A Modified Sliding Window Architecture for Efficient BRAM Resource Utilization

Murad Qasaimeh, Joseph Zambreno and Phillip H. Jones
Department of Electrical and Computer Engineering, Iowa State University, Ames, Iowa, USA 50010
Email: {qasaimeh, zambreno, phjones} @iastate.edu

***Abstract*—Sliding window is one of the most commonly used techniques in image processing algorithms. Implementing it in hardware requires buffering image rows on-chip to exploit data locality and avoid redundant off-chip pixel transfers. However, scaling this architecture to large window sizes or high resolutions linearly increases on-chip memory utilization. This imposes limitations on porting many image processing algorithms into hardware efficiently. In this paper, we propose a new sliding window architecture that utilizes less on-chip memory resources while maintaining performance as compared to the traditional method. The proposed architecture exploits that most natural images have smooth color variations with fine details in between these variations to compress images. It decomposes non-zero image pixels into their wavelet components and represents each wavelet coefficient with a minimum number of bits. The architecture is also flexible to use lossless or lossy compression based on a configurable threshold value. The FPGA implementation of our proposed architecture shows memory saving of 25-70% compared to the traditional architecture using lossless compression, and for lossy compression with up to a mean square error of 5 achieves up to 84% in memory savings.**

***Keywords*—*Image processing, Sliding Window, Compression, BRAMs, IWT, FPGA*.**

I. Introduction

Many image processing algorithms use a sliding window technique as part of their algorithm. The sliding window operation repeatedly gathers a rectangular region of pixels, calculates an output for that window, and then slides across the input image. This operation is computationally and data intensive and benefits from hardware acceleration on FPGAs, especially for real-time applications [1]. Hardware implementations typically buffer image rows on-chip to exploit data locality and avoid redundant pixel transfers. Although these implementations provide significant performance improvement, scaling them to large window sizes or high resolutions linearly increases on-chip memory utilization. This imposes limitations on porting many image processing algorithms into hardware efficiently.

There are many image processing algorithms currently limited by the number of Block RAMs (BRAMs) available on FPGAs [2][3][4]. For example, in object detection algorithms, the maximum detectable size is limited by the window size supported in hardware. Increasing the window size will increase the chances of detecting more objects, but will also require more BRAMs to store additional image rows on-chip. A common solution is to scale down and re-scan the whole image [2]. Implementing lens distortion correction on FPGAs is another example. The maximum distortion coefficients supported by a hardware implementation is also limited by the window size supported. Increasing the window size will increase the mapping range supported by the distortion correction core, but increases on-chip memory requirements [3]. Supporting larger window sizes for image filters often increases their accuracy. For example, for a Gaussian smoothing filter, the size of the window should be at least 5 times its standard deviation to not loose precision by trimming the kernel's small values. Moreover, most image processing algorithms consists of 2-5 sequential sliding window operations, where the output of one operation is fed via line buffers to the following operation. These implementations require a high number of BRAMs for implementing multiple sets of buffer lines [4].

In this paper, we propose a modified sliding window architecture that utilizes on-chip memory resources more efficiently than the traditional architecture (introduced in Section III) by storing compressed instead of raw pixels values. It uses the 2D Haar wavelet transform to decompose the active window's pixels into its wavelet coefficients: approximation, horizontal details, vertical details, and diagonal details sub-bands. Then, it finds the minimum number of bits required to represents these coefficients. Because natural non-random images have most of their information in the approximation sub band and small details in the other sub-bands, the compression algorithm is able to represent the coefficients in the details sub-bands using less bits. The proposed sliding window architecture uses this simple compressing method to pack the compressed bits, and store them into the buffering system without any degradation in computing throughput performance. The architecture is also parametrizable to have the flexibility to change the compression ratio based on available on-chip memory and a threshold parameter that allows lossless or lossy compression.

Contributions. In this work, we propose a new sliding window architecture that takes advantage of most of an images' information residing in low frequencies to reduce the required on-chip memory. The main contributions of this paper are: (1) A novel sliding window compression algorithm that can be efficiently implemented in hardware and gives comparable compression ratios to the state of the art compression algorithms, (2) A parametrizable sliding window architecture that has the flexibility of changing its compression ratio based on a threshold parameter, (3) A fully pipelined architecture similar to the traditional sliding window architecture with up to 84% in BRAM resource saving.

Organization. The rest of the paper is organized as follows. Section II discusses related work and compares it to our approach. In Section III, we present the traditional line-buffering sliding window architecture. Section IV explains the proposed algorithm we use for compression. In section V, we provide a detailed description of our hardware architecture and its main building blocks. Section VI discusses the experimental results and architecture performance. Finally, Section VII concludes the paper with outlooks for future work.

II. Related Work

A number of works can be found in the literature that aim to reduce on-chip memory requirements of the traditional sliding window architecture. Some works try to solve the problem by proposing new buffering methods. For example, in [5] and [6], instead of using the traditional line-buffering method, they use a block buffering technique. This method starts by reading a block of pixels with size greater than the size of the operation window. This allows for processing multiple windows without the need to load new data. While processing the current block, data for the next block is buffered. This method reduces the required on-chip memory but it is not as efficient as the traditional architecture, as its average number of off-chip accesses is greater than 1 pixel per window operation.

Others attempt to reduce the required memory by dividing the input image into segments and process each individually [7]. They partition the data array into segments along a row. Once the current segment has completed processing, the next segment of data is processed until the current row is finished. This approach can save some BRAMs, but is not efficient for streaming applications when pixels come directly from a camera sensor, as it requires pixels to be in off-chip memory.

The authors in [4] try to avoid utilizing sliding window buffering between an algorithm's operations by using a larger window size for the first operation and uses several parallel processing units to compute the next operations. This method reduces the BRAMs usage but it consumes high amounts of combinational logic, and is only applicable when sequential operations can be composed into one composite operation.

In this work, we investigate reducing memory requirements for sliding window architectures by storing compressed instead of raw pixels values. The compression algorithm, to be suitable for our purpose, must compress an entire column of pixels in the current window every clock cycle. It should have a good compression ratio and can be implemented in hardware with low resource overhead. It should be lossless to recover the original image and have the flexibility to be lossy to recover an approximation of the original image with different mean square errors when additional compression is required.

