# Hardware Edge Detection using an Altera Stratix Nios2 Development Kit

Mahmoodi A, Kraut J, Jahns M, Mahmood S

*Abstract*— **Edge detection is a computer vision algorithm that is very processor intensive. It is possible to increase the speed of the algorithm by using hardware parallelism. This paper presents an implementation of edge detection in an FPGA, the Altera nios2 development kit. The paper focuses on providing the often missing link from the algorithm development to the FPGA implementation. In addition to a discussion of the edge detection algorithm, memory access and data transfer to the FPGA is discussed. Memory access is achieved by developing a generic component that handles the memory transfer on the Avalon Bus. The design is open so other memory intensive algorithms can be used with only a slight modification of components. In addition the testing software and firmware development is described. The results show how a highly parallel algorithm can run faster on a 50 MHz FPGA then a modern PC in the GHz.**

*Index Terms*—**Computer Vision, FPGA, Altera Stratix, Edge Detection.**

## I. Introduction

One of the algorithms used for computer vision is edge detection. It can be the first stage in processing a bitmap into a representation of an area that can be used for autonomous decisions. The edge detection is a highly parallelizable algorithm that can be sped up dramatically with a custom circuit, in particular a FPGA. An FPGA running at 50 Mhz can perform faster then a PC running in the gigahertz.

In order to fully make use of an edge detection circuit a method to transfer the bitmap data to the FPGA, have it stored, processed and saved then have it returned to a pc is necessary. Several interfaces exist to a FPGA but among the fastest is the Ethernet interface. This led to the choice of using a development kit with a Stratix FPGA with a nios2 embedded processor [1][2][3]. In additional to the hardware Ethernet module, it contains a software network stack and sample code to allow the development to focus on the edge detection circuitry.

To perform the edge detection a bitmap is captured from a web cam with a laptop using the DirectShow interface. It is then converted into 8 bit monochrome and sent to the FPGA. Since UDP is faster it is used so a custom packet format is used to seamlessly packetize the data and re-form the bitmap in the FPGA. Then c code on the nois2 processor acts as the firmware and instructs the edge detection circuit what operation to perform, the memory locations, etc. The firmware can be used to customize the algorithm used for edge detection. As soon as the edge detection is complete the nios2 sends the bitmap back to the laptop, which complete the edge detection cycle for a single frame. The received bitmap can be compared with a bitmap that has had the edge detection formed with software to verify that the circuit is operating correctly.

## II. Edge Detection

Edge detection consists of taking a 3 by 3 mask and performing a 2D-convolution on the image [6]. The mask is placed at location (0,0) and pixels under the mask are multiplied with corresponding value of mask and then added together. This would gives new pixel value for location (1,1). The resulting image is generated by sliding the mask over by one and so forth. The general equation to find the resulting image is shown below,

$$Im(x,y)=\sum \sum w(s,t) * f(x+s,y+t)$$ , where w is mask and f is image

Detecting edges for the boundary would result in black boundary therefore they were ignored. Hence this convolved image is transformed from n x m to n-1 x m-1 image.

| | | |
|---|---|---|
| 1 | -2 | 1 |
| 0 | 0 | 0 |
| 1 | -2 | 1 |

Horizontal Mask

| | | |
|---|---|---|
| 1 | 0 | 1 |
| -2 | 0 | -2 |
| 1 | 0 | 1 |

Vertical Mask

Fig. 1. Sobel Mask

Above shows Sobel mask which detects edges. The Sobel mask uses horizontal and vertical mask to determine horizontal and vertical edges and then added together.

$$Im_{final} = abs(Im_{hori}) + abs(Im_{Vert})$$

Since 8 bit black and white image is used for edge detection therefore $Im_{final}$ should also be 8 bit. In order to maintain this criterion any values which are greater 255 are truncated to 255 and if any values are less than zero they are set to 0.

