Edge Detection in a Color Image

Swarna Kamakshi Jayaraman
M.Sc., Embedded Systems
School of Computer Science and Engineering
Nanyang Technological University, Singapore
s160015@e.ntu.edu.sg

Deepshikha
M.Sc., Embedded Systems
School of Computer Science and Engineering
Nanyang Technological University, Singapore
deep0023@e.ntu.edu.sg

Sinduja Vijayakumar
M.Sc., Embedded Systems
School of Computer Science and Engineering
Nanyang Technological University, Singapore
sinduja004@e.ntu.edu.sg

Agatheswaran Rajashree Sundaram
M.Sc., Embedded Systems
School of Computer Science and Engineering
Nanyang Technological University, Singapore
rajashre001@e.ntu.edu.sg

***Abstract*— Heterogeneous architectures promise an improved processor system design leveraging the benefits of high-speed programmable computing elements that seamlessly work in unison [1]. Programmable systems on chip integrate the software programmability of a processor combined with hardware programmability of FPGA, offering unparalleled flexibility, scalability, performance and improved power economics [2]. While there may be a myriad of applications that can be accelerated with the help of hardware, the application we intend to accelerate is the edge detection of a color image. The intent is to establish communication between the Processing System, consisting of 2 ARM Cortex-A9 Processor cores and Programmable Logic of the Zynq Zedboard 7000 All-programmable SoC. Offloading the compute-intensive convolution part to the Programmable Logic (FPGA) guarantees improvement in various performance metrics.**

Keywords—Sobel; PSoC; Image Processing; Hardware software codesign; Zynq Zedboard;

I. INTRODUCTION

Edge detection involves identifying variations in brightness levels in an image, to find the boundaries of objects in an image. It is the preliminary step for various image processing techniques such as feature detection, image segmentation and computer vision. Computationally, edge detection is performed by convolving the image with an appropriate kernel which would identify the gradient or difference in brightness levels. Canny edge detector, Sobel operator and Prewitt operator are popular convolution kernels used for edge detection. The choice of convolution matrix depends on design requirements such as speed, accuracy and availability of resources. The Sobel operator is relatively less sensitive to noise or sudden spike in image gradient, and hence is useful for performing edge detection with respect to orientation. The Sobel operator consists of two matrix kernels, which must be convolved with the image matrix to identify edges.

The following section describes our implementation of edge detection on a color image using Sobel Operator.

```
Gx = {1, 0, -1,       Gy={ 1, 2, 1,
      2, 0, -2,            0, 0, 0,
      1, 0, -1}           -1, -2, -1};
```

II. IMPLEMENTATION

Hardware-software co-design of the Edge detection algorithm involves working with three tools, namely Vivado High Level Synthesis (HLS), Vivado Design Suite and Xilinx SDK. Vivado HLS allows description of the design in algorithmic language (C, SystemC and C++) and generation of digital hardware design in RTL form, based on given area and time constraints. The custom IP Core created in HLS post implementation, is imported to Vivado and necessary interfaces for communication are defined by a detailed block design. Bitstreams are generated for this newly created design and PS software code is written appropriately to establish communication with PL.

A. IP Core creation in Vivado HLS

The code to be executed in the Programmable Logic is written in Vivado HLS, tested with a test bench, synthesized and implemented as an IP Core. The steps followed in achieving edge detection in IP Core are:

i. Extraction of RGB Components of an image using OpenCV routines.
ii. Streaming in a 1-D array of RGB values to the top-function "edge_detect".
iii. Conversion of RGB Values to grayscale by forming a weighted sum of R, G and B components [5].
iv. Performing edge detection, by convolving the grayscale output with Sobel operator kernels Gx and Gy.

