Midterm Progress Report
CS 462 - Senior Capstone

February 9th, 2016

The HawkEyed Crew - Group 4

Hailey Palmiter

Ryan Kitchen

Scott Griffy

Abstract

This document is a midterm progress report that covers a brief introduction of our senior project, our current progress, problems that have impeded our progress, and any preliminary results that have been gathered. It also describes what is left to do before all of the requirements are met for the project along with the potential for some stretch goals if time allows. This report includes a few interesting pieces of code and a description of what this code does. It is constructed using the IEEEtran style guidelines.

Contents

I. Introduction

Rockwell Collins, our project sponsor, designs video systems for pilots to use during flight. Pilots often used enhanced imaging to help them see better in rough weather conditions, and to generally assist during different flight operations, such as landing. Some of these enhanced imaging functions could be overlaying a pilot?s view with graphics to help the pilot locate a landing strip in a storm, or automatically turning a night-vision camera off when it is not needed to save power. The hardware used for this system must limit the power use it pulls from the plane, and also be lightweight in order to have the least affect on the plane it can. Even five extra pounds during a year can add up to thousands of dollars in fuel costs. In order to provide pilots with this specific low power, low weight, and enhanced image processing, Rockwell Collins develops software on Field Programmable Gate Arrays (FPGAs), which makes the code very complex and very costly to develop. New vision enhancements can take weeks or months to develop on the FPGAs. The FPGAs are currently Rockwell Collins only option that meets the requirements needed to create practical systems that pilots can use effectively.

Our goal is to provide an alternative to Rockwell Collins' FPGAs that has the needed performance metrics while providing faster implementation time and reducing the cost of production. Specifically, we are designing a proof of concept using single board computers (SBUs). Single board computers differ from FPGAs because they have a standardized execution environment which allows simplified code to be executed on it, reducing the development time. SBUs also use low cost hardware and don't consume much power. If we can prove that the video quality produced by SBUs is adequate for pilots to use, SBUs will meet all the requirements needed to develop a practical vision system on. In the air, FPGAs often take feeds from multiple cameras and run a lot of processing algorithms on them, so in order to be effective, SBUs should be able to do this as well. Our project aims to test and measure the capabilities of a single board computer by delivering a multiple-stream video display which has been processed to a high degree. We believe the best candidate for this is to use NVIDIA's single board computer, the Jetson TX1. Our goal is to fully test the Jetson TX1's ability to provide enhanced imaging. This proof of concept will result in measurements that will help Rockwell Collins determine the practicality of using single board computers for their vision systems.

II. Current Progress

A. Refined Requirements Document

At the beginning of the term, we revised our requirements document. The revision was needed in order to refine our project with the discovery of problems/improvements. The changes were reviewed with our sponsor and agreed upon before any final changes were made. The most current requirements document was uploaded to the SharePoint as Revised_ReqDoc on January 15th.

Only a few minor things were changed, but they had a big impact on how we define our deliverable. Our client was very happy to see the revisions we made and all revisions were agreed upon. One of these changes was making sure we have two working cameras for our system and allowing us to use only the Jetson TK1 or the Jetson TX1. As we move forward we plan to rely more heavily on the TX1, but want to allow ourselves the room to be able to fall back on the TK1 if need be to complete our project. Either of these single board computers would meet the requirements of our client.

B. Jetson TK1

We have installed graphics drivers on the TK1 as well as a driver for the original PointGrey camera we were loaned by Rockwell Collins, but had a lot of difficulty working with the USB 3.0 port, due to kernel support for it being disabled. We found that the Jetson TK1's operating system had to be recompiled in order to grab full 2048x2048 pixel images from the camera. Without the recompiled kernel we were able to grab images at a lower resolution using the FlyCapture API, and could only utilize the PointGrey camera as a USB 2.0 device, not USB 3.0. Even after the recompile of the kernel we still had connection issues and display latency with the video. We quickly began to realize that development on the TK1 may not even be worth the struggle, and have moved our development process to the TX1.

It would be possible to continue development on the TK1, but it does not seem to be the best option to obtain the greatest results moving forward with development. If time permits we will test our software system on the TK1 to measure its capabilities in comparison with the Jetson TX1.

C. Jetson TX1

After discovering a manufacturer error on the original TX1s, we continued development on the TK1 until the new, fully functional, TX1 arrived. This was the point when development shifted from the TK1 to the TX1.

