HW-Accelerated Real-Time Video Streaming and Processing

Authors: Ilan Biala, Brandon Takao, Edric Kusuma
Electrical and Computer Engineering
Carnegie Mellon University
Email: {ibiala, brt, ekusuma}@andrew.cmu.edu

***Abstract*—This capstone project is a distributed, hardware-accelerated platform that provides functional video processing on live streams with the goal of scalability and speed. With the prevalence of low-cost video devices, coupled with the use of intense video processing algorithms, the need to offload said processing is on the rise. Our system streams video feeds over Wi-Fi from low-power, low-performance devices through an Avnet Ultra96 SoC and FPGA to easily and capably handle real-time video processing before sending the processed streams to a monitoring computer over Wi-Fi. The algorithm we implement is the Canny edge detection algorithm, one that is the basis of many computer vision algorithms, and is quite computationally-intense and powerful as a preprocessing step for further algorithms such as image segmentation and object detection. Our implementation will improve on a software-only approach and thus improve performance and unlock additional scalability while keeping costs low.**

***Index Terms*—Distributed Systems, Edge Detection, High-Level Synthesis, FPGA, Hardware-Accelerated, Computer Networking, Video Processing.**

I. Introduction

As augmented reality, self-driving cars, and other technologies become mainstream, having the ability to do real-time video processing will play a critical role in the adoption of these technologies. Our project will explore and analyze the implementation of hardware-accelerated real-time video streams with Canny edge detection in a security system context. We implemented a networked video streaming and processing system that receives a 720p30 H.264 video stream from a Raspberry Pi Zero W over Wi-Fi and performs hardware-accelerated Canny edge detection on the Ultra96's Xilinx FPGA before routing the processed video stream to a laptop that displays the results similar to a security system's central monitoring room. Before routing the stream to the monitoring room, the CPU offloads the computational steps of Canny edge detection to our custom implementation on FPGA fabric to reduce computational latency from approximately 1.25 seconds to just 20 milliseconds. Our system reaches scalability limitations on server-side CPU due to lack of support for hardware-accelerated video encoding and decoding, which could be mitigated by sourcing Xilinx's V-variant Zynq SoC that includes dedicated video processing functionality including hardware-accelerated video encode and decode.

Our implementation and findings have the potential to be applied to a wide spectrum of video processing applications, especially the security, transportation, and telecommunication sectors. With new developments in video-based security systems, self-driving cars, and video conferencing systems, our work can really improve on the latency for all of these applications. This is the main advantage of our approach. Currently, Canny edge detection is accomplished mainly using software either on CPUs or on GPUs, which can take anywhere between 100s milliseconds to over 1 second to to go through the multi-step algorithm. Our system significantly reduces this latency, resulting in a much faster response time that opens up new possibilities in all of these sectors both in terms of functionality and scalability. Also, the networking and memory interfacing aspects of our system allow for a multiple feed approach to any problem that these sectors will inevitably encounter. Our main goal is to use our implementation as a proof of concept for an easily scalable, hardware-accelerated video processing system.

II. Design Requirements

Our main design requirements were that we can handle at least two Raspberry Pi streams simultaneously within our system, and communicate successfully across the entire architecture. Additionally, we must be able to accomplish the canny edge detection algorithm on our FPGA, while minimizing latency. Finally, we must be able to achieve a real-time video stream, and demonstrate the scalability of the system as a whole. Qualitatively, most of this was verified by visual analysis, as we had the final output of the video feed displayed on a browser, such that we can both see the video feed and analyze the results of the canny edge detection process. Quantitatively, we were aiming for a goal of < 100 ms of latency throughout the entire system. In order to verify that our design met these specifications, we planned to and succeeded in conduct unit tests throughout the process, in order to isolate each component, and eventually, conducted integration tests. Aside from performance, there was also a scalability requirement for our system. Because we are targeting a network of security cameras, the ability to add an additional feed is crucial to our design. As such we need to be able to quantify how many resources a feed takes up in the form of FPGA fabric (LUTs, FFs, BRAMs, DSPs are the main metrics of interest), Wi-Fi bandwidth, and DRAM address space. With these

metrics we are able to extrapolate the cost of an additional stream if we were to add more to a larger FPGA.

A. Success Metrics

1) Software Networking

TABLE I. NETWORKING SUCCESS METRICS

Metric	Goal	Result	Pass/Fail
Concurrent streams	At least 2	1 for end-to-end 3 for SoC-exclusive pipeline	Fail
Resolution	1280x720	1280x720	Pass
Framerate	30 FPS	5 FPS	Fail
Packet loss	< 3%	~1%	Pass
Communication latency	75 ms	50 ms	Pass
Visual analysis of result	Compare to software implementation	Visual inspection passed	Pass

In order to demonstrate the scalability of the system, we originally aimed for at least two streams, which we actually achieved on the Pis (we demonstrated three streams, and those streams all exceeded our metrics of 30 FPS and ~50 ms latency, which achieved real-time streams). Unfortunately, when sending the streams through the ARM core, we realized that because of the lack of dedicated video encode and decode functionality, we could not support more than one stream (we could not even support one at our target metrics) since the H.264 decode and encode maxed out CPU and memory utilization. Additionally, this limited our FPS on the send side, as we ended up receiving from the Pis at 30 FPS, but sending out at 5 FPS. While all of our other metrics pa ssed - resolution at 1280x720, ~1% packet loss due to TCP, and a communication latency of 50 ms - this FPS bottleneck resulted in a jittery, delayed result stream. Thus, we were unsuccessful in achieving an end-to-end real-time video stream, and we did not have enough time to source and integrate a dedicated H.264 IP block. However, our solution could easily be migrated to use Xilinx's V-variant SoC with dedicated hardware encode/decode functionality or hardware-accelerated encode/decode as part of our overall edge detection compute pipeline.

