DESIGN AND IMPLEMENTATION OF A 2D CONVOLUTION CORE FOR VIDEO APPLICATIONS ON FPGAs

Dr. Khaled Benkrid **Mr. Samir Belkacemi**

The Queen's University of Belfast, 18 Malone Road, Belfast BT7 1NN, UK
k.benkrid@qub.ac.uk

Abstract

This paper presents the design and implementation of a 2-D convolution core for video applications optimised for the Xilinx low cost 3.3V SpartanXL™ FPGA family. The core is parameterised and scaleable in terms of the convolution window size and coefficients, the input pixel word length and the image size. The window coefficients are represented as sum/subtract of power of twos in Canonical Signed Digit (CSD) representation, which means that the usually costly multiplication operation can be easily implemented by a small number of simple shift-and-add operations, leading to considerable hardware savings. Optimised FPGA configurations capable of processing real time PAL video are automatically generated from high-level descriptions of generic 2-D convolutions, in the form of EDIF netlists, in less than 1 sec.

I. INTRODUCTION

Image Processing applications deal with computationally demanding operations often under real time requirements. The spatial-domain two-dimensional (2-D) convolution is one of the essential operations in real-time image and video processing [1]. It involves passing a 2-D window over an image and performing a Multiply-Accumulate operation between the image pixels and window coefficients. This operation presents a very high degree of parallelism. Reprogrammable hardware in the form of FPGAs [2] have been proposed as a convenient platform to exploit this parallelism and achieve high performance. Indeed, FPGAs offer both the performance of a direct hardware solution with the flexibility offered by the reprogrammability feature. However, in order to get the most of their potential performance, FPGAs need to be programmed at the low hardware level and not at the application level. This process needs a considerable hardware knowledge, which means that FPGA programming is still reserved to specialists.

One way of accelerating FPGA programming process while maintaining hardware efficiency has been to assist users with optimised core generators that are parameterisable and scaleable according to the user's needs. This paper presents the design and implementation of a high-level core generator for 2-D convolution operations. This is parameterised in terms of the window size, the window coefficients, the input pixel word length and the image size. The core is optimised for the Xilinx low cost 3.3V SpartanXL™ series [3]. It has been written in a Prolog-based [4] Hardware Description

0-7803-7984-5/02 $10.00 © 2002 IEEE

Environment, called HIDE, developed at Queen's University Belfast [5]. HIDE enables highly scaleable and parameterised composition of blocks using a small set of constructors (e.g. *horizontal*, *vertical*). It generates optionally pre-placed configurations in EDIF format for Xilinx FPGAs.

The following will first start by presenting a generic hardware architecture for 2-D convolution. Next, the design of the key basic building blocks of this architecture are presented, before the high level generation of the core is illustrated. Timing and area measurements of an implementation of a particular instance of this core for processing PAL video are then presented. Finally conclusions are drawn.

II. HARDWARE ARCHITECTURE

As mentioned above, a 2-D convolution operation involves passing a 2-D window over an image, and carrying out a Multiply-Accumulate operation at each window position. To allow each pixel to be supplied only once to the FPGA, line delays (buffers) are required. These line delays synchronise the supply of input pixels to the Multiply-Accumulate elements, ensuring that all the pixel values involved in a particular neighbourhood operation are processed at the same instant [1][6]. Figure 1 shows the architecture of a generic 2-D convolution operation with a P by Q elements window.

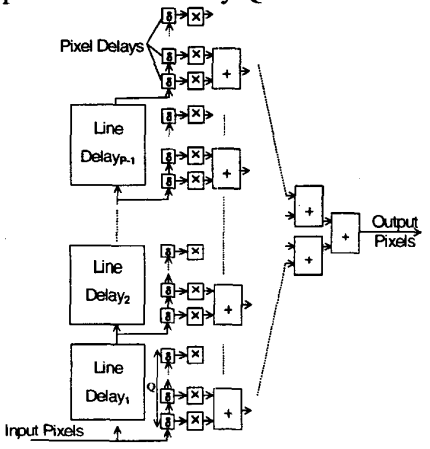


Figure 1. A generic PxQ 2D convolution architecture

As we are targeting Xilinx SpartanXL™ FPGAs, we have chosen bit parallel arithmetic to implement our architectures. Indeed, it has been shown that bit parallel implementations often lead to a better time area product on Xilinx FPGAs compared to bit serial ones [5]. The key for this is the availability of dedicated fast carry logic on these FPGAs [7].