Existing lossless and lossy compression algorithms have good compression and signal-to-noise ratios, but are not suitable for our purposes. For example, FPGA implementations of the standard lossless compression algorithm, JPEG-LS [8], consume too many resources and reduces system performance. It has a 6-stage pipeline and its maximum operational frequency is around 27MHz. The JPEG compression algorithm [9] is also not a good choice as it uses a fixed size window of 8×8, while in our case we want to compress the pixels in the left-most column of the window. The number of pixels in these columns depends on the window size.

Several other wavelet transform based compression algorithms exist in the literature. The most popular algorithms are: Embedded Zerotree Wavelet, Set Partitioning In Hierarchical Trees, and Embedded Block Coding with Optimized Truncation [10]. These algorithms are designed for variable bit rate image transmission and require three dynamically updated lists that make them unsuitable for high speed applications. This lead us to propose a new simpler image compression algorithm based on 2D Haar transform that satisfy our needs and can be implemented in hardware with relatively low hardware resources compared to the other compression algorithms.

III. Traditional Sliding Window Archtiecture

Line-buffering is the most popular approach for implementing sliding window architectures [1]. It consists of a set of FIFOs connected to shift registers, as shown in Figure 1. The number of FIFOs and their size depends on the input image resolution and the window size. For an image of resolution (W×H) pixels and a window of size (N×N), the number of FIFOs is (N-1) with depth (W-N) pixels, and the size of the window should be N×N shift registers. The input pixels are pushed into the first line of the active window and the processed pixels are pushed out from the last line. The shift registers in the right-most column are connected to the input of the next FIFO lines. The outputs of the FIFOs are connected as inputs to the shift registers in the left-most column, as shown in Figure 1.

The architecture has three main states: (1) Fill the FIFOs: in which we need to wait until the FIFOs are full with valid pixels. We only receive input pixels and push them into the buffer lines, and no output or operations are done in this stage; (2) Processing stage: we read one input pixel and process the active window to generate an output. This processing is done in one clock cycle; (3) Empty the FIFOs: in this stage, there are no more input pixels to read. But there are still valid pixels inside the FIFOs that need to be processed until no pixels are left inside the FIFOs.

The sliding window architecture produces an output of size (N×N), in other words, one value for each pixel in the input image [11]. For example, for an image of size 512×512 and a window of size 3×3, the first window to process is a square between (0,0) and (2,2). The required on-chip memory for this example, assuming 8-bit pixels, equals (512-3)×2×8 bits.

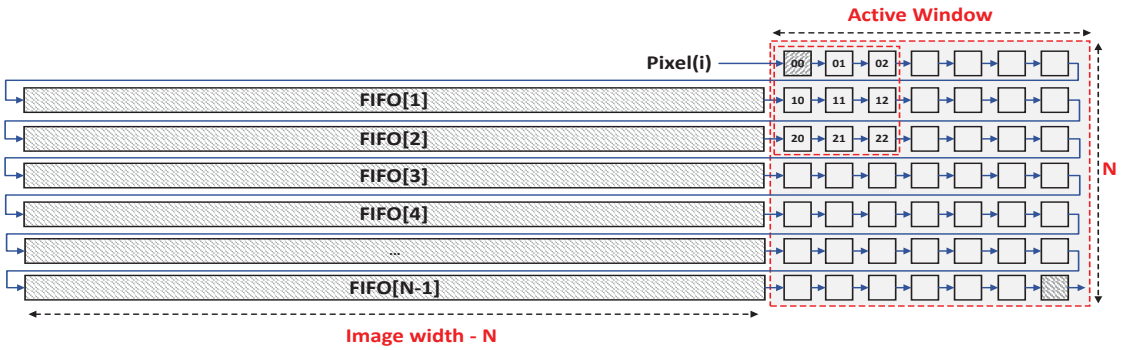

Fig. 1: Line-buffering Sliding Window Architecture

This relatively small example can be implemented in hardware. However, for high resolutions and large window sizes, such as a window of size 120×120, an image of HD resolution (2048×2048), and 24-bit colored pixels, the required on-chip memory is at least (2048 - 120)×120×24 bits = 5,422Kb. While FPGAs like the XC7Z020 has a total on-chip memory of 5,018Kb.

IV. Proposed Compression Algorithm

This section describes the proposed compression algorithm used in our modified sliding window architecture. The algorithm is composed of three main steps. First, the integer wavelet transform *(IWT)* decomposes the pixels in the active window into its wavelet components. Second, the *Bitpacking* step finds the minimum number bits required to represent each wavelet coefficient in each of the four sub-bands. The bits of the non-zero coefficients, only, are packed into chunks and stored in the buffering system. Third, *BitUnpacking* reconstructs the original pixels from the compressed bits to be used by the next window. The following sub-sections describe these three steps in further detail.

A. Integer Wavelet Transform (IWT)

Forward IWT transforms image pixels into a set of integer coefficients. The transformation is reversible, meaning the pixels can be recovered without any loss using the inverse integer wavelet transform (IIWT). In this paper, we use a 2D single-level Haar wavelet transform [12] to generate four sub-bands: (1) Approximation (LL), (2) Horizontal details (LH), (3) Vertical details (HL), and (4) diagonal details (HH) sub-bands. Figure 2 shows an example of a window of size 8×8 after it has been decomposed into its wavelet sub-bands.

B. Bit Packing

In the Bit Packing step, wavelet coefficients are compared to a threshold value. If the absolute value of the coefficient is less than the threshold, it is considered insignificant and replaced with zero, otherwise it will not be modified. Figure 2 shows this step for a threshold value of zero (lossless). The coefficients did not change because the thrshold value is zero. In the next step, for each column of each sub-band the minimum number of bits (NBits) required to encode the largest pixel value in 2's complement is determined and stored in the compressed window. Then, the least significant NBits bits of each coefficient for each sub-band column are packed together to form the compressed version of the original coefficients.

