A Compact FPGA Implementation of a Bit-Serial SIMD Cellular Processor Array

Declan Walsh and Piotr Dudek
School of Electrical and Electronic Engineering
The University of Manchester
Manchester, United Kingdom
declan.walsh@postgrad.manchester.ac.uk; p.dudek@manchester.ac.uk

***Abstract*— An FPGA implementation of a fine grain general-purpose SIMD processor array is presented. The processor architecture has a compact processing element which is encapsulated into two configurable logic blocks (CLBs) and is then replicated to form an array. A 32 × 32 processing element array is implemented on a low-cost Xilinx XC5VLX50 FPGA using four-neighbour connectivity with the possibility to scale up using a larger FPGA. The processor array operates at a frequency of 150 MHz and executes a peak of 153.6 GOPS (bit-serial operations). Binary and 8-bit greyscale image processing is performed and demonstrated.**

I. Introduction

Massively parallel processor arrays have been long known to be suitable for implementing image processing [1]-[5]. More recently, some state of the art processor arrays have been used for real-time machine vision tasks in applications such as intelligent transport systems [6] and video processing on mobile platforms [7], providing a much more powerful solution than a conventional processor. A reason for the improved performance of Single Instruction Multiple Data (SIMD) processor arrays in image processing, as compared with single or multi-core processors, is that a lot of low-level image processing operations rely closely on local data and the same operation needs to be performed on every pixel. Such operations map very well onto parallel array processors.

By employing a processor-per-pixel architecture, each processor can access its local data which eliminates the need to transfer data throughout the array. The designs in [8]-[12] are also advantageous as each processor is connected to a photodetector which significantly improves the processor's ability to input an image without having to route data throughout the array. In [8], [11] and [12], analogue computations in a pixel-per-processor array can save a lot of area on the chip. However, as fabrication technologies are scaled, analogue processors are not as robust as digital processors so digital processor architectures are preferred [9].

Many SIMD processor arrays have been implemented on FPGAs [13]-[18]. The design in [16] presents a 48 × 48 processor array that performs binary operations showing FPGAs to be capable of implementing moderate sized fine grained processor arrays which could even be scaled up to larger FPGAs. Even though FPGAs cannot offer performance as high as ASICs and despite their lower frequency of operation, they can still offer significant performance due to their parallel nature, with their overall performance being limited by FPGA size [17]. Due to their reconfigurabilty there is a much lower development cost with FPGAs and they are much more easily sustainable in the long term as designs can be implemented on newer FPGAs.

The aim of the presented design is to create a processor array where the processing elements are as small as possible in terms of area while still being able to provide useful functionality in terms of an ALU and substantial local memory sufficient for processing greyscale images. Despite area being of primary concern, it is important that performance is not significantly impaired. The rest of this paper is organised as follows. Section II will describe the processor and array design, Section III will show the FPGA implementation and resource utilisation. Section IV will describe some of the processor operations and some example algorithms will be presented with results shown. Section V will give a brief summary and conclusion of the paper.

II. FPGA and Processor Design

A. FPGA Architecture

The FPGA used for the implementation of the processor array is a Xilinx Virtex-5 FPGA. This FPGA [19] has configurable logic blocks (CLBs) with two 'slices' per CLB. Each slice includes four 6-input lookup tables (LUTs), four flip-flops and some additional logic. At least one in every four slices is a SLICEM (as opposed to a SLICEL) which means that the LUTs in this particular slice can be implemented as RAM [19].

B. Processing Element Design

The primary aim of this design is to have a small processing element (PE). As such, it is vitally important that the processing element can occupy as few resources as

This work has been supported by EPSRC; Grant No.: EP/H023623/1

possible meaning that a relatively simple design is required. In turn, this allows the implementation of a larger array on the FPGA. The connectivity between PEs is with the four nearest neighbours in the north, east, west and south directions. This reduces the multiplexing requirements for wires coming into the PE. By limiting each PE to two CLBs (four slices), a 32 × 32 array of PEs can easily fit on the XC5VLX50 FPGA. Using a much larger FPGA, for example, the Xilinx Virtex-5 XC5VLX330 FPGA, an array of 112 × 112 PEs can be implemented. The ability to include one SLICEM into each PE, allows 256 bits of RAM to be implemented on each PE.