2.1. Basic component architectures

In this section, we will present the implementation of the basic building blocks needed for the architecture shown in Figure 1. These are:

- Bit parallel multiplication
- Bit parallel accumulation (addition)
- Pixel delays

2.1.1 Bit parallel multiplication.

Multiplication involves two basic operations: the generation of the partial products and their accumulation. In order to speed up the multiplication process, either the number of the partial products should be reduced, or the accumulation process should be speeded up. Note that the partial products corresponding to '0' bits in the multiplier input are zeros, and therefore do not have to be included in the sum. If the number of '1' bits in a constant coefficient multiplier is small, then the multiplication may be realised with shifts (for multiplication with powers of twos) and few adders. Moreover, in the case where the multiplier contains a string of consecutive 1's, a proper recoding of the multiplier can reduce the number of partial products. Consider the example of a multiplication by 15. Note that 15=1111 can be expressed as 16-1= $1000\bar{1}$ in Signed Digit notation where $\bar{1} = -1$. Thus the multiplication is reduced to a simple subtraction (with input shift). The recoding scheme reduces the number of partial products by a half on average. In common neighbourhood operations, window coefficients are often sparse (contain many 0's) and thus yield to a considerable reduction in hardware complexity. For instance, no multiplication is needed in a Laplacian filter, which is a convolution with the following window:

0	-1	0
-1	4	-1
0	-1	0

The minimal number of partial products required for a given multiplier is equivalent to the problem of finding the smallest number of non-zero digits in a Signed Digit (SD) representation of the multiplier. The algorithm for obtaining the minimal representation of a SD number is called *canonical signed digit recoding* (CSD) [8].

In general, if R is the number of 1's and –1's in the minimal SD representation of a multiplier coefficient C, the multiplication is then performed by the accumulation of R values. Each value is in fact equal to the multiplicand value (M) multiplied by a certain power of two (N_k) as shown in the following equation:

$$C \times M = \left(\sum_{k=1}^{R} sgn_k \, 2^{P_k} \right) \times M = \sum_{k=1}^{R} \left(sgn_k \, 2^{P_k} \times M \right) \quad \text{where } sgn_k = 1 \text{ or } -1$$

$$= \sum_{k=1}^{R} (N_k \times M)$$

Multiplication by a power of two is simply a shift operation and does not consume any hardware. The whole multiplication then reduces to a reduction operator which can be performed by a tree structure as shown in Figure 2. In order to speed up the circuit, a pipeline stage is added after each two columns of the tree (see Figure 2). Note that it is still possible to speed up the circuit even further by inserting a pipeline stage at each tree column at the expense of extra latency and hardware.

We provide a utility for generating a generic reduction operator block based on a tree structure. The corresponding generator header is:

is_tc_par_reduction_op(Op_2, Input_List, Delay_pos, Max_width, OWL, Latency, B)

Op_2 defines the 2-input operation in each node of the tree. In our multiplication case, to get a global accumulation, **Op_2 = add_sub**.

Input_List is a list of 2-tuples of the following form:

$$[(PW_1, N_1), (PW_2, N_2), \ldots, (PW_i, N_i), \ldots]$$

in which each tuple contains a particular input word length PW_i and its corresponding power of two coefficient N_i.

Delay_pos is a constant which determines the position of the delay elements and can have 4 possible values:

- '0' in which case no delays are used.
- '1' in which case the delay elements are inserted after each odd numbered column of the tree.
- '2' in which case the delay elements are inserted after each even numbered column of the tree (as in Figure 2).
- '3' in which case the delay elements are inserted after each column of the tree.

Max_width is the maximum output word length. This is needed to restrict the pixel width from growing unnecessarily.

OWL is the resulting output pixel word length and **Latency** is the latency of the resulting block **B**.

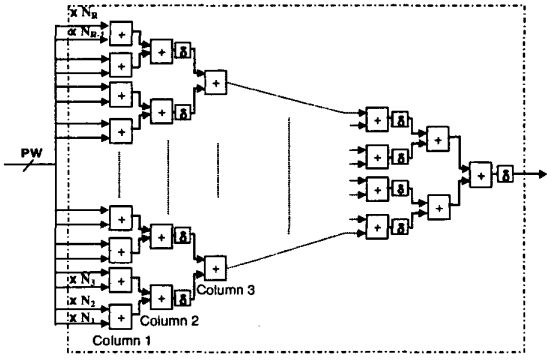


Figure 2. Bit parallel Multiplier architecture: Reduction operator

Note that the tree nodes in Figure 2 (particularly in the first column) perform a weighted addition/subtraction since each input should be first multiplied by a power of two value (perhaps negative).

