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 beta level release progress report that covers a brief introduction of our senior project, our current progress, problems that have impeded our progress, and the preliminary results that have been gathered. It also describes what is left to do to increase our projects performance and/or more concrete testing of our software system before it is presented at the Engineering Expo, along with the potential for implementing any of our stretch goals before then as well. This report includes a few interesting pieces of code and a description of what this code does to improve or enhance our system. It is constructed using the IEEEtran style guidelines.

Contents

I. Introduction

Rockwell Collins, our project sponsor, designs video vision systems for pilots to use during flight. Pilots often use these enhanced image systems to help them see better in rough weather conditions, and to generally assist during different flight operations, such as landing. An example of this functionality 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 consumption it pulls from the plane, and also must be lightweight in order to have the least affect on the plane it can. Even an extra five pounds added to the plane 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). This piece of hardware 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 (SBCs). 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. Single board computers also use low cost hardware and don't consume much power. If we can prove that the video quality produced by the SBCs is adequate for pilots to use, they 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 those images. In order to be effective, the SBCs should also be able to handle this operation. Our project aims to test and measure the capabilities of a single board computer by delivering a multiple-stream video display that 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. Hardware

This section includes a list of all the hardware we have been (very generously) leant and have been working with for the duration of this project. The following list and images should be able to supply a better understanding and familiarization of the project.

Cameras (figure 1):

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

Lenses (figure 1):

- EVS-3000 (figure 6)
- 2 x LS-TP-08 (Standard) (figure 7)

Single Board Computers:

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

Extras:

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

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

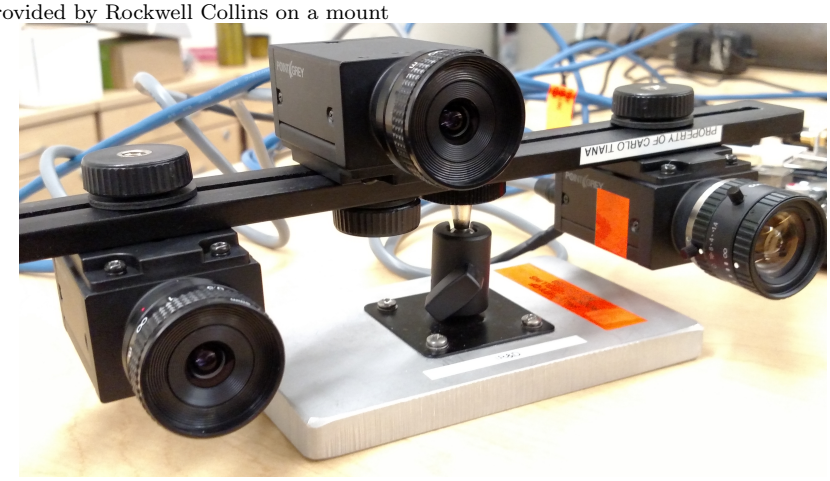
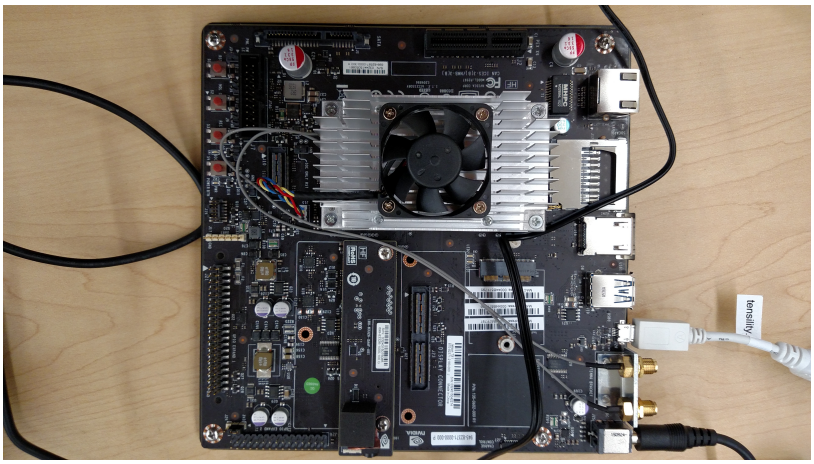