The basic processing element is shown in Fig. 1. It is a bit-serial processor and includes a 256-bit single-port synchronous RAM block, three general-purpose registers (X, Y and Z), an accumulator register, a FLAG register for local autonomy, a NEWS register for local communication, an ALU and a carry register. A 22-bit instruction word is provided to the PEs to indicate which operations are to be performed (these are shown in Fig. 1 by the dashed lines).

A schematic of the ALU is given in Fig. 2. There are three inputs (Input, Acc and Carry In) and two outputs (Output and Carry Out). This does not include the instruction inputs (five bits controlling the multiplexers). One of the inputs comes from either local storage (RAM or register) or a neighbouring PE, another always comes from the accumulator register and the third always comes from the carry register. The ALU can perform the operations COPY, AND, XOR, OR, SUM, CARRY, SET ‘0’, and SET ‘1’ with the required operation being chosen using an 8:1 multiplexer. As these are all logic operations, in practice, they will be implemented on the FPGA using LUTs. The output of any selected operation (can also be optionally inverted) is written to the accumulator and any selected registers or RAM, and the carry output of the full adder is written to the carry register only when the SUM operation is selected. These operations are seen as the fundamental operations which could offer the best performance while consuming minimal area. Any further operations would have required more resources and possibly an even larger multiplexer and would have occupied a larger area on the FPGA. If the inverse of an operation is required, then this can be achieved by selecting the appropriate input of the 2:1 multiplexer at the output of the ALU which effectively allows the implementation of the NOT, NAND, XNOR and NOR functions as well. Similarly, if the inverse of the accumulator is required as the input to the ALU, then this can be achieved by selecting the appropriate input to the 2:1 multiplexer at the input to the ALU.

There are multiple reasons for having the COPY function. The first reason is that the PE uses an accumulator based architecture which requires a value to be written to the accumulator register before the next operation can occur. Secondly, it is used when shifting data through the array by reading from a particular direction and writing to the NEWS register. Thirdly, it can be used to transfer a value from one storage location to another. Data, however, cannot be transferred from one RAM location to another RAM location in the same clock cycle. As the RAM is to be implemented in one of the slices (four 6-input LUTs), this allows 256 single bit locations with an address of 8 bits and only one of these locations can be accessed on any given clock cycle resulting in a single-port RAM. In the event that data must be transferred between two different RAM locations in the same PE, the value can be read from the RAM and written to one of the three registers on the first clock cycle, and then read from this register and written to the appropriate RAM location on the second clock cycle. However, a data value can be read from a RAM location and a new value written to that same location in the same clock cycle. An alternative solution could have 128-bit dual-port RAM implemented over the same area enabling reading and writing to different locations in one clock cycle.

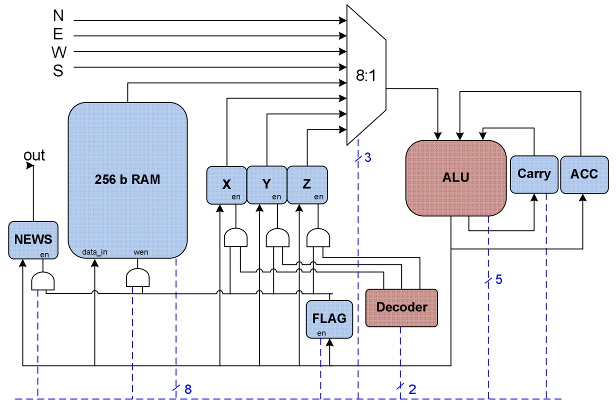

Figure 1. Processing element schematic.

