International Journal of Technology 13(5) 1045-1054 (2022)
Received April 2022 / Revised August 2022 / Accepted August 2022

International Journal of Technology

http://ijtech.eng.ui.ac.id

Video Chunk Processor: Low-Latency Parallel Processing of 3×3-pixel Image Kernels

Daniel Cheok Kiang Kho[1], Mohammad Faizal Ahmad Fauzi[1], Sin Liang Lim[1*]

[1]*Faculty of Engineering, Multimedia University, Persiaran Multimedia, 63100 Cyberjaya, Selangor, Malaysia*

Abstract. Video framebuffers are usually used in video processing systems to store an entire frame of video data required for processing. These framebuffers make extensive use of random access memory (RAM) technologies and interfaces that use them. Recent trends in the high-speed video discuss the use of higher speed memory interfaces such as DDR4 (double-datarate 4) and HBM (high-bandwidth memory) interfaces. To meet the demand for higher image resolutions and frame rates, larger and faster framebuffer memories are required. While it is not feasible for software to read and process parts of an image quickly and efficiently enough due to the high speed of the incoming video, a hardware-based video processing solution poses no such limitation. Existing discussions involve the use of framebuffers even in hardware-based implementations, which greatly reduces the speed and efficiency of such implementations. This paper introduces hardware techniques to read and process kernels without the need to store the entire image frame. This reduces the memory requirements significantly without losing the quality of the processed images.

Keywords: Framebuffer; Image processing; Kernel processing; Parallel processing; Video processing

1. Introduction

Video processing solutions (Park et al., 2020) typically require incoming video streams to be captured and stored first, and then to be processed later. This technique of buffering incoming frames is still the mainstream technique used to process videos today. However, as the consumer demands for clearer and faster videos continue to rise, initiating processing of videos after a time- lapse of several frames reduces efficiency and increases latency between the incoming video and the consumer of the video at the endpoint of the video pipeline. This paper introduces techniques that enable real-time processing of videos to reduce such latency while still maintaining the quality of the video being received, enabling potential improvements to existing hardware-based solutions that require the use of external RAM memory for frame buffering.

Traditionally, random-access memory (RAM) (Kechiche, 2018; Vervoort, 2007; Wang, 2019; Cerezuela-Mora et al., 2015; Gonzalez, 2002) has been extensively used to store entire image frame data before actual processing is performed in image and video processing. As the resolution of images increase, so do the need for more RAM. To be able to process video at high speed, access to the memory now becomes a bottleneck. Newer DDR and HBM RAM interfaces have been designed to cater for a surge in need for faster

*Corresponding author's email: lim.sin.liang@mmu.edu.my, Tel.: +603-8312 5366; Fax: +603-83183029
doi: 10.14716/ijtech.v13i5.5865

RAM access from the host (eg. graphics processing unit (GPU)). This paper proposes a technique to perform image processing without requiring the use of external DDR or HBM RAM memory. A much smaller amount of on-chip RAM is required to buffer the incoming video stream. Figure 1 shows a video processing system that incorporates the proposed image processing technique. This paper is an extension of work originally presented in the 2020 edition of the IEEE Region 10 Conference (TENCON) (Kho et al., 2018).

2. Literature Review

Chen (Chen, 2008) has shown that it was feasible to implement Full-HD (1920×1080p)-resolution video processing on an application-specific integrated circuit (ASIC) at 55 fps. However, it is not known what type of edge detection or video sharpening algorithms were supported, or if any exist at all. Our research has shown that video sharpening, edge detection, and potentially many other video processing algorithms, can be implemented feasibly on a low-cost and much slower FPGA at a speed of at least 60 fps.

He (He, 2019) showed that it was feasible to implement high-speed HDMI video transfer and display using an FPGA device. Similar to our research, they have also used the HDMI as the video input interface to receive and perform video capture. However, their solution focused more on video capture, transmission, and display, with median filtering being the only algorithmic block. We have also used something similar to the image storage module, in what we refer to as the Row Buffers shown in Figure 1. However, our research focused more on improving algorithm efficiency and quality (Kho et al., 2018; Kho et al., 2019; Kho et al., 2020), as well as efficient video data encoding techniques. Our Row Buffers re-encode the VGA or HDMI input video stream into a native 15×3-pixel chunk structure which is then provided to the Chunk Processor. Our Chunk Processor efficiently repacks a chunk into an array of fifteen 3×3 kernels, which are then passed directly into our algorithm blocks for further video processing. As shown in Figure 7, there are 15 such kernel processors within the Chunk Processor that processes each kernel in parallel. Our algorithmic blocks can receive these kernels directly and commence the processing without further delay.