Figure 2, for simplicity, shows only the bit packing process for the vertical (HL) and diagional (HH) sub-bands. It shows the NBits bits required to represents the first column of HL (Pixels: 13, 12, -9 and 7) is 5. So, the least significant 5 bits of the non-zero pixels (01101, 01100, 10111 and 00111) will be packed together and stored in the compressed window. BitMap is used to distinguish between zero or non-zero coefficients. The BitMap of the first column is 1111 because all the coefficients have non-zero values, while the BitMap of the last column is 0011 because the first two coefficients are zeros.

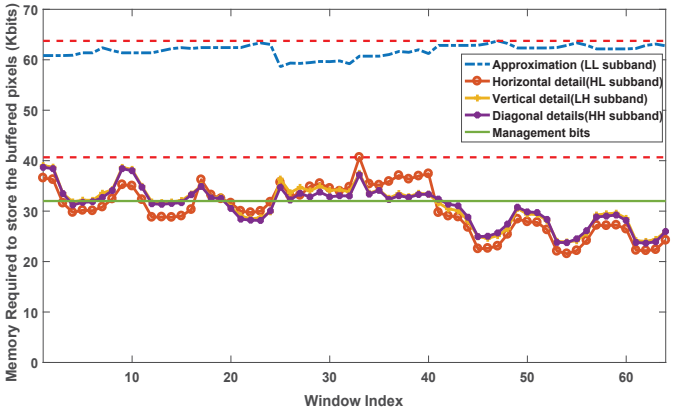

Fig. 3: Memory Requirement as the Window slides across the Image

To see the efficiency of this algorithm on real images, Figure 3 shows the amount of memory required to buffer image rows (lossless compression) as a window of size 64×64 slides across an image of size 512×512. It shows that the number of bits required for the approximation (LL) sub-band is almost two times higher than the three detail sub-bands. It also shows that in the worst case we need around 40 Kbits to store the coefficients of the LH, HL and HH sub-bands and around 65 Kbits for the LL sub-band. In total, we can store the coefficients of the 64 image rows in 185 Kbits plus 32 Kbits of management bits (total = 217 Kbits) compared to 230 Kbits using the traditional sliding window method. As image resolution increases so does the memory efficiency of this algorithm (see section VI).

In this example, the threshold value equals zero (lossless compression), increasing the threshold value (lossy compression) reduces the total number of bits because the number of zeros in the sub-bands increases and the algorithm uses only one bit for zero coefficients and compresses only non-zero coefficients. Increasing the threshold value increases the compression ratio, but decreases the quality of the reconstructed image.

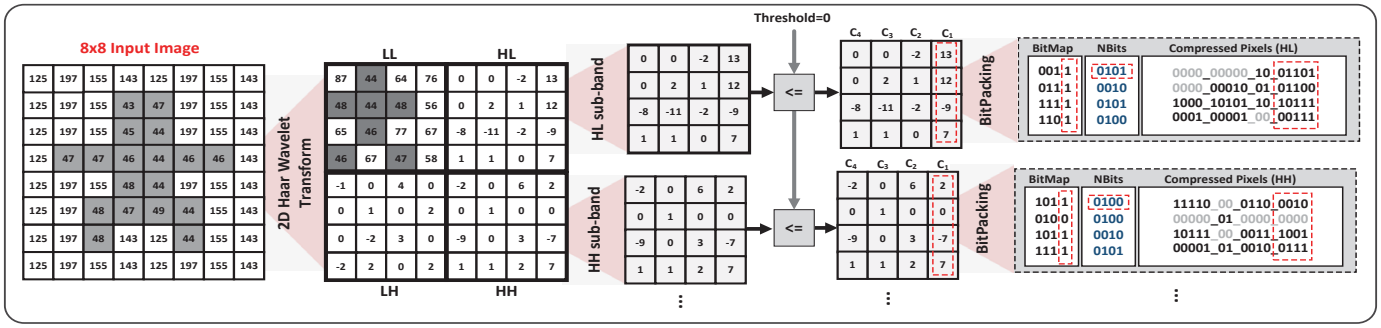

Fig. 2: Example of the Compression Algorithm for an 8×8 Window Size

C. Bit Unpacking

To reconstruct the original coefficient values, we first read a BitMap value. If it equals zero, then the output should be zero, otherwise we read the least significant NBits and sign extend to the pixel size (8 bits) and send it to the active window. If NBits is less than 8, the remaining bits are kept for the next output. Even if this compression algorithm seems simple, it shows good compression ratios. But it also introduces management bits (BitMap and NBits) that need to be taken into consideration. For an image resolution (W×H) and a window size (N×N), the total management bits equals 2×(W-N)×4 bits for NBits and (W-N)×N bits for BitMap.

In this algorithm, we used a one-level Haar wavelet transform because adding more levels complicates the architecture for both the forward and inverse wavelet transform blocks. Moreover, using 2 or 3 levels of decomposition did not increase the compression ratio significantly. We also chose the Haar wavelet transform instead of other transformations like 5/3 and 7/9 for the same reasons. In the Bit Packing step, we find the minimum number of bits for each column in each sub-band instead of other options like for each coefficient or for each sub-band because there was a tradeoff between the compression ratio and the number of management bits.

V. Proposed Hardware Architecture

This section presents the proposed hardware architecture and its building blocks. The architecture consists of five modules: (1) 2D integer wavelet transformation (IWT), (2) Bit Packing, (3) Memory Units, (4) Bit Unpacking and (5) 2D inverse integer wavelet transformation (IIWT). Figure 4 shows an abstract high-level overview of the modified sliding window architecture and its components. The active window is implemented using shift registers so that a processing kernel can directly access all pixels of the active window each clock cycle. As an example, a 2D image filter could multiply each pixel in the active window with a corresponding constant in the filter kernel, and output these results as a sum or weighted sum.