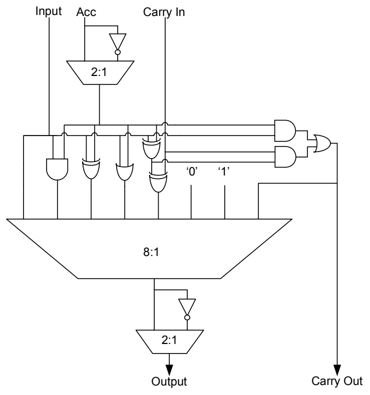

Figure 2. Arithmetic Logic Unit.

The array uses four-neighbour connectivity where the output from the NEWS register is connected to each of its four nearest neighbours. These can be seen as inputs in Fig. 1; each input is denoted by the letters N, E, W and S. These four inputs are connected to an 8:1 multiplexer as well as the RAM block and the three general-purpose registers and one of these inputs is always selected as one of the inputs to the ALU.

Local autonomy is provided by means of the FLAG register. This register can be written to directly from the ALU similarly to the NEWS and general-purpose registers. When this register is set to ‘0’, the three general-purpose registers, the RAM, and the NEWS register are disabled, effectively putting the PE in an idle state. The output of the FLAG register undergoes the AND operation with the enable signals of these other elements resulting in them being disabled.

The proposed architecture possesses some similarities with the design described in [16] but there are some differences which make the proposed design more flexible. Firstly, there is a more versatile ALU providing additional operations such as an adder and XOR function in addition to all the functions offered in [16]. As well as this, the proposed architecture has 256 bits of RAM as opposed to eight 1-bit registers for local storage allowing for the storage of larger images or even multiple images. The FLAG register also provides the ability to disable particular PEs which is not possible in [16].

C. *Processing Element Control*

The control for the processing element is provided by a 22-bit instruction word which provides signals to a decoder and multiplexers as well as some enable signals. Nine of these bits are used for the RAM block with eight used as the address and the other bit being a write enable signal. Three bits are used as the select lines for the 8:1 multiplexer which selects the input to the ALU and a further three bits are used for the 8:1 multiplexer in the ALU to select which operation to perform. Two individual bits are used as the select lines for the 2:1 multiplexers at the input and the output of the ALU. A decoder is used to determine which of the general-purpose registers is to be written to, this uses another two bits. The final three bits are the enable signal for the NEWS register, the enable signal for the FLAG register and the clear carry signal which clears the carry register. There is not a set carry signal. To set the carry register to ‘1’, first the carry register can be cleared and then by using the SUM operation to add two ‘1’s, the carry register can be set to ‘1’.

D. *Processing Element Implementation*

When the PE design is synthesised, constrained into four slices and implemented, it occupies 7 slice registers and 16 slice LUTs (12 of which are implemented as logic and four as RAM). The properties were optimised for area. The floorplan of the four slices is shown in Fig. 3. The shaded areas are the occupied resources with LUTs at the left hand side of each slice and flip-flops at the right hand side of each slice and multiplexers in the middle. The SLICEM can be seen as the slice furthest to the left in which the RAM is implemented. There may appear to be a lot of LUTs being used and the main reason for this is that the four flip-flops on each slice share the same enable signal. Because all of the registers in the design have different enable signals, this would require each register to be placed on a slice on its own which would significantly increase the processor area. Instead, a LUT is used along with each register to allow multiple registers to be implemented on the same slice.

E. *Processor Array Design*

The processor array, named FPAC (Field Programmable Array Core), is constructed as in Fig. 4 with a four-neighbour connectivity. A 32 × 32 array with each PE occupying two CLBs can fit on the XC5VLX50 FPGA with a large section remaining for other components. The array employs pipelined column-parallel I/O operations. Data is input from the west and is output to the east. The result of this is that individual PEs cannot be accessed randomly and to read data, the entire array must be shifted out. This saves valuable resources in the aim of achieving a minimal size processor array. The north, south and east inputs are connected to some dummy elements which can be either set to ‘0’ or ‘1’ to control boundary effects.