2) Hardware-accelerated Computation

TABLE II. HARDWARE SUCCESS METRICS

Metric	Goal	Result	Pass/Fail
Scalable with more streams	2 streams	> 7 supported streams	Pass
High resolution and framerate	720p30	720p60 stream processing	Pass
High pipeline throughput	320 MHz clock frequency	333 MHz clock frequency	Pass
Computation latency	25 ms/frame	15ms/frame	Pass

We initially targeted 2 streams as a proof-of-concept, and since we were working with a smaller FPGA that might be limited if the pipeline proved to use all of the 128 DSP slices/edge detection pipeline that we originally estimated. However, we were able to optimize many of the operations to either not use DSP slices at all while still maintaining good performance or to use look-up tables and other workaround functionality. Our design was also further optimized from initial experimentation and estimates and used significantly less fabric per edge detection pipeline. Regarding the timing metrics, we are able to achieve nearly double the 30 FPS target frame rate since our design is both pipelined within each stage as well as between stages. This yields higher throughput by sacrificing a small amount of register-transfer latency, but our computation latency still remains about 1.67x our original estimate. Our target clock frequency was also surpassed after some optimization of the compute pipeline. We were not able to push this clock frequency any higher due to AXI port clock frequency limitations.

3) Final Hardware Resource Utilization Metrics

TABLE III. HARDWARE UTILIZATION METRICS

Base components	LUTs	FFs	BRAMs	DSPs
Zynq	0	0	0	0
Reset	17	37	0	0
Total	17	37	0	0
Percent	0.02%	0.03%	0	0
Per-compute pipeline components	LUTs	FFs	BRAMs	DSPs
2 data width converters	0	0	0	0
AXI DMA	1475	1960	5	0
2 AXI Interconnects	2752	2837	4	0
Canny edge detection pipeline	3061	1697	8.5	2
Total	7288	6494	17.5	2
Percent	10.33%	4.60%	8.10%	0.56%
Total for 1 stream	7305	6531	17.5	2
	10.35%	4.63%	8.10%	0.56%
Total for k streams (k defined below)	58321	51989	140	16
8	82.65%	36.84%	64.81%	4.44%

Here we show the final hardware resource utilization metrics, with a breakdown between base components and the per-compute pipeline components. We can see that the base components use a negligible amount of resources, and each pipeline uses approximately ~10% of the FPGAs LUTs after a moderate amount of optimization. This is how we come to our conclusion of being able to support over 7 simultaneous streams in hardware. Aside from hardware resource utilization, memory bandwidth is the other factor to consider. Note that we pivoted from a main memory buffering scheme to a streaming computation design, which yields much lower memory and bandwidth utilization. The AXI port's bandwidth is also on the order of gigabits/second, so it is not a limiting factor. As a result, we conclude that we could support a much higher number of streams, likely between 7-8 depending on place-and-route results and optimization tweaks to fit in this SoC's FPGA fabric.

B. Testing and Validation Plan

Testing and validation is a critical step in ensuring that any project's implementation works correctly and meets the

requirements set forth during the design, and this is no different for this project. On the software side, many components were unit tested and mocked out to easily test the implementation along the way. Methods such as dummy and random frame generation for sending and receiving video frames allowed us to easily test our implementation, as well as visual inspection when using images and actual video streams. Packet loss is also both easily visually inspected as well as programmatically measured, since any dropped packets are observed as missing data and incomplete frames. On the hardware side, our compute pipeline design is fully testable in simulation, and as such time has been allocated for testbenches to be developed that will test both correctness and reliability of our hardware implementation. Our testbenches will again use dummy and randomized data generation to feasibly test as much of our design as we can without requiring a significant amount of time creating test cases. At the integration stage, most of work will be testable by visual inspection as well as by verifying against a software library implementation. Our interfaces are the least testable, and as such we will mainly be focusing on testing these interfaces and verifying realistic actual bandwidth and throughput numbers based on their theoretical values. Our memory interaction between the ARM core and FPGA is tested through the use of a starter Vivado HLS block that is known to be functional and performs a Gaussian blur. Throughout the development process, these testbenches were used to quickly root cause bugs and unwanted behavior during bring-up, development, and integration. As always, avoiding integration failures and failing earlier during the component and module implementation phase is preferable, so our integration tests are mostly for the purpose of testing before other components of the system are ready, and they also act as simulated, controlled functional testing that will prove helpful for simulating actual bugs during the integration process.

III. Design Trade Studies

A. Software

On the software side, there were two main decisions that were analyzed for design tradeoffs and ultimately determined the major design of the software system. These two decisions were UDP vs TCP and format transmission protocol (raw arrays vs JPEG vs H.264 compressed encoded stream).

For the UDP vs TCP decision, we implemented and tested both protocols early on, and compiled some latency numbers. While UDP was significantly lower latency than TCP (~3x), in packet loss testing, it resulted in enough packet loss to significantly negatively affect the quality of the video (based on visual inspection). Even though our project was highly focused on minimizing latency, we felt that the quality of the video must reach a certain standard (<3% packet loss), and that UDP wasn't achieving this. Thus, we decided to pivot to TCP, and accept the higher latency. Fortunately, we were still able to satisfy our software communication latency requirement.

