Sylvain Bechet, Sakayan Sitsabesan, Mark Yep

Group 4

compsys 701 final report

Object Detection Image Processing System

# Introduction

Real-time digital image processing is an essential application in the computer systems of many different science and technological fields. Some of these applications which rely heavily on image processing include autonomous vehicles, medical diagnostics, and security. A growing proportion of these digital image processing applications are hard real-time systems, which implies that time constraints are increasingly important. This research and development project utilises a hardware/software co-design approach to create improvements to a provided embedded digital image processing system. The digital system is implemented to run on a field programmable gate array system on chip (FPGA SoC), which is an integrated circuit chip that contains both a processor and FPGA logic elements. The improvements which we developed were focussed mainly around the processing speed of the system, with the aim of producing a system as close to real-time as possible. Furthermore, the project is given context as the resulting system should be able to be used as an object detection system.

The embedded development system provided for the project was a Terasic DE1-115 board, with an Altera Cyclone V FPGA SoC. This particular FPGA SoC contains an FPGA section with 85K logic elements, and an ARM Cortex-A9 Dual Core processor. The FPGA section contained an SRAM of 1024 KB. The board also contained 1 GB of hardware processing system (HPS) DDR3 SDRAM and 64 MB of FPGA SDRAM. The board ran Poky 8.0 (Yocto Project 1.3 Reference distribution of Linux) which booted from an SD Card, which also contained the video files for image processing. The Linux system’s shell was accessible via a UART to USB port, while the video output was displayed via VGA on a monitor.

The initial embedded digital image processing system was implemented completely in the C programming language. It involved several sequential stages - decryption, decoding, filtering, and output. These stages are able to be developed and modified independently of one another. To provide structure, our project was split into 3 phases. The first phase focussed on improving the performance of the decryption stage. The second phase focussed on improving the performance of the decoding and filtering stages. Both of these stages involved designing a hardware/software co-designed solution on the FPGA section of the SoC. The final phase focussed on the output and overall integration of the image processing system. In this phase we also added a basic object detection algorithm to our system to detect simple polygons and circles of an input video file.

The development of the improved image processing system was only a small part of the overall project. This is because a large part of this project was undertaking research around the possible improvements we could implement. Multiple improvements for each stage were identified, with a variety of implementation techniques available for each improvement. By researching before starting any development, we ensured that we would only be developing the necessary implementations. This allowed us to compare trade-offs around design complexity, performance increases or decreases, and logic resource allocation. This increased the efficiency of our design and development flow as we were limited by the time we could spend on this project. Additionally, a large part of this project was also invested in the testing and validation of each improved component. For example, every VHDL entity had its own ModelSim test bench so that we could be sure that the individual parts were working as intended. This would aid us in detecting and diagnosing issues before they were integrated into the working solution.

The overall aim of this project was to improve a digital image processing application through extensive research and the development of hardware/software co-design systems. We also added an object detection algorithm to our development to give our resulting system a useful context. The next sections of the report describe and discuss the considerations, research, and development which we undertook during each phase of the project. We also include a discussion which details the overall improvements and characteristics of our final solution.

# Decryption

## Considerations

For the first development phase of our project, we considered different methods of implementing the Trivium stream cipher in a hardware solution. This was necessary as the input files to the object detection system were encrypted via this cipher. A hardware implemented Trivium decryption stage would increase the processing speed of the object detection system since the Trivium algorithm can be transferred to a concurrent system with a fast clock speed. This is because the Trivium stream cipher was originally designed to be implemented in a hardware concurrent application, rather than as a software algorithm.

One of the major factors in our considerations was the communication method from the ARM core to the FPGA system. This was a complicated design decision to make, as both large initialisation values and single cipher stream bits had to be transferred between the hardware and software at different times via the same channel. The communication method needed to be efficient for the 80 bit initialisation values, as well as for the individual bits which the stream cipher would encode. Among the communication methods which we considered were the standard and lightweight AXI Bridges. Another communication method would be to transfer data via the FPGA SDRAM. These options were considered in relation to their trade-offs in latency, throughput, logic utilisation, and difficulty in development.