Kolonko (Kolonko, 2019) used a Raspberry Pi single-board computer to repack the standard VGA or HDMI input video source to a format suitable for their FPGA design to manipulate. Such a system was intended to test the algorithmic blocks within the FPGA, and it was not meant to be a production-ready solution. Our chunk processor has shown to quickly process incoming VGA or HDMI video and pass the information quickly to the algorithmic blocks, hence, this solution can be used as a stand-alone production-ready solution.

Kechiche (Kechiche, 2018) shows a typical implementation that uses an ARM processor connected to an AXI bus, and making use of a 1-Gigabit DDR3 external RAM device. Sousa (Sousa, 2020) uses the million of connection updates per second (MCUPS) metric to characterize the throughput of their neural-network based application, which is different from the video processing throughput. Park (Park, 2020) discussed a 29×29-pixel Retinex algorithm at length and reported that the latency in a 1080p video processing environment at 60 fps is unnoticeable. The latency proposed in this paper is also unnoticeable and has been used successfully in gradient-based calculations. Our proposed techniques could be scaled to the same kernel size as Park's, enabling the scaling of such applications to higher resolutions and framerates.

Wahab (Wahab et al., 2015) shows an automatic face-image acquisition system designed using a microcontroller that can be used for facial recognition applications. Pasnur (Pasnur et al., 2016) described methods to retrieve a partial image by specifying a region of interest (ROI). Our proposed techniques aim to help improve the speed of recognition and retrieval processes.

3. Hardware Architecture

3.1. Devising a new kernel processing scheme

Image processing algorithms usually implements with kernels, which are small fixed-size pieces of an image. These kernels contain pixels that form a square, with the width and height of the square usually as an odd number. Even-numbered kernel dimensions exist but are less common. Kernel sizes may range from 3-by-3, 5-by-5, 7-by-7, 9-by-9 pixels to larger dimensions. In this paper, we show how we have carefully devised a new image processing scheme that considers these odd-dimension kernel sizes.

We would like to process at least some kernels in parallel to leverage the parallel processing advantages inherent in hardware-based designs. A bunch of kernels will be transferred to the processing block to be processed in parallel. In our paper, we shall denote this group of kernels as a “chunk”. The block that processes these chunks is shown in Figure 1 as the Chunk Processor. In selecting a suitable size of a chunk, we considered the various kernel sizes from 3×3, 5×5, up to 15×15. We settled on a chunk size of 15×3 pixels due to practical reasons.