The input stream of the edge_detection function is a 32-bit value, packed with data in the following format:

Bits [31:24] – No data

Bits [23:16] – Blue component of i-th pixel
Bits [15:8] – Green component of i-th pixel
Bits [7:0] – Red component of i-th pixel
Where i = 0 to 16383 in a 128x128 image

The pixels are correctly interpreted in the edge detection module and RGB to Grayscale conversion is done using the below formula for each RGB set in the input:

Gray-scale pixel = 0.2989 * R + 0.5870 * G + 0.1140 * B

This is followed by invocation of the 'edge_detect' function which involves convolution. The first convolution operation is performed between the grayscale output array and the Gx kernel of the Sobel operator and the intermediate result is stored in a temporary array. The second convolution uses the intermediate result from the previous step and convolves it with the Gy kernel of the Sobel operator. Convolution is performed by using the 1-dimensional equivalent of the Gx and Gy matrices. The center of the convolution kernels is identified as half the length of the 1-dimensional version of the kernel. The first matrix's element is multiplied with the element of the flipped version of the second matrix. The product is cumulatively added to obtain the final convolution output of each iteration. The optimization techniques used in our implementation rely on the effects of loop unroll and pipeline of the iterations in the convolution function.

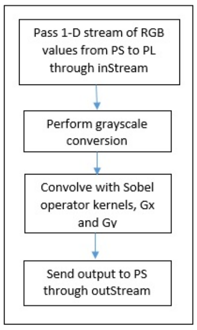

Pass 1-D stream of RGB values from PS to PL through inStream
Perform grayscale conversion
Convolve with Sobel operator kernels, Gx and Gy
Send output to PS through outStream

Fig. 1. Steps involved in edge detection

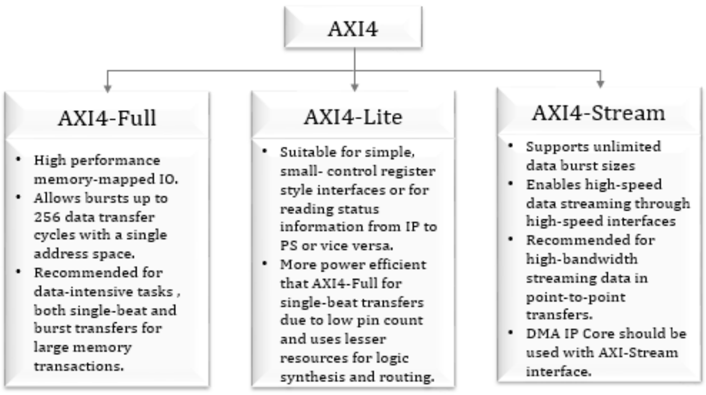
B. Block Design in Vivado Design Suite

Xilinx Vivado Design Suite allows users to do a bottom-up creation of design flow with in-built IP Integrator. It allows custom IP blocks to be integrated with the hardware blocks in the SoC.

Firstly, an instance of Zynq Processing System is added to the block design. The high-performance AXI ports (HP0) of PS are enabled to allow high performance streaming data transfers between PL IP and PS memory [3].

Data movement in PSoC from one functional block to another on-chip is accomplished using standard peripherals and interfaces. The key design challenge is choosing the right data transfer mechanism to connect the peripheral to memory subsystem. While small data transfers can be achieved using software instructions, huge chunks of data require a more sophisticated and efficient data transfer resource [3]. DMA module has been chosen for transfer of data between PS and PL, as it operates independent of CPU. The CPU just needs to set up some parameters for data transfer such as keep, strobe, user, destination, id, last (to indicate the last value to be transferred), etc., and can perform other operations during data transfer or go to low-power mode. This assures that the CPU needn't be active for the whole duration of data transfer and hence guarantees a more power-aware design.

The next crucial factor in deciding the performance of the system is the bit-widths used for data transfer. DMA supports 8, 16 and 32 bits for transfer. The maximum bandwidth of 32 bits is chosen in this design and in one shot, the three components (Red, Blue, Green pixel values) each of 8-bits length are sent via the AXI interface. Thus, the frequency of fetching data and writing back is reduced. AXI4 protocol supports three different interfaces as illustrated in Figure [3].



AXI4-Full
- High performance memory-mapped IO.
- Allows bursts up to 256 data transfer cycles with a single address space.
- Recommended for data-intensive tasks , both single-beat and burst transfers for large memory transactions.