The input pixels *Pixel(i)* coming directly from a camera sensor or off-chip memory are stored in the first register of the first row in the active window and previous pixels are shifted to the right. The Integer Wavelet Transform module reads the right most column of the active window each clock cycle, and decomposes the pixels into their wavelet components. The resulting coefficients are fed to the Bit Packing module that represents these coefficients with the minimum number of bits and compresses them. When 8 bits have been accumlated in the Bit Packing unit, it writes the packed bits into the Memory Unit along with its management bits: (1) NBits: number of bits of each compressed coefficient, and (2) BitMap: one bit to distinguish whether the coefficient has a zero or non-zero value. When the number of coefficients in the memory unit equals the image width, the Bit Unpacking unit reads the compressed bits from the FIFOs and reconstructs the coefficients' values. Finally, the reconstructed coefficients are transformed back to the original pixels by the Inverse Integer Wavelet Transform module. The results from IIWT module are written into the left most column of the active window and the previous pixels are shifted to the right. This process is repeated until all pixels in the image pass through the architecture. The following subsections describe the architecture's blocks in further detail.

A. Integer Wavelet Transform Module

This module receives input from the active window right-most registers, and sends its output to the Bit Packing module. Each clock cycle, it reads N pixels and generates N/2 low frequency coefficients (LL), and N/2 Horizontal details (LH) coefficients or N/2 vertical details (HL) coefficients, and N/2 diagonal details coefficients. Where N is the window size. In this work, we used the Haar wavelet transform because it maps to a simple hardware structure. The Haar wavelet transform equations are shown in Equations (1) and (2), where i and j are pixels coordinates.

$$L(i,j) = X(i,j+1) + H(i,j)/2 \tag{1}$$

$$H(i,j) = X(i,j) - X(i,j+1) \tag{2}$$

The hardware implementation of the 2D forward Haar wavelet transformation is shown in Figure 5. Each 1D block consists of one adder, one subtractor and one division by 2 (implemented as a shift right by 1). The 2D transformation is implemented by connecting four 1D blocks together as follows: the two low-frequency outputs (L) in the first stage are connected as inputs to the top block in the next stage, and the two high-frequency outputs (H) are connected to the bottom

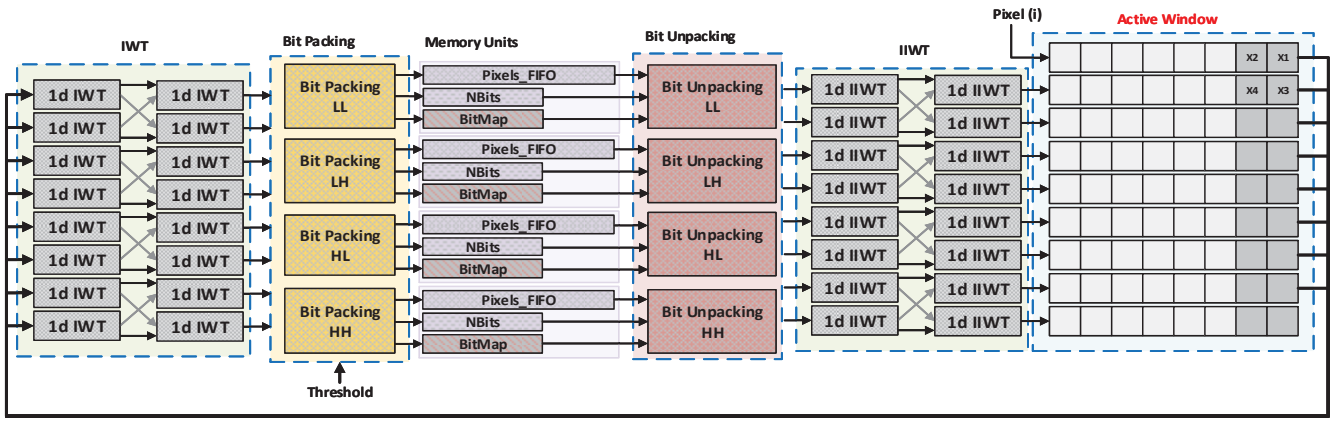

Fig. 4: The Proposed Sliding Window Architecture

block in the next stage. The inputs are four pixels (X1, X2, X3 and X4) and the results are four sub-band coefficients: LL (Low-low: Approximation sub-band), LH (Low-High: vertical details), HL (High-low: horizontal details) and HH (high-high: diagonal details).

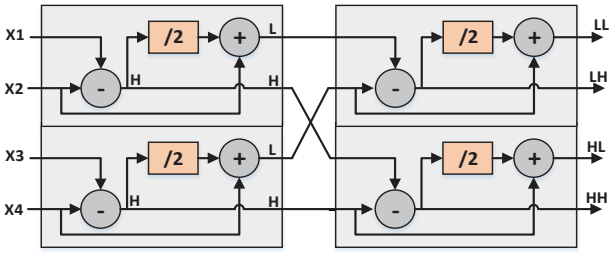

Fig. 5: 2D Haar Integer Wavelet Transform Block

B. Bit Packing Module

The Bit Packing block receives input from the IWT module, compresses each coefficient, and further packes the compressed coefficients together before storing to the Memory Unit. It has three steps: First, it finds the minimum number of bits required to represent the largest of the input coefficients (NBits). Second, it compares the coefficients values with a threshold parameter that determines whether the compression is lossless or lossy. If coefficients values are less than the threshold, they will be considered as insignificant and replaced by zero. The third step involves the actual compression. Each clock cycle, the block collects a coefficient's NBits least significant bits. Once the number of collected bits reach the maximum bit width, it writes these packed bits to the Memory Unit along with the management information needed to restore the original pixels later.

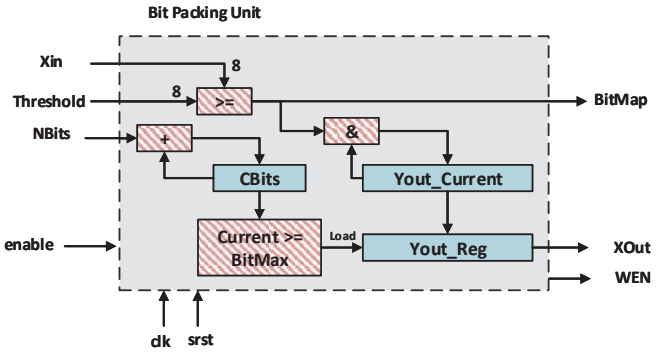

Fig. 6: Bit Packing Unit

Figure 6 shows the hardware architecture of the Bit Packing unit. It consists of three Registers: (1) Current number of bits (CBits): a 4-bit register that keeps track of the number of bits in the Yout_current register; (2) Current concatenation of bits (Yout_Current) is an 8-bit register that stores bits during the concatenation process; (3) The output register (Yout_Reg): is an 8-bit register used when the number of valid bits in Yout_Current reaches BitMax (8 bits), and the value of Yout_Current will be copied to Yout_Reg. The Bit Packing block also sets write enable WEN=1 to write valid output to the Memory Unit. The block has two comparators: one compares the input (Xin) with a threshold value. If the input coefficient value is less than threshold, the BitMap value is set to 0 otherwise to 1. The other comparator compares the current number of bits (Cbits) to BitMax. It also has one adder that is used to add Cbits to the input number of bits (NBits). The number of Bit Packing blocks in the proposed architecture is equal to the window size, i.e. if the window size is 100×100, there will be 100 Bit Packing blocks.

