Diseño de una Arquitectura de Flujo de Datos para la Estimación de Movimiento en Tiempo Real Mediante la Técnica de Búsqueda Exhaustiva de Macrobloques

Eduardo Serrano, Jesús Barba, Julián Caba, M. Soledad Escolar,
Manuel J. Abaldea, Fernando Rincón, Juan Carlos López
Escuela Superior de Informática
Universidad de Castilla-La Mancha
Ciudad Real, España

Resumen— **La estimación de movimiento es la etapa de mayor coste computacional en los estándares de compresión de vídeo, los cuales tratan de reducir la cantidad de datos aprovechando la redundancia temporal existente entre dos *frames* consecutivos. Aunque el mecanismo es simple - dado un macrobloque el algoritmo tiene que encontrar su mejor emparejamiento dentro de una zona de búsqueda - su coste computacional es elevado. El mejor método, *Full Search* o búsqueda exhaustiva, utiliza un enfoque de fuerza bruta, el cual no es apropiado para aplicaciones en tiempo real.**

Este trabajo introduce una propuesta de arquitectura para FPGAs que implementa el algoritmo de estimación de movimiento mediante la técnica de búsqueda exhaustiva de macrobloques (*FSBM*). La solución propuesta ha sido modelada con la herramienta Vivado HLS en lenguaje C++, implementándose en la placa de prototipado ZC702 de Xilinx. El IP implementa una arquitectura de flujo de datos para el procesamiento en tiempo real de una fuente de vídeo. La arquitectura propuesta es configurable para adaptarse a diferentes alternativas.

Los resultados obtenidos en placa muestran una frecuencia de fotogramas de 746fps, 247fps y 110fps para resoluciones VGA, HD y Full HD, respectivamente. Con una frecuencia de reloj de 115Mhz, el IP consume en la FPGA un tercio de los FF y BRAMs y un 60 % de LUT.

Palabras clave— **FPGA, Síntesis de Alto Nivel, HLS, Estimación de Movimiento, Búsqueda Exhaustiva, Macro-bloques, Visión por Computador**

I. Introducción

MUCHOS estándares de codificación de vídeo (e.j. H.263, H.264, MPEG, ITU-T) hacen uso de técnicas de Estimación de Movimiento (ME) para eliminar redundancia temporal entre frames. Con el rápido crecimiento de las aplicaciones de vídeo y la mejora de resolución de las imágenes, la fase de ME se ha vuelto cada vez más crítica, alcanzando entre un 60 % y 80 % del tiempo total, dependiendo de la estrategia elegida para realizar la implememtación del algoritmo de Block Matching (BM).

Para cada macrobloque (MB), grupo de NxN píxeles, el algoritmo de BM trata de encontrar su mejor ajuste dentro de una zona de búsqueda ($(N+p)x(N+p)$), donde el parámetro p equivale al área de búsqueda. El objetivo es encontrar, en base a un criterio de similitud, la posición relativa del macrobloque dentro de la zona de búsqueda. Este proceso se repite para todos los macrobloques del frame.

El algoritmo *Full-Search Block Matching* (FSBM) consigue resultados de mayor precisión respecto a otras implementaciones, ya que realiza todas las comparaciones posibles de un macrobloque dentro de la zona de búsqueda. Pero esta precisión se traduce en un coste computacional prohibitivo para aplicaciones de tiempo real.

Sin embargo, las características que una plataforma basada en FPGA (*Field Programmable Gate Array*) ofrece permite solventar este inconveniente, alcanzando una ejecución en tiempo real.

Hasta hace poco tiempo, los lenguajes empleados en el desarrollo de soluciones para FPGA requerían de una alta curva de aprendizaje para introducirse en el diseño de tecnología de lógica reconfigurable. Sin embargo, con la aparición de entornos de trabajo HLS (del inglés *High Level Synthesis*), esta curva de aprendizaje se ha reducido, mejorando el acceso a esta tecnología.

En este trabajo, se detalla una arquitectura - diseñada en C++ con la herramienta Vivado HLS - que implementa el algoritmo FSBM, capaz de analizar vídeo en tiempo real.

Existen diferentes propuestas para solventar este problema, mayoritariamente se centran en la implementación de los módulos encargados de calcular la función de similitud entre dos macrobloques ([1], [2]). En cambio, el trabajo aquí presentado aborda una mejora global del algoritmo, introduciendo nuevas propuestas no contempladas anteriormente.

En [3] se aplica una técnica denominada *Online Arithmetic* que permite acelerar el cálculo del operador SAD. El acelerador propuesto es capaz de procesar 17.2 fps en formato VGA utilizando el dispositivo Virtex-II con una frecuencia de reloj de 425 Mhz y un tamaño de macrobloque de 16x16. Mientras que nuestra solución para VGA es capaz de procesar 186 fps con una frecuencia de reloj de 125 Mhz.

Las arquitecturas de tipo array sistólico son también consideradas como una solución óptima para la implementación del algoritmo FSBM por el uso óptimo de recursos, bajo consumo energético y configu-

rabilidad [4] [5]. Por ejemplo, [6] presenta un procesador escrito en VHDL capaz de procesar 60 fps (resolución CIF, reloj 192Mhz) en una FPGA Virtex-II para un tamaño de bloque de 8x8, ocupando un 11 % del área de la FPGA. Aunque es complicado proyectar una comparación, una configuración idéntica ($N = 16$ y $p = 8$) de la arquitectura propuesta es capaz de obtener el mismo rendimiento en resoluciones mucho más exigentes. Además, hay que tener en cuenta que en este trabajo se incluye toda la lógica que mueve la imagen desde memoria, y sería interesante ver cómo estas propuestas escalan al incorporar esta funcionalidad y aumenta la resolución del flujo de vídeo.