AXI4-Lite
- Suitable for simple, small- control register style interfaces or for reading status information from IP to PS or vice versa.
- More power efficient that AXI4-Full for single-beat transfers due to low pin count and uses lesser resources for logic synthesis and routing.

AXI4-Stream
- Supports unlimited data burst sizes
- Enables high-speed data streaming through high-speed interfaces
- Recommended for high-bandwidth streaming data in point-to-point transfers.
- DMA IP Core should be used with AXI-Stream interface.

Fig. 2. Comparison of Interfaces supported by AXI4 Protocol

We observe from the features that AXI-Stream best suits our application which mandates high-speed data transfer via DMA. The input stream of the custom IP Core (edge_detect) is connected to AXI Memory-mapped-to-stream-Master (S_AXI_MM2S) and the output stream of the IP is connected to AXI Memory-mapped Slave interface (S_AXI_S2MM). The Scatter-Gather option of DMA IP Core has been disabled for our application. This optional mode enables higher-performance DMA Transfers at the cost of additional FPGA resource consumption [3].

An AXI Timer core is also included and configured to operate in 64-bit mode. This timer's helper APIs can be used to profile the execution time in the Processing system.

After creation of Block design, the design is verified and HDL wrapper is added. The bitstream is generated for the design and SDK is launched.

C. SDK Software Implementation

A new Application Project is created in SDK using the ".bit" generated by Vivado. Zynq Zedboard is chosen as the target device to include the necessary Board Support Packages in the design. The software code to be performed in the ARM Core is written in the SDK. The necessary headers such as xaxidma.h and xedge_detect.h, auto-generated by Vivado design are included in the source file. The memory addresses for transmission (MEM_BASE_ADDR + 0X00100000) and reception (MEM_BASE_ADDR + 0X00300000) buffers are pre-defined. The RGB Pixels of the input image are extracted via OpenCV routines in the Vivado HLS Testbench and stored in a file "image_pixels.h". This file is used as input to the edge_detect() function which shall be invoked from Processing System to Programmable Logic.

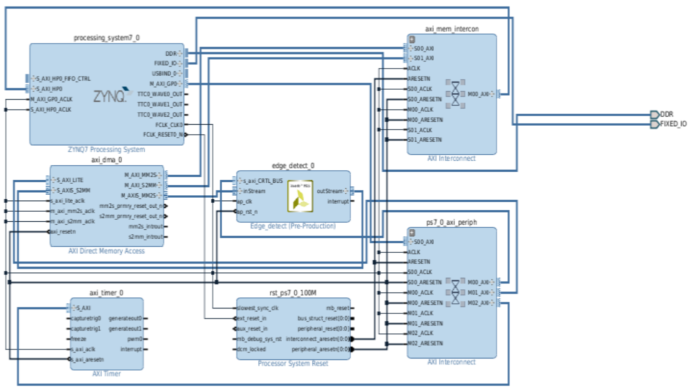


Fig. 3. Implemented Block Design in Vivado

Steps Involved [4]:

- Configure DMA module with suitable device ID (XPAR_AXI_DMA_0_DEVICE_ID, auto-generated and defined in the design's BSP).
- Instantiate edge detection module and load configurations for the same.
- Extract the RGB pixel values from "image_pixels.h" into an array, that shall be passed as input to the IP core.
- Start Edge detection by calling the API "XEdge_detect_Start()" autogenerated from Vivado.
- Perform data transfers from PS to PL and back to PS.

Once the output is written to memory, the following commands can be keyed into Xilinx Microprocessor Debugger (XMD Console) to verify the output:

```
set logfile [open "/home/jsk_027/PSOC_ED/EdgeDetection/imgDump.txt" "w"]
puts $logfile [mrd 0x01300000 16384 b]
close $logfile
```

XMD Console is deprecated and the latest versions of Vivado use Xilinx Software Command-line Tool (XSCT) instead. The equivalent commands in XSCT to dump output pixel values to a logfile are as follows:

```
set logfile [open "/home/jsk_027/PSOC_ED/EdgeDetection/imgDump.txt" "w"]
puts $logfile [mrd -file $logfile 0x01300000 16384]
close $logfile
```

- The first command opens the logfile specified by the path in write mode.
- The second command signals to read 16384 values from memory address 0x1300000, which is the starting address of RX_Buffer specified in PS, and dump the values to the logfile.
- Then the logfile is closed.