The AXI Bridge is an ARM interface bus between the hardware processing system and the FPGA. The lightweight AXI Bridge is a version of the AXI Bridge with a lower logic resource usage but with fewer features available. In both of these cases, the 80-bit initialisation values and stream cipher bits would be communicated sequentially between the HPS and FPGA. This meant that a data transfer overhead would be present on each bit of the stream cipher. On the other hand, the FPGA SDRAM is accessible from both the HPS and the FPGA, and hence would be an effective medium to transfer entire encrypted or decrypted files between the two processing components. This is because if an entire file is transferred in a single transfer, then the overall transfer overhead would be smaller than that of sequential streams. The communication method complexity would also be decreased.

## Implementation

Utilising the online Trivium documentation, we implemented the Trivium stream cipher in hardware. As Trivium is already designed to be optimised for hardware, there weren’t any optimisation techniques available for the algorithm implementation. Our resulting hardware Trivium implementation consisted of four components, the two lowest level components consisted of a key setup component and a stream cipher generation component. The key setup component sets up the 288-bit Trivium state from an input key and an input initialisation vector, both 80 bits wide. This state was then passed to the stream cipher generator which outputted one bit of stream cipher each clock cycle. The overall Trivium component controlled both of these lower level components with control signals. The final component was a top level interface between the hardware solution and the software implementation. This interface is displayed in Figure 1. Figure 2 shows the control signals for the interface.

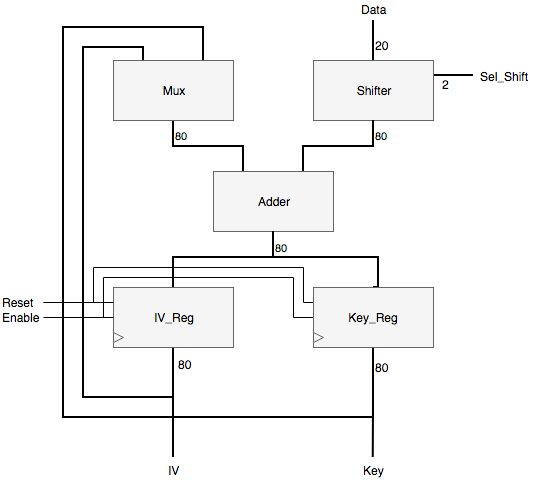


Figure 1: Hardware Trivium Interface

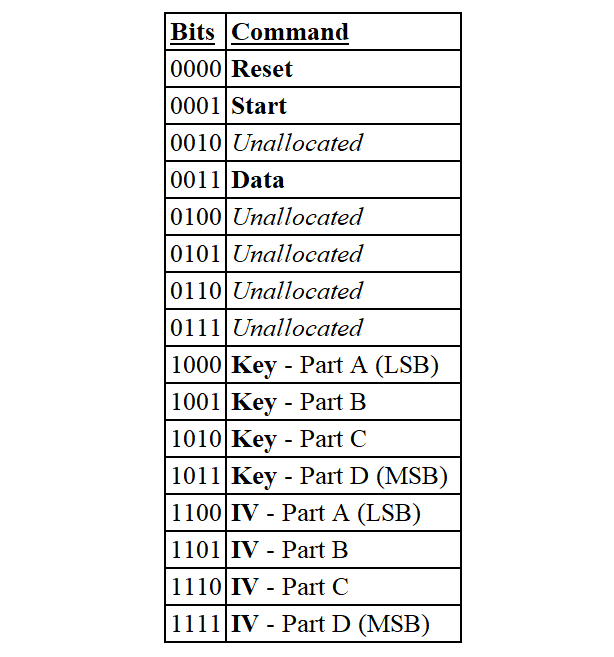


Figure 2: Trivium Control Signals Table

The communication method we chose to implement was the AXI lightweight bridge. We utilise this bridge to transfer data between the HPS and the FPGA. This bridge is lightweight in terms of logic resources, meaning that it only supports single data transfer mode and not the burst or pipelined modes. Due to its lightweight nature, it has a lower latency and throughput than the AXI Bridge. We chose to use the lightweight bridge for its lower latency, which is more useful to us than a higher throughput, as after the initialisation values the data transfer between the HPS and the FPGA would only be a single cipher bit each clock cycle. We decided not to use the FPGA SDRAM to communicate between the FPGA and HPS as this would have added on a significant amount of research and design time for a very small amount of additional performance.