F. *Peripheral Components*

As well as the processor array, there are additional components that have to be placed on the FPGA. Buffers are needed to store the image data to be input into the array and the image data to be output from the array. Memory is also required to store the instructions to be executed by the processor array. A controller is required to issue instructions to the SIMD array and control program flow. These buffers, memory and control circuits all form one complete component named Instruction Processing Unit (IPU). It executes a sequence of instructions which perform operations including reading the image into the input buffer, shifting the image into the array, processing the image, shifting the image out of the array, and writing the image from the output buffer. There is also a Digital Clock Manager (DCM) which is used to supply a defined clock speed to the array and IPU.

III. FPGA IMPLEMENTATION

The 32 × 32 processor array is synthesised and implemented on a XC5VLX50-1 FPGA. When implemented, the entire array as shown in Fig. 4 occupies 7,168 out of 28,800 slice registers, 13,671 out of 28,800 slice LUTs, a total of 7,032 out of 7,200 available slices and 90 I/O pins. With this design and before the IPU and DCM are added to the FPGA, the minimum clock period is given as 4.31 ns, which gives a potential maximum operational frequency of the array alone of 232 MHz. The critical path for this speed is shown in Fig. 5. This is found to be inside the PE with the limiting factor being the delay from the output of the RAM block, through the ALU and back into the RAM block. When a larger array of 80 × 80 PEs is synthesised for a larger FPGA, the critical path remains the same indicating that it is not associated with the distribution of control signals across the

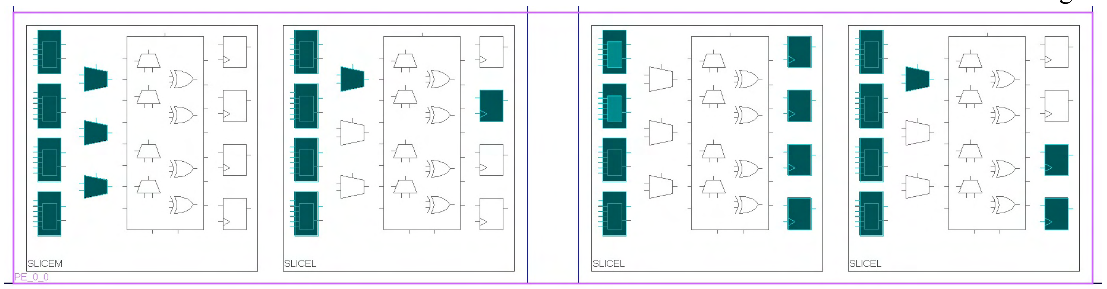

Figure 3. Floorplan of an individual processing element as implemented on a Virtex-5 FPGA. The PE is spread over two CLBs (four slices) and includes one SLICEM and three SLICELs. The components that are used are shaded.

array. As such, it is estimated that even larger arrays than this can be implemented on the FPGA before speed performance is impaired. The entire system is implemented on an Opal Kelly XEM5010 integration board [20] which allows communication between a computer and the board using USB. It is fully supported by Opal Kelly's FrontPanel programmer's interface which allows software to be interfaced to the XEM board. In this particular case, the array is programmed to the FPGA using the C++ programmer's interface which sends the bit file to the FPGA. This interface also sends data from a computer to the FPGA and reads data from the FPGA.

A schematic of the processor array with the IPU components is shown in Fig. 6. The input image is sent to IPU through the USB interface. Through a sequence of IPU instructions the IPU reads this data from the USB interface and sends it to input of the processor array. When the entire image data has been written to the array, the processing instructions are then executed. When the image processing is complete, the processed image data is then read out by the IPU block and is sent back to the computer through the USB interface.

The entire design, including the peripherals uses 7,149 slices (99%), 14,685 LUTs (50%) and 7,796 slice registers (27%) of the XC5VLX50 device. The floorplan of this is shown in Fig. 7. The clock speed in this case is now limited by the IPU rather than the processor array itself so a slower clock is used. To get the maximum operational frequency the DCM uses the 48 MHz external clock and supplies an operating frequency of 150 MHz (6.67 ns period) to the array. As each PE executes 150 MIPS (million instructions per second) the total array executes a peak of 153.6 GOPS (bit-serial operations). The peripheral components in the design only require an additional 1,014 LUTs and 628 registers which is approximately 2-4% of the entire FPGA. This FPGA is one of the smallest Virtex-5 FPGAs; a much larger array can be implemented on a larger device.