For the format transmission protocol decision, most of the semester was spent on this decision. Initially, we started with raw grayscale array transmission, which is extremely inefficient, and not very realistic in achieving the low latency and high FPS in our goals. However, sending arrays is a much simpler task, and these arrays are therefore much more accessible than the other options. Our first pivot was into an H.264 stream, which we implemented by the interim demo. Unfortunately, with bandwidth issues in the Pis, we weren't able to observe a significant increase in FPS, and it was a fairly complex process to extract individual frames from the stream, so we pivoted a second time into JPEGs. This method was less efficient than the H.264 stream, but still much more efficient than raw grayscale arrays. However, even with a threading system implemented, and after fixing the bandwidth issues on the Pis, we were still failing to achieve the FPS goals that we were hoping for. We finally pivoted for a third time back to H.264 streams, but using a different transmission method to allow for frame extraction, and this method worked very well overall. The differences in these qualities, along with ease of visualization, is demonstrated in Table IV below.

TABLE IV. VIDEO STREAM FORMAT TRADEOFFS

Format	Accessibility	Efficiency	Visualization
Raw arrays	1	3	2
H.264	3	1	1
JPEG	2	2	3

**Qualities are ranked from 1 (best) to 3 (worst).*

B. Hardware

On the hardware side, the bulk of the design tradeoffs were made in 1) the passing of data through the pipeline, and 2) the interfaces between each phase.

1) Pixel Data

We decided early on that we would be using 8-bit pixel values as edge detection doesn't require much, if any granularity with pixel color values. There was an issue around having to use floats for the Sobel phase's gradient calculation, but we noticed that the HLS built-in arctangent function conveniently rounds values to integers. Thus we were able to keep an 8-bit stream throughout the pipeline.

2) Memory Interfacing

TABLE V. MEMORY INTERFACING TRADEOFFS

Interface	Advantages	Disadvantages
DRAM	Default interface, and extremely easy to implement.	Extremely slow to access values, and convolution access patterns are low in locality, thus burst reads are not possible.
BRAM	Also easy to implement, and is much faster than DRAM.	Not enough BRAM on the Ultra96 to buffer the entire frame. Too expensive to scale up for multiple streams.
AXI streams	Very low BRAM utilization, streamed accesses have high throughput, leading to lower overall latency.	More complicated to implement. Unable to access values already read without something like line buffers.

The table above summarizes our experiences with each of the interface types we tried. In the end we decided to use AXI

streams as they were most in line with our requirements of low utilization and low latency. Our entire code had to be overhauled to adhere to the streaming access pattern, but the gains from doing so were well warranted.

IV. Architecture and System Description

The RPis with cameras act as clients, streaming their video via Wi-Fi with TCP to the Ultra96 SoC. The server Ultra96's ARM core is then utilized to handle the receiving, decoding, and storing of input frames to the edge detection pipeline and re-encoding and sending of the processed frames. Due to the memory requirement of storing an entire frame, we decide to store frame data in DRAM as a contiguous array of pixels. We note that adding more streams to the system will reduce the space allowed for each stream; however, DRAM has become relatively inexpensive and is not the limiting factor in cost scalability of this system, as we will later go in detail.

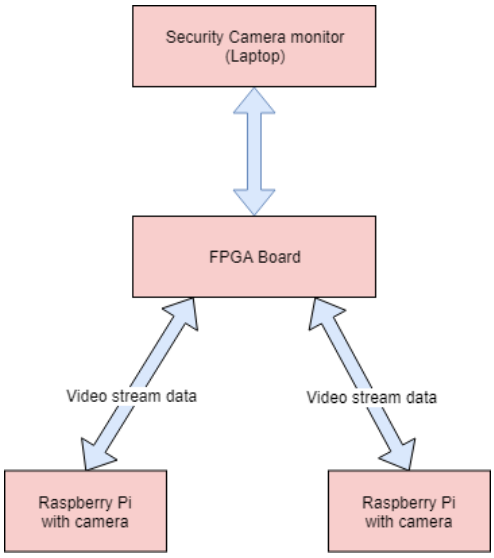

Fig. 1. Example system architecture

A. Software System/Video Transmission

After numerous instances of trial and error where we implemented raw array frame transmission, H.264 stream transmission, and JPEG transmission, we decided to revert to the H.264 stream transmission protocol to minimize network bandwidth consumption. The Raspberry Pi Camera feeds were recorded using the Raspivid framework, and then piped into an FFmpeg TCP transmission block on the RPi. On the Ultra96's ARM core, an FFmpeg receive and decode block is used to convert the incoming TCP stream to a raw array of 24-bit RGB pixel data. This image is then converted to grayscale on the ARM core before being stored into contiguous memory blocks for the fabric to access and process. The resulting processed frames are outputted back into the ARM core into a dedicated contiguous receive buffer. These frames are encoded back into an H.264 stream and sent using FFmpeg over TCP to the laptop "monitoring room". The stream is received and displayed using Mplayer, which allowed for real-time visualization. H.264 as a streaming and compression system really helped us attempt to maximize the FPS of our video stream, as in our tests with JPEG and raw arrays, the amount of data being sent and the inherent computational overhead of producing this data really limited the FPS of the resulting video. We also decided to use TCP vs UDP because of the packet loss we were experiencing through UDP. Finally, Mplayer was the easiest media visualizer that was compatible with H.264 streams, so it was the natural choice for video stream visualization.