Nuno et. al validan en [7] varias estrategias de optimización para alcanzar cotas de eficiencia mayores en este tipo de arquitecturas sistólicas, todo ello sin bajar la calidad del resultado. Algunas de estas técnicas (ej. reducción de precisión de píxel) son de interés y su utilización en futuras versiones de la arquitectura con el objetivo de reducir recursos y mejorar los resultados del IP FSBM.

II. Descripción general de la arquitectura

El principal objetivo en el diseño del IP FSBM es realizar el cálculo de los vectores de movimiento, a partir de un flujo de datos, con el máximo rendimiento posible. Para este fin, se ha desarrollado una arquitectura de procesamiento de vídeo para la placa de prototipado ZC702 de Xilinx (ver Figura 1).

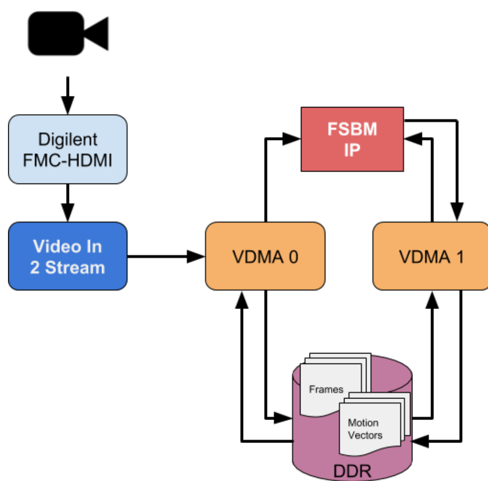

Fig. 1.Plataforma de procesamiento de vídeo para el cálculo de los vectores de movimiento.

Conforme se van recibiendo los frames desde la fuente de vídeo (tarjeta capturadora Digilent FMC-HDMI), éstos son almacenados en memoria DDR, en formato YUV 4:2:2 (16 bit/pixel). Dos Vídeo DMA se encargan de suministrar los datos de los dos frames que alimentan al IP FSBM. La sincronización entre los VDMA se realiza tanto a nivel hardware (configuración dinámica [1]) como software (configuración de un buffer circular).

Además de perseguir la máxima productividad (idealmente un ciclo de procesamiento por píxel), en el diseño se persigue que el uso de recursos sea el mínimo posible.

figure

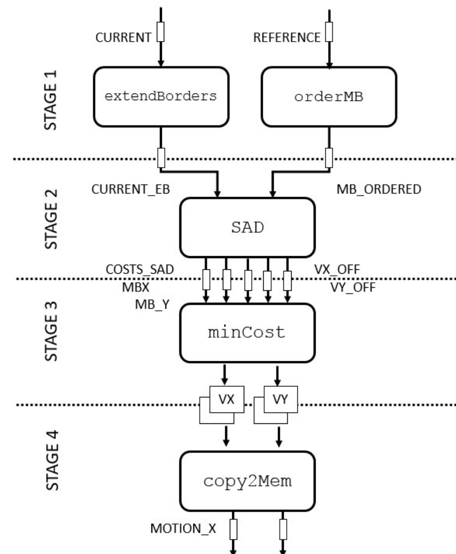

Fig. 2. Arquitectura propuesta de dataflow en el IP FSBM.

El modelo del IP FSBM consta de cuatro etapas, tal como se muestra en la Figura 2. A continuación se describe el trabajo realizado por cada etapa:

- Etapa 1: Recepción de datos y acomodación. Por un lado, se extienden los bordes del frame etiquetado como *CURRENT* para reducir la complejidad de los cálculos en etapas posteriores. Para el frame etiquetado como *REFERENCE* se realiza un reordenamiento y encapsulado de los macrobloques.
- Etapa 2: Función de coste. Para cada macrobloque de *REFERENCE* se calcula la similitud con todos los macrobloques de la zona de búsqueda. La función de coste implementada es *Sum of Absolute Differences* (SAD).
- Etapa 3: Selección del macrobloque con el mínimo coste dando lugar a la actualización de las coordenadas del vector de movimiento.
- Etapa 4: Copia de los vectores de movimiento en memoria externa.

LISTADO 1 muestra el modelo C++ simplificado para Vivado HLS. Debido a las restricciones de espacio, sólo se muestran las estructuras de datos utilizadas por los parámetros de la función principal FSBM, las directivas `INTERFACE` no aparecen. El modelo está parametrizado gracias a la utilización de directivas *#define* que permiten elegir diferentes configuraciones de resolución, tamaño de macrobloque y zona de búsqueda.

Todas las etapas reciben y consumen los datos en forma de stream, implementados como canales FIFO, evitando la necesidad de utilizar memorias intermedias con estrategia de *ping-pong buffers* que requieren BRAMs. La única excepción a esta regla se produce entre la etapa 3 y 4 que utilizan dos *ping-pong buffer* para comunicar los valores de Vx y Vy. Así, para las diferentes resoluciones y un tamaño de ma-

[1] *Genlock Synhcronization* (página 39), Xilinx AXI Video Direct Memory Access v6.2 Product Guide (PG020).

crobloque de 16x16 (utilizado en este trabajo durante la fase experimental de obtención de resultados del IP FSBM), las dimensiones de las matrices son 40x30 (VGA), 80x45 (HD) y 120x68 (FHD) con elementos de 4 bits. Las memorias BRAMs utilizadas por Vivado HLS para realizar el mapeo de las variable son duplicadas para implementar la sincronización por medio de la estrategia *ping-pong buffer*.