IV. Processor Operations

In this section, the processor's functionality is demonstrated using algorithms for binary image processing and 8-bit greyscale image processing. To implement these algorithms, APRON software [21] is used which is a dedicated

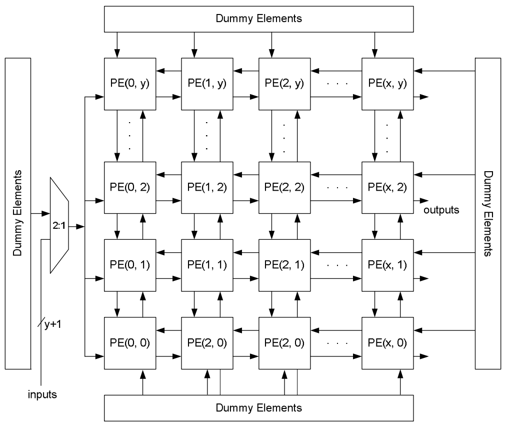

Figure 4. FPAC schematic.

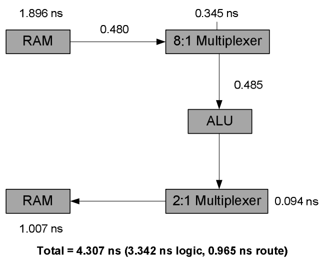

Figure 5. Critical path for processor array.

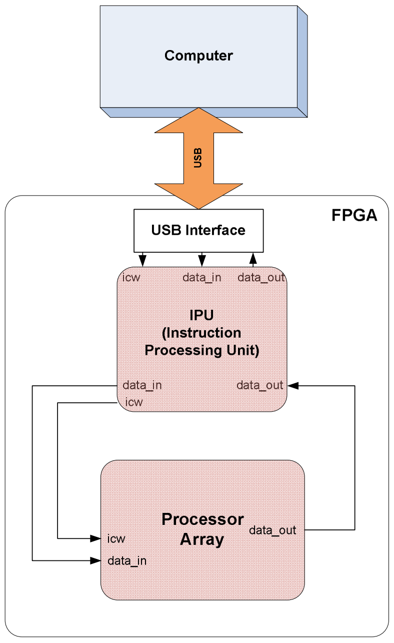

Figure 6. IPU and array schematic.

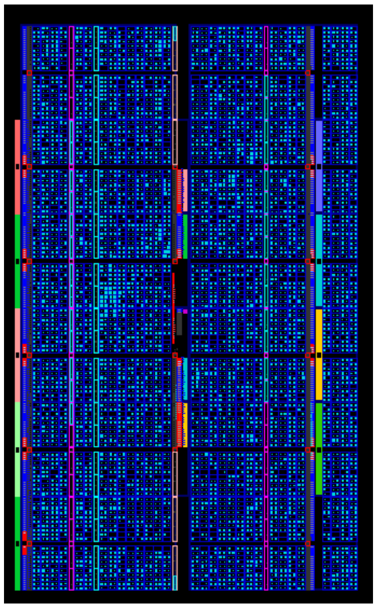

Figure 7. Floor plan of processor array and peripheral components. The bright blue areas are the occupied area on the FPGA with the larger blue dots being multiple occupied LUTs or registers and the smaller dots being individual LUTs or registers.

tool for cellular array processor code development, emulation and interfacing. APRON also captures and displays the images. Assembler instructions are created and are compiled by APRON in order to create the sequence of Instruction Code Words (ICWs). An example of instructions is shown in Fig. 8. These instructions will be used in one of the algorithms described.