## III. Nios Avalon Interface

Peripherals that interface directly to the nios processor use the Avalon Bus. The Bus consists of masters and slaves. The masters initiate communication and the slave responds to

them. The Avalon bus is a slave side arbitration bus. If multiple masters are connected to a single slave each master gets its own dedicated lines connecting to the slave. The slave then controls the arbitration between multiple masters. The arbitration circuit is automatically generated by the SOPC program provided with in the development IDE. The only thing that had to be taken into consideration to use the bus is to properly generate and read signals, and to adhere to the bus timing.

The edge detector circuit requires an interface to the nios processor so the processor can control its operations. It also requires a direct interface to memory, which significantly speeds up the operation versus having the processor act as a interface between the edge detector and the memory. This leads to having one slave port to receive data from the nios processor and 2 master ports, one to read, the other to write to memory that allows it to use dual port.

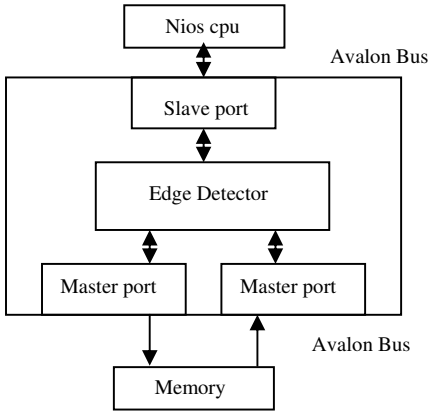


Fig 2 DMA Architecture

The slave interface is simple. When *read* is asserted, place the data request by the address on the read bus. When *write* is asserted take the data from the write bus and place it in a register at the location given by the address.

The master bus is slightly more complicated. Both reading and writing look similar as shown by figure 3 [4]. The transfer begins on the rising edge of the clock. Place the address, *byteenable read_n* on the bus. On the first rising edge when *waitrequest* is '0' then the data is available and can be read in. This requires a simple state machine. There is also the option to pipeline the reads and writes but this is not used for the project.

## IV. The Framework

To interface to the memory a framework is used that can be used for other projects. The framework consists of an input block to read the data and an output block to write to the memory. Between the input and output block is the edge detection unit that performs the arithmetic operations. The interface to the input block, that the edge detection unit sees is three, 8 bit data signals, a *dataout_valid,* and *dataout_request* control lines. When *dataout_request* is '1' and *dataout_valid* is '1' then data is clocked into the addition unit. The output block has a single 8 bit input line and a *datain_request* and *datain_valid* signal that works the same as the output block.

To reduce the memory transfers the edge detectors takes in consideration that for any output pixel other then the first of a row the six pixel values from the previous edge detection operation can be used. So other then the first pixel of the row only three bytes are required to generate an output pixel.

The 32 to 8 bit converter component are necessary because the Avalon bus supports 32 bit transfers so it makes sense to transfer in 32 bits and then shift it out 8 bit in a time. Since the Avalon bus requires at least two clocks for any transfers (without pipelining) using this scheme the slowest operation is the read. Using fast onboard memory a read is performed using 2 clocks. Since 3 words need to be loaded in for a single output it takes 6 clock cycles to read in the 3 words. Since each read generates 4 bytes the framework can calculate 6/4 = 1.5 pixels per clock cycle if the operation does not have anything that takes more then 1 clock cycle to calculate.

## V. Edge Detection Component

This section discusses the design of the 9-pixel input mask for the edge detection algorithm and the implementation of the arithmetic operations required to perform edge detection.

### A. 9-Pixel Input Mask

The input mask for the edge detection algorithm interfaces with the memory via the framework, using handshaking signals *data_input_valid* and *data_input_request*. Similarly, the input mask communicates with the first arithmetic module of the edge detector using handshaking signals *data_output_valid* and *data_output_request*.

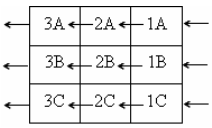


Figure 4: Data flow through 9-Pixel Mask

The 9-pixel mask, shown in figure 4, is moved from the left to right end of an image line as it performs its algorithm. When a new pixel column is received from memory, the contents of the mask are shifted one column to the left and the new pixel information is shifted into the far left column of the mask.