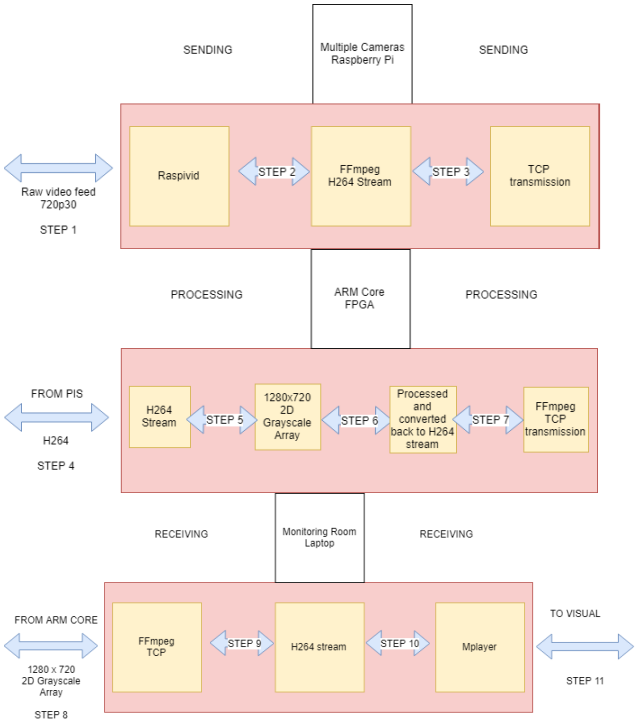

Fig. 2. Software video stream flow

B. Server Architecture

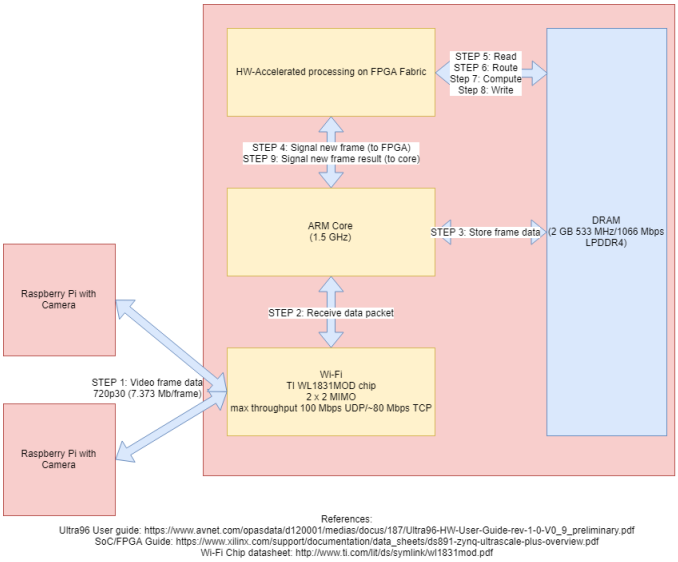

Fig. 3. High-level receive-side and compute server architecture

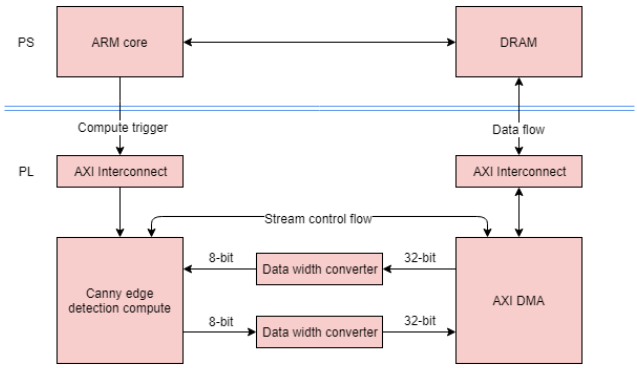

Fig. 4. PS/PL memory interface

Our server architecture diagrams mainly focus on the communication portion of the system, since this is somewhat isolated from the compute pipeline in functionality (aside from data widths, packing schemes, etc.). Here we focus on the receive-side of the system, since the send-side communication is simply to display the processed video streams on a monitoring computer and is quite straightforward. The Ultra96 has a built-in TI WL1831MOD Wi-Fi chip that in practice yielded approximately 40 Mbps throughput, which will be more than sufficient for two streams and leaves a small amount of room for future scalability depending on the exact bitrates used for the H.264 streams. All data is transmitted over Wi-Fi, and the ARM core is responsible for allocating contiguous memory for pre-processed and post-processed frames. Our memory interface starts by receiving 32-bit streaming grayscale pixel data, with the lower 8 bits of each data beat containing 1 8-bit pixel value. This is the simplest approach and we did not anticipate memory bandwidth being the initial bottleneck, so we left this unoptimized. The DRAM memory interface on the Ultra96 supports 1066 Mbps and the AXI port interface supports a theoretical maximum of 42.6 Gbps. Both of these interfaces will again be more than sufficient for two streams while still having room to scale. Once the frames are in DRAM, the ARM core will communicate with the hardware implementation in FPGA fabric with a producer consumer handshake. The FPGA fabric itself will stream and convert the 32-bit pixel data from the contiguous memory buffer through the compute pipeline since each frame is too large to store in block RAM or distributed RAM. Once processing is done, the ARM core will push out the processed frames over Wi-Fi to the monitoring computer, which will display the streams for users.

C. Compute Block Architecture

When a frame is ready to be processed, the ARM core will signal the compute block with a ready signal and a starting address. From there the frame is pulled from DRAM and the processing can begin.