During this phase, we decided not to transfer the decrypted data back to the HPS as we were considering the possibility of keeping the image data on the FPGA for further processing before sending it back. The main reason behind this decision was the fact that a time overhead is created whenever data is transferred between the ARM core and the FPGA. Additionally, the Trivium stage of the object detection system was determined to only make up a very small proportion of the total time overhead of the overall processing time, forcing us to consider whether or not we wanted to include this hardware solution in our final implementation, since it carried a communications overhead. At this point, we considered the possibility of the logic resources which the Trivium utilised to be potentially used for improving our performances in the next stages. This meant that developing a return mechanism would be inefficient and unnecessary at this stage.

After comparing the performances of the hardware and software Trivium implementations, we confirmed that the hardware implementation was indeed faster and showed an improvement of 23%. This percentage does not appear significant because the Trivium decryption stage only accounts for a very small proportion of the total processing time.

## Extensions

By the end of the decryption implementation stage, we decided that the hardware implementation of the Trivium cipher would be used with the AXI lightweight bridge. The hardware implementation of the cipher was chosen over the software implementation as it performed better than its software counterpart. The main drawback behind this decision was that it came at the cost of a small increase in logic element usage, albeit insignificant when compared to the rest of the project.

As the data would be encrypted on hardware, a communication protocol had to be established between the ARM core and the FPGA. The considered options included using the AXI Bridge, the AXI lightweight bridge or the FPGA SDRAM. Both AXI Bridge methods involved data being sent and passed to the FPGA Bridge, the main bridge interface between the HPS and the FPGA. Furthermore, the data could be sent across in multiple manners: through single, pipelined transfer, as well as burst transfer for the AXI Bridge. As single transfer was chosen, the AXI lightweight bridge was the preferred option since it offered the same features as the AXI Bridge but used a smaller amount of logic resources. Although the use of burst mode may have proven to offer better performances, the limited understanding and knowledge around it on top of the small time frame available to develop it discouraged us from using it and hence using the AXI Bridge. The remaining option, the use of the FPGA SDRAM, was not considered for as long as the AXI Bridge, as it had little appeal to us. The first main issue it suffered from was that it appeared quite complex to implement when compared with the AXI Bridges, preventing us from being able to plan how long development for it would last. The other main issue was that, even if it was implemented, it would still be using the FPGA Bridge and may not have offered preferable performances when compared to the AXI Bridges. As a result, the FPGA SDRAM option was quickly abandoned while the AXI Bridge was discarded due to a higher amount of logic resource use, explaining why the AXI lightweight bridge was chosen.

Overall, the insignificant speed increases offered by alternative communication solutions was not worth the research and development time required to implement them.

# Filtering

## Considerations

The ideas that we considered for this phase of our object detection system involved hardware accelerating the filtering stage of our image processing system. The main idea was to move the filtering functions to a hardware FPGA solution on the system on chip. This was considered as the filtering stage is a very significant component of the total MJPEG image processing time. Additionally, the filtering process involves a lot of code structures which could easily be made parallelised, and so would be relatively easy to implement in a hardware concurrent solution. These code structures included multiple nested for loops and repeated matrix calculations. Due to the repetitive nature of these filtering functions, we determined that there could be a significant speed improvement by moving the filtering stage to a hardware implementation.

The other ideas which we considered were methods of implementing a communication method to transmit the frame data from the ARM system to the FPGA hardware solution, and to transmit the processed data back to the ARM system. This would be necessary for integrating the hardware filtering solution into our final system. Several ideas were proposed and researched for this. We could use the AXI Bridge from the HPS to the FPGA to transmit individual pixel frames and run the filtering functions directly on those. Another option with the AXI Bridge could be to transfer an entire frame at a time, and concurrently calculate the filtering functions on the entire buffered frame. Both methods would include writing back to the HPS via the AXI Bridge. A different option could be to utilise the on-chip SRAM or FPGA SDRAM to write the frame to and from the ARM core, written into a specific address. The FPGA could then perform calculations directly on the stored frame. We considered and researched multiple ideas for the methods of communicating because each method had different characteristics and requirements. We had to consider the trade-offs between speed, logic utilisation, difficulty, and feasibility for our specific system.