The first example of image processing is binary image processing. For these images, black pixels are indicated by '0' and white pixels are indicated by '1'. This algorithm demonstrates erosion and edge-detection of an image. The erosion part of the algorithm detects whether there are any isolated pixels (white completely surrounded by black or black completely surrounded by white) and eliminates them by making them identical to their eight neighbours. The edge detection part detects any black pixels that have a white neighbour and recognises this pixel as an edge.

As the erosion algorithm requires the knowledge of its eight neighbours rather than just four, some additional operations are required. First all the neighbours are read and written to RAM. The four nearest neighbours can be read in four clock cycles because of the four neighbourhood connectivity whereas reading the four diagonal neighbours requires eight cycles. For the diagonal neighbours, to read the two diagonal pixels to the north, all the data has to be shifted south to the NEWS register, and then each processor will read the processor to the west, then east and store both of these values in the RAM before rewriting the original NEWS register value back into its corresponding register. This takes a total of four clock cycles and similarly a further four cycles are required to read the data in the two diagonal pixels to the south. The erosion and dilation part of the algorithm is completed in 34 instructions with each instruction being executed sequentially with each clock cycle.

The edge detection algorithm then begins by reading the four nearest neighbours again and storing their values in RAM. Then, the black pixels are activated by setting their FLAG registers high. By using the OR function of the four neighbours, a white pixel can be detected and a black pixel can be written to the NEWS register indicating an edge. This takes a total of 10 clock cycles. The total processing time of this particular algorithm is 44 clock cycles which, at a clock speed of 150 MHz, is 293.33 ns. The result of this process is shown in Fig. 9. The binary edge detection is clear from Fig. 9(c). The effect of the erosion algorithm is also noticeable: as can be seen from the thresholded image in Fig. 9(b), there is an isolated pixel on its own which can no longer be seen in the processed image in Fig. 9(c).

This complete algorithm was used to show some of the features of the processor but a simple edge detection algorithm on its own could be implemented in as few as 5 instructions without reading the neighbours and storing in RAM first. At 150 MHz operation, the total processing time for this basic binary edge detection algorithm is 33.33 ns.

The second algorithm is 8-bit greyscale Sobel edge detection which can be seen in Fig. 10. As the storage of 8-bit greyscale images requires eight times more data than the binary images, the entire image cannot be stored in the PE

```
//activate black pixel
LDAINV(FLAG, RAM255)

//read neighbours and AND together to detect edge
LDA(NORTH)
OR(EAST)
OR(WEST)
ORINV(NEWS RAM255, SOUTH)
```

Figure 8. APRON assembler code.

registers so the RAM must also be used to store the image. The Sobel operation calculates edges in the image through a summation of absolute value images obtained from convolving the original image with two 3×3 kernels:

$$G_1 = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}, G_2 = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} \quad (1)$$

The entire processing time for the Sobel edge detection operator requires 416 instructions which, at 150 MHz, gives a total processing time of 2.77 µs.

For performance comparisons, these figures are compared to similar results from different designs in literature. The binary edge detection is achieved in 5 clock cycles, 33.33 ns at 150 MHz. The design in [16] can achieve the same result in 6 clock cycles, 89.15 ns running at 67.3 MHz on an array of 48 × 48 PEs. Despite our architecture being a smaller array, if the design was increased to 48 × 48 PEs, the processing time would remain the same as the size of the array does not affect the processing time of the image due to all processing being performed in parallel. Thus, the same algorithm could be implemented in 5 clock cycles on a larger array.

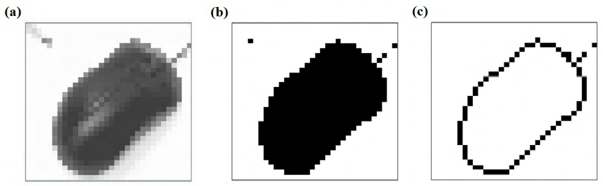

Figure 9. Binary edge detection; (a) the image from the camera; (b) the thresholded binary image that is uploaded to the FPGA; and (c) the processed image downloaded from the FPGA.