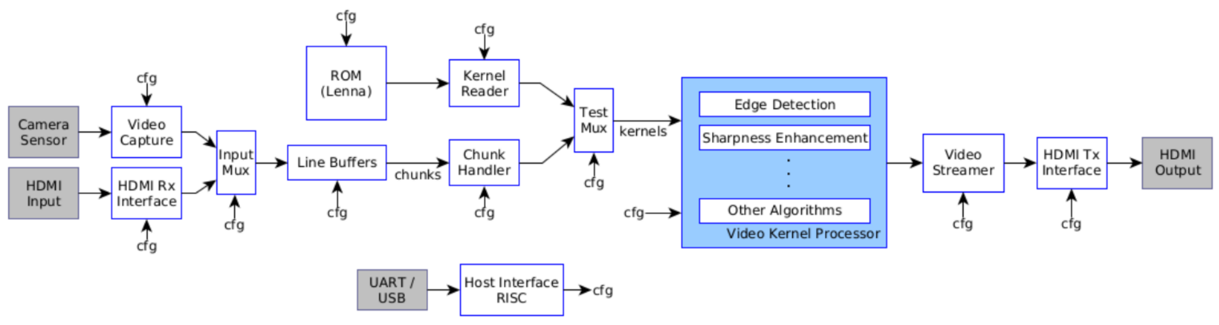


Figure 1 Video processing system block diagram

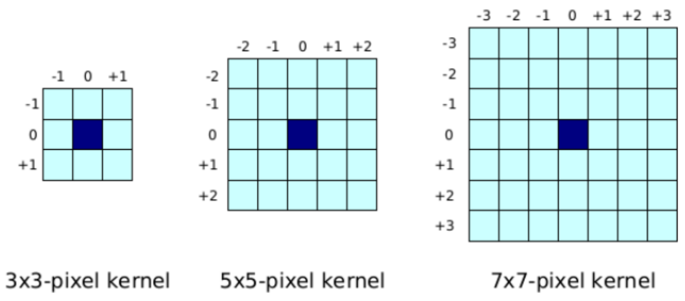


Figure 2 Examples of odd-sized kernel structures

Figure 2 shows several examples of odd-sized image kernels. The dark pixels denote the centre pixels. Because we are not able to support each and every possible kernel dimension, we wanted to make sure at least several kernels could be processed simultaneously using common chunk. By selecting a 15×3 chunk size, 3×3-sized kernels can be supported. Although other kernel dimensions such as 5×5 and 7×7 are gaining in popularity, kernels having dimensions of 3×3 pixels are still very much in use, therefore will be the focus of this paper.

To perform video processing continuously, one would shift the center pixel by one pixel, (either to the right, or to the bottom) by assuming that processing is done from the left to right and top to bottom. For example, by shifting the center pixel to the right, this would result in having 13 full kernels in a 15×3 chunk, and 2 partial kernels. The partial kernels are what we denote as the edge kernels or boundary kernels, as shown in Figure 3.

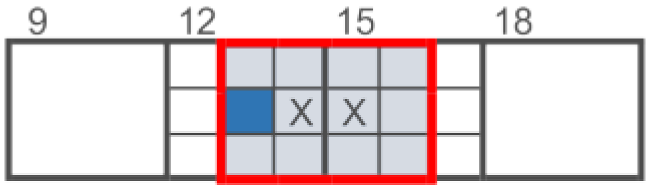


Figure 3 Boundary kernel structures

We will show that, from the viewpoint of the Kernel Processors, the pixel shifts are occurring in parallel, which means that we are processing all 15 kernels within a chunk simultaneously. This means that within a single chunk_valid cycle, all 15 kernels within a chunk are being processed in parallel, while the next chunk of video data is being buffered by the row buffers and transformed into chunks by the chunk processor.

3.2. Kernel processing scheme

The input video source is read in pixel-by-pixel by a pre-processing block, which is comprised of row buffers that stores several rows of pixels of a frame. These Row Buffers, as shown in Figure 1, receive the incoming pixels serially and buffer them in internal RAM memory. This block then transforms the buffered pixels into our chunk format, which is subsequently transferred to the chunk processor block. The row buffers will also generate a chunk valid pulse every time a 15×3-pixel chunk has been buffered and the data is available for the chunk processor. The chunk processor then transforms chunks into individual kernels which are fed into the parallel algorithmic kernel processors. This will process all the 15 kernels within the chunk in parallel.

In our research, we wanted to perform Sobel edge detection with a 3×3 kernel. The pixel arrangement of such a kernel is shown in Figure 2, where the highlighted pixel in black is the center pixel. For a typical software-based solution, the algorithm would sweep across the kernels from the left-most of the frame to the right, each time reading a 3×3-pixel kernel across 3 rows of the image. For our proposed hardware-based solution, we would read in 3 rows of 15 pixels directly (a chunk) and begin to process all the 15 kernels simultaneously. The proposed chunk structure is shown in Figure 5.

In the case of our Sobel gradient computations, it takes 3 clock cycles to complete the processing of a kernel. Because all the kernels within a chunk commences processing in parallel in which the same algorithm is used to process each kernel, all the kernels will also complete the processing at the same time. We shall denote this duration as the chunk processing time δTc. The chunk processing time varies depending on which algorithm is being used to process the kernels, and the actual hardware implementation involved including the number of pipeline stages used.

After a chunk has been processed, the hardware moves on to the next chunk and repeats the process until the end of the frame is reached. In the context of a Full-HD (1920x1080p) frame, our proposed chunk structure would be as shown in Figure 6.
Figure 7 shows a block diagram of our proposed chunk processing scheme. The Chunk Processor in Figure 7 corresponds to the same Chunk Processor module as was shown in Figure 1. Likewise, the Kernel Processors in Figure 7 corresponds to the same module as depicted in Figure 1. Here, we show the parallel processing architecture of the 15 Kernel Processors.