LISTADO 1
MODELO HLS DE LA ARQUITECTURA DE FLUJO DE DATOS DEL IP FSBM.

```
1 #if defined(SA_16)
2 typedef ap_uint<4> MB_OFFSET_T;
3 #define SA 16
4 #endif
5
6 #if defined(VGA)
7 #define PIXELS_H 640
8 #define PIXELS_V 480
9 #define V_MB 30
10 #define H_MB 40
11  typedef ap_uint<6> MB_X_T;
12 #endif
13
14 #if defined (INPUT_BUS_WIDTH_16)
15 #define PIXELS_WORD 1
16 typedef hls::Mat<PIXELS_V, PIXELS_H, HLS_8UC4>
          YUV_IMAGE_T;
17 typedef hls::stream<ap_axiu<16,1,1,1>>
       AXI_STREAM;
18 #elif defined(INPUT_BUS_WIDTH_32)
19 #define PIXELS_WORD 2
20 typedef hls::Mat<PIXELS_V, PIXELS_H, HLS_8UC4>
       YUV_IMAGE_T;
21 typedef hls::stream<ap_axiu<32,1,1,1>>
       AXI_STREAM;
22 #elif defined(INPUT_BUS_WIDTH_64)
23 ...
24 #endif
25
26 #define NPIXELS_IMG (PIXELS_H*PIXELS_V)
27 void FSBM(AXI_STREAM& IMG_REF, AXI_STREAM&
       IMG_CURRENT, MB_OFFSET_T MOTION_X[V_MB][
       H_MB], MB_OFFSET_T MOTION_Y[V_MB][H_MB]){
28 ...
29     YUV_IMAGE_T CURRENT;
30     MB_OFFSET_T VX[H_MB];
31 #pragma HLS STREAM variable=VX off
32         MB_X_T MB_X[NPIXELS_IMG/PIXELS_WORD];
33 #pragma HLS STREAM variable=MB_X depth=2 dim=1
34
35 #pragma HLS dataflow
36         //Stage 0: Xilinx HLS Video data types
37         hls::AXIvideo2Mat(IMG_REF, REFERENCE);
38         hls::AXIvideo2Mat(IMG_CURRENT, CURRENT)
              ;
39         //Stage 1
40         extendBorders(CURRENT,CURRENT_EB);
41         orderMB(REFERENCE,MB_ORDERED);
42         //Stage 2
43         SAD(CURRENT_EB,MB_ORDERED,COSTS_SAD,
              MB_X,MB_Y,VX_OFF,VY_OFF);
44         //Stage 3
45         minCost(COSTS_SAD,VX_OFF,VY_OFF,MB_X,
              MB_Y,VX,VY);
46         //Stage 4
47         copy2Mem(VX,VY,MOTION_X,MOTION_Y);
48     return;
49 }
```

Debido al equilibrio entre las etapas, la profundidad de los canales de las FIFO han sido configurados con el menor valor posible (2 palabras), ayudando a moderar el uso de recursos en la FPGA. El ancho y la profundidad de los canales depende de la configuración de las interfaces Axi-Stream y de la resolución de las imágenes.

Respecto al consumo interno de recursos por parte de los módulos que implementan las diferentes etapas, el esfuerzo se ha centrado en utilizar una estrategia de ventana deslizante junto al empleo de *line buffers*. Esta estrategia permite en las etapas `orderMB` y `SAD` la utilización de *kernels* con un II=1 (intervalo de inicialización) aplicando la directiva `PIPELINE`.

A continuación, se analiza el detalle de la arquitectura para ayudar al lector a comprender el funcionamiento de cada etapa, los mecanismos de sincronización utilizados y la estrategia seguida para reducir la utilización de recursos.

III. ETAPA 1: ADAPTACIÓN DEL FLUJO DE ENTRADA

Esta etapa se encarga de la preparación de los frames de vídeo antes del cálculo de la función de coste SAD. Antes del comienzo de la etapa 1, se realiza una conversión de formato de los streams de entrada a una estructura de tipo Mat (tipo de datos básico de la librería para procesamiento de vídeo *hls_video* en Vivado HLS) utilizando las funciones *hls::AXIVideo2Mat* de Xilinx (lineas 37 y 38) del LISTADO 1.

El propósito general del algoritmo FSBM consiste en encontrar o estimar para cada macrobloque de referencia su relativa posición dentro de la zona de búsqueda establecida para cada macrobloque en el frame *CURRENT*. Los macrobloques situados en los bordes de los frames son un caso especial debido a que ciertas posiciones dentro de la zona de búsqueda no pueden ser obtenidas ya que se exceden los límites del frame. Esta problemática se puede resolver detectando cuando se está trabajando con información situada en el borde de los frames. Sin embargo, este enfoque genera un conjunto de sentencias condicionales *if-then-else* que añaden una lógica que rompe el estilo recomendado para HLS de bucle perfecto, provocando peores intervalos de iniciación al aplicar la primitiva `PIPELINE` y aumentando los periodos de ciclo de reloj.

En consecuencia, para lograr una implementación los más eficiente posible, en vez de emplear sentencias condicionales se ha optado por realizar una extensión de bordes que permita obtener los valores que exceden los límites de las dimensiones de los frames, completando la zona de búsqueda. Esta tarea es realizada por la función `extendBorders`. La figura Figura 3 representa gráficamente el resultado de este proceso. La técnica llevada a cabo consiste en replicar la información de los bordes, sin realizar interpolaciones, evitando aumentar la complejidad de los cálculos. Para este fin, es necesario la utilización de dos buffer con un tamaño de una línea de la imagen para guardar la información relativa a la primera línea y la ultima línea de la imagen. Para el resto de las líneas no es necesario almacenamiento intermedio ya que el orden de secuencia de extracción de los datos se corresponde con el orden de secuencia de rellenado de los stream de datos. Como salida de la función `extendBorders`, se genera un stream (*CURRENT_BE*) de 32-bit de ancho de palabra, que

empaqueta el valor de cuatro píxeles (8-bit de luminancia) por palabra.