1) Video Output: Our first step exploring the Jetson TX1's capabilities was getting a working video output from the onboard camera. We were able to display this image at 1080p with the option of 30 or 60 frames per second. This video stream used embedded GStreamer code to transform the image buffer generated from the camera into a Video4Linux compatible format, which is a widely compatible format. The image was then read with a standard linux application and displayed to the monitor. After we were able to get the video output from the PointGrey cameras working we implemented two basic edge detection functions using several of OpenCV's built-in functions to gain familiarity with the OpenCV API.

With further development in OpenCV we found that our visual frame rate was not matching the metrics that were being displayed to the screen. We were capturing and processing images at 30 frames per second, but that was not the resulting visual display feed. The output stream had significant lag. Once we found that the bottle neck was happening when frames were being copied from the CPU to the GPU we knew we needed to change out approach. We have left the capture and processing system in place, but simply replaced the way we display the images, giving us more control over the display process. We have changed development to a low level API that allows us to directly manipulate the frames on the GPU. This API is called Cuda, which is a window system based on OpenGL that allows us to have zero copy. Zero copy is implemented using GLUT, which allows us to write the camera images directly to a buffer that is read by the GPU because the GPU and the CPU share memory. We now have eliminated the need for slow copy procedures, resulting in multiple camera inputs stitched together into one output stream with very high frame rate capabilities that are consistent. Currently we are outputting at roughly 35 frames per second with one camera, 33 frames per second with two cameras, and 31 frames per second with three cameras. The video quality is much better than when using OpenCV and actually appears to be at the output frame rate that is being displayed.

2) Hardware: Our next focus was on getting two cameras' video streams displayed to a monitor. We prioritize this before moving on to fixing the quality of the output or adding custom filters. We decided that using two USB 3.0 cameras would be the best way to progress in order to accomplish alpha release goals. But first, we had to get the USB 3.0 PointGrey camera that Rockwell Collins had supplied for us working. Once the camera was working, we were able to display the stream from the PointGrey camera on our TV. We were even able to apply some built-in filters from OpenCV. Now that we had this capability, we discussed the need for additional cameras that our sponsor, Carlo has offered for us to use.

Our next major goal after getting the working TX1 was to get a second camera. This would allow us to be able to begin developing our software to handle processing of multiple video streams into one output. Multiple cameras are a requirement that is needed for us to be able to have a complete deliverable at the beta release. We were able to get in contact with Carlo, and set-up a meeting allowing him to (very generously) loan us two more PointGrey cameras with additional lenses as well. A current list of all of our hardware can be found below.

Cameras (figure 1):

- GS3-U3-41C6NIR-C (PointGrey/GrassHopper)
- GS3-U3-41C6C-C
- GS3-U3-23S6C-C

Lenses (figure 1):

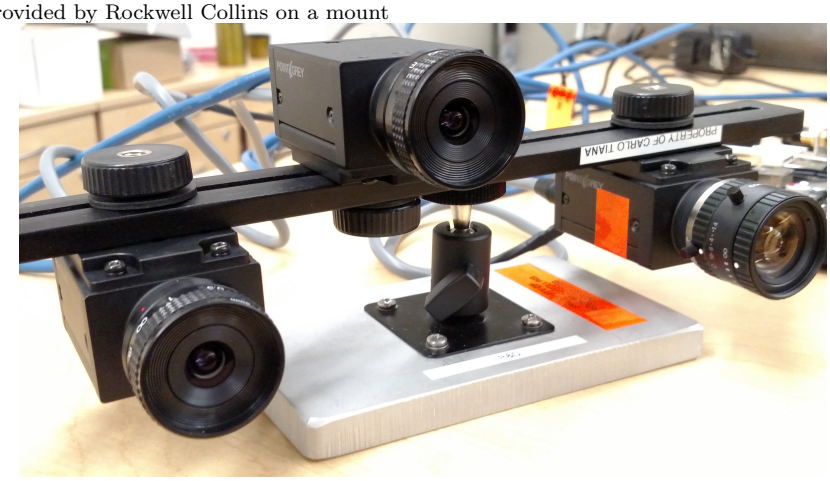
- EVS-3000
- 2 x LS-TP-08 (Standard)

Single Board Computers:

- Jetson TK1 (figure 2)
- Jetson TX1 developer kit with onboard camera (figure 3 and 4)

3) System: During the wait of obtaining two more Point Grey cameras, we implemented a full modular software architecture for our vision system. Our software reads a JSON configuration file to set-up the modules. This allows for more flexibility with the use of different cameras and their software. It also gives us the ability to implement

Fig. 1. The 3 cameras provided by Rockwell Collins on a mount



different filter modules and only apply the ones we want at a specific time. It could handle more flexibility with the use of different types of cameras only being used at specific times or for specific reasons. For example, a long range camera may only be activated when needed to find the runway from a given distance. It could then be turned on and processed when needed, but deactivated to increase the performance when it is not needed.