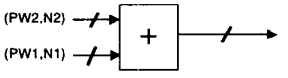


Figure 3. Basic component: A weighted adder/subtracter

Based on this weighted adder/subtracter unit (see Figure 3) and using the generic reduction operator of Figure 2, a multiplication unit can be easily generated by:

is_tc_par_mult(Coeff, IWL, Max_width, OWL, Latency, B)

where **Coeff** is the multiplier's constant coefficient, **IWL** is the input word length, **Max_width** is the maximum output word length allowed.

2.1.2 Bit parallel accumulation

This unit is based on dedicated fast carry logic as explained in the previous section. It can be implemented by the weighted adder/subtracter unit of Figure 3 where $N_1 = N_2 = 1$.

2.1.3 Pixel delays

Delays are implemented on Xilinx SpartanXL™ FPGAs using the logic blocks' 16x1 synchronous distributed RAMs with address counters [9]. With a modulo-16 counter,

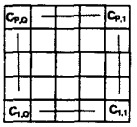
each logic block (CLB) can implement two 16x1 synchronous RAMs, hence a 32 bit delay. The CLBs' distributed RAMs can share the same address counter.

III. HIGH LEVEL GENERATION OF 2-D CONVOLUTIONS

In this section, we will present the high level generation of the whole 2-D convolution architecture. Any 2-D convolution instance will be derived from a suitable instance of the following generic constructor:

is_tc_par_convolution(Window, Buffer_size, IWL, OWL, Latency, B)

Consider a convolution operation with the following generic PxQ window:



This can be represented by a list of lists as follows:

$$[[C_{P,Q}, \ldots, C_{P,1}], [C_{P-1,Q}, .., C_{P-1,1}], ..,[C_{1,Q}, .., C_{1,1}]]$$

Unused window positions are represented by '~'.

Once all the PxQ pixels of a particular neighbourhood are available (after proper line and pixel buffering as shown in Figure 1), the local operation (multiplication) needs first to be performed, followed by the global operation (accumulation). In this particular case, we can do a useful optimisation by noting that the local multiplication (which essentially is an accumulate of partial results) and the global accumulation can be combined in a single large accumulator. This will result in the saving of delay elements, which would have been necessary if each local multiplication has been performed separately. Figure 4 illustrates this for two coefficients 21 and 1. In effect, we are just balancing the complete accumulation tree.

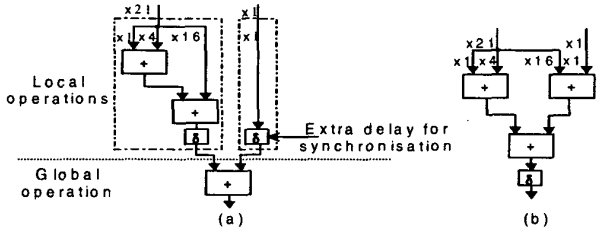


Figure 4. Convolution process: (a)Local multiplications performed separately (b) Merging the local multiplications with the global accumulation

For a coefficient $C_{i,j}$, each incoming pixel within the neighbourhood is replicated $R_{i,j}$ times, where $R_{i,j}$ is the number of 1's and –1's in the minimal CSD representation of $C_{i,j}$. In general, the whole multiplication process within the window neighbourhood will result in $Pn = \sum_{i,j} R_{i,j}$ pixels ($1 \le i \le P$, $1 \le j \le Q$), where each replicated pixel has a particular power of two weight $(N_{i,j})_k$, $1 \le k \le R_{i,j}$, such as:

$$C_{i,j} = \sum_{k=1}^{R_{i,j}} sgn_{i,j}\, 2^{(P_{i,j})_k} = \sum_{k=1}^{R_{i,j}} (N_{i,j})_k \quad \text{where } sgn_{i,j} = 1 \text{ or } -1$$

The multiplication operation results effectively in the replication of each set of pixel wires in '*Pix-delay*' unit (see Figure 5 where a horizontal scanning of the input image is assumed). This process is performed by the following constructor:

is_tc_par_local_mult(Window, New_Coeff_List, Latency, B)