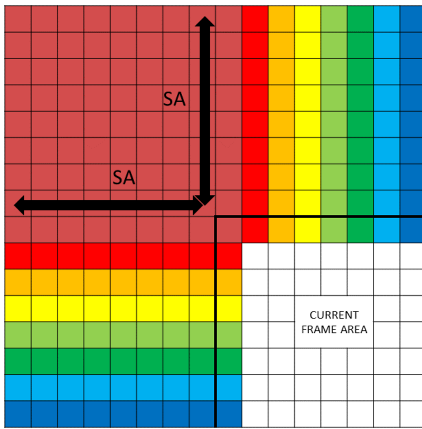

Fig. 3.Extensión del frame para completar el área de búsqueda en los bordes.

En paralelo a la extensión de bordes, la función `orderMB` realiza un reordenamiento de los píxeles del frame *REFERENCE*. La siguiente etapa espera recibir la secuencia ordenada de macrobloques de dicho frame, ya que permite un procesamiento secuencial de los píxeles del frame *CURRENT_BE*.

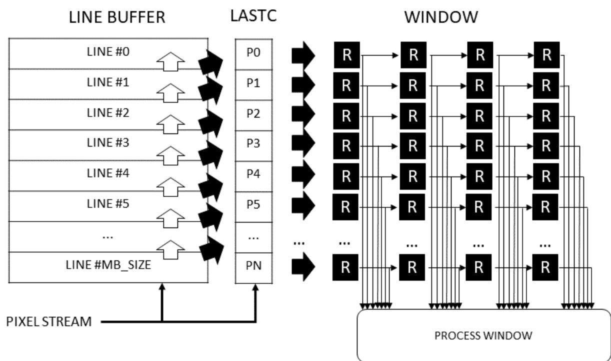

Fig. 4.Implementación de la técnica de ventana deslizante que se utiliza en las Etapas 1 y 2.

Para realizar el reordenamiento de los píxeles, se utiliza un enfoque de ventana deslizante con *line buffers*, reduciendo las necesidades de memoria BRAMs. La Figura 4 representa la arquitecura empleada en el IP FSBM.

El mecanismo modelado en HLS se muestra en LISTADO 2. La directiva `ARRAY_PARTITION` (línea 11) es utilizada para mapear las filas de la matriz que representa el line buffer en otras tanta memorias BRAM. En la línes 9, esa misma directiva divide la ventana de procesamiento a nivel de registro. Esta asignación a recursos de la arquitectura permite el acceso concurrente, lo que posibilitará la planificación de varias operaciones de lectura/escritura en un mismo ciclo, permitiendo conseguir un intervalo de iniciación de un ciclo de reloj.

En `orderMB` se empaquetan macrobloques con una dimensión de NxN píxeles con un ancho de X-bit en una palabra de $NxNxX$-bit. En el prototipo desarrollado en este trabajo es $16x16x8 = 2048$-bit. El empaquetado se realiza siempre que los índices `row` y `col` sean múltiplos del tamaño de macrobloque (esquina inferior del macrobloque), cumpliéndose la condición de guarda. Por lo tanto, la palabra empaquetada representa el contenido de un MB en el frame de referencia.

LISTADO 2
Plantilla HLS para la implementación de la técnica de *line buffer* empleada.

```
1  typedef ap_uint<32> pixel_t;
2
3  #define W_SIZE ((MB_SIZE*MB_SIZE)/4)
4  #define W_MB (MB_SIZE/4)
5
6  pixel_t lastc[MB_SIZE];
7  pixel_t window[W_SIZE];
8  #pragma HLS ARRAY_PARTITION variable=window
       complete dim=0
9  pixel_t lineb[MB_SIZE][H_Lines/PIXELS_WORD];
10 #pragma HLS ARRAY_PARTITION variable=lineb
       complete dim=1
11
12 L1: for(row = 0; row < PIXELS_V; row++) {
13  L2: for(col = 0; col < PIXELS_H/PIXELS_WORD;
        col++) {
14 #pragma HLS PIPELINE II=1
15   // Line Buffer fill
16   for(idxMBSIZE_t i = 0; i < MB_SIZE-1; i++) {
17    lastc[i] = lineb[i][col] = lineb[i+1][col];
18   }
19   //Read Pixel Stream
20   PIXEL_STREAM >> p1;
21   pixel(7,0) = p1.val[0]; //Only Luminance
22 #if defined(INPUT_BUS_WIDTH_16)
23   PIXEL_STREAM >> p1;
24   pixel(15,8) = p1.val[0];
25   PIXEL_STREAM >> p1;
26   pixel(23,16) = p1.val[0];
27   PIXEL_STREAM >> p1;
28   pixel(32,24) = p1.val[0];
29 #endif
30 #if defined(INPUT_BUS_WIDTH_32)
31   pixel(15,8) = p1.val[2];
32   PIXEL_STREAM >> p1;
33   pixel(23,16) = p1.val[0];
34   pixel(31,24) = p1.val[2];
35 #endif
36 #if defined(INPUT_BUS_WIDTH_64)
37   pixel(31,24) = p1.val[2];
38   pixel(23,16) = p1.val[4];
39   pixel(31,24) = p1.val[6];
40 #endif
41   lastc[MB_SIZE-1] = lineb[MB_SIZE-1][col] =
         pixel;
42   //Shift Window
43   L3:for(idxMBSIZE_t i = 0; i < MB_SIZE; ii++)
44    L4:for( j = 0; j < (W_MB)-1; j++){
45     window[i*(W_MB)+j] = window[i*(W_MB)+j+1];
46    }
47    L5:for(idxMBSIZE_t i = 0; i < MB_SIZE; i++)
        {
48     window[i*(W_MB)+(W_MB)-1] = lastc[ii];
49    }
50    //if(condition){
51    //Process Windows: User Logic
52    //}
53  }
54 }
```

IV. Etapa 2: Cálculo de la función de coste

En esta etapa, cada macrobloque del frame *REFERENCE* es comparado con todos los macrobloques del segundo frame (*MB_ORDERED*) situados dentro de la zona de búsqueda. Se establece, por tanto, una relación de similitud entre macrobloques, basada en el valor calculado por la función de coste SAD *Sum of Absolute Differences*: cuanto menor sea el valor calculado por esta función mayor será la similitud entre macrobloques.