Fig. 2. The Jetson TK1

Fig. 3. The Jetson TX1

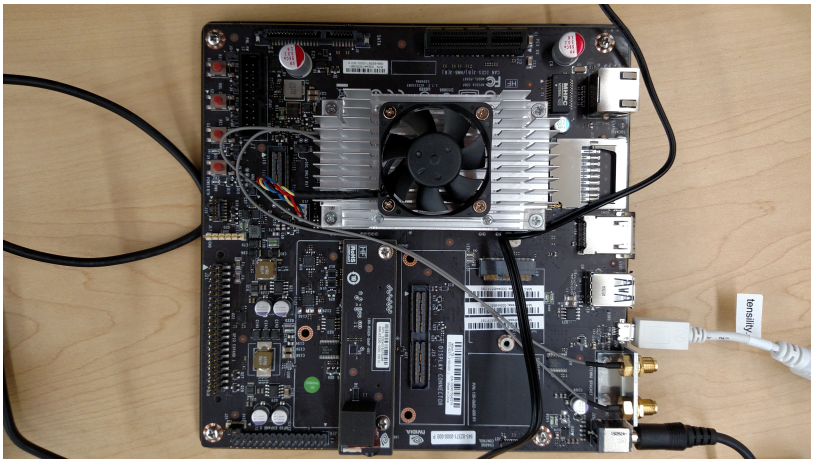
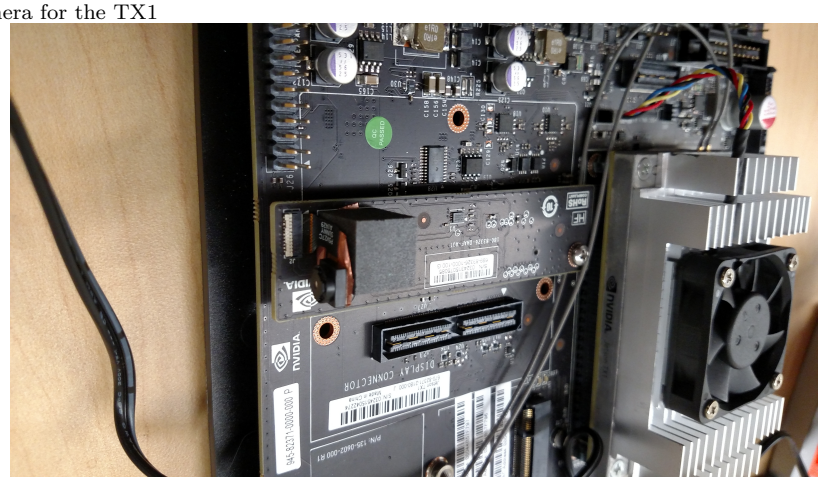


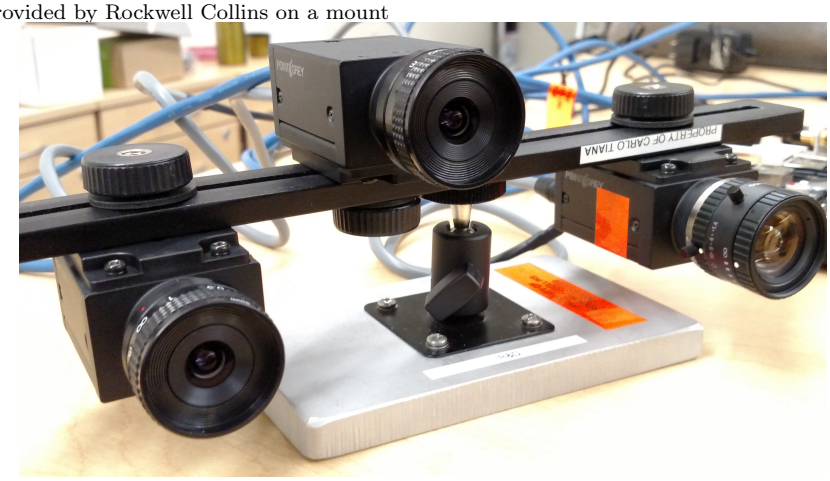
Fig. 4. The on-board camera for the TX1



Extras:

- 2-port USB 3.0 supersede PCIe Card (figure 5)
- Camera mount
- 3 x USB 3.0 cables
- 1080p monitor (TV)

Fig. 5. The 3 cameras provided by Rockwell Collins on a mount



4) Modular Software System: While we were waiting for two more Point Grey cameras, we implemented a modular software architecture for our vision system. Our software reads a JSON configuration file to set-up the modules. Each module is comprised of a series of camera captures, filters, and sinks. This module system allows for flex without the need for recompilation, which can be complex and time consuming. It also gives us the ability to implement different filter modules and quickly apply only desired filters. We have plans to expand the system so that it could intelligently use different types of cameras only for specific reasons to conserve power consumption. An example of this is using the normal camera until the image becomes too dark to see, turining on a night-vision camera. Another example could be that a long range camera may only be activated when needed to find the runway from a given distance. It could then be turned on and processed when needed, but deactivated to increase the performance when it is not needed. We also want to implement multiple camera overlay. An example of this system is having one camera capture the normal image while an infrared camera is detecting bright spots (i.e. runway lights). Once the camera filter has detected the bright lights, it then draws circles around each spot to make a clear indication of where a runway is. Then this image will be overlaid onto the normal image to give the pilots the best possible accuracy and location of the runway. The impact of the flexibility of this modular system is limitless with being able to assist is safe flight travel in almost all situations.

Fig. 6. The Jetson TX1

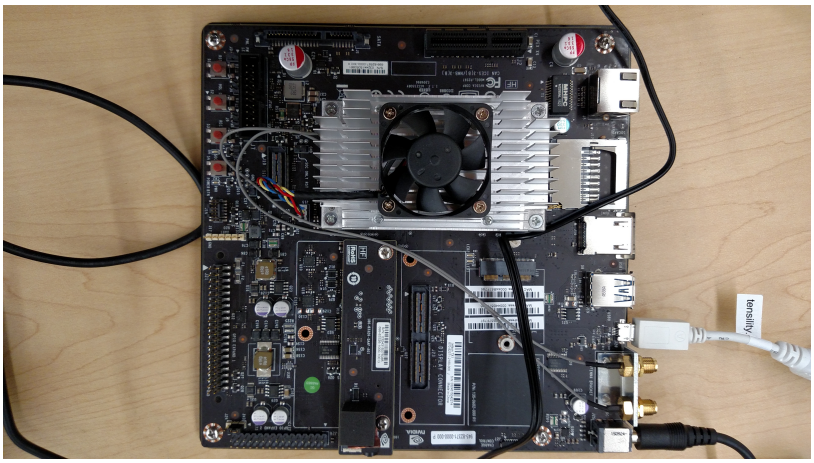
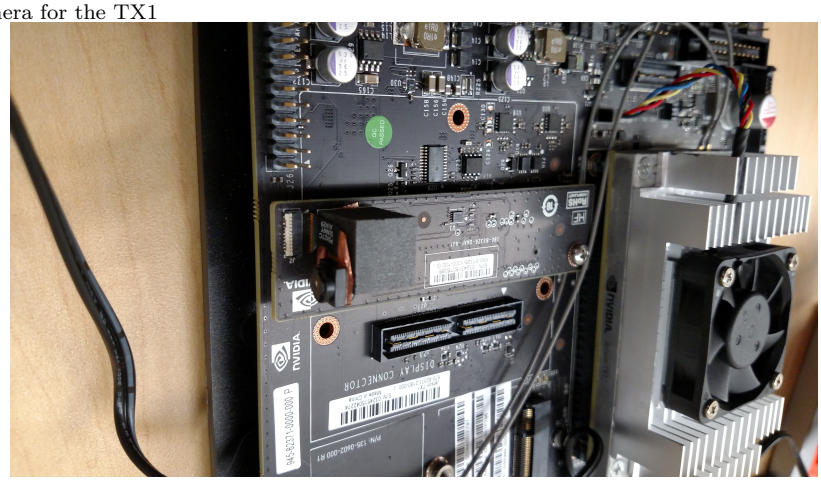