We also considered and researched several optimisation methods for our hardware accelerated filtering phase. One such idea was that we could utilise NEON vectorisation in our software implementation to compute operations on multiple registers at a time. Another idea was to utilise a pipelined approach to our communication method, so that we could use multiple hardware filtering components concurrently. Both of these methods would take up significant development time but would also offer significant speed improvements to our filtering stage.

## Implementation

For the filtering phase of our project we implemented a hardware solution for computing pixel values, given a filtering window and a filter type. The solution was developed for the FPGA section of the Altera system on chip embedded in our system board, and it communicated with the ARM hardware processing system on the same chip. The hardware solution developed is able to compute various noise suppression and edge detection filtering algorithms, such as Sobel filtering, Gaussian filtering, median filtering, and edge detection filtering. These filters are a necessary intermediate stage before object detection can take place.

The implementation consists of an overall filter interface component, which utilises several different filtering components. These components are designed to compute a new pixel value for the centre pixel of a two dimensional pixel array of three pixels height and width. This is called window filtering, as the three by three pixel window is shifted across an entire frame to compute new pixel values for every pixel, excluding border pixels.

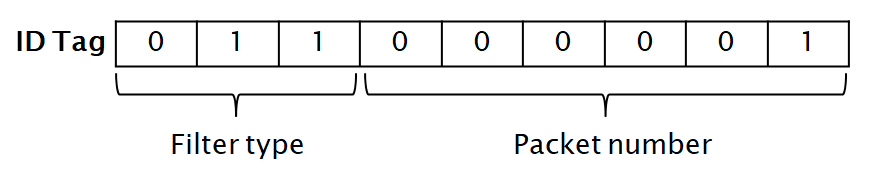


Figure 3: Filter Packet ID Tag Bits

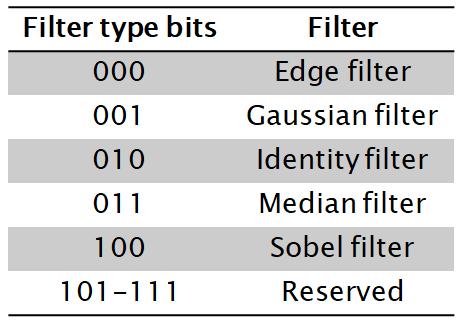


Figure 4: Filter Type Bits Table

During this phase, the filter interface received nine pixels values at a time from the hardware processing system, along with an identification number and a flag which determined filtering type. The ID tag bit structure and type selection is displayed in Figure 3 and Figure 4. In the final integration phase we improved upon this method with a pipelined approach. The received pixels are used as the inputs to multiple different filter components. The filter interface then muxes the output of the different filters, and the filtering type flag determines which filter component output is outputted back to the hardware processing system.



Figure 5: Filter AXI Bridge Diagram

Figure 5 shows the communication of the data between the hardware processing system and the FPGA via AXI Bridges, one in each direction. This introduces some communication overhead as the data has to be sent down to the FPGA and back to the hardware processing system. At this time our implemented system worked in a non-pipelined manner. Each three by three pixel window was sent down to the hardware solution, processed, and then the computed pixel value was sent back - before the next three by three window was sent. This method is inefficient as we were not utilising the concurrency of the hardware solution to its full potential. In the next phase of our project we looked at some of the optimisation methods which we could use to improve the performance of our hardware filtering stage. We decided to implement the communication and integration functionality of our hardware filtering stage before the optimisation methods, so that we could measure performance before and after optimisation techniques to determine their impact on our overall system.

In terms of the software functionality, we replaced the window filter function with a function which sends the three by three pixel window to the input FPGA register, and then waits for the computed value to be written to the output FPGA register.

At this stage we measured a 21% speed increase due to the basic hardware filtering stage. We aimed to improve this value using the possible optimisation methods explained in the next section.