The logfile contents are the gray-scale edge-detected pixel values, which are used to plot the image in MATLAB.

Image Verification: A C script is written to convert 1-D edge-detected output to 2-D, a form suitable for reconstruction of the output image in MATLAB. Image is plotted using MATLAB functions: imshow() and image()

III. Results

A. Test Bench Results in Vivado HLS

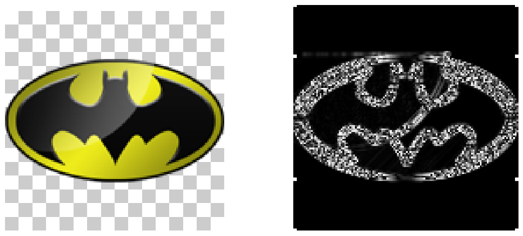


Fig. 4. 128 x 128 Input and Output Images

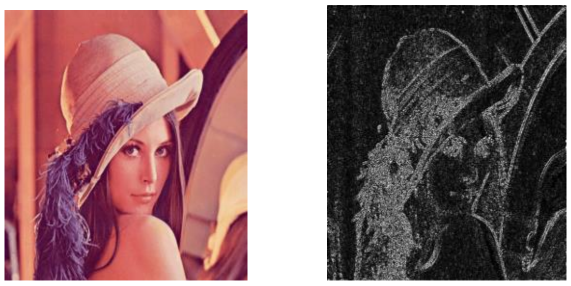


Fig. 5. 512 x 512 Input and Output Images

Although the edge-detection algorithm works well for any image size in Simulation, the design is not synthesizable for huge image sizes such as 512 x 512, as the hardware chosen does not have enough LUTs to fit the design. So, an optimal size of 128 x 128 was chosen for IP Core design.

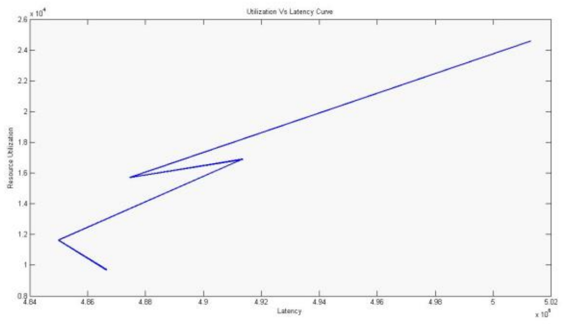


Fig. 6. Area vs. Latency graph

Intuitively, with increase in area, latency is expected to decrease. However, we see an increasing trend in this graph. The reason for this could be that there are not enough LUTs for implementing the design. Hence, unroll directive (which requires more resources) has no effect, and results in increase in execution cycles.

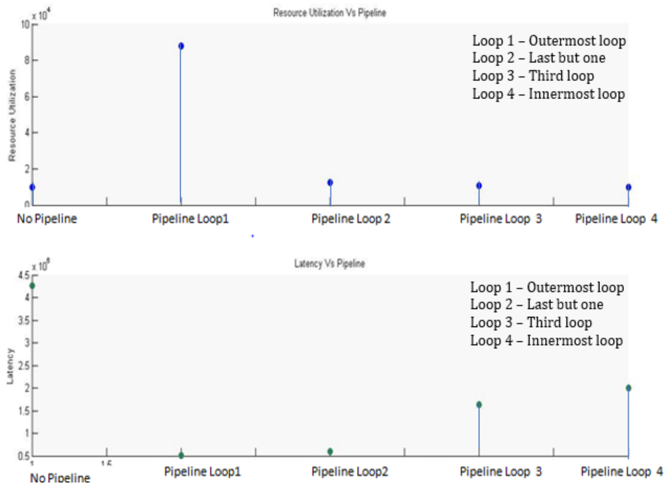


Fig. 7. Latency and Resource Utilization trends with Pipelining

Figure [7] shows that of the 4 for loops, pipeline is the most efficient in the outermost loop of convolution in terms of Resource utilization and Latency (measured in cycles).