El objetivo es realizar todos los cómputos SAD con una latencia equivalente al número de píxeles del frame *CURRENT* con los bordes extendidos. Para este fin, al igual que en la anterior etapa, se utiliza una arquitectura de line buffer (ver LISTADO 2). Sin embargo, se introduce lógica para sincronizar la información de los streams `CURRENT_BE` y `MB_ORDERED`.

LISTADO 3

Código HLS para la etapa de cálculo de los costes SAD.

```
1  ap_uint<1> load_MB = 0, fillMB = 0;
2  L1: for(row = 0; row < PIXELS_V; row++) {
3   L2: for(col = 0; col < PIXELS_H/PIXELS_WORD;
        col++) {
4  #pragma HLS PIPELINE II=1
5     if (load_MB) {
6      MBs[idxMBs_w++] = MB_ORDERED[idxMBs_stream
           ++];
7      load_MB = 0;
8      if (idxMBs_w == MB_H) {
9       idxMBs_w = 0;
10     }
11     //Stop consuming MBs
12     if (idxMBs_stream == MB_V*MB_H) {
13       fillMB_st = 1;
14     }
15    }
16    if ((col >= MB_SIZE) && ((col & (MB_SIZE-1))
          == MB_SIZE-1) && ((row & (MB_SIZE-1)) ==
          (MB_SIZE-1))) {
17     if (fillMB == 0) {
18       load_MB = 1;
19      }
20    }
21    //Line buffer architecture template
22    //...
23    //User logic
24    if(row >= MB_SIZE && col >= MB_SIZE){
25     idxMB_x = ((col-MB_SIZE)) >> 4;
26     idxMB_y = ((row-MB_SIZE)) >> 4;
27
28     MB_ref = MBs[idxMB_x];
29     sad_off = costSAD(window, MB_ref, &sadCost,
          (col & (MB_SIZE-1));
30     COSTS_SAD[idxCostsSAD] = sadCost;
31     MB_X[idxCostsSAD] = idxMB_x;
32     MB_Y[idxCostsSAD] = idxMB_y;
33     VX_OFF[idxCostsSAD] = ( (col + sad_off) & (
           MB_SIZE-1);
34     VY_OFF[idxCostsSAD_out++] = row & 0xF;
35    }
36   }
37 }
```

La Figura 5 muestra el mecanismo de sincronización y el patrón de consumo de los macrobloques. La imagen representa una versión simplificada de cómo solapa la información del frame *REFERENCE* (zona sombreada con gris claro) con la información del frame *CURRENT*. Cada celda representa a un subbloque de 8x8 píxeles y se asume en esta representación un tamaño de macrobloque de 16x16 píxeles, con un área de búsqueda de 8 píxeles. El momento en el que los macrobloques del frame *REFERENCE* son consumidos se representan con un diamante negro, a medida que se rellena el buffer `MBs` de un tamaño igual a una fila de macrobloques (pasos de a a d) en Figura 5). Las líneas 5-20 del LISTADO 3 implementan este comportamiento, mediante la activación de la bandera `load_MB` que será tenida en cuenta en la siguiente iteración.

Los valores de coste SAD se calculan en dos fases para cada macrobloque de referencia; la mitad

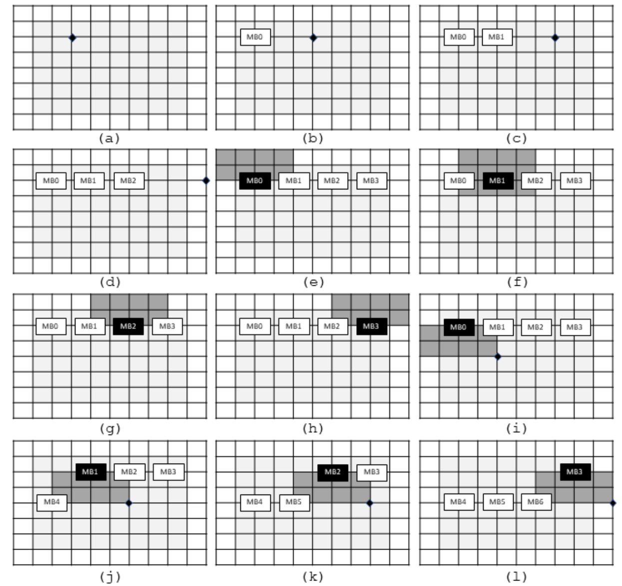

Fig. 5.Patrón de consumo de los macrobloques del frame de referencia y áreas de búsqueda.

superior (pasos e hasta h) y la parte inferior (pasos i hasta l). Después de la finalización de la segunda fase, los macrobloques en el buffer son remplazados por nuevos macrobloques, ya que no van a ser necesarios para los siguientes cálculos. Las celdas en color gris oscuro representan el área de búsqueda para un macrobloque de referencia (resaltado en color negro). Las líneas 25-28 en LISTADO 3) implementan la lógica de selección para el macrobloque de referencia.

A. Función de coste

La función `costSAD` es responsable de calcular la similitud entre macrobloques. Esta función recibe una nueva ventana de procesamiento cada ciclo de reloj, para poder mantener el rendimiento objetivo (un ciclo por píxel) de la ruta de datos. El stream `CURRENT_BE` tiene un ancho de palabra de 32 bits y en cada iteración se reciben cuatro componentes de luminancia (información del blanco y negro de los píxeles). Esto significa que, en cada iteración, cuatro valores tienen que ser calculados por la función `costSAD`. De otra forma, se perdería la información de tres columnas de píxeles motivado por el desplazamiento de la ventana deslizante.