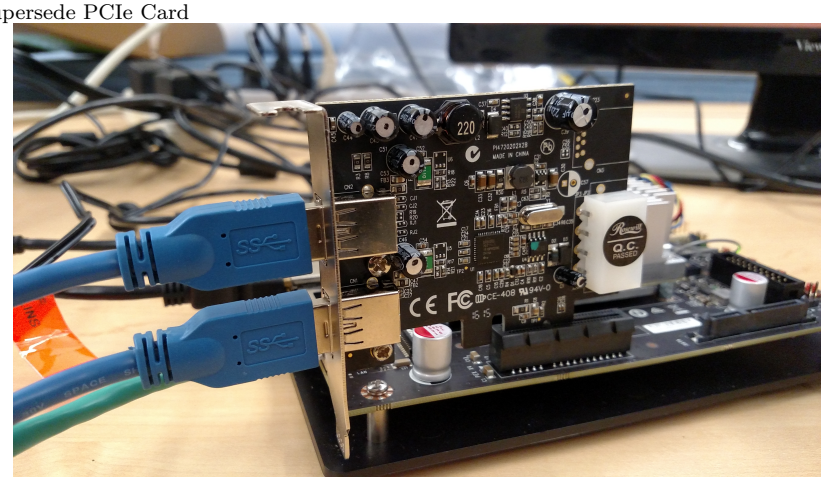
Fig. 7. The Jetson TK1

Fig. 8. The on-board camera for the TX1



We first explored the modular system by first being able to output a static image demonstrating that the system was working. We then created a module to handle input from the PointGrey camera by implementing its given software. This allows us to be able to interact with multiple PointGrey cameras. Now that we have received the USB 3.0 PCIe card, we have been able to implement up to three cameras at once over USB 3.0, all overlaying their video streams on top of each other to provide one output stream. Currently we have it set up so that one camera is providing 50% of the display output and the other two are split by 25% each. When only two cameras are running, the video output is provided with a split 50% input from the cameras equally. We have also added line detection

Fig. 9. 2-port USB 3.0 supersede PCIe Card



filter, which can run without dropping our frame rate any significant amount. The ability and ease of being able to switch between how many cameras are running and applying filters to each has proven the efficiency of our modular system.

III. Problems That Impeded Progress

A. Jetson TX1 Kernel Recompile

We found that the Jetson TK1's operating system had to be recompiled in order to grab full 2048x2048 pixel images from the camera. Without the recompiled kernel we were able to grab images at a lower resolution using the FlyCapture API, but we needed to obtain the higher resolution to match our requirement of 1080 pixels at 30 frames per second. We had to recompile the TK1 kernel with a special version of Linux4Tegra, named "Grinch," to get the output from the USB 3.0 PointGrey camera. The recompile proved to be a difficult task and we still have some connection and display issues with the video stream. Despite these setbacks, we were able to get some video output.

B. Jetson TX1 Manufacture Error

Development on the TX1 began slowly as we found that our first TX1 had a manufacturing problem, preventing it from connecting to wifi and causing unacceptable system lag. We notified our instructor who gave us another board that also turned out to be defective in the same way. We then had to return the TX1 we currently had, and order a new board. In the meantime we were working with the TK1, until the new fully functional TX1 arrived. When it arrived and had been tested to detect that the manufacturer error wasn't present, we were finally able to switch development from the TK1 to the TX1.

C. Jetson TX1 and PointGrey Camera Compatibility

Once we were able to interact with the USB 3.0 PointGrey camera and the Jetson TX1, we ran into the issue of getting the TX1 to register the camera and to display an image. We were finally able to get a video output from the PointGrey camera after realizing we were missing two big elements. To fix the issue, we needed to recompile the Linux kernel to increase the USB buffer size to accommodate for higher resolution image capturing from the camera, and acquired the proper camera software needed from PointGrey directly to interface between the proprietary camera API and OpenCV.