The kernel outputs from the Chunk Processor are fed to the algorithmic Kernel Processors for parallel computation. The kernel processor outputs q0 to q14 correspond to the processed pixels after every kernel has been processed. Each kernel processor works on the center pixel of a 3×3-pixel kernel and computes the resulting output pixel qN based on the other 8 pixels adjacent to the center pixel.

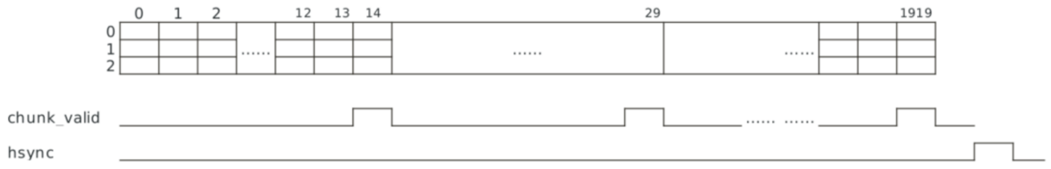


Figure 4 Chunk valid timing

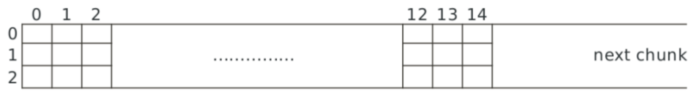


Figure 5 Chunk structure

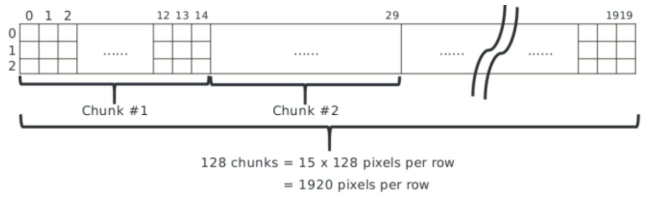


Figure 6 Chunk structure for a Full-HD Frame

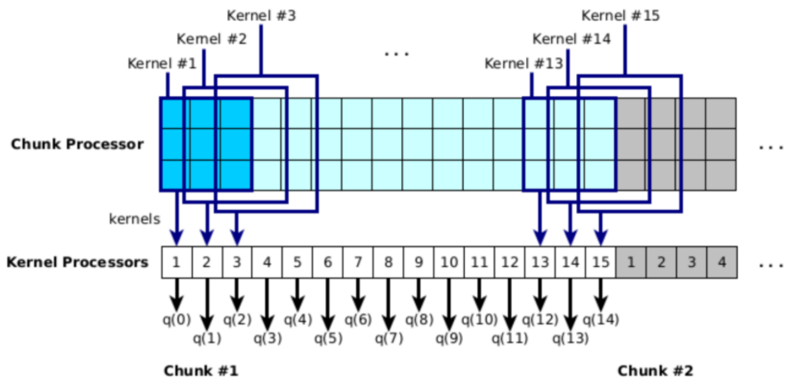


Figure 7 Parallel kernel processing scheme

At every chunk_valid pulse, data from 15 kernels is available. The 15 Kernel Processors will commence the parallel processing of all the kernels coming from the Chunk Processor, producing the pixel field q consisting of 15 pixels per chunk_valid cycle.

4. Experimental Setup

Figure 8 shows the experimental setup to characterize our video buffering scheme and processing algorithms. Our hardware implementation currently supports Full-HD (1080p) resolution video running at 60 frames-per-second (fps).

In our laboratory environment, we used a 100-MHz oscillator input from our board as our system clock. A Xilinx Mixed-mode Clock Manager (MMCM) module is used to synthesize the output frequencies required for 1080p video. From the 100-MHz input clock, a pixel clock running at 148.5 MHz is generated by the MMCM. Also, a high-speed clock running at 750 MHz, required for transmitting and receiving serialized video data at the HDMI connectors, is also generated by the MMCM.

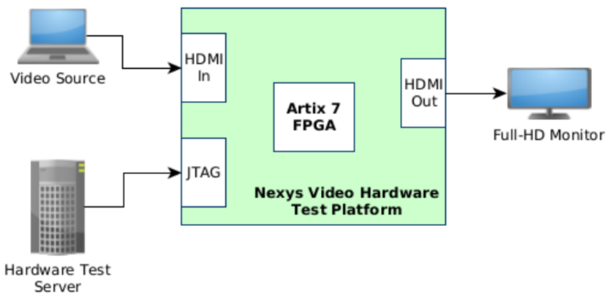


Figure 8 Experimental laboratory setup

To characterize the performance of our chunk processor in terms of signal latencies, we shall first denote the signals of interest. The reset signal resets the system to default values. During this time, all counters and registers are zeroed out. The Vsync signal is the vertical synchronization signal of the incoming video input. Together with the vertical blanking (Vblank) signal, the Vsync signal is used to flag that a new video frame has arrived. Both the reset and Vsync signals are inputs to our chunk processing module.