## Extension

The method we chose for communicating the frame between the hardware processing system and the FPGA was via AXI Bridges. The AXI Bridges provided a maximum of transfer 64 bits per clock cycle, which was sufficient for our purpose. Additionally, the data was streamed between the FPGA and ARM components, hence a bus was suitable for this purpose. An alternative option which we could have used would be to store entire frames starting at a specific address location in SRAM or FPGA SDRAM. The FPGA could then access the pixels directly, theoretically leading to a significant increase in performance as multiple concurrent components could access the frame pixels simultaneously. However we chose not to use the SRAM or FPGA SDRAM for this purpose due to a few reasons. The SRAM could not be used as it was not of sufficient size to store an entire 640x480 pixel frame. This is the minimum possible size of each frame of the video to be processed. Additionally, we found that the communication method from the FPGA SDRAM to the ARM core would be no faster than the AXI Bridges, as it would still be using the FPGA Bridge and therefore would not be concurrent.

The optimisation methods which were considered included a pipelined implementation and NEON vectorisation. A pipelined implementation would involve shift registers at the hardware filtering interface so that pixels can be shifted into the interface from the AXI Bridge. This would increase performance and logic utilisation as smaller data packets are sent to the hardware interface - utilising the overlapping pixel windows. NEON is the ARM implementation of advanced single instruction multiple data (SIMD) technology. NEON vectorisation increases the performance of our system as it allows the software to operate on multiple registers in a single clock cycle. This means that loading the pixels from the frame to the AXI Bridge would be significantly faster, decreasing the communication overhead of our system. We chose not to implement these methods for this phase, as we decided that our limited time could be better spent developing the integration method with the ARM core. These optimisation methods would provide significant performance increases, however we preferred to have a fully working solution at this stage so that we could measure performance improvements as we developed.

# Final Integration

## Considerations

For our final integration we considered a large number of possible methods to improve the speed performance of our object detection system. After the previous phase we had an implemented hardware/software integrated system, so the only steps left to do were to optimise and implement object detection.

The optimisation methods which we considered mainly involved improving the concurrency of both our hardware and software implementations. NEON vectorisation would improve the software implementations by allowing the processor to compute a single instruction over multiple data values in a single clock cycle. This would be useful for both the filtering and output stages as multiple colour channels could be operated on simultaneously. This optimisation could be easily achieved using NEON intrinsics.

We also considered daisy chaining the filter components of the hardware filtering solution. This would allow us to sequentially compute multiple filters on each frame without being required to send data back to the ARM core between filter operations. This would significantly decrease our communication overhead if multiple filters would be required before object detection.

Another optimisation method which we considered would be to run some of the software code concurrently on two separate threads, utilising the dual cores of the system on chip. This would be particularly effective for the Trivium and filtering stages, as one core could send data to the FPGA while the other core could receive data. This would significantly decrease time spent waiting for data to be returned before sending a new value.

Finally, we considered the options around implementing object detection on our system. The most relevant library to use would be OpenCV, however it is implemented for C++ while our system had been implemented for C up to this phase. In order to use OpenCV we would have to install the libraries on our system on chip, and modify our existing system to be compatible with C++.

## Implementation

For the final integration phase of our project we introduced many optimisation methods to further improve our overall performance. For each implemented optimisation method, we iterated upon our design and tested after each optimisation to ensure it did not introduce errors into our object detection system.