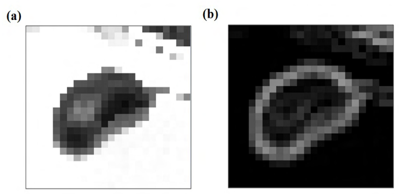

Figure 10. Sobel edge detection; (a) the image from the camera that is uploaded to the FPGA; and (b) the processed image downloaded from the FPGA.

The Sobel edge detection is achieved in 416 clock cycles, 2.77 μs at 150 MHz. A similar process is performed on a 32 × 32 array of SIMD PEs in [22]. This design uses Sobel edge detection as part of a larger algorithm and is also implemented on an FPGA. With a maximum frequency of 100 MHz, the edge detection is performed in 52.2 μs. If the clock speed is increased to 150 MHz, the performance of [22] is still approximately 12 times slower than the proposed design. In [9], Sobel edge detection is performed on a SIMD cellular processor array implemented on a custom ASIC chip. The array size in this case is 19 × 22. Again, array size does not affect performance times as processing is done in parallel. The time reported in [9] is 5.59 μs using a 75 MHz clock. In comparison to a DSP processor, the TMS320C64x range of processors [23] require 1,343 clock cycles to execute a Sobel edge detection on a 32 × 32 image, over three times more than the 416 required in this paper (although the TMS320C64x range of devices are capable of running at higher clock speeds).

V. Conclusion

A compact processing element is designed so that it only occupies two CLBs of a Xilinx Virtex-5 FPGA and can perform basic Boolean and bit-wise binary arithmetic operations. A 32 × 32 processor array implemented on the FPGA with each processor connected to its four nearest neighbours, operates at a frequency of 150 MHz and has a peak execution of 153.6 GOPS (bit-serial operations). The presented architecture performs binary image processing and 8-bit greyscale image processing effectively and should be capable of doing the same on much larger arrays. A basic binary edge detection algorithm can be completed in 33.33 ns. Larger FPGAs will allow more resources to be implemented showing how the design can be easily reconfigured for a much larger array without having to go through an entire new design process. Using the proposed design, it is estimated that an array of 276 × 276 PEs could be implemented on the largest device (XC7V2000T) in the Virtex-7 FPGA family [24].

The performance of the proposed design is comparable or better to similar FPGA architectures in literature and also shows processing times similar to that demonstrated by some ASIC processor arrays. With a low development cost, low cost of migration to future devices, and a good performance, FPGAs are suited to the design of cellular array processors for pixel-parallel processing. With larger FPGAs being available in the future, it seems promising that arrays comparable to state of the art custom designed ICs can be implemented on these devices.

References

[1] M. J. B. Duff, “Review of the CLIP image processing system,” in *Proc. of the National Computer Conference*, Anaheim, CA, 1978, pp. 1055-1060.

[2] S. F. Reddaway, “DAP- A distributed array processor,” in *Proc. 1st Annual Symp. on Computer Architecture (ISCA ‘73)*, 1973, pp. 61-65.

[3] K. E. Batcher, "Design of a massively parallel processor," *Proc. IEEE Trans. Comput.*, vol. C-29, no. 9, pp. 836-840, Sept. 1980.

[4] E. Cloud, "The geometric arithmetic parallel processor," in *Proc. IEEE 2nd Symp. Frontiers of Massively Parallel Computation*, Orlando, FL, 1988, pp. 373-381.

[5] E. S. Gayles, T. Kellihr, R. M. Owens and M. J. Irwin, “The design of the MGAP-2: a micro-grained massively parallel array,” *Proc. IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 8, no. 6, pp. 709-716 Dec, 2000.

[6] S. Kyo and S. Okazaki, “IMAPCAR: A 100 GOPS in-vehicle vision processor based on 128 ring connected four-way VLIW processing elements,” *Journal of Signal Processing Systems*, vol. 62, no. 1, pp 5-16, Jan, 2011.

[7] A. Abbo et al., “Xetal-II: a 107 GOPS, 600 mW massively parallel processor for video scene analysis,” *Proc IEEE J. Solid-State Circuits*, vol. 43, no. 1, pp. 192-201, Jan., 2008.