The chunk_valid signal is an output of our chunk processor block, indicating that a chunk of video data (15×3 pixels) has been processed and it is available for use by the downstream algorithmic processing blocks.

As part of our characterization work, all these signals are captured using Xilinx ChipScope, and they produce similar results with our logic simulations. The Results section discusses these in terms of signal latencies and performance.

5. Results

Our simulation testbench closely emulates our hardware configuration. All hardware blocks have been written in the VHDL (Ashenden, 2010) hardware language. The design was functionally simulated in Mentor Graphics ModelSim. A sample screen capture of the final simulation results are shown in Figure 9.

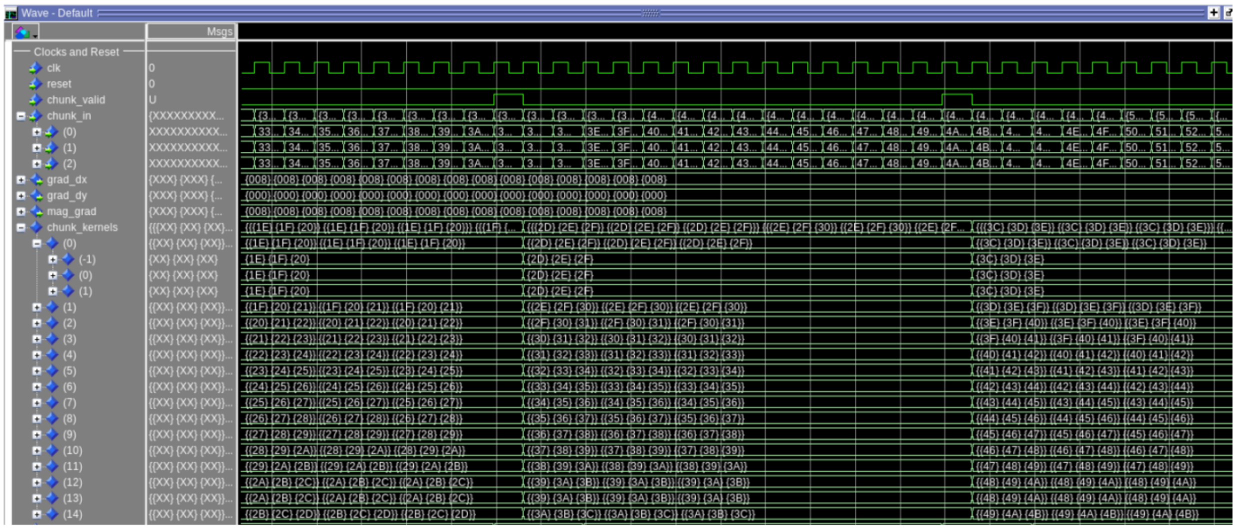


Figure 9 Partial view of output data from chunk processor.

After gaining enough confidence from our simulations, we synthesised our design to FPGA hardware using Xilinx Vivado. The final implementation was downloaded into the Xilinx Artix XC7A200T device and the hardware results are acquired in real-time using

Xilinx ChipScope. The realtime hardware results are compared and verified against the ModelSim simulation results.

5.1. Functional Simulations

We have implemented this design on a Xilinx Artix 7 (XC7A200T) FPGA device and used Xilinx's ChipScope integrated logic analyzer to acquire real-time waveforms from our development hardware. Figure 9 shows a partial view of the functional simulation results in ModelSim.

From Figure 10, we note that the latency from the deassertion of the reset signal until the first assertion of the chunk_valid signal is 59.3535 microseconds (μs). A similar amount of time will be taken between the assertion of the Vsync video synchronization signal until the assertion of the chunk_valid signal. This means that the initial latency after a reset or power-up -up event, until data becomes available for processing by the algorithmic blocks, is 59.3535 μs.

Figure 11 shows that the processing latency between one chunk_valid pulse and the next is 101.01 nanoseconds (ns).