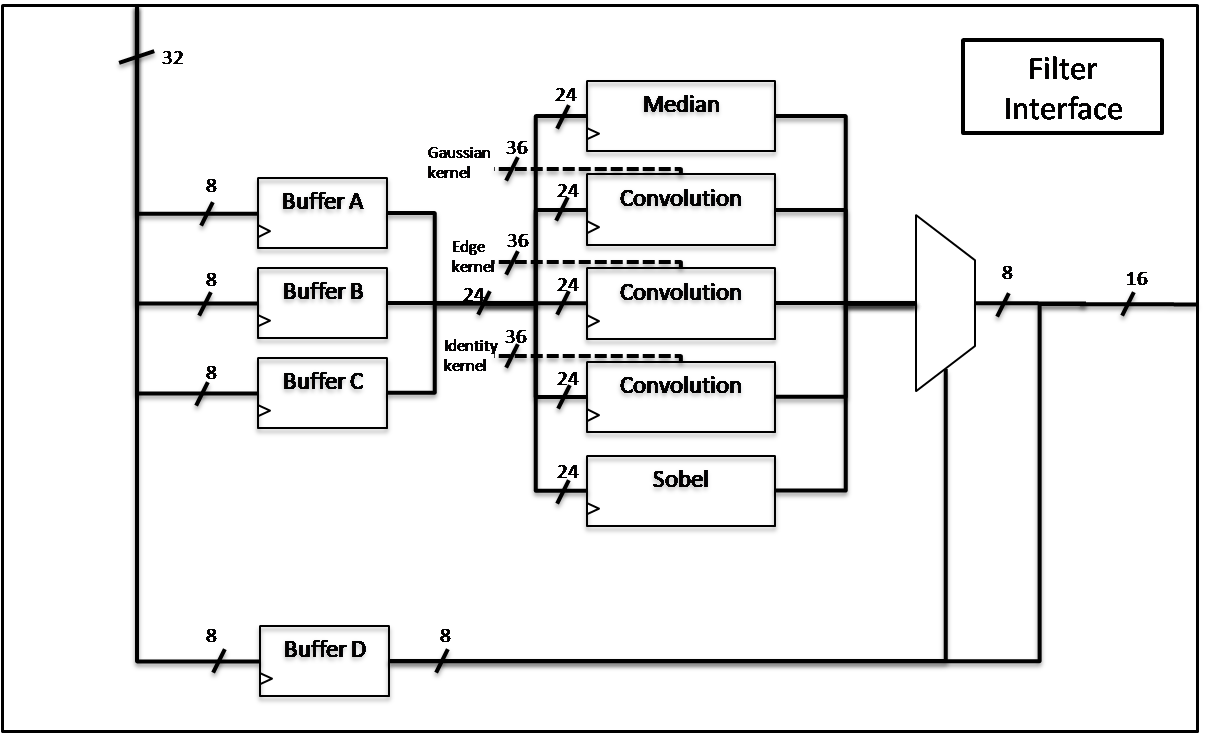


Figure 6: Pipelined Filter Implementation

In terms of our filtering stage, we implemented a pipelined approach to optimise our solution. This is effective because the nine pixel window is shifted along the image, with each new nine pixel window overlapping with six pixels of the previous window. The hardware components of this new system are displayed in Figure 6. Instead of passing in nine pixel values at a time, we utilised shift registers to buffer data being received. In the new communication method, three pixel values are passed at a time into a series of three shift registers as a way of reducing the overall communication overhead. This is efficient because only three new pixel values are required to be sent to the hardware solution each clock cycle - the remaining six pixels are stored in the shift registers. This reduces the communication overhead of our hardware filtering solution. Another filtering optimisation which we implemented on our system was filtering input frames in a greyscale colour model rather than in RGB. This meant that we did not have to perform the filtering on all three colour channels, and instead we only filtered a grayscale layer for each frame. This has some additional overhead in the colour model transformation but significantly increases the performance of our filtering stage.

During the final phase of our project, we also implemented an object detection subsystem in software. This subsystem detects various two dimensional shapes present in each frame, and overlays a text message on top of any detected shape - stating which shape has been detected. The implementation utilised the OpenCV libraries and added on a small overhead to our project. The OpenCV object detection only modifies each frame by overlaying the text layer, no internal image processing is visible on the resulting image. This improves the aesthetic of our object detection system as the user can see the objects on the original image rather than on a heavily filtering image.

The OpenCV libraries also contained libraries for filtering each frame before running object detection. This code was significantly faster than both the provided software implementation and our implemented hardware solution for filtering, due to its optimised single instruction multiple data (SIMD) implementation. Therefore for our final implementation we disabled the implemented hardware filtering solution and computed the filtering using only OpenCV. This resulted in a final system with a speed improvement of 101% over the initial system.

The hardware filtering stage is still able to be enabled or disabled using a definition in the software. This is mainly for display purposes as utilising this hardware stage reduces the speed improvement of our system.

The final optimisation which we made to our system was improving the memory access of the output VGA function. The initial output VGA function accessed the pixel locations in memory by iterating vertically and then horizontally. This increased the probability of processor cache misses as it was not accessing memory locations which are right next to each other. By modifying this function to access pixel location by iterating horizontally first, we greatly increased the probability of cache hits. This is because the processor loads nearby memory locations into the cache after a memory location is accessed. Overall this increased the speed performance of our object detection system.