A state machine regulates the loading of this 9-pixel input mask to correspond with the size of the chosen bitmap. When the mask is to start operation on a new image line, three new pixel columns must be inputted before the mask data can be considered valid. At this point the input mask asserts *data_output_valid*, and waits for *data_output_request* to be

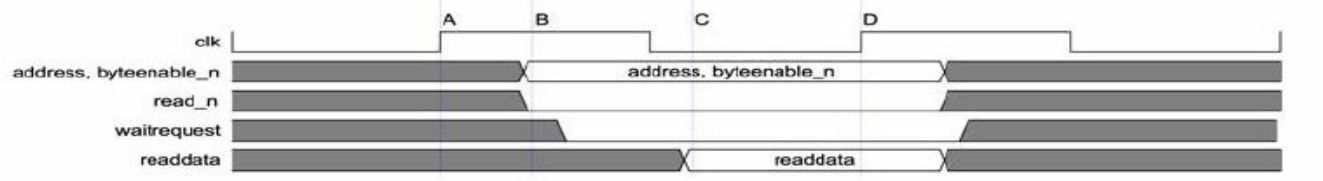


Fig 3 Avalon Bus read waveform from [4]

asserted by the multiplier modules. When this is satisfied, the input mask sends its valid information, deasserts *data_output_valid*, and requests one new pixel column of information from the memory framework. When the input mask receives the valid input, it is again full of valid pixel information, and *data_output_valid* is asserted. This process continues until the counters present in the FSM indicate that the mask has received the last pixel column of an image line. At this point, the mask must once again receive three new pixel columns from memory for the new image line, before its output can be considered valid. The state machine for the input mask proceeds in this fashion until it recognizes that it has received all the pixels in the image.

### B. *Arithmetic Implementation*

A top level diagram of the arithmetic implementation for the edge detection algorithm is shown in figure 5. Notice that the implementation of the arithmetic operations has been subdivided into several VHDL modules. By pipelining the arithmetic operations, faster clock speeds can be obtained. The trade off is that the larger the pipelining, the longer the latency to receive the output data. The flow of data between the arithmetic modules is regulated with the same handshaking signals used in the interface with the 9-pixel input mask, but a different state machine.

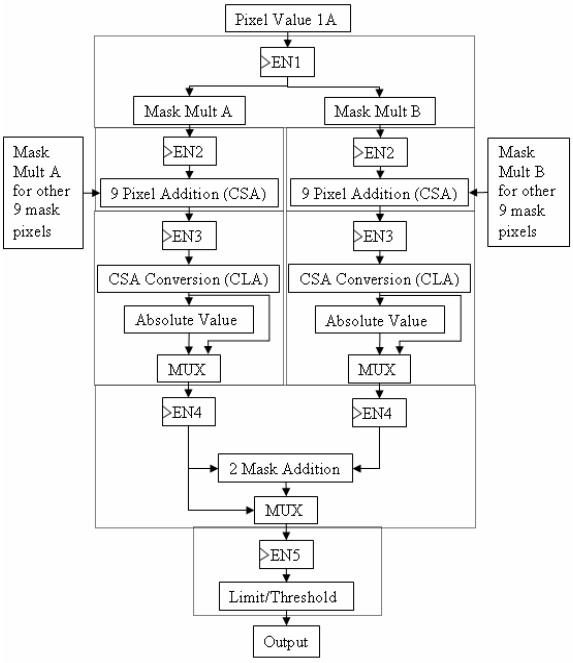


Figure 5: Top Level Arithmetic for Pixel Output

As an overview, each arithmetic module is in *wait_ state* when it is requesting new input data. When the valid data is received, it enters the *execution_state*, for the number of clock cycles required to perform the respective arithmetic operation. During this time, the module does not request new input data, nor is it considered to have valid output data. When the arithmetic operation has complete, the module enters the *complete state*, where it waits to output its valid information. When the module has sent its valid data to the next module, it once again requests new input information in the *wait_state.*

The implementation of each of the arithmetic modules shown in figure 5 will now be summarized. Since the circuit is to be customizable to use other masks, the required operations to output a single pixel are: 18 multiplications (2 masks), the addition of 9 numbers (for each mask), ensuring the numbers are positive (if desired), adding the results from the two results together (if desired) and thresholding the output pixel.