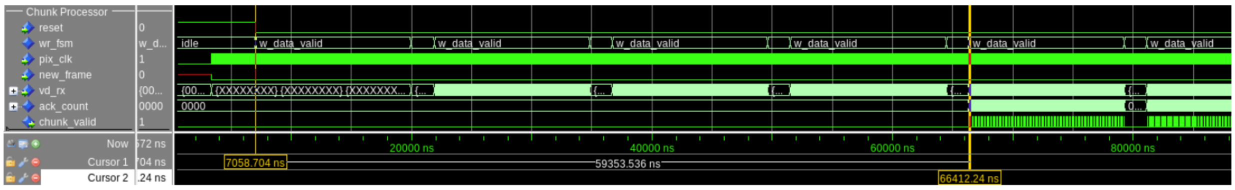


Figure 10 Start-up latency between reset and first chunk_valid

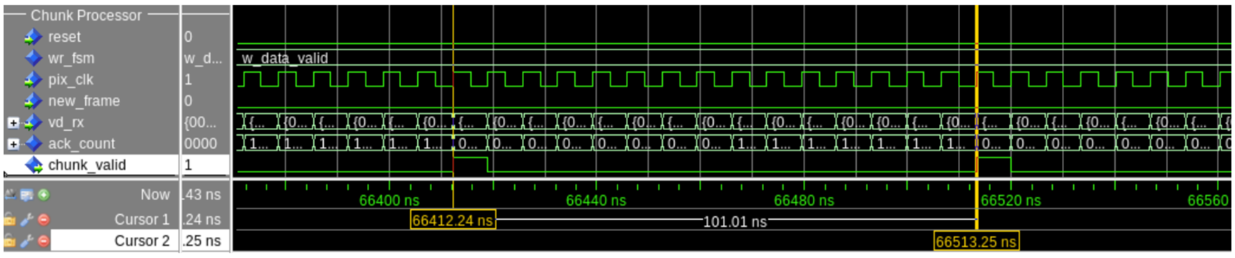
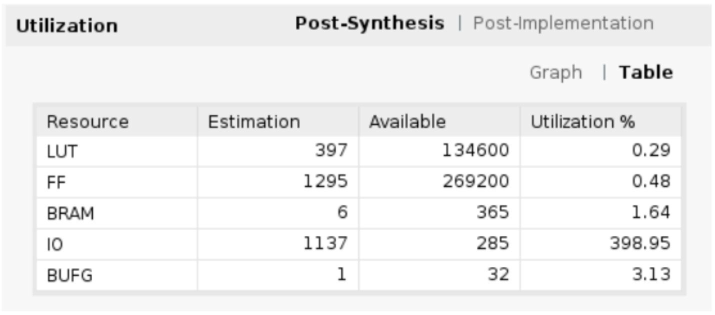


Figure 11 Latency between two consecutive chunk_valid pulses

5.2. Hardware Synthesis, Place & Route, and Design Assembly

The Xilinx post-synthesis report shows that our row buffers and chunk processing scheme utilize only 397 look-up tables (LUTs), which on the Artix XC7A200T device, translates to just 0.29% of all the LUT resources in the chip. The post-synthesis results are shown in Figure 12.

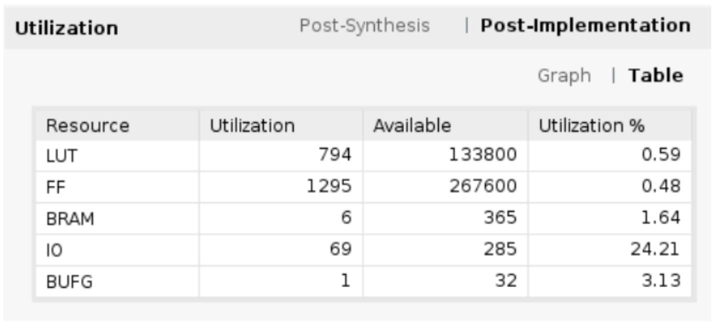

Utilization Post-Synthesis | Post-Implementation

Graph | Table

Resource	Estimation	Available	Utilization %
LUT	397	134600	0.29
FF	1295	269200	0.48
BRAM	6	365	1.64
IO	1137	285	398.95
BUFG	1	32	3.13

Figure 12 Post-synthesis utilization results for the chunk processing module

Xilinx's post-place-and-route (PAR) report shows that our chunk processing scheme was physically implemented with only 794 LUTs, which translates to 0.59% of all the LUT resources in the Artix 7 chip. The results are shown in Figure 13.