La Figura 6 representa esta funcionalidad. La ventana se extiende una palabra (las últimas cuatro columnas resaltadas) para evitar la ya mencionada pérdida de píxeles. En HLS, la función `costSAD` (línea 30, LISTADO 3) implementa esta tarea. La lectura tanto de la ventana como del macrobloque de referencia (ambos mapeados en registros) tarda un ciclo de reloj en contemplarse. Después, por cada coste SAD calculado, se realiza la diferencia en valor absoluto de cada grupo de píxeles (4 en total) y se suman (línea 17, LISTADO 4). El balanceo automático de expresiones realizado por Vivado HLS produce la latencia mínima posible para el árbol de sumas.

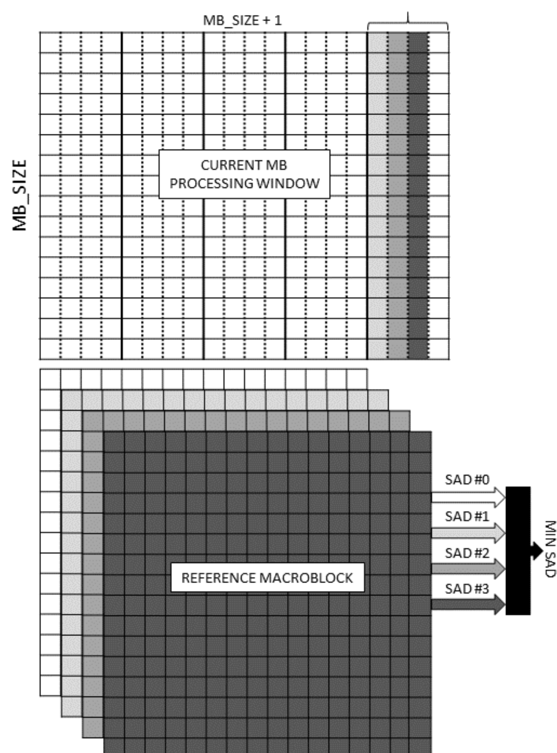

Fig. 6.Cálculo en paralelo de los valores de la función de coste SAD para la ventana de procesamiento extendida.

LISTADO 4
CÁLCULO DE COSTES SAD.

```
1 typedef ap_uint<2048> MBreg_t; //For 16x16 MB
      size
2
3 ap_uint<2> costSAD(pixel_t MB_curr[W_SIZE],
      MBreg_t MB_ref, COST_SAD_T *cost,
      MB_OFFSET_T offset){
4 #pragma HLS INLINE off
5  ap_uint<2> sad_select0, sad_select1,
      sad_select;
6
7  coste_MAD_i:for(i=0; i<MB_SIZE; i++){
8   coste_MAD_j: for(j=0; j<MB_SIZE/4; j++) {
9    p1R = MB_ref((i*MB_SIZE+j*4)*8+7,(i*MB_SIZE+
         j*4)*8);
10   p2R = MB_ref((i*MB_SIZE+j*4)*8+15,(i*MB_SIZE
         +j*4)*8+8);
11   //p3R, p4R
12
13   p1C = MB_curr[i*((MB_SIZE/4)+1)+j](7,0);
14   p2C = MB_curr[i*((MB_SIZE/4)+1)+j](15,8);
15   // p3C, p4C, p5C, p6C, p7C
16
17   SAD_0 += abs(p1R-p1C)+abs(p2-p2C)+abs(p3R-
         p3C)+abs(p4R-p4C);
18   //SAD_1, SAD_2, SAD_3
19  }
20 }
21 //1st Cycle
22 minSAD0 = SAD_0;
23 sad_select0 = 0;
24
25 if (SAD_1 < SAD_0)
26  minSAD0 = SAD_1;
27  sad_select0 = 1;
28 } else if (SAD_0 == SAD_1 {
29    if (offset < MB_SIZE/2) {
30     minSAD0 = SAD_1;
31     sad_select0 = 1;
32   }
33 }
34 //Idem for SAD_2 and SAD_3
35 //2nd Cycle
36 *costSAD = minSAD0;
37 sad_select = sad_select0;
38
39 if (minSAD1 < minSAD0) {
40  *costSAD = minSAD1;
41   sad_select = sad_select1;
42 } else if (minSAD0 == minSAD1){
43  if (offset < MB_SIZE/2) {
44    *costSAD = minSAD1;
45    sad_select = sad_select1;
46  }
47 }
48 return mb_select;
49}
```

El parámetro `offset` proporcionado a `costSAD` representa la posición relativa del macrobloque de referencia respecto al área de búsqueda. Este parámetro se usa, en caso de igualdad entre valores SAD obtenidos, para seleccionar el macrobloque más próximo a la referencia (líneas 25-47, LISTADO 4). Esta decisión de diseño ha sido tomada debido a la alta tasa de frames que procesa el IP FSBM y a la naturaleza de la secuencia de vídeo (movimiento global), para poder detectar mejor pequeños desplazamientos entre frames. Como resultado, se selecciona el valor mínimo de SAD y el índice (`sad_select`) es retornado. De vuelta al módulo padre, la variable de selección de coste SAD sirve para ajustar la componente X del vector de movimiento (líneas 29 y 33, LISTADO 4).

V. ETAPA 3: SELECCIÓN DEL COSTE MÍNIMO

Esta etapa recibe cinco flujos de datos de entrada, salidas de la anterior etapa SAD. El significado de estos flujos es el siguiente. Para un macrobloque de referencia (`MB_X, MB_Y`), un nuevo coste intermedio (`COST_SAD`) necesita ser procesado y comprobar si es el valor mínimo para ese macrobloque. En caso afirmativo, ese valor del vector de movimiento es actualizado guardando las componentes `VX_OFFSET` y `VY_OFFSET`.