Because the convolution operation requires the values of surrounding pixels, we are unable to process a frame and modify its value in-place. Moreover, because each phase is dependent on the output of the previous, we would require a significant amount of memory to store a copy of every processed frame. Doing so in DRAM is unacceptable, as the latency around pulling from DRAM would be too high for our requirements. To account for this limitation, we decided to stream the frame data through the pipeline using 8-bit AXI streams. As edge detection pixels are either black or white there is no need to waste bandwidth on higher-bit streams. Due to the nature of the convolution operation which requires accessing previously used pixels, we use a line buffer to keep temporary copies of the surrounding pixels. We find that through using streams and line buffers we are able to both remove the excessive latency involving DRAM accesses while keeping our overall utilization low. Once a phase is complete the next phase in the pipeline takes in the streaming output of the previous and operates on it, whereby this process repeats until completion.

In implementing the Canny compute pipeline, we decided to do the entirety of it in Vivado's High-Level Synthesis (HLS) suite. Doing so gave us the following benefits: removing the development time needed to implement an AXI interface and interface with other Vivado IP blocks, use of convenient utilities like line buffers and complex math functions, detailed synthesis reports with both timing and utilization summaries, and the convenient use of C++ testbenches for behavioral validation.

In the Gaussian blur, Sobel, Non-maximum Suppression and edge tracking phase, a 3x3 convolution is necessary. To do this we utilized three line buffers which are shifted down the frame once an entire row is finished processing. This allows us to access pixel values that we would normally be unable to use given our streaming data access pattern. While the operations performed for each phase's convolution were different, the usage of this line buffer structure stayed the same. This made implementation of each phase quite simple once we got past the learning curve of using HLS's built-in utilities.

We can see during the Intensity Gradient Phase that two separate operations need to be performed: calculating pixel magnitude and gradient. Initially we attempted to write to two

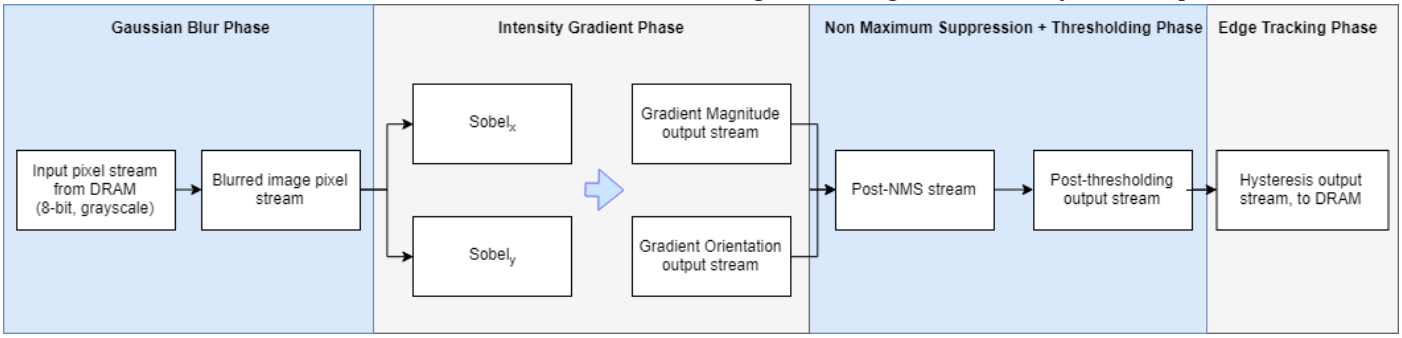

Fig. 5. Canny compute pipeline

concurrent AXI streams, which were successful in the C simulation but ended up being too out of sync with each other going into the non-max suppression block and deadlocked when run on the RTL simulator and on the FPGA. In the end we decided to output a single 16-bit stream where the magnitude and gradient results are concatenated together. The NMS phase is then responsible for extracting these values. It is worth noting that thresholding does not require the use of neighboring pixel data, so we managed to include this in the NMS phase. This reduced the overall latency of the pipeline.

Once a frame is complete the compute block signals the ARM core that it is finished. It is then up to the core to pull the frame from DRAM and transmit it to the monitor. This includes encoding the frame data back to H.264 before sending it out.

V. Project Management

A. Schedule

Since our schedule is not readable as an inline image here, we have included it at the end of the report. The overall schedule has not changed noticeably with respect to the tasks to do, however there were a number of delays that had to be made.

For the software side, we spent so much time fixing the bandwidth issues on the Pis, that we were delayed up until the demo with a lot of backed up work. Without accurate FPS numbers on the video feeds, it was extremely difficult to work on integration, visualization and concurrent streams. Thus, that work got pushed back as we explored the various transmission formats and bandwidth fixes. Once we were able to get the bandwidth fixed on the Pis, we accelerated intensely to complete integration and visualization for the demo. Unfortunately, due to the lack of dedicated video processing functionality on the ARM core, our system was unable to accommodate multiple streams. However, we were still able to implement them separately, just to demonstrate the real-time, reasonably high FPS that we were able to achieve without the ARM core bottleneck. At the end of the day, we were satisfied with our results considering what was under our control, what was out of our hands, and the numerous delays caused by such intertwined issues.

Hardware interfacing was originally planned to be finished just after the start of the realization phase after spring break, but significantly more time was needed to fully understand how to use Vivado's block diagram, AXI interconnects, DMA cores, and other components to make the interface work. Additionally, the initial start with Vivado HLS was slow, and considering our whole hardware focus relied on significantly accelerating a heavyweight algorithm, we decided to devote more time to this initially. However, this proved to be both good and bad, since the compute pipeline's memory interface was modified a few times since the overall memory interface was not solidified until quite late in the realization phase.

The delays with the compute pipeline stem from two parts: understanding the operations done in each phase of the Canny algorithm, and the learning curve with using HLS. The former was not a significant issue, as there were plenty of resources online that were simple to understand. The slack we had originally put in place for implementing the pipeline absorbed all of this. The latter issue however proved to be quite an obstacle. Our pains with HLS are detailed more in the "Summary" section.