Resource	Utilization	Available	Utilization %
LUT	794	133800	0.59
FF	1295	267600	0.48
BRAM	6	365	1.64
IO	69	285	24.21
BUFG	1	32	3.13

Figure 13 Post-PAR utilization results for the chunk processing module.

The extra LUTs reported by the post-PAR report are due to deliberate insertion of multiplexers to serialize the chunk output results, which otherwise would have used too much input/output (I/O) pins to implement. During normal usage, this module will be integrated with the algorithmic processing modules which will be able to receive all the parallel chunk data directly. This will in effect, require only 397 LUTs in a typical implementation.

Notice that in our implementation, we have also used 6 Xilinx's block random-access memory (BRAM) memory for buffering of the incoming Full-HD video data. This amounts to only 1.64% of the Artix 7 device. This means that our implementation is more easily scalable to support higher video resolutions such as 4K and 8K video, without requiring additional hardware.

5.2. Computational Throughput

The chunk processor depends on the video transfer rate of the incoming video stream. As such, certain amount of pixel buffering is required before repacking the pixels into the 15×3 chunk format. Once a 45-pixel chunk is available, the downstream algorithmic blocks will process all 15 kernels in parallel. In effect, there are 15 overlapping kernels, each with 3×3 size, that are being processed in parallel by the algorithmic kernel processing blocks, resulting in a total of 135 concurrent pixel operations per chunk_valid cycle.

We are therefore processing 135 pixels within a 15-cycle period of the pixel clock for a single channel of video data, which translates to 9 pixels per clock per channel. For a 3-channel RGB or YCbCr video system as in our case, we have 27 pixels per cycle of the pixel clock. Consider our pixel clock to have a frequency of 148.5 MHz (as in the case of Full-HD (1080p) video), hence the pixel clock period is 6.734 ns. As a result, we have a computational throughput of 4.0095 gigapixels per second, as shown in Equation (1).

$$\begin{aligned} R &= mf \\ &= 27p \times 148.5MHz \\ &= 4.0095Gps^{-1} \end{aligned} \tag{1}$$

where:

m = number of pixels being processed

f = pixel clock frequency

p = symbol unit, in term of pixels

Table 1 provides comparisons between our computational throughput against the throughput results from other implementations. Due to differences in the scope of work, fair comparisons could not be made. For example, we discuss the computational throughput of the buffering and pixel-packing stages before any algorithmic computations are performed, whereas the other implementations discuss the throughputs of the actual algorithms being performed.

Table 1 Computational throughput comparisons between several implementations.

Implementation	Throughput (Mpixels/s)
He et. al (He, 2019)	995
Zhou et. al (Zhou, 2011)	530
This Work[a]	4009.5

[a] Fair comparison could not be made due to differences in scope

Due to the scarcity of pre-algorithmic throughput data, Table I gives an idea of existing computational throughputs (Zhou et al., 2011; He et. al, 2015; He et al., 2019) related to video processing in general. The fact that our computational throughput is much higher at 4.0095 gigapixels per second implies that our solution could be applied to other existing kernel-based algorithms without incurring too much additional latency, and instead help to increase the throughput of such systems by helping to process kernels in parallel.

6. Conclusions

Our research has shown that we are able to process video edges and perform video sharpening in real time at 60 fps frame rate for incoming Full-HD video. By utilising our line buffers and chunk processor blocks, incoming serial video streams from an HDMI input interface can be quickly transcoded to a suitable data format for other downstream video processing algorithms, such as our video edge detection and enhancement algorithms. The ability of our chunk processor block to directly produce kernel data structures for seamless integration with our algorithmic blocks helped increase the performance and efficiency of our solution. The chunk processing scheme has shown to be feasibly implemented with a small amount of digital hardware (397 LUTs, and 6 internal BRAMs), and could help lower the cost of existing video processing systems.

Acknowledgements

The authors are thankful to the Ministry of Higher Education Malaysia for the Fundamental Research Grant Scheme FRGS/1/2015/TK04/MMU/02/10 to support this project. Also, we are thankful to Jeannie Lau and Ang Boon Chong for the many technical discussions that helped us complete this project in one way or another.