D. OpenCV Limiting Frame Rate

After further development in OpenCV we found that our visual frame rate was not matching the framerate that were being logged. The logs were saying we were outputting 30 fps, but the video was obviously less than that. The output stream had significant lag. We determined that this lag was due to a bottle neck when frames were being copied from the CPU to the GPU. Fixing this in OpenCV would've taken some serious investigation into OpenCV's code that we didn't have the time for. So, we left the capture and processing system in place, but simply replaced the way we display the images, giving us more control over the display process. We have changed development to a low level API that allows us to directly manipulate the frames on the GPU. This API is called Cuda, which is well

supported on the Jetson TX1. We also used GLUT, a window system based on OpenGL. Using these two low level libraries to replace OpenCV allows us to use zero copy. Zero copy means capturing the camera images directly to a buffer that is read by the GPU because the GPU and the CPU share memory. We now have eliminated most of the need for slow copy procedures, resulting in multiple camera inputs stitched together into one output stream with very high frame rate capabilities that are consistent. Performing image format conversions in-place on the GPU will improve performance further.

IV. Preliminary Results

Our project requirements specification describes specific latency, frame rate, and resolution requirements in order for our system to be useful. Our end-to-end latency requirement is that it should take less than 100ms to capture video from the camera, process it, and display the outputs. Additionally, we are required to process a minimum of 30 frames per second at 1080p resolution. Our goal is to perform as much image processing as we can while maintaining these performance metrics.

In order to measure the latency of our application, our software contains a detailed, built-in performance logging system. Each component of the software is separated into polymorphic, class-based HawkEye modules. Each individual module measures execution time including the time it takes to transfer the results to the next module. The overall latency for each frame is determined by the time it takes for all modules to finish executing from input to output, counted as the sum of the longest path. It is important to measure each individual module's latency as well as the overall latency because not all cameras are going to be subjected to the same filtering algorithms, and the additional information will enable us to predict the outcome latency based on the set of modules used. In order to measure the frame rate, we simply count the number of frames processed in a second.

Our goal is to support independent image processing of three separate inputs. Rockwell Collins has given us some challenges outside of our requirements which we will try to implement. We have multiple types of cameras with different lenses: two standard RGB cameras and one NIR (Near infrared) camera. The IR camera will be used to identify high intensity point lights, which we will then overlay onto the processed input from one of the RGB cameras. In the industry, this would be used to identify runway lights for landing aircraft. The third camera would provide a more focused look at the target. The first performance benchmark we need to determine if this is even possible with our hardware, is to simply get three simultaneous camera feeds running at the full 30 frame per second.

Frame Rate			
	One Camera	Two Cameras	Three Cameras
USB 3.0	37 FPS average		
USB 2.0	11 FPS average	10 FPS consistent	6 FPS

Total Latency			
	One Camera	Two Cameras	Three Cameras
USB 3.0	16-30 ms		
USB 2.0	26-45 ms	38-56 ms	42-67 ms

Note: One-camera usb 2.0 results are shown for comparison purposes.

Our initial results are promising, with a single camera on USB 3.0 meeting our requirements for both FPS and latency. However, due to having only a single USB 3.0 port, our current multi-camera frame rate is much lower than the required specification. Our application is multithreaded and should not suffer full delays from additional cameras, but due to having two of the cameras using a single USB 2.0 port the delay scales linearly with each additional camera. Once we get our USB 3.0 expansion card, we should be able to significantly increase the frame rate and reduce the latency to approximately the one-camera, USB 3.0 level.

An additional source of latency is the output windows. Currently we are running three separate windows for the three separate cameras, however to view all three at once, they must be scaled down (as currently we are running the inputs at 1080p or higher resolution). This is a processing intensive action, and can take up to 15ms to complete. Our final version is planned to combine multiple inputs and overlay them onto each other, so this operation will not have to be performed multiple times.

A. Implications of Preliminary Results

Our preliminary results leave us with plenty of leeway as far as latency goes. We having taken only about one-fourth of the maximum total latency, however our frame rate is less than optimal. In order to increase our frame rate, we will improve our modular system to function as a buffered, pipelined system that can begin processing a frame while the previous frame is still being processed. We can do this because our latency is well below the required level, and our CPU usage is not yet at its peak. If we can begin loading a new frame while the previous frame is being processed, we will be able to maximize our usage of the USB 3.0 connection to copy new frames into memory. This is currently the slowest process and also provides the most room for improvement, as we are not yet even close to the maximum USB bandwidth. We hope to be able to maintain the same 16-25 ms latency while processing multiple frames from each camera at the same time, though if the latency increases even by doubling we will still be well within our maximum latency limit. We have already tested and seen that performing the basic operation of a colorspace conversion on a frame (which requires operations to be done on every single pixel) adds less than 2ms of latency, so we believe that we will still be able to do quite a bit of image processing within our 100ms time budget.