## Extensions

One of the optimisation methods which we did not implement was daisy chaining the hardware filtering solution. This would allow us to compute multiple filters on each frame in the hardware solution before sending data back to the ARM core. Daisy chaining would be utilised by mapping the output of a noise suppression filter such as a median filter to the input of a edge detection filter such as the Sobel filter. This increases the speed performance of the hardware filtering stage if multiple filters are required. However, we found that the noise suppression filter was unnecessary for computing object detection. Therefore multiple filters during the filtering stage were not necessary. This meant implementing daisy chaining would not offer us any improvement. Additionally, our final system utilised OpenCV filtering instead of hardware filtering, therefore implementing further improvements to our hardware filtering would have been a misuse of our limited time.

Another optimisation method which we did not implement was splitting the C code into two threads at each stage and running each thread on a separate core. This would be efficient as the ARM Cortex-A9 processor on the system on chip is a dual core processor. This would create general speed improvements to our software code. Additionally, by restricting specific threads to run on specific cores we could create instances where this offers additional improvements. For example, during the hardware Trivium or hardware filtering stages we could utilise one thread to send data to the hardware while the other thread receives concurrently. However, we found that we could not implement such a mechanism as the FPGA was operating at a much higher speed than the software code and therefore output data too fast to the ARM core. We could also get resource access issues due to the software concurrency in our system, leading to potentially indeterminate memory access errors. We decided that these disadvantages outweighed the potential speed improvements to our system.

# Discussions

Overall our design performance showed significant speed improvements over the initial image processing system. We implemented a 23% faster Trivium implementation and an 81% faster filtering implementation (run-time reduction), both as hardware solutions - including communication overhead. We increased our system FPGA logic resource usage as the initial system did not have any hardware computation sections.

Our final implemented solution consists of sequential stages. This linear stage arrangement means that there is only one processing path through the system for each video MJPEG file. This file is decrypted and decompressed and then each frame is filtered and object detection is performed on it. Finally the frame is output to the VGA monitor. As there is only one path, this is the critical path as improving any stage on this path will decrease the overall system run time.

During our project development, most of the processing time was being taken up by the filtering stage. In the filtering phase of the project, we aimed to decrease this time as much as possible utilising a hardware solution. This allowed us to reduce the total time taken up by the filtering stage from 807s to 465s. Following further optimisation such as pipelining and grayscaling we were able to reduce this to 150s.

After OpenCV was integrated into our project, we were able to use the optimised filtering functions of this library. This dramatically reduced the filtering time down to 0.13s. This changed the limiting stage of our overall project from the filtering stage to the VGA output stage, which took a total time of 100s. Following processor cache optimisation on the output stage we were able to further decrease this output VGA time to 87s, although it still remains the limiting section of the overall system.

Finally, we added the -O3 compilation flag to the G++ compiler. This further reduced our total run time to 48s (19 FPS). For comparison, the initial total run time was 904s (1 FPS). This shows a speed increase of 1900%. All run times were measured using GPROF.

During development we came across a number of issues and roadblocks. These issues included segmentation fault issues and display issues. Segmentation faults were a major issue when developing the software interface with the AXI Bridge. We eventually worked out that we had not initialised some of the pointers which pointed to hardware registers, hence we were attempting to write data to random locations. Display issues occurred in hardware, as our software code was outputting entire frames to the hardware to be written onto the VGA monitor. However the provided hardware VGA interface had some faults which caused only half of the image to be written correctly to the screen. This was fixed after some debugging time.

# Conclusions

The aim of this project was to ameliorate the performance of a given image processing system. The chosen metric to improve on was the overall runtime, in order to be able to maximise the number of frames per seconds achieved. Through a series of research and development phases we significantly improved the speed performance of the initial image processing application. We were also to add a shape detection algorithm to the application to provide additional context and purpose to the project. Through a mix of research, lectures, and development, we believe to have furthered our skills and knowledge relating to hardware/software co-design, particularly for image processing applications.

# Acknowledgements

We would like to thank Morteza Biglari-Abhari, Zoran Salcic, Benjamin Tan, Tyrone Sherwin,

Nicholas Harvey-Lees-Green and Maryam Hemmati for their assistance and support throughout this project.