References

Ashenden, P.J., 2010. *The Designer's Guide to VHDL.* 3rd Edition. USA: Elsevier

Cerezuela-Mora, J., Calvo-Gallego, E., Sanchez-Solano, S., 2015. Hardware/Software Co-Design of Video Processing Applications on a Reconfigurable Platform. *In:* IEEE International Conference on Industrial Technology (ICIT), pp. 1694–1699

Chen, J.C., Chien, S., 2008. CRISP: Coarse-Grained Reconfigurable Image Stream Processor for Digital Still Cameras and Camcorders. IEEE *Transactions on Circuits and Systems for Video Technology*, Volume 18(9), pp. 1223–1236

Floating-Point Reciprocal Square Root. *Journal of Electrical & Electronic Systems,* Volume 7(4)

Gonzalez, R.C., Woods, R.E., 2002. Digital Image Processing. 2nd Edition. New Jersey: *Prentice-Hall*

He, G., Zhou, D., Li, Y., Chen, Z., Zhang, T., Goto, S., 2015. High-Throughput Power-Efficient VLSI Architecture of Fractional Motion Estimation for Ultra-HD HEVC Video Encoding. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Volume 23(12), pp. 3138–3142

He, X., Tang, L., 2019. FPGA-Based High Definition Image Processing System. *In:* International Conference on Wireless and Satellite Systems, pp. 208–219

Kechiche, L., Touil, L., Ouni, B., 2018. Toward the Implementation of an ASIC-Like System on FPGA for Real-Time Video Processing with Power Reduction. *International Journal of Reconfigurable Computing*, Volume 2018, pp. 1687–7195

Kho, C.K., Fauzi, M. F. A., Lim, S., L., 2020. Hardware Parallel Processing of 3x3-Pixel Image Kernels. *In*: 2020 IEEE Region 10 Conference (TENCON), pp. 1272–1276

Kho, C.K., Fauzi, M.F.A., Lim, S., L., 2018. Hardware Implementation of Low-Latency 32-Bit

Kho, C.K., Fauzi, M.F.A., Lim, S.L., 2019. Hardware-Based Sobel Gradient Computations for Sharpness Enhancement. *International Journal of Technology*, Volume 10(7), pp. 1315–1325

Kolonko, L, Velten, J., Kummert, A., 2019. A Raspberry Pi Based Video Pipeline for 2-D Wave Digital Filters on Low-Cost FPGA Hardware. *In:* IEEE International Symposium on Circuits and Systems (ISCAS), Sapporo, Japan

Park, J.W., Lee, H., Kim, B., Kang, D., Jin, S.O., Kim, H., Lee, H., 2020. A Low-Cost and High-Throughput FPGA Implementation of the Retinex Algorithm for Real-Time Video Enhancement. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Volume 28(1), pp. 101–114

Pasnur, Arifin, A.Z., Yuniarti, A., 2016. Query Region Determination Based on Region Importance Index and Relative Position for Region-Based Image Retrieval. *International Journal of Technology*, Volume 7(4), pp. 654–662

Sousa, M.A.A., Pires, R., Del-Moral-Hernandez, E., 2020. SOMprocessor: A High Throughput FPGA-Based Architecture for Implementing Self-Organizing Maps and Its Application to Video Processing. *Neural Networks*, Volume 125, pp. 349–362

Vervoort, L.A., Yeung, P., Reddy, A., 2007. *Addressing Memory Bandwidth Needs for Next Generation Digital Tvs with Cost-Effective, High-Performance Consumer DRAM Solutions*. White Paper, Rambus Inc. Sunnyvale, CA, USA

Wahab, W., Ridwan, M., Kusumoputro, B., 2015. Design and Implementation of an Automatic Face-image Data Acquisition System using IP Based Multi Camera. *International Journal of Technology*, Volume 6(6), pp. 1042–1049

Wang, Z., Huang, M., Zhang, G., Qian, L., 2019. The Design of MIPI Image Processing Based on FPGA. *In:* SPIE Optics+ Optoelectronics, 2019, Prague, Czech Republic

Zhou, D., Zhou, J,J., He, X., Zhu, J., Kong, J., Liu, P., Goto, S, 2011. A 530 Mpixels/s 4096x2160@60fps H.264/AVC High Profile Video Decoder Chip. *IEEE Journal of Solid-State Circuits*, Volume 46(4), pp. 777–788