To find the minimum number of bits required for encoding n coefficients (X_0-X_{n-1}) of the just inputted column of a sub-band, we first compare the sign bit (bit7) with bits 0-6 to find the first location that has a different value than the sign bit for each coefficient. We used 2-input XOR gates to do the equality comparison, as shown in Figure 7. The first input to all XOR gates is the sign bit and the second input is bits 0-6. To find the minimum number of bits required to represent all the coefficients, we use n-input OR gates to find if any of the XOR gate's outputs was 1. If the OR gates output is 1, the minimum number of bits should be at least 2 bits, else if the OR gates output is less than 4, it should be 3 bits, and so on. Figure 7 shows the block's architecture when n=3. As an example, for X_1= (-6) 0b'11111010, X_2= (-2) 0b'11111110 and X_3= (6) 0b'00000110, the output of the XOR gates will be 0000101, 0000001 and 0000110 respectively. The output of the OR gates will be 0000111, which indicates for each coefficient in this sub-band column the minimum number of bits required will be 4.

C. Bit UnPacking Module

The task of the Bit Unpacking module is to reconstruct the original pixels values from the compressed coefficients stored in the Memory Unit. The module first reads one value from each of the three Memory Unit buffers: (1) number of bits (NBits), (2) Bit map, and (3) Pixel FIFO. If the Bitmap bit equals zero, that means the original coefficient was less than the threshold value or zero. Otherwise, it extracts the lower NBits of the value read from the Pixel FIFO, then sign extends and sends it to the output. If NBits is less than 8 bits, the module keeps the remaining bits to be used for the next coefficient.

Figure 8 shows the hardware architecture of Bit UnPacking module. It has three registers: (1) CBits, similar to Bit Packing: it is a 4 bit register that keeps track of the remaining bits in the Yout_current register; (2) The remaining bits register, Yout_rem, a 16 bit register that is used to store the remaining bits after each output. For example, if the block reads 8 bits and NBits is only 2 bits, it will keep the remaining 6 bits to be used in the next output; (3) The output register, Yout_Reg. The

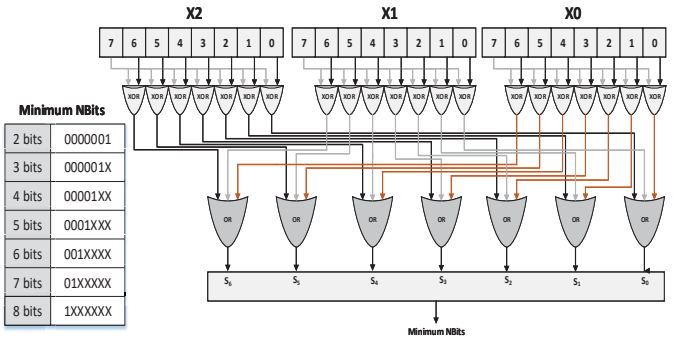

Fig. 7: Find Minimum Number of Bits Architecture

block also has two comparators: one to check if the Bitmap bit is zero or one. Another to make sure that the block always has enough bits for the next output by checking if the CBits register value is less than 8. It has a multiplexer that selects bits from Yout_rem and/or Xin to be cpoied to Yout_reg. It also has one adder connected as shown in Figure 8.

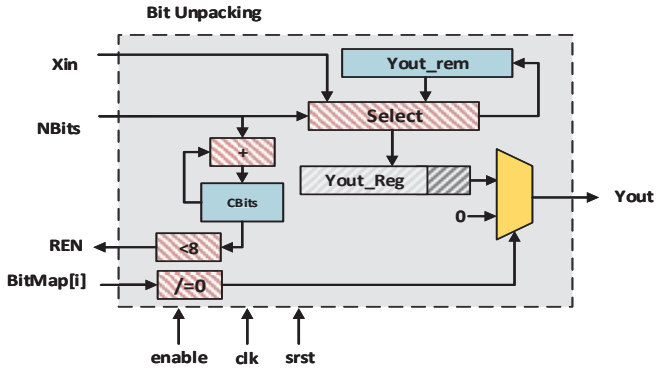

Fig. 8: Bit Unpacking Unit

Figure 9 shows an illustrative example of the bit unpacking process. It shows the compressed bits of five pixels (A, B, C, D and E), and the values of Yout_rem and Yout_reg in the first four steps. In the first step, the block will read 8 bits that contain pixel (A)'s bits and part of pixel (B)'s bits. Then it will extract the lower NBits, and sign extend and write it into Yout_reg. Because the current number of bits (CBits) is now less than 8 bits, in the next step it will read another 8 bits as shown in Figure 9. The same process is applied to extract the lower NBits and sign extend the value and write it into Yout_reg. The size of Yout_reg is 16 bits because the worst case is when the previous step has NBits equals to 1 and in the next step NBits equals the max number of bits (8). In this case, we need the size of Yout_rem to be enough to store two consecutive reads from Pixel FIFO.