The multiplier module was implemented using the onboard DSP blocks provided on the FPGA. To ensure the validity negative multiplications, all multiplications were performed using unsigned arithmetic. If a negative multiplication was desired, this was indicated by a control signal, allowing for correction by taking the two's complement of the product.

9 adders were needed to add the outputs of each mask. Instead of using 9 adders, Carry Save Adders were used since CSA has lower complexity and lower delay. Each CSA converts 3 inputs to two carry and sum outputs. This could be done in 4 stages and using an adder to add the output of the last CSA. Because the size of the input and outputs of every stage are the same, to avoid overflow, before doing the addition, size of the inputs should be increased.

All other additions used in the algorithm were implemented using CLA modules.

As previously mentioned a pixel threshold was required. Simply stated, any values which are greater 255 are truncated to 255 and if any values are less than zero they are set to 0.

In order to troubleshoot the VHDL code before loading it to the FPGA, a testbench was created. The tesbench reads an input 100*100 bitmap file and will send 3 pixels in each stage for edge detection block using handshaking signals. The resulting one pixel output of the edge detection block for each computation is returned to the testbench. Received pixels will be written to a 98*98 output bitmap file. The testbench uses handshaking signals to transfer data to and from the edge detection block. Comparing input image and output image we can ensure that edge detection block works properly. Problems with the testbench were encountered in that the size of the output image could not be properly sized to 98x98. As a result, a 2 pixel skew per line was developed in the resulting image. However, the testbench still verified the functionality of the edge detection algorithm.

## VI. DIRECTSHOW, PACKETIZING, FIRMWARE AND TESTBENCH

This section covers any miscellaneous items required to test the circuitry. It consists of how a bitmap is obtained from a web camera, how the data is packetize, and the firmware required for the nios processor.

### A. *DirectShow Interface*

Every web cam that operates in windows uses a standard driver interface. This allows for software makers to not have to customize code for different manufactures. For media streaming the standard interface is DirectShow. In the interface, each component of it is called a filter. A web cam source filter is a single component as is a frame grabber filter. DirectShow specifies how each filter connects with each

other, and the data transfer method. Most of the connection work is automatically done with DirectShow with each filter just having to specify its possible formats.

To interface to DirectShow a frame grabber filter is created. The frame grabber is programmed to just accept 24 bit RGB 320x240 frames. Once it connects to the web cam filter, the web cam can start streaming data. For each frame the frame grabber receives an *IMediaSample* interface which contains the functions to obtain the frame data. The frame grabber contains no processing and just sends the *IMediaSample* to the main image processing application via a callback function. The main application contains the code to send the data to the laptop.

### B. *Packet Transfers*

The main application contains code to (un)/packetize the data to/from the FPGA, display the data and perform software edge detection to verify the FPGA edge detection is correct. It contains three display windows. One is the image directly from the web cam. The second is the software edge detection. The third is the hardware edge detection. It also contains various controls mainly to change the thresholding and scaling value, which the application is running to see its effects.

UDP is used to transmit to the FPGA because it is faster the using TCP. The since the FPGA and laptop are connected through a hub and Ethernet is inherently reliable and speed is more important then the extra reliability.

The data integrity is verified by having the data sent to the nois and then echoed using 76800 bytes, the size of s a single 320x240 8 bit bitmap. The speed of the operation is around .2 seconds, which means that the nois2 board is capable of around 5 bitmaps per second. There are some speeds up possible but the speed found of 76800*2*5 = .768 MB/S is very close to the maximum posted on the Altera product spec, so it is unlikely that more data can be sent.