LISTADO 5
CÓDIGO HLS PARA LA ETAPA MINCOST.

```
1 void minCost(COST_SAD_T COSTS_SAD[NPIXELS_IMG],
      MB_OFFSET_T VX_OFF[NPIXELS_IMG],
      MB_OFFSET_T VY_OFF[NPIXELS_IMG], idxMB_t
      MB_X[NPIXELS_IMG],idyMB_t MB_Y[NPIXELS_IMG
      ], MB_OFFSET_T VX[V_MB][H_MB], MB_OFFSET_T
      VY[V_MB][H_MB]) {
2 ...
3 ap_uint<4> distMX[MB_SIZE][MB_SIZE] = {...};
4
5 #pragma HLS ARRAY_PARTITION variable=distMX
      complete dim=0
6 COST_SAD_T COSTS_MIN[V_MB][H_MB];
7
8 //Init MIN_COSTS matric (MAX_COST_SAD)
9 ...
10 L1: for(idx = 0; idx < NPIXELS_IMG; idx++) {
11 #pragma HLS PIPELINE II=1
12  sad = COSTS_SAD[idx];
13  vx_offset = VX_OFF[idx];
14  vy_offset = VY_OFF[idx];
15  mbx = MB_X[idx];
16  mby = MB_Y[idx];
17  minSAD = MIN_COSTS[mby][mbx];
18
19  if (sad < minSAD) {
20   minSAD = sad;
21   MIN_COSTS[mby][mbx] = sad;
22   VX[mby][mbx] = vx_offset;
23   VY[mby][mbx] = vy_offset;
24  } else if(sad == minSAD){
25   vx_min = VX[mby][mbx];
26   vy_min = VY[mby][mbx];
27   dist_sad = distMX[vy_offset][vx_offset];
```

```
28      dist_min = distMX[vy_min][vx_min];
29
30      if (dist_sad < dist_min){
31       VX[mby][mbx] = vx_offset;
32       VY[mby][mbx] = vy_offset;
33      }
34    }
35  } // end of L1
36 }
```

La función `minCost` (ver LISTADO 5) inicializa la matriz `MIN_COSTS` con el máximo valor posible para coste SAD. Después, comienza el proceso de comparación entre el valor mínimo actual, guardado en `MIN_COSTS`, con al nuevo valor obtenido del flujo `COSTS_SAD`. En el caso de que estos valores sean iguales, se selecciona aquel macrobloque cuya distancia respecto al centro de la zona de búsqueda sea menor (lineas 23-31). Una ROM (variable `distMX`) es utilizada para almacenar los valores de las distancias precalculadas para reducir la latencia, sustituyendo una operación aritmética por un acceso a memoria.

En el final de esta etapa, los canales ping-pong buffer de `VX` y `VY` contendrán los valores calculados del vector de movimiento para el par de frames procesados. La función `copy2Mem` (Etapa 4) únicamente lee del *ping-pong buffer* y empaqueta los resultados del vector de movimiento en palabras con hasta 8 componentes (dependiendo del ancho de palabra seleccionado para la interfaz AXI-Stream).

VI. Resultados experimentales

El IP FSBM ha sido diseñado e implementado utilizando las herramientas de Xilinx Vivado HLS y Vivado en su versión 2016.4. El modelo y plataforma comentados en este trabajo fue migrado a la última versión disponible de las herramientas pero, sorprendentemente, los resultados de la síntesis HLS obtenidos por la versión 2018.3 fueron peores si los comparamos con los resultados de la versión 2016.4, mostrando un aumento considerable de los recursos del IP (especialmente el número de BRAMs).

El código C++ ha sido parametrizado completamente, permitiendo la posibilidad de variar la resolución de vídeo (VGA, High Definition, y Full HD), el ancho de palabra de la interfaz AXI-Stream (16, 32 y 64 bits) para acceder a la memoria donde se encuentran almacenados los frames y modificar la función `costSAD`, permitiendo la posibilidad de realizar 1, 2 o 4 (ver Figura 6 en la Sección IV) comparaciones a la vez con los valores de un macrobloque. La versión del componente utilizada para realizar las medidas de rendimiento tiene unos valores fijos para el tamaño de macrobloque ($N = 16$) y un valor del área de búsqueda ($p = 8$).

La Tabla I muestra las latencias (expresadas en ciclos de reloj) que resultan de procesar dos frames que entran en el cauce del IP FSBM. Dependiendo de la configuración de los parámetros, el rendimiento del IP se ve alterado por la modificación del ancho de palabra de la interfaz de memoria (e.j. la segunda fila en la Tabla I), y el intervalo de iniciación de la función `costSAD` (e.j. primera y tercera filas en la Tabla I). El intervalo de iniciación del cauce generado para la función `costSAD` es dos (II=2), debido a las dependencias de datos, sin importar su versión (1, 2 o 4), lo que retrasa el flujo de datos ya que el resto de los módulos cumplen con el objetivo (II = 1).

TABLA I

Valores de iniciación del *pipeline obtenido* (II) para diferentes configuraciones del IP FSBM (número de ciclos en cosimulación)

Conf.		Resolución de vídeo		
AXIS W	#SAD	VGA	HD	FHD
16,32,64	1	615765	1847206	4155822
16	2,4	325399	981699	2163569
32,64	2	308563	925210	2081650
32	4	170601	491961	1083441
64	4	154810	464802	1044850

El impacto en la utilización de los recursos para las diferentes síntesis realizadas (después del Place & Route) para la FPGA Xilinx ZC702 (ZedBoard), se recogen en la Tabla II. Debido a las restricciones de espacio en el documento, sólo se recoge un conjunto de configuraciones significativo para mostrar una serie de conclusiones. La primera es que cuanto mayor es la resolución mayor es la demanda de recursos; siendo esto un resultado esperado. Sin embargo, este incremento no sigue una relación lineal, sino que es un aumento moderado. La segunda, tiene que ver con la versión de la función `costSAD`, en este caso se produce un cambio significativo en los recursos utilizados. Esto es motivado por la cantidad de memoria extra que se necesita (mayor número de columnas en las ventanas de procesamiento) y por la lógica de adaptación de la ruta de datos, consumiendo un mayor un número de LUTs y FFs.