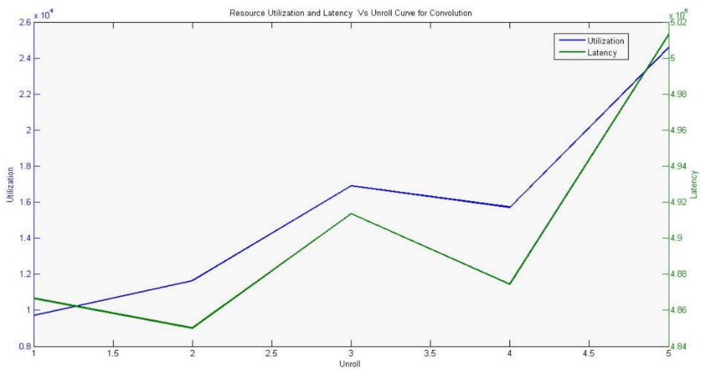


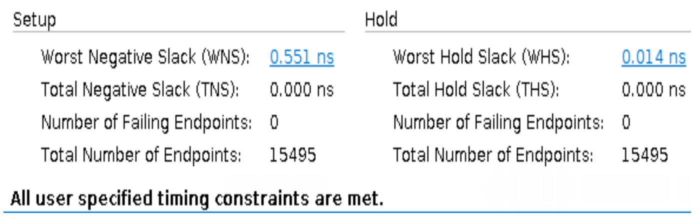
Fig. 8. Latency and Resource Utilization trends with Unroll directive

Figure [8] shows that resource utilization increases when increasing the unroll factor. An increase in Latency is also observed, owing to lack of enough resources to enable unroll directive. If enough resources are available in the hardware, unroll ideally results in decrease in latency or execution time.

B. Vivado Results

i. Timing Summary

Setup time is the minimum amount of time before the clock's active edge for which the input has to be stable for it to be properly latched or reflected at the output. **Hold time** is the minimum amount of time after the clock's active edge for which the input has to be stable for it to be properly latched [6].



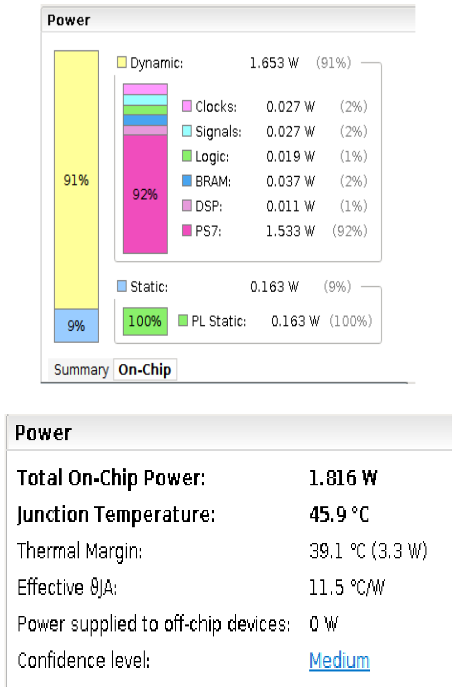
Setup		Hold	
Worst Negative Slack (WNS):	0.551 ns	Worst Hold Slack (WHS):	0.014 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	15495	Total Number of Endpoints:	15495

All user specified timing constraints are met.