### C. *Firmware*

The firmware in the nios processor is a modification from the Simple_socket_server program that comes with the ide [5]. The modifications are to alter the server to accept UDP data. The data is passed into the packetizer. When the packetizer returns with a completed packet a function is called to perform the edge detection. When the edge detection is complete the data is then sent back.

## VII. Results

The edge detection component described in this report was tested and proven successful in simulations. However undetermined problems caused it to fail in hardware implementation. Fortunately, two versions of the edge detection block were created simultaneously. Although both blocks used a similar implementation with an identical top level interface with the memory, the other block was successfully implemented in hardware.

The edge detection component described in this report was verified in simulations using a 100x100 pixel input bitmap. At the start of the image, 18 clock cycles are required to fill the pipeline before the first output pixel is generated. However once the pipeline is filled, a new output pixel is generated every 5 clock cycles. Since there are 9,604 output pixels for the edge detection of a 100x100 pixel image, approximately (9,604*5) + 18 = 48,038 clock cycles are required. For comparison with a 320x240 image used in the hardware implementation approximately (76,160*5) + 18 = 380,818 clock cycles are required per image. Timing analysis determined a maximum operating frequency of 82.91 MHz. In addition, the entire VHDL implementation utilized only 12% of the total logic elements available on the FPGA.

The alternate edge detector component used in hardware implementation was tested and proven to be working correctly. In total 114249 clock cycles are used. Since 320x240 reads times 6 waits per 3 reads, divided by 4 bytes per word is = 76160*6/4 = 114240. Taking into account that there are 10 pipeline stages before the last pixel is ready the +9 difference makes sense.

The Original goal of continuous real time edge detection is not achieved. The Ethernet connection is too slow to support 320x240 bitmaps at 30 times a second. Also there are several minor glitches that prevent the edge detection from being continuous, such as a heap error in MicroOS.

## VIII. Conclusion

This project is a good example of implementing an algorithm that is parallel and can be sped of with a FPGA. The Altera Stratix Nios 2 contained ready out of the box Ethernet which sped up development. The Avalon interface took a bit of time to implement but after that the edge detection circuit development was relative straight forward.

The edge detection component that was implemented in hardware operates at 114249 clock cycles for a 320x240 bitmap on a 50 MHz FPGA. This translates into 437 bitmaps per second which is quite a lot faster then unoptimized c code can perform the edge detection on a new x86 processor.

## References

[1] Altera, “Stratix Device Handbook” [Online document], 2005 Jul (ver3.3), [cited 2005 Dec 2], Available HTTP: www.altera.com/literature/hb/stx/stratix_handbook.pdf

[2] Altera, “FPGA Features, Package & I/O Matrix: Production Devices” [Online document], 2004 Nov (ver3.0), [cited 2005 Dec 2], Available HTTP: www.altera.com/literature/manual/mnl_nios2_board_stratix_1s40.pdf

[3] Altera, “Nios Development Board Reference Manual, Stratix Professional Edition” [Online document], 2005 Oct (ver 5.1), [cited 2005 Dec 2], Available HTTP: www.altera.com/literature/hb/nios2/n2cpu_nii5v1.pdf

[4] Altera, “Avalon Bus Specification Reference Manual” [Online document], 2005 May (ver3.1), [cited 2005 Dec 2], Available HTTP: www.altera.com/literature/manual/mnl_avalon_spec.pdf

[5] Altera,”Using Lightweight IP with the Nios II Processor Tutorial” [Online document], 2005 Jan (ver 1.1), [cited 2005 Dec 2], Available HTTP: http://www.altera.com/literature/lit-nio2.jsp

[6] R. Gonzalaz, R. Woods, “Digital Image Processing”. New Jersey: Prentice-Hall 2002, ch 10.