGAs utilizando código C, al que la mayoría de programadores de aplicaciones HPC están acostumbrados. También, los informes de compilación resultan de utilidad de cara a identificar los puntos del código que limitan las mejoras de rendimiento, además de proporcionar información detallada sobre con qué hardware se implementan los distintos elementos del código.

En general, OpenCL cuenta con una ventaja clave sobre cualquier otro lenguaje de síntesis de alto nivel, siendo esta su amplia adopción por parte de desarrolladores para sistemas heterogéneos, especialmente en los ámbitos del desarrollo para GPUs. Al ser común el aprendizaje de OpenCL para programar cualquier sistema heterogéneo, OpenCL es una de las mejores opciones para comenzar a desarrollar aplicaciones HPC para FPGA, frente a tener que aprender una tecnología distinta desde cero.

Sin embargo, también hemos detectado carencias importantes de OpenCL como herramienta de desarrollo para FPGAs, especialmente cuando se compara con otras alternativas. Una de las principales es que rompe con la portabilidad que pretende ofrecer OpenCL: al funcionar las FPGAs mejor con *kernels* de tipo tarea frente a *NDRange*, la escritura de *kernels* para FPGAs resulta diferente a la de *kernels* para otros aceleradores más ampliamente usados (GPUs o CPUs). El programador tiene que escribir dos *kernels* considerablemente distintos si quiere explotar al máximo las capacidades de aceleradores vectoriales y FPGAs a la vez. También limita al usuario a la utilización del estándar 1.0 de OpenCL, siendo el último estándar de OpenCL a día de hoy el 3.0. Por otra parte, hemos tenido problemas con el sistema de emulación de FPGAs que ofrece Intel, no pudiendo compilar los *kernels* cuando se utilizan ciertos *pragmas* para optimizar a grano fino su rendimiento en la FPGA.

Consideramos una limitación importante que el informe generado por el compilador no reporte ninguna estimación del tiempo en ciclos de reloj que tomaría la ejecución completa de un *kernel* o de alguna de sus secciones. La información proporcionada por el *Scheduler Viewer (Beta)*, que trata de localizar en una línea de tiempo la ejecución de los bloques que conforman el diseño, no nos resultó fidedigna, o por lo menos no supimos interpretarla. Esta limitación destaca especialmente cuando se realiza una comparación con las herramientas de síntesis de alto nivel

Versión	**milisegundos / fotograma**	**fotogramas por segundo (fps)**
Referencia secuencial (-O2)	1627.39	0.614
-O2 + MMX	145.31	6.882
-O2 + SSE	126.64	7.896
-O3 (autovectoriza con SSE)	89.49	11.174
FPGA - versión con lógica de detección de bordes	90.12	11.096
FPGA - versión que trabaja con fotogramas extendidos	88.43	11.309

Tabla IV: Comparación de rendimiento entre las dos versiones FPGA desarrolladas y varias implementaciones CPU.

que ofrece Xilinx, que sí que proporcionan una estimación en ciclos de reloj de la latencia de los diseños implementados.

Como conclusión sobre OpenCL para FPGAs de Intel, consideramos que supone una oportunidad muy prometedora para el desarrollo de aplicaciones HPC para FPGAs, pero que todavía le falta algo de madurez para optimizar el flujo de desarrollo para FPGAs, como logran otras herramientas.

VII. Conclusiones

En este trabajo hemos implementado un algoritmo de Estimación de Movimiento por *block-matching* para FPGAs utilizando OpenCL para FPGAs de Intel (*Intel FPGA SDK for OpenCL*). Nuestra propuesta procesa fotogramas Full HD ($1\,920 \times 1\,080$ píxeles) en escala de grises, utilizando búsqueda completa. El procesamiento de cada fotograma completo se realiza dentro de la FPGA, sin intervención del *host*. Hemos desarrollado dos versiones del *kernel*: una que trabaja con fotogramas con bordes extendidos, y otra que añade lógica de detección de bordes para evitar los accesos fuera de los límites de memoria. Nuestra propuesta realiza las operaciones de SAD de forma completamente paralela, y es capaz de alcanzar un alto rendimiento gracias a la explotación de la memoria interna de la FPGA mediante el uso de memoria local de OpenCL.