D. Inverse Integer Wavelet Transform (IIWT) Module

The inverse wavelet transform module takes input from Bit Unpacking and regenerates the original pixels values from the four wavelet sub-bands. The inverse Haar wavalet transformation equations are shown in Equations (3) and (4):

$$X(i,j) = (H(i,j)/2 - L(i,j)) + H(i,j) \quad (3)$$

$$X(i,j+1) = H(i,j)/2 - L(i,j) \quad (4)$$

Figure 10 shows the architecture of the 2D inverse wavelet transformation block. It looks similar to the Forward Wavelet Transform block. Each 1D block has one addition, one subtraction and one division by 2. The two low frequency outputs (L) in the first 1D stage are connected to the input of the top block in next stage. The two high-frequency outputs (H) are connected to the input of the bottom block in the second stage. For an architecture with window size N, the number of 2D inverse wavelet transform modules equals N/2.

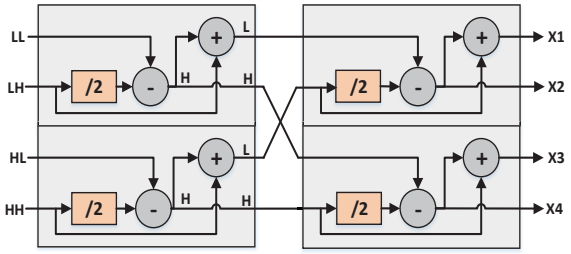

Fig. 10: 2D Haar Inverse Integer Wavelet Transform Block

E. Memory Unit

The memory unit is where all the compressed pixels and management bits are stored. It contains buffers for storing the packed bits (the output of BitPacking blocks), number of bits (NBits), and BitMap. Based on the compression ratio, the output of the Bit Packing blocks can be mapped to FIFO buffer lines by one of the following options: First, store one packed image row in one FIFO buffer line. This option has memory savings equals 0% because it is similar to the traditional sliding window architecture. The second option is to store two packed image rows in one FIFO. The memory savings will be around 50% compared to the traditional method. Third, storing four image rows in one FIFO to have an approximate saving of 75%. Fourth, to store eight image rows in one FIFO. The memory saving will be around 87.5%. Figure 11 shows the four options of mapping the packed bits to buffer lines.

Because each column in the decomposed image has two sub-bands (LL and LH or HL and HH), as shown in Figure 11, the total number of bits required for NBits equals $2\times4\times$(Image width - window_size) bits. For BitMap, we need one bit for each pixel, so the total number of bits equals (image width-window size) $\times$ window size bits. Because for each column we need 2×4 bits for NBits, mapping NBits to Block RAMs can be done by configuring 18Kb BRAMs to be $2K \times 9$ in simple

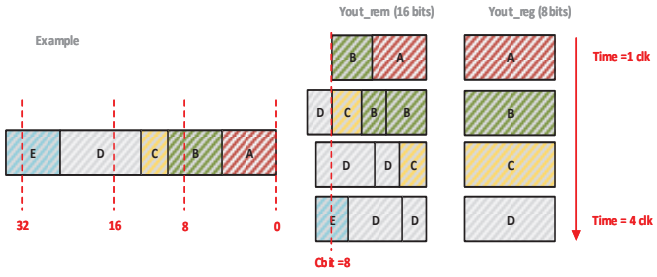

Fig. 9: Example of Unpacking Process

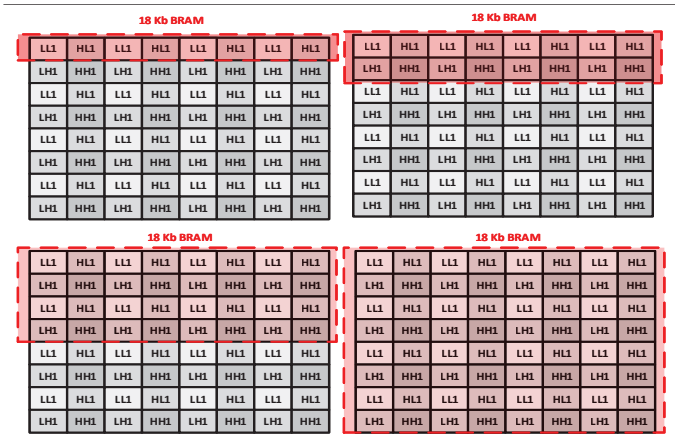

Fig. 11: Memory Mapping Options

dual port mode. Mapping BitMap bits depends on the window size. For example, if the window size is 8, 16, 32, 64, or 128, and image width 512, the 18Kb BRAMs will be configured as 2k×9, 1k×18, 512 ×36, 2×(512×36) and 4×(512×36) respectively.

Current Limitations. Our architecture currently has one limitation that the compression ratio should be known at design time. That means, the number of BRAMs and their configurations used in the memory unit should be known. This will not be an issue in cases where the image scenes do not change significantly. In this case the memory unit will be configured to the worst-case scenario. But in cases of bad frames or random images, the compression ratio will be very low and the size of the packed bits will be greater than the available BRAMs. This can be fixed in the future by making threshold values automatically adjustable based on the available memory and the current frame compression ratio. Moreover, the Bit Unpacking block in our architecture consumes a large number of LUTs resources compared to the other blocks, due to a large multiplexer. But our architecture LUTs resources is only a function of window size. While increasing image resolution does not impact the architecture's LUT resources, allowing for increased BRAM savings with higher resolutions at no additional LUT cost.

VI. Results and Analysis

This section presents the experimental results for testing the performance of the proposed compression method, and evaluates the memory savings gained by using it in our sliding window architecture compared to the traditional architecture. It presents the hardware resources used to implement each block in the architecture, and the overall system.

A. Memory resource savings

To evaluate our compression method, we used 10 randomly selcted images from the MIT Places Database for Scene Recognition [13]. Figure 12 shows examples of the images used. It includes indoor and outdoor scenes. We computed the averege memory saving gained by compressing these images for different window sizes and image resolutions. Equation 5 shows the memory saving formula.

$$MemorySaving = (1 - \frac{Compressed}{Uncompressed}) \times 100\% \qquad (5)$$

Figure 13 shows the memory savings of our compression algorithm in comparison to the traditional sliding window approach for an 2048×2048 image resolution. These numbers take into account the overhead associated with the management bits (BitMap and NBits). For lossless compression, the saving is around 26-34% for different window sizes. The compression ratio increases as we increase threshold value from 0 to 2, 4 and 6 (i.e. as the algorithm becomes more lossy). The saving is around 41-54% when threshold value is set to 6.