B. Team Member Responsibilities

Broadly, our project is split into two main areas, hardware design and software systems (signal processing is still present through the video processing algorithm theory, but it is not a core focus of this project). As such, our team is composed of three qualified individuals specializing in these two areas. Brandon focused on the software system, while Edric and Ilan on the hardware design and component interfacing and system integration, respectively. Within the software system, Brandon implemented client code on the Raspberry Pis, server code on the ARM core, and the communication protocol between these devices. His work mainly focused on the low-level networking between these devices and ensuring that frame data is efficiently and reliably transmitted. Additionaly, Brandon and Ilan worked together on the revamping the transmission pipeline to use FFmpeg video processing offloading between the ARM core and the FPGA fabric. Edric focused on hardware design, implementation, and verification for the FPGA's Canny compute pipeline. He and Ilan worked together to design the communication between software, hardware, and DRAM and the high-level interfacing for the edge detection algorithm's DSP pipeline. Ilan worked on hardware interfaces at the beginning to support both Brandon and Edric (Wi-Fi, Raspberry Pi bring-up, ARM core bring-up, FPGA bring-up), and transitioned in the middle to support Edric in the implementation and verification of some of the pipeline's stages. Finally, Ilan transitioned back to implementing the memory interface between PS and PL, helping Edric integrate the pipeline into the design and debug any issues, and working with Brandon to switch to FFmpeg for increase streaming throughput from the RPis. Budget

Since most of our project's cost is the Ultra96 board, which is provided by the 18-500 course staff, our project comes in well under the $600 budget with just under $400 remaining. However, it is worth noting that for a commercial implementation the total cost would still likely differ from our bill of materials since the Ultra96 is a development kit and not ideal for commercial applications.

C. Budget

Since most of our project's cost is the Ultra96 board, which is provided by the 18-500 course staff, our project comes in well under the $600 budget with just under $300 remaining. The increase in cost led from our attempts to improve the video stream bandwidth via antennae. It is worth noting that for a commercial implementation the total cost would still likely differ from our bill of materials since the Ultra96 is a development kit and not ideal for commercial applications.

The final, itemized Bill of Materials can be found on the next page.

Item	Source	Model number	Quantity	Unit Cost	Line Item Cost	Provided by Course
Hardware						
Avnet Ultra96	18-500 Staff	AES-ULTRA96-G	1	$249.00	$249.00	x
Avnet Ultra96 power adapter	18-500 Staff	AES-ACC-U96-PWR	1	$8.50	$8.50	x
Avnet USB To JTAG/UART Adapter	Newark	AES-ACC-U96-JTAG	1	$39.00	$39.00	
Raspberry Pi Zero W with Camera Pack	Adafruit	3414	3	$44.95	$134.85	
Raspberry Pi SD Card	Adafruit	3259	3	$9.95	$29.85	
Micro USB power adapter	Adafruit	1995	4	$7.50	$30.00	
Micro USB OTG cable	Adafruit	1099	3	$2.50	$7.50	
USB Micro-B cable	Sparkfun	13244	3	$1.95	$5.85	
Mini HDMI cable	Sparkfun	14274	3	$4.95	$14.85	
Shipping for second order	Sparkfun	N/A	1	$11.06	$11.06	
USB Hub	Amazon	B01BKYM8ZY	2	$8.49	$16.98	
Wifi Dongle	Amazon	B003M0NURK	3	$6.99	$20.97	
Software						
Vivado (FPGA toolchain)	18-500 Staff/Ultra96 license		1	$0.00	$0.00	x
Python 3	Open source/free		1	$0.00	$0.00	
Git	Open source/free		1	$0.00	$0.00	
GitHub	Open source/free		1	$0.00	$0.00	
Budget					$ 600.00	
Total Cost					$310.91	
Total remaining					$ 289.09	

Fig. 6. Project bill of materials

D. *Risk Management*

In terms of software system design, many aspects of our system were flexible and did not carry a large amount of risk. We mitigated packet loss risk by leaving TCP open as an option for the transmission protocol. We expected very little risk in the software component of our system, and thus didn't have a lot in place against this. We unfortunately didn't anticipate the bandwidth issue, since it was such an obscure issue, so it took us a long time to develop a solution. Our hardware component of the system carried much more risk for the following reasons: (1) difficulty of algorithm implementation, (2) longer time-to-completion, (3) fewer support resources. The Canny edge detection algorithm involves multiple stages, with many parts of the algorithm having data dependences on adjacent pixels. This did not lend itself to simple, easy-to-implement RTL implementations in the time allotted. We ended up utilizing HLS, peer resources, and Xilinx documentation to mitigate against the hardware risk overall. Finally, our project had a significant amount of unallocated budget, and so we could easily afford to order replacements or additional parts in case of any issues. This helped with our Wi-Fi antenna purchases as a potential solution to our Pi bandwidth issue.

VI. Related Work

On the software side, we found a paper by Chi-Fai Wong, Wai-Lam Fung, Jack T. Chi-Fai, and S.H. Gary Chan, entitled *TCP Streaming for Low Delay Wireless Video*. This paper debates the usage of TCP over UDP for video transmission, and cites the following reasons against UDP: unreliable complex error handling, network unfriendliness, unselective data loss, and firewall penetration. For TCP, they cite reliable transmission, network fairness, and ease of deployment. We experienced almost all of these qualities for both, and thus concluded TCP as superior similar to the writers of this paper.