Nuestra propuesta ha sido diseñada para ser sintetizada en una FPGA Intel Stratix 10, apropiada para computaciones intensivas. La utilización de recursos de este modelo de FPGA por parte de nuestra propuesta es baja en valores porcentuales: menos del 10 % para el *kernel*, y aproximadamente el 25 % para el sistema completo.

El trabajo futuro incluye más experimentación, para evaluar el rendimiento de nuestra propuesta en comparación con: (1) una versión paralela usando OpenMP, (2) una versión GPU utilizando OpenCL, y (3) una implementación en VHDL. También incluye el desarrollo de una versión de la propuesta que utilice registros de desplazamiento para mejorar la utilización de recursos, y expandir las implementaciones a otro tipo de técnicas de búsqueda más eficientes, como la búsqueda en diamante.

Agradecimientos

Esta investigación ha sido financiada parcialmente por el Ministerio de Economía, Industria y Competitividad y por el programa ERDF de la Unión Europea a través del proyecto PCAS (TIN2017-88614-R); la Junta de Castilla y León - FEDER Grants a través de los proyectos PROPHET y PROPHET-2 (VA082P17, VA226P20); la Fundación General de la Universidad de Valladolid a través del contrato 062/204051; el Ministerio de Ciencia e Innovación (PID2019-104184RB-I00, AEI/FEDER/EU, 10.13039/501100011033); y por la Xunta de Galicia y fondos FEDER (Centro de Investigación de Galicia acreditación 2019-2022, ref. ED431G 2019/01, así como el Programa de Consolidación y Estructuración de Unidades de Investigación Competitivas, ref. ED431C 2017/04).

Referencias

[1] Toshio Koga, "Motion compensated inter-frame coding for video conferencing," 1981.

[2] Liang-Gee Chen, Wai-Ting Chen, Yeu-Shen Jehng, and Tzi-Dar Chiuch, "An efficient parallel motion estimation algorithm for digital image processing," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 1, no. 4, pp. 378–385, 1991.

[3] Mohammed Ghanbari, "The cross-search algorithm for motion estimation," *IEEE Transactions on Communications - TCOM*, vol. 38, 07 1990.

[4] Reoxiang Li, Bing Zeng, and M.L. Liou, "A new three-step search algorithm for block motion estimation," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 4, no. 4, pp. 438–442, 1994.

[5] Shan Zhu and Kai-Kuang Ma, "A new diamond search algorithm for fast block-matching motion estimation," *IEEE Transactions on Image Processing*, vol. 9, no. 2, pp. 287–290, 2000.

[6] Erich Marth and Guillermo Marcus, "Parallelization of the x264 encoder using opencl," 01 2010.

[7] Jinglin Zhang, J.F. Nezan, and Jean-Gabriel Cousin, "Implementation of motion estimation based on heterogeneous parallel computing system with opencl," 06 2012, pp. 41–45.

[8] Usman Shahid, Ashfaq Ahmed, Maurizio Martina, Guido Masera, and Enrico Magli, "Parallel h.264/avc fast rate-distortion optimized motion estimation by using a graphics processing unit and dedicated hardware," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 25, pp. 1–1, 04 2015.

[9] Mateus Melo, Gustavo Smaniotto, Henrique Maich, Luciano Agostini, Bruno Zatt, Leomar Rosa, and Marcelo Porto, "A parallel motion estimation solution for heterogeneous system on chip," in *2016 29th Symposium on Integrated Circuits and Systems Design (SBCCI)*, 2016, pp. 1–6.

[10] Vladimir Afonso, Henrique Maich, Luan Audibert, Bruno Zatt, Marcelo Porto, and Luciano Agostini, "Memory-aware and high-throughput hardware design for the hevc fractional motion estimation," in *2015 28th Symposium on Integrated Circuits and Systems Design (SBCCI)*, 2015, pp. 1–6.

[11] Khronos OpenCL Working Group et al., "The OpenCL Specification, version 1.0. 29, 8 December 2008," .

[12] Intel, "Intel Stratix 10 FPGAs & SoC FPGA," `https://www.intel.com/content/www/us/en/products/details/fpga/stratix/10.html`, último acceso: junio de 2022.

[13] Karl Olav Lillevold and Robert Danielsen, "TMN encoder," 1996, `https://gitlab.inria.fr/citi-lab/dycton/-/blob/master/src/platform_tlm/software/h263/mot_est.c`, último acceso: junio de 2022.