Fig. 12: Example Images

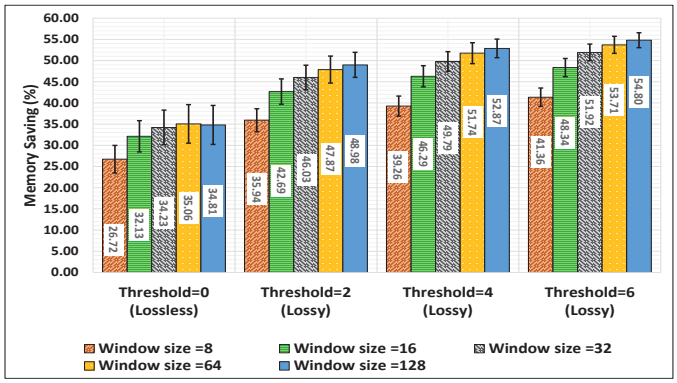

Fig. 13: Percentage of memory savings with 90% confidence intervals

Table 1 shows the number of BRAMs used in the traditional sliding window architecture for different image widths (512, 1024, 2048 and 3840) and window sizes (8, 16, 32, 64 and 128). It shows that each FIFO line is realized in hardware by one 18Kb BRAM except for image width 3840. This is because each pixel is 8 bits and an 18Kb BRAM configured as 2k×9 can fit up to 2048 pixels. Thus, image rows of width 512, 1024 and 2048 can fit in one BRAM, while image widths greater than 2048 require cascading multiple BRAMs to store one image row.

TABLE I: Number of BRAMs (18Kb)

Window size	512	1024	2048	3840
8	8	8	8	16
16	16	16	16	32
32	32	32	32	64
64	64	64	64	128
128	128	128	128	256

Tables II-V can be used to compare our approach's memory usage to the traditional sliding window approach. We examine the impact of varying window size, image resolution, and specified lossiness. Table II shows that with our compression algorithm the packed bits for window size 8 can fit in two 18Kb BRAMs for the lossless case (T=0) with two BRAMs for management bits. This represents a 50% memory saving compared to the traditional architecture. Increasing the threshold value to 6 (most lossy case considered) makes it possible to fit more rows in one Block RAM. This represents a memory saving of 62.5%. These numbers show that the proposed architecture is more efficient in exploiting BRAMs by compressing the pixels and packing more than one image's row into one BRAM.

Next, we take a closer look at the managment bits (BitMap and NBits) memory requirements. For window size 8 and image resolution 512×512, the size of BitMap equals 8×(512-8) bits and the size of NBits equals 4×2×(512-8) bits. The BitMap can be buffered in one 18Kb BRAM with a 2k×9 configuration. The NBits can also fit in one 18Kb BRAM with a 2k×9 configuration where bits 0-3 used for low sub band and bits 4-7 are used for the high sub band. Table 2 shows a total of two BRAMs for window size 8 and image resolution 512×512.

TABLE II: Number of BRAMs (18Kb) for Resolution (512×512)

Window size	Packed bits				Management bits
	T=0	T=2	T=4	T=6	
8	2	2	2	1	2
16	4	4	2	2	2
32	8	8	4	4	2
64	16	16	16	8	3
128	32	32	32	16	5

TABLE III: Number of BRAMs (18Kb) for Resolution (1024×1024)

Window size	Packed bits				Management bits
	T=0	T=2	T=4	T=6	
8	4	4	2	2	2
16	8	8	4	4	2
32	16	16	8	8	3
64	32	32	16	16	5
128	64	64	32	32	9

TABLE IV: Number of BRAMs (18Kb) for Resolution (2048×2048)

Window size	Packed bits				Management bits
	T=0	T=2	T=4	T=6	
8	4	4	4	4	2
16	8	8	8	8	3
32	16	16	16	16	5
64	32	32	32	32	9
128	64	64	64	64	16

TABLE V: Number of BRAMs (18Kb) for Resolution (3840×3840)

Window size	Packed bits				Management bits
	T=0	T=2	T=4	T=6	
8	8	8	8	8	4
16	16	16	16	16	6
32	32	32	32	32	9
64	64	64	64	64	16
128	128	128	128	128	28

Increasing window size to 16 shows similar results for 512×512 resolution. The compressed bits of every four rows can fit in one BRAM for lossless compression and for lossy compression with threshold equals 2. This shows a memory saving of 62.5%, if the compression ratio is increased by increasing the threshold value to 4 and 6 to make every 8 rows fit in one BRAMs. The color in Tables II-V represents number of image rows mapped to one BRAM. Green cells represent mapping each 8 rows in the input image to one BRAM. Blue cells represent mapping each 4 rows to one BRAM. Yellow cells represent mapping each two rows to one BRAM.

Increasing image width from 512 to 1024 and 2048 increases the compression ratio, but the number of pixels in each row increases, so increases the total number of bits. This can be seen clearly when the image width increases from 1024 to 2048 in Tables III and IV. When the Threshold value is 4 and 6, the memory saving is around 75% for 1024. That means the architecture was able to pack 8 rows of 1024 pixels in one BRAM. Increasing the image width to 2048 allows for packing 4 rows of 2048 pixels in one BRAM. A saving of approximately 50% compared to the traditional architecture.

The architecture currently uses a threshold to configure the system for lossless to varying degrees of lossy compression. Our evaluations show thresholds of 2, 4 and 6 gives mean square errors (MSEs) of 0.59, 3.2 and 4.8 respectively. Other options can be investigated to vary lossiness, such as using the average of previous pixels.