[8] P. Dudek, and S. Carey, “General-purpose 128×128 SIMD processor array with integrated image sensor,” *Electronics Letters*, vol. 42, no. 12, pp. 678-679, June, 2006.

[9] A. Lopich and P. Dudek, “A SIMD cellular processor array vision chip with asynchronous processing capabilities,” *Proc. IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 58, no. 10, pp. 2420-2431, Oct., 2011.

[10] T. Komuro, S. Kagami and M. Ishikawa, “A dynamically reconfigurable SIMD processor for a vision chip,” *Proc. IEEE J. Solid-State*, vol. 39, no. 1, pp. 265-268, Jan, 2004.

[11] J. Poikonen, M. Laiho and A. Paasio, “MIPA4k: A 64× 64 cell mixed-mode image processor array,” in *Proc. of the IEEE Int. Symp on Circuits and Systems*, Taipei, Taiwan, 2009, pp. 1927-1930.

[12] A. Rodríguez-Vázquez et al., “CMOS architectures and circuits for high-speed decision-making from image flows,” *in Proc. of the Society of Photographic Instrumentation Engineers (SPIE) 6940*, Orlando, FL, 2008, 69402F, pp. 1-10.

[13] D. L. Andrews, A. Wheeler, B. Wealand and C. Kancler, “Rapid prototype of an SIMD processor array (using FPGA's),” in *Proc. of the Fifth International Workshop on Rapid System Prototyping*, Grenoble, France, 1994, pp. 28-33.

[14] I. Vassanyi, “Implementing processor arrays on FPGAs,” in *Field-Programmable Logic and Applications From FPGAs to Computing Paradigm*, R. Hartenstein and A. Keevallik, Eds., Berlin/Heidelberg, Germany: Springer, 1998, pp. 446-450.

[15] S. Y. C. Li, G. C. K. Cheuk, K. Lee and P. H. W. Leong, “FPGA-based SIMD processor,” in *11th Annual IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*, Napa, CA, 2003, pp. 267-268.

[16] A. Nieto, V. Brea and D. Vilarino, “SIMD array on FPGA for B/W image processing,” in *Cellular Neural Networks and Their Applications (CNNA)*, Santiago de Compostela, Spain, 2008, pp. 202-207.

[17] T. Saegusa, T. Maruyama and Y. Yamaguchi, “How fast is an FPGA in image processing?,” in *Int. Conf. on Field Programmable Logic and Applications (FPL)*, Heidelberg, Germany, 2008, pp. 77-82.

[18] P. Bonnot et al., “Definition and SIMD implementation of a multi-processing architecture approach on FPGA,” in *Design Automation and Test in Europe (DATE ’08)*, Munich, Germany, 2008, pp. 610-615.

[19] *Virtex-5 FPGA User Guide UG190*, v5.3, Xilinx Inc., San Jose, CA, 2010.

[20] Opal Kelly Inc., “XEM 5010,” *XEM 5010 FPGA USB 2.0 module with Virtex-5 | Opal Kelly*. Opal Kelly Inc.. [Online]. Available: http://www.opalkelly.com/products/xem5010. [Accessed: Apr. 27, 2012].

[21] D. R. W. Barr and P. Dudek, “APRON: A cellular processor array simulation and hardware design tool,” *EURASIP Journal on Advances in Signal Processing*, vol. 2009, pp. 1-9, Jan. 2009.

[22] Y. J. Li, W. C. Zhang and N. J. Wu, “A novel architecture of vision chip for fast traffic lane detection and FPGA implementation,” in *Proc. of IEEE 8th Int. Conf. on ASIC (ASICON ’09)*, Changsa, China, 2009, pp. 917-920.

[23] *TMS320C64x Image/Video Processing Library Programmer’s Reference SPRU023B*, Texas Instruments Inc., Dallas, TX, 2003.

[24] *7 Series FPGAs Overview DS180*, v1.7, Xilinx Inc., San Jose, CA, 2011.