Fig. 9. Vivado Timing Summary

We can observe that both setup and hold time have a positive slack. Hence, timing constraint is met and the clock period need not be altered in the resultant design.

ii. Power Estimates



Power

Dynamic:		1.653 W	(91%)
	Clocks:	0.027 W	(2%)
	Signals:	0.027 W	(2%)
	Logic:	0.019 W	(1%)
	BRAM:	0.037 W	(2%)
	DSP:	0.011 W	(1%)
	PS7:	1.533 W	(92%)
Static:		0.163 W	(9%)
	PL Static:	0.163 W	(100%)

Power

Total On-Chip Power:	**1.816 W**
Junction Temperature:	**45.9 °C**
Thermal Margin:	39.1 °C (3.3 W)
Effective ϑJA:	11.5 °C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

Fig. 10. Vivado Power Summary

The dynamic power consumption of an FPGA is decided by the operating clock frequency (f), node capacitance (C), FPGA operating voltage (V), and the activity (α) on various nodes in the design [7]. The dynamic power consumption of the implemented design is observed to be about 91%. Some optimization techniques in Vivado involving the concept of clock gating can help reduce dynamic power consumption results to less than 30%.

With increasing junction temperature, leakage current also exponentially increases, thereby causing high static power. Junction temperature is usually estimated based on board selection, heat sink and ambient temperature.

iii. Utilization Estimates

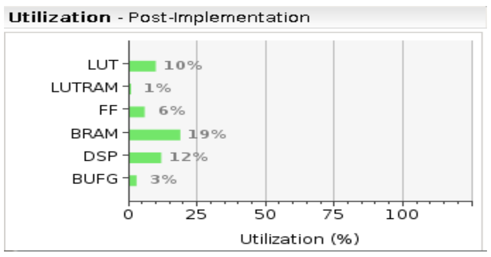


Fig. 11. Vivado Utilization Report

C. SDK Results Verified using MATLAB

The output grayscale pixels read to the logfile from memory are fed to MATLAB. The resultant image is verified to be the edge-detected output of input image.

The test bench also includes software verification, wherein RGB pixels are converted to grayscale using OpenCV and convolved with Sobel kernel in a hand-written convolution function. The image output is compared with hardware output and verified to be the same.

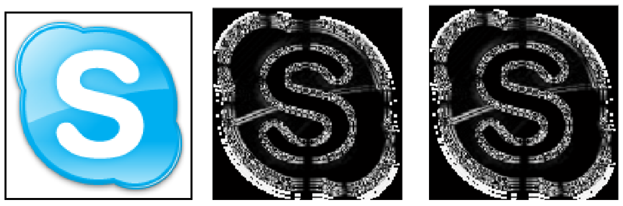


Fig. 12. a. Input Image b. Edge-detected output image from IP Core c. Software output for edge detection using Sobel kernel

D. Abbreviations and Acronyms

PS	Processing System
PL	Programmable Logic
PSoC	Programmable System On-Chip
AXI	Advanced Extensible Interface
MM2S	Memory-mapped-to-stream-Master
S2MM	Memory-mapped Slave interface
IP	Intellectual Property
RGB	Red, Green, Blue
RTL	Register Transfer Level
HLS	High-Level Synthesis
FPGA	Field Programmable Gate Array
CPU	Central Processing Unit
SDK	Software Development Kit
DMA	Direct Memory Access
LUT	Look-Up Table

ACKNOWLEDGMENT

We extend our sincere gratitude to our professor, Dr. Sharad Sinha, for his constant guidance and feedback during the course of this project, without which we couldn’t have completed the project on time. The contents that were taught during the lectures and lab sessions of the course “Programmable Systems on Chip”, coupled with the numerous reference materials shared by the professor gave us a lot of insight and helped us understand the rudiments of the subject with great ease.

REFERENCES

[1] http://developer.amd.com/resources/heterogeneous-computing/what-is-heterogeneous-system-architecture-hsa/

[2] https://www.xilinx.com/products/silicon-devices/soc.html, PSoC Lecture 2 Slides – Introduction

[3] https://www.xilinx.com/support/documentation/white_papers/wp459-data-mover-IP-zynq.pdf

[4] https://forums.xilinx.com/xlnx/attachments/xlnx/EMBEDDED/17960/1/sdk_main.cc

[5] https://es.mathworks.com/help/matlab/ref/rgb2gray.html

[6] http://www.edn.com/design/analog/4371393/Understanding-the-basics-of-setup-and-hold-time

[7] https://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_3/ug907-vivado-power-analysis-optimization.pdf