Window is a list of lists containing the window coefficients.
New_Coeff_List is a list containing each output pixel word length $PW_{i,j}$ along with its corresponding power of two coefficient $(N_{i,j})_k$:

$$[..,(PW_{i,j},(N_{i,j})_1), (PW_{i,j},(N_{i,j})_2),..], \; i=1, 2,.., P \text{ and } j=Q, Q-1,.., 1$$

The global operation (accumulation) can then be seen as a tree of two-operand additions/subtractions. Each addition/subtraction operand has an associated power of two weight (particularly in the first column of the tree). The maximum pixel word length (i.e. the minimum necessary processing word length) depends on the input pixel word length and the window coefficients. We provide the following utility to compute the maximum pixel word length:

is_maximum_pixel_width(IWL, Window, Max_width)

The convolution processing unit (see Figure 5) can then be described by the following Prolog rule:

```
is_tc_par_convolution_proc_unit(Window, Buffer_size, IWL, OWL, Latency_in, Latency_out, B):-
is_tc_par_local_mult(Window, New_Coeff_List, Latency_Loc, Local),
is_maximum_pixel_width(IWL, Window, Max_width),
is_tc_par_reduction_op(add_sub, New_Coeff_List, 2, Max_width, OWL, Latency_Glo, Global),
B =horizontal([Local, Global]),
Latency_out is Latency_in+Latency_Loc + Latency_Glo.
```

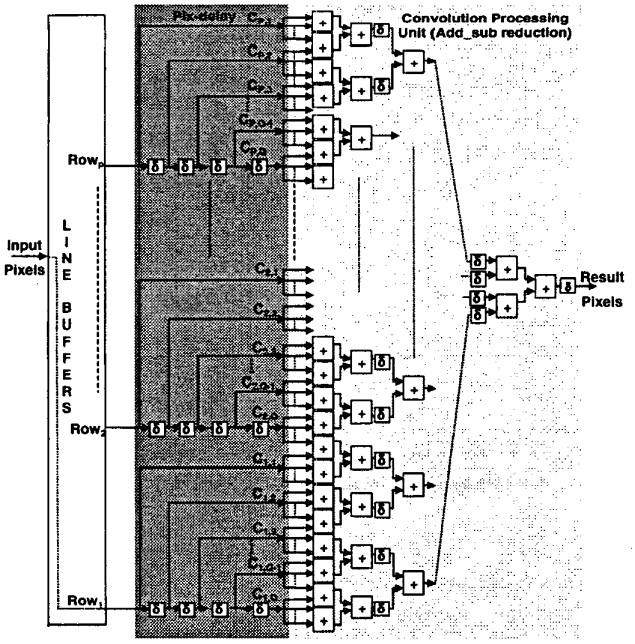


Figure 5. Architecture of a PxQ 2-D convolution

Having built the convolution processing unit from a small number of basic building components, we can now build the full convolution architecture by adding the necessary line and pixel delays. Based on pixel delays (see section 2.1.3 above), and given the input pixel word length, the convolution window and the buffer size (image width or height depending on the image scanning direction), the following generates the necessary line and pixel delays:

is_tc_par_line_and_pixel_delays(IWL, Window, Buffer_Size, B)

The full convolution architecture can then be easily built as follows:

```
is_tc_par_convolution(Window, Buffer_size, IWL, OWL, Latency, B):-
is_tc_par_line_pixel_delays(IWL, Window, Buffer_size, Buffers),
is_tc_par_convolution_proc_unit(Window, Buffer_size, IWL, OWL,0, Latency ,Proc_Unit),
B =horizontal([Buffers, Proc_Unit]).
```

IV. IMPLEMENTATION RESULTS

FPGA configurations of instances of the above core can be generated in less than 1 sec. The core generates FPGA configurations in EDIF netlist format. FPGA bitstreams are then generated from the resulting EDIF descriptions using Xilinx placement and routing tools. For instance a Laplace filter configuration for PAL video (720x576 images of 8-bit/pixel) is generated automatically from the following high-level description:

is_tc_par_convolution([[~,-1,~],[-1,4,-1],[~,-1,~]], 576, 8, OWL, Latency, B)

Table 1 gives the timing and area measurements of the resulting implementation on a Xilinx Spartan XCS30XL-4 FPGA chip for two levels of pipelining:

1. Partially pipelined: This corresponds to a delay element inserted after each even numbered column of the reduction tree of Figure 2.
2. Fully pipelined: This corresponds to a delay element inserted after each column of the reduction tree of Figure 2. This can be easily done by supplying the reduction tree constructor with a parameter 3 for **'delay_pos'** instead of 2 (see section 2.1.1).