On the hardware side, we found a paper by Qian Xu, Chaitali Chakrabarti, and Lina J. Karam, entitled *A distributed Canny edge detector and its implementation on FPGA*. This paper similarly investigates and exploits the parallelism and pipelining of the canny edge detection algorithm, and yielded a 16x decrease in processing time compared to the software implementation using a Xilinx Virtex-5 FPGA.

Finally, we found a paper by Calliope-Louisa Sotiropoulou, Christos Gentsos and Spiridon Nikolaidis, entitled *FPGA-based Canny Edge Detection for Real-Time Applications*. This paper is almost identical to the hardware side of our project, but involved much more optimization by hand that couldn't be achieved by us due to time-constraints. They achieved ~300 FPS while we achieved ~60 FPS on the hardware.

VII. Summary

In the end, our system was able to perform the Canny edge detection algorithm on an incoming video stream, and stream this result to a client. While our functionality requirements were met, our performance requirements were not. Due to the bottleneck caused by the ARM core's lack of dedicated video decoding/encoding hardware we were unable to achieve our success metrics. The camera streams were each 720p30 streams, and we demonstrated that without the ARM core bottleneck we could send three concurrent streams to the monitor computer. On the hardware side, the Canny compute pipeline was able to completely process video at around 60 FPS, far above our target. Moreover the utilization came in at just over 10%, so we are fairly confident that barring the addition of routing logic for multiple streams we can achieve 7-8 concurrent video feeds.

A. *Future Work*

Because we realized our true bottleneck far too late in the project, any future work would likely tackle that issue. With some searching, there is a variant of the chip used by the Ultra96 that has the dedicated hardware necessary for video encoding with 8k streams, but this solution may be unnecessary for the processing intensity of our project. Instead, with more time we would likely look into offloading the stream decoding/encoding to the fabric itself, effectively implementing our own custom hardware.

On the compute side, there is further work to be done in HLS. Referencing the Sotiropoulou paper, clearly it is possible to push the Canny algorithm's latency much further. For now the critical path remains in the arctan and square root functions. No obvious solutions come to mind for this, although there are likely resources out there that discuss better ways to perform these operations in hardware. It is unlikely that the built-in HLS implementation is the fastest. We also received a fair number of warnings in HLS regarding the lack of memory ports, so improving memory management and this memory port limit by duplicating memory will lead to significant improvements. On the memory interface side, we found out that keeping the pixel data in discrete 8-bit values led to a significant amount of overhead in the streams. Packing multiple pixel data in larger blocks will amortize said overhead and improve our memory access latency.

B. *Lessons Learned*

Apart from the usual "start early" advice, we found too late that better initial research on existing designs and protocols leads to less pains in implementation. Knowing what is and isn't possible before development is something that would have been incredibly helpful. This particularly came up when looking into the video stream formatting, as we flipped back and forth between JPEG, h264, and raw video far more often than we should have. Another lesson learned is to commit to a design early and deal with the problems later. When we discussed the different hardware interfaces, we made the mistake of delaying the decision until after the implementation. Once we decided on AXI streams, we then found out that the entire implementation had to be overhauled to adhere to the interface, leading to further delays. Committing to AXI much earlier would have alleviated this. Finally, set aside time for learning a tool set if it is not a familiar one. HLS certainly had its issues when we started using it, and the learning curve proved to be a significant obstacle in the first half of the semester.

REFERENCES

[1] Chi-Fai Wong, Wai-Lam Fung, Chi-Fai Jack Tang and S. -. G. Chan, "TCP streaming for low-delay wireless video," Second International Conference on Quality of Service in Heterogeneous Wired/Wireless Networks (QSHINE'05), Lake Vista, FL, 2005, pp. 6 pp.-41.

[2] Sotiropoulou, Calliope-Louisa & Gentsos, Christos & Nikolaidis, Spyridon & Rjoub, Abdoul. (2011). FPGA-based Canny Edge Detection for Real-Time Applications.

[3] Q. Xu, C. Chakrabarti and L. J. Karam, "A distributed Canny edge detector and its implementation on FPGA," 2011 Digital Signal Processing and Signal Processing Education Meeting (DSP/SPE), Sedona, AZ, 2011, pp. 500-505.

[4] Ultra96 User Guide, https://www.avnet.com/opasdata/d120001/medias/docus/187/Ultra96-HW-User-Guide-rev-1-0-V0_9_preliminary.pdf

[5] SoC/FPGA Guide, https://www.xilinx.com/support/documentation/data_sheets/ds891-zynq-ultrascale-plus-overview.pdf

[6] Zynq Ultrascale Technical Reference Manual, https://www.xilinx.com/support/documentation/user_guides/ug1085-zynq-ultrascale-trm.pdf

[7] DSP48E2 Slice Datasheet, https://www.xilinx.com/support/documentation/user_guides/ug579-ultrascale-dsp.pdf

[8] Wi-Fi Chip Datasheet, http://www.ti.com/lit/ds/symlink/wl1831mod.pdf

[9] Liang, Justin, Canny Edge Detection, http://justin-liang.com/tutorials/canny/

[10] Edge Detection in Python, https://towardsdatascience.com/canny-edge-detection-step-by-step-in-python-computer-vision-b49c3a2d8123

[11] Vivado HLS overview, https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug902-vivado-high-level-synthesis.pdf

[12] Vivado HLS tutorial, https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug871-vivado-high-level-synthesis-tutorial.pdf

[13] Vivado HLS methodology guide, https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_1/ug1270-vivado-hls-opt-methodology-guide.pdf