V. What is left to do

A. Bugs

We currently have a few bugs in the system, causing strange output or system failure.

- System freezes during program exit Our system doesn't uninitialize the hardware properly and ends up freezing up the hardware. This shouldn't be difficult to fix.
- Camera output is split Another issue while quitting our application is improper deinitialization of the cameras. This can cause cameras to act strangely on restart, such as partitioning their image output in half before sending it to the screen.

B. Stretch Goals

If time permits, we should have the ability to implement other functions. We have planned some of these out and are considering them as stretch goals. Some of these we have already or are close to meeting:

- Improve the framerate of our system (above 30fps)
- Display our video feed on more than one monitor.
- Perform an object tracking filter.
- Design a real-world model to test our cameras on, such as with miniatures (two of our cameras have macro lenses which would be perfect for miniatures)
- Display video operation metrics on screen. We currently log these to a file, but it would be easier to demonstrate the capabilities of the SBC if we displayed these on the video stream.
- Compare the Jetson with different single board computers. This would mean re-implementing our code on different platforms and comparing their processing speed and hardware capabilities.

VI. Interesting Piece of Code

The core of our program is the modular video processing pipeline. This polymorphic object-oriented system allows us to easily implement new video processing algorithms, support for different types of input devices, and new methods of outputting results. It is also responsible for measuring performance of each individual component.

Each module is defined in its own class, which is a child class of HawkEye_Module. Each module has two required parts, the constructor and the run() command, as well as an existing setup() command which can be overloaded as needed.

```
1     class MyModule: public HawkEye_Module{
2   public:
3     MyModule(){
4       numinputs = 2;
5       numoutputs=1;
6       setup();
7     }
```

The initializer MUST set the number of inputs and outputs. This is extremely important because the setup() function creates the appropriate buffers and sets some internal variables needed by the module class to determine when the needed inputs have been satisfied (and thus when to execute).

```
1     void run(){
2     *outputs[0] = *inputs[0]/2 + *inputs[1]/2;
3   }
```

This is an example of what a simple module might do. The HawkEye_Module class defines two arrays of pointers to OpenCV Mat objects, which are a kind of smart matrix/image object. This code would take two images and combine them together by a simple averaging method. OpenCV allows mass mathematical operations to be performed on all elements of a matrix, and allows adding two identically sized matrices together via simple operations as shown above. Other algorithms may require more complex operations, but this shows how simple an implementation can really be.

Once such a module is defined, it can then be added to the parser. Currently this is a hardcoded list of module names which checks an input type name string against the predefined types and creates the associated module, however we have plans to implement a dynamically-loaded module system that allows adding new modules automatically via a dynamically linked library. After the module is added to the parser, instances can then be created in the JSON configuration file. In order to create an instance, an entry is added to the "modules" section of the config file.

A JSON configuration for our program consists of three sections: The "modules" section, which defines named instances of HawkEye_Modules, the "start" section, which tells the program to start a camera or input, and the connections section, which defines where the output(s) from each module goes to (as a module could have multiple outputs, and some modules require multiple inputs).

```
1     {
2     ''modules'':{
3       ''MyModule_instance'':''MyModule'',
4       ''inputModule'':''camera'',
5       ''outputModule'':''basicOutput''
6     },
7     ''start'':''inputModule'',
8     ''inputModule'':{TX1
9       ''output0'':''MyModule_instance0''
10    },
11    ''MyModule_instance'':{
12      ''output0'':''outputModule0''
13    }
14  }
```

In this example, we define two camera inputs, a combination algorithm module (as defined above in the previous code example), and an output window module. The start section tells the program that it needs to run both camera inputs, which it does separately. The code immediately after the start section tells the program to connect output 0 of inputModule1 to input 0 of MyModule_instance, output 0 of inputModule2 to input 1 of MyModule_instance, and the output of MyModule_instance to the display module. When the program is executed, it will capture input from both cameras, combine it using MyModule, and then display it to a window. This configuration system makes it easy to implement complex networks of algorithm modules, and makes it so that any reconfiguration of the modules does not require a recompile of the software. This also defines the names used by the logger, in this case inputModule1, inputModule2, MyModule_instance, and outputModule. These names can be anything the user decides to define, as long as they are .

Fig. 10. Video output from the TX1 using a PointGrey camera with an EVS-3000 lens attached

Fig. 11. Video output from the TX1 using a PointGrey camera with a LS-TP-08 lens attached