Table 1. Laplace Filter implementation results

	Speed (MHz)	Frame Rate (frame/sec)	Total Area (CLBs)
Partially pipelined	85	25	360
Fully pipelined	90	27	378

As can be seen from Table 1, the speed improves with full pipelining, at the expense of little extra CLBs (18). Nonetheless both implementations can achieve real time performance.

Other examples of instances of our core are:

- *Image Sharpening:* generated automatically from the following high level description:

  is_tc_par_convolution([[~,-1,~],[-1,6,-1],[~,-1,~]], 576, 8, OWL, Latency, B)

- *Image Blurring:* generated automatically from the following high level description:

  is_tc_par_convolution([[1,1,1],[1,1,1],[1,1,1]]), 576, 8, OWL, Latency, B)

In general, our core's performance rivals with our carefully handcrafted designs. It has been used successfully to implement a video coprocessor running on an FPGA based video board (Visicom's VigraVision™) [10][11].

V. CONCLUSION

In this paper, we have presented the design and implementation of a 2-D convolution core for video applications optimised for the Xilinx low cost 3.3V SpartanXL™ FPGAs. The core is parameterised and scaleable in terms of the window size, the window coefficients, the input pixel word length and the image size. It generates optimised FPGA configurations capable of processing real time PAL video, in the form of EDIF netlists, from high-level descriptions of generic 2-D convolutions using a small set of basic building blocks in less than 1 sec. The core exploits the window coefficient values to reduce the number of adders/subtractors needed in multiplications. Indeed, canonical signed digit recoding is used in order to minimise the number of partial products in a multiplication. This is particularly useful in the case of image processing operations where integer coefficients are often used. Further, these are often powers of two or zeros, which makes the multiplication operation redundant.

Our core's performance rivals with our manually crafted designs. It has been used successfully to implement many real time video-processing operations running on a commercial FPGA based video board.

VI. REFERENCES

[1] Kamp, W., Kunemund, H., Soldner and Hofer, H., 'Programmable 2D linear filter for video applications', IEEE Journal of Solid State Circuits, pp 735-740, 1990.

[2] Rose, J., and Sangiovanni-Vincentelli, A., 'Architecture of Field Programmable Gate Arrays', Proceedings of the IEEE Volume 81, No7, pp 1013-1029, 1993.

[3] Xilinx Ltd, 'Spartan and Spartan-XL Families Field Programmable Gate Arrays Data Sheets', June 2002. http://direct.xilinx.com/partinfo/ds060.pdf

[4] Clocksin W. F. and Melish C. S., 'Programming in Prolog', Springer-Verlag, 1994.

[5] Benkrid, K., 'Design and Implementation of a High Level FPGA Based Coprocessor for Image and Video Processing', PhD Thesis, School of Computer Science, The Queen's University of Belfast, 2000. http://www.cs.qub.ac.uk/~K.Benkrid/MyThesis.html

[6] Shoup R. G., 'Parameterised Convolution Filtering in an FPGA', More FPGAs, W. Moore and W. Luk (editors) Abington EE&CS books, pp. 274, 1994.

[7] Xilinx Ltd., 'Using the Dedicated Carry Logic', Xilinx Application Notes, XAPP 013, July 1996. http://www.xilinx.com/xapp/xapp013.pdf

[8] Koren, I., 'Computer arithmetic algorithms', Prentice-Hall, Inc, pp 99-126, 1993.

[9] Xilinx Application Notes, 'Efficient Shift Registers, LFSR Counters, and Long Pseudo-Random Sequence Generators', XAPP 052, July 1996. http://www.xilinx.com/xapp/xapp052.pdf

[10] Benkrid, K., Crookes, D., Smith, J. and Benkrid, A., 'High Level Programming for Real Time FPGA Based Video Programming', Proceedings of ICASSP'2000, Istanbul, June 2000. Volume VI, pp. 3227-3231.

[11] Benkrid, K., Crookes, D., Smith, J., and Benkrid, A., 'High Level Programming for FPGA Based Image and Video Processing using Hardware Skeletons', IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM'2001, April 2001, Preliminary Proceedings.