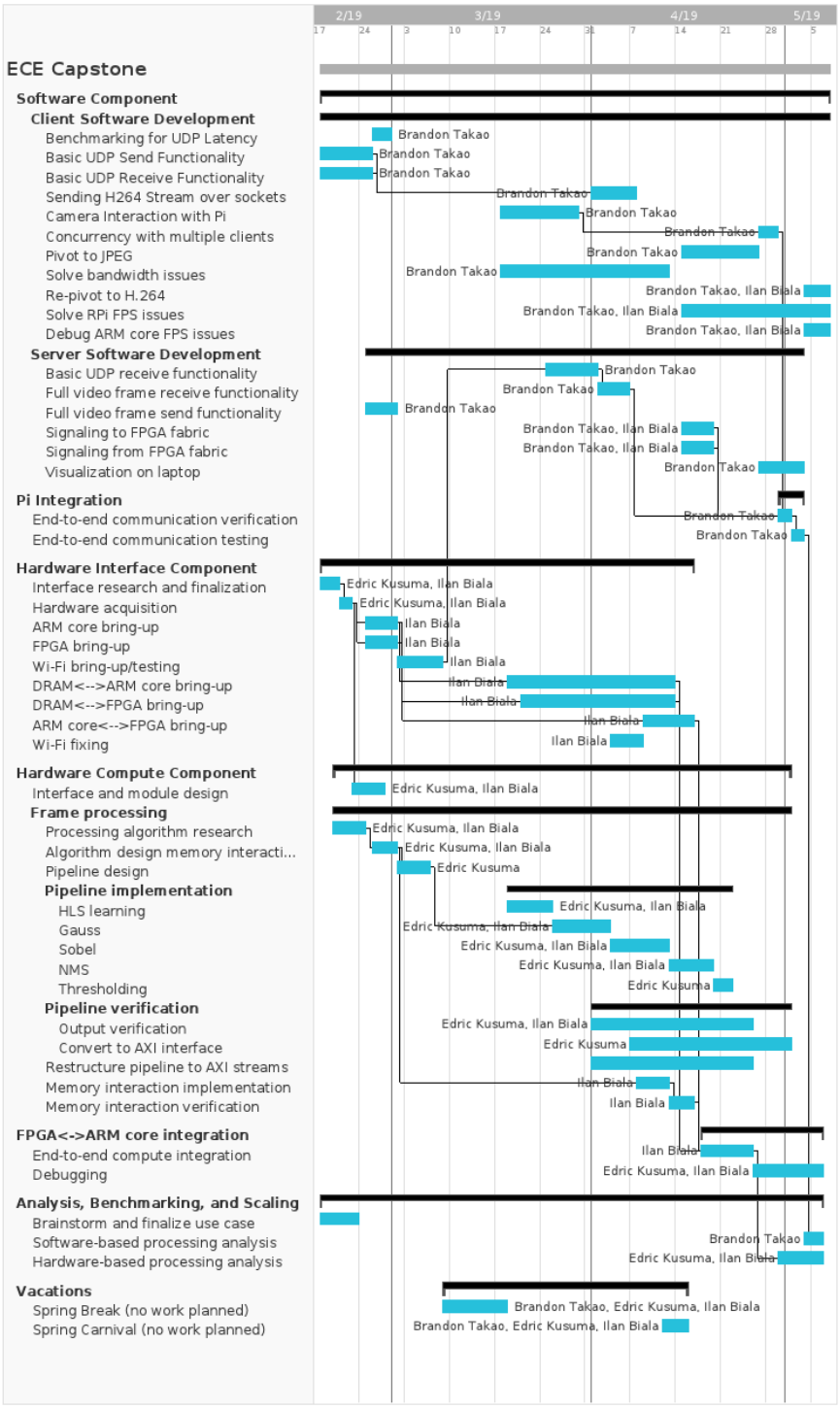

Fig. 7. Final Gantt chart

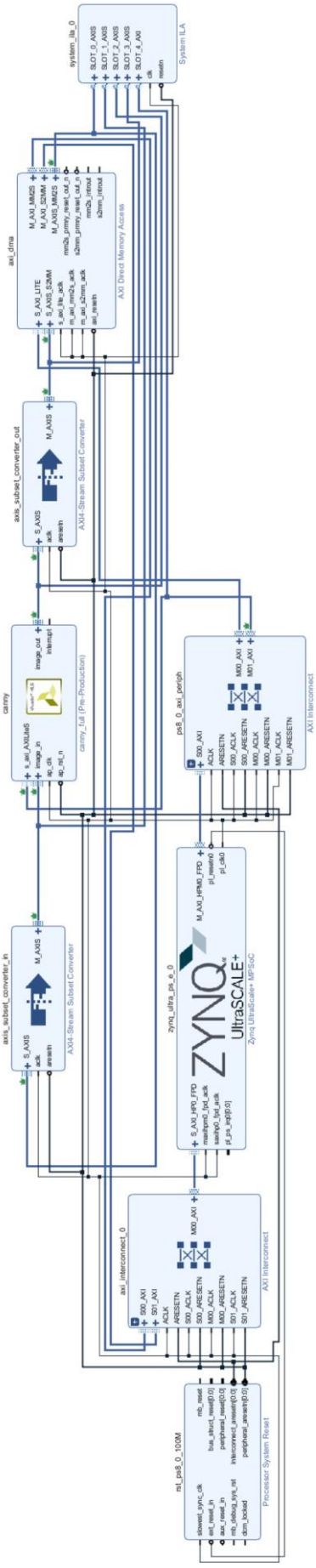

Fig. 8. Vivado block diagram for the Ultra96