Respecto al rendimiento del diseño, en la Tabla III se muestra la tasa de frames por segundo (fps) procesados para cada configuración. Los resultados post-síntesis muestran una ligera variación en el período de reloj final para el IP FSBM según se modifican los parámetros del modelo. No obstante, se puede decir que el período promedio (8.69ns $\pm 0{,}19ns$) es constante, independientemente de la resolución de vídeo, la interfaz AXI-Stream y la versión de la función `costSAD` utilizada. La frecuencia de trabajo nominal ($\simeq$115 Mhz) es suficiente para realizar el procesamiento en tiempo real de un stream de vídeo, para las resoluciones VGA y HD con una tasa de refresco de 60Hz, gracias a la combinación de ciertos parámetros del IP que aseguran una óptima utilización de recursos en la FPGA. En cambio, para resoluciones Full HD, el componente no puede alcanzar la restricción del tiempo de píxel (esto es, 148.5Mhz/6.7ns para Full HD a 60 Hz). Por lo tanto, el diseñador debe encontrar una solución a esta restricción mediante la selección de una combinación de parámetros más agresiva, que permita cumplir el objetivo de realizar un procesamiento en tiempo real. Esta estrategia también se aplica en contextos donde el alto rendimiento es el principal objetivo de la aplicación.

TABLA II
UTILIZACIÓN DE RECURSOS (PLACA ZC702)

Conf.	LUT	FF	BRAM
	VGA		
16_SAD1	11647(21,89 %)	16780(15,77 %)	48,5(34,64 %)
32_SAD2	18186(34,18 %)	16824(15,81 %)	49,5(35,36 %)
64_SAD4	31424(59,07 %)	30820(28,97 %)	51,5(36,79 %)
	HD		
32_SAD2	18211(34,23 %)	17845(16,7 %)	50,5(36,07 %)
32_SAD4	30720(57,74 %)	24176(22,72 %)	52,5(37,50 %)
64_SAD4	31522(59,25 %)	31045(29,18 %)	52,5(37,50 %)
	FHD		
32_SAD2	18207(34,22 %)	19884(18,69 %)	53,5(38,21 %)
32_SAD4	30680(57,67 %)	26880(25,26 %)	55(39,29 %)
64_SAD4	31548(59,30 %)	31450(29,56 %)	55(39,29 %)

TABLA III
TASA NOMINAL DE FPS OBTENIDA (DISPOSITIVO ZC702, $\overline{T}$=8.69NS±2,23 %)

	Ancho palabra AXI-STREAM								
	16			32			64		
	Versión SAD			Versión SAD			Versión SAD		
	1	2	4	1	2	4	1	2	4
VGA	186	353	353	186	373	674	186	373	743
HD	62	117	117	74	124	234	74	124	247
FHD	27	53	53	41	55	106	41	55	110

VII. CONCLUSIONES

En este trabajo se ha presentado un IP que realiza una implementación de alto rendimiento del algoritmo de estimación de movimiento FSBM (*Full Search Block Matching*. La arquitectura propuesta se basa en una ruta de datos modelada con la herramienta Vivado HLS 2016.4. El modelo está parametrizado (resolución de vídeo, ancho de la interfaz de memoria, cálculos SAD en paralelo), permitiendo explorar fácilmente el espacio de soluciones.

El prototipo desarrollado sobre una placa ZC702 de Xilinx funciona a 115 Mhz, frecuencia suficiente para cumplir los requisitos temporales para VGA y HD 60Hz, con un consumo de recursos moderado. La implementación en tiempo real para Full HD necesita de un mayor ancho de banda a memoria y paralelizar los cálculos de coste SAD, alcanzando un máximo de 110 fps.

AGRADECIMIENTOS

Este trabajo ha sido financiado por el Ministerio de Economía y Competitividad de España bajo el proyecto `PLATINO (TEC2017-86722-C4-4-R)` y por la Junta de Comunidades de Castilla-La Mancha bajo el proyecto `SymbIoT (SBPLY/17/180501/000334)`.

REFERENCIAS

[1] S. Ghosh and A. Saha, "Speed-area optimized fpga implementation for full search block matching," in *2007 25th International Conference on Computer Design*, Oct 2007, pp. 13–18.

[2] H. Loukil, F. Ghozzi, A. Samet, M. A. Ben Ayed, and N. Masmoudi, "Hardware implementation of block matching algorithm with fpga technology," in *Proceedings. The 16th International Conference on Microelectronics, 2004. ICM 2004.*, Dec 2004, pp. 542–546.

[3] "Sad computation based on online arithmetic for motion estimation," *Microprocessors and Microsystems*, vol. 30, no. 5, pp. 250 – 258, 2006.

[4] A. Ryszko and K. Wiatr, "An assessment of fpga suitability for implementation of real-time motion estimation," in *Proceedings Euromicro Symposium on Digital Systems Design*, Sep. 2001, pp. 364–367.

[5] N. Roma and L. Sousa, "Efficient and configurable full-search block-matching processors," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 12, no. 12, pp. 1160–1167, Dec 2002.

[6] M. Mohammadzadeh, M. Eshghi, and M. M. Azadfar, "Parameterizable implementation of full search block matching algorithm using fpga for real-time applications," in *Proceedings of the Fifth IEEE International Caracas Conference on Devices, Circuits and Systems, 2004.*, vol. 1, Nov 2004, pp. 200–203.

[7] N. Roma, T. Dias, and L. Sousa, "Customisable core-based architectures for real-time motion estimation on fpgas," in *Field Programmable Logic and Application*, P. Y. K. Cheung and G. A. Constantinides, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 745–754.