B. Hardware Resource Utilization

This section presents the Post-Synthesis hardware resource utilization for each block in our architecture. Each table shows the number of LUTs, registers and the maximum operating frequency. We used Xilinx Vivado 2015.3 tool and Xilinx Zynq 7020 (XC7z020) FPGA [14] in our experiments. It has a total of 53,200 LUTs and 106,400 registers. Table VI and Table IX show the resources of the forward and inverse integer wavelet transform blocks. The resources of these two blocks are similar because they have similar architectures. Table VII shows the resources of the Bit Packing unit for different window sizes. It shows that resources are linearly increasing with the window size from 1% LUTs for window size 8 to 3%,7%,16% and 32% for window sizes 16, 32, 64 and 128. The Bit Unpacking block consumes more resources compared to the other blocks. It consumes 15%, 29% and 59% LUTs for window size 32, 64 and 128. This is due to a large multiplexer in the block that selects bits from both the remaining bits from the previous read and the new input. Table X shows the overall resources of the whole architecture. The LUTs for window size 32 and 64 was around 33% and 67% of the total LUTs on the chip. For a window size of 128 the LUTs exceed this device resources.

TABLE VI: Integer wavelet transform (IWT)

Window size	LUTs	Registers	Frequency
8	386 (0%)	166 (0%)	592.1 MHz
16	770 (1%)	326 (0%)	592.1 MHz
32	1538 (2%)	646 (0%)	592.1 MHz
64	3074 (3%)	1276 (1%)	592.1 MHz
128	6146 (11%)	2566 (2%)	592.1 MHz

TABLE VII: Bit Packing unit hardware resources

Window size	LUTs	Registers	Frequency
8	1061 (1%)	200 (0%)	538.6 MHz
16	2083 (3%)	400 (0%)	538.6 MHz
32	4047 (7%)	801 (0%)	538.6 MHz
64	8598 (16%)	1856 (1%)	538.6 MHz
128	17179 (32%)	3712 (3%)	538.6 MHz

TABLE VIII: Bit UnPacking unit hardware resources

Window size	LUTs	Registers	Frequency
8	2130 (0%)	203 (0%)	343.1 MHz
16	4246 (7%)	387 (0%)	343.1 MHz
32	8039 (15%)	817 (0%)	343.1 MHz
64	15660 (29%)	1637 (1%)	343.1 MHz
128	31660 (59%)	3237 (3%)	343.1 MHz

TABLE IX: Inverse IWT unit hardware resources

Window size	LUTs	Registers	Frequency
8	386 (0%)	130 (0%)	592.1 MHz
16	770 (1%)	258 (0%)	592.1 MHz
32	1538 (2%)	529 (0%)	592.1 MHz
64	3074 (3%)	1055 (1%)	592.1 MHz
128	6146 (11%)	2108 (2%)	592.1 MHz

TABLE X: The overall architecture hardware resources

Window size	LUTs	Registers	Frequency
8	4994 (9%)	1643 (1%)	230.3 MHz
16	9432 (17%)	2792 (2%)	230.3 MHz
32	17773 (33%)	5091 (4%)	230.3 MHz
64	35751 (67%)	9680 (9%)	230.3 MHz
128	-	-	-

VII. Conclusion

In this paper, we present a new sliding window architecture that efficiently utilizes the available Block RAMs on chip. The proposed image compression algorithm gives good compression ratio and can be used in our architecture to reduce BRAMs at the expense of introducing more LUTs resources. The architecture can be configured to perform sliding window operations using lossless or lossy compression based on an application's requirement. Evaluating the proposed architecture on a set of images selected from a benchmark dataset shows promising results. The memory saving reached 25-70% for lossless compression and up to 84% for lossy compression. The proposed architecture is fully pipelined, giving similar performance to the traditional architecture. The compression ratio is currently configured at design time, our future work will investigate making this automatically adjustable at runtime based on the previous frame compression ratio.

References

[1] J. Fowers, G. Brown, P. Cooke, and G. Stitt, "A performance and energy comparison of FPGA, GPUs, and multicores for sliding-window applications," in *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, pp. 47–56, ACM, 2012.

[2] C. Huang and F. Vahid, "Scalable object detection accelerators on FPGAs using custom design space exploration," in *2011 IEEE 9th Symposium on Application Specific Processors (SASP)*, pp. 115–121, June 2011.

[3] H. Blasinski, W. Hai, and F. Lohier, "FPGA architecture for real-time barrel distortion correction of colour images," in *Multimedia and Expo (ICME), 2011 IEEE International Conference on*, pp. 1–6, IEEE, 2011.

[4] A. Amaricai, C. E. Gavriliu, and O. Boncalo, "An FPGA sliding window-based architecture harris corner detector," in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–4, Sept 2014.

[5] H. Yu and M. Leeser, "Optimizing data intensive window-based image processing on reconfigurable hardware boards," in *IEEE Workshop on Signal Processing Systems Design and Implementation, 2005.*, pp. 491–496, Nov 2005.

[6] H. Yu and M. Leeser, "Automatic sliding window operation optimization for FPGA-based," in *2006 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 76–88, April 2006.

[7] Y. Dong, Y. Dou, and M. Liu, "A design space exploration algorithm incompiling window operation onto reconfigurable hardware," *International Journal of Computers and Applications*, vol. 30, no. 1, pp. 36–43, 2008.

[8] P. Praveena, "Implementation of loco-i lossless image compression algorithm for deep space applications," *International Journal of Reconfigurable and Embedded Systems*, vol. 3, no. 3, 2014.

[9] A. M. D. Silva, D. G. Bailey, and A. Punchihewa, "Exploring the implementation of jpeg compression on FPGA," in *2012 6th International Conference on Signal Processing and Communication Systems*, pp. 1–9, Dec 2012.

[10] J. Jyotheswar and S. Mahapatra, "Efficient FPGA implementation of dwt and modified spiht for lossless image compression," *Journal of Systems Architecture*, vol. 53, no. 7, pp. 369–378, 2007.

[11] G. Stitt, E. Schwartz, and P. Cooke, "A parallel sliding-window generator for high-performance digital-signal processing on FPGAs," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 9, no. 3, p. 23, 2016.

[12] W. Zhenhua, X. Hongbo, T. Yan, T. Jinwen, and L. Jian, "Integer Haar wavelet for remote sensing image compression," in *6th International Conference on Signal Processing, 2002.*, Aug 2002.

[13] B. Zhou, A. Lapedriza, J. Xiao, A. Torralba, and A. Oliva, "Learning deep features for scene recognition using places database," in *Advances in neural information processing systems*, pp. 487–495, 2014.

[14] Xilinx, "Zynq-7020." https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf, 2016.