Fig. 2. The Jetson TK1

Fig. 3. The Jetson TX1



III. Current Progress

A. *Refined Requirements Document*

Before real development began, we had reread our requirements document and noticed that some changes and additions needed to be made. The revision was needed in order to refine our project with the discovery of problems/improvements we had found. It was a needed process as we wanted to have full clarity on what needed to be accomplished with this project, and be able to display those requirements with our sponsor. The changes were

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

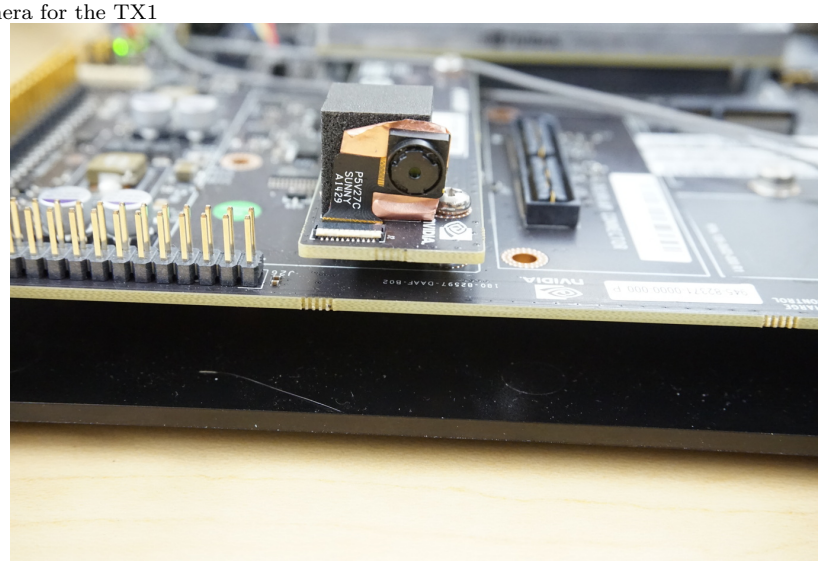
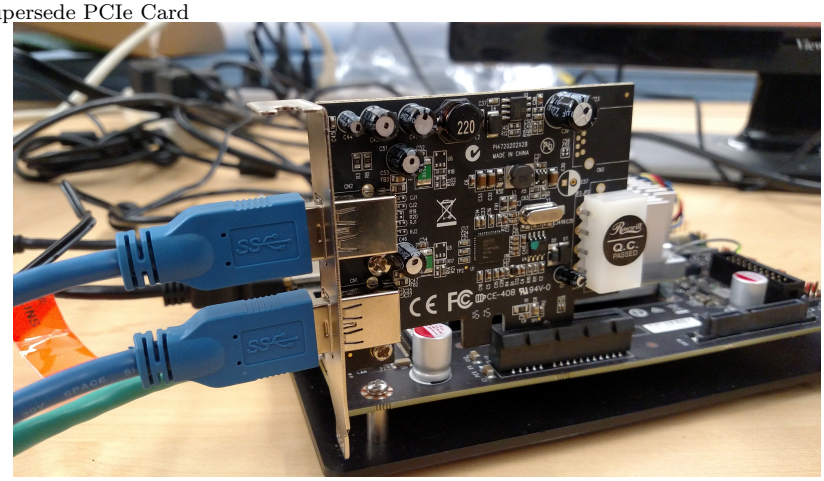


Fig. 5. 2-port USB 3.0 supersede PCIe Card



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 according to their standard. Our client was very happy to see the revisions we had made and all final revisions were agreed upon. Theses changes included that we have two working cameras for our final system, and to use either the Jetson TK1 or the Jetson TX1 for development. Before the revision, the requirements document was specifying the use of only the Jetson TK1. After much research on our end we had decided that either Jetson's TK1 or TX1 would be adequate for development, and actually plan to rely more heavily on the Jetson TX1. This change allows us to have more flexibility between the two Jetson SBC's, which provides protection if we cannot get one to work properly or to the meet the requirements needed.

B. Jetson TK1

We first began development on the Jetson TK1, as it was what we had on hand while we were waiting for the TX1 to arrive. We had installed graphics drivers on the TK1 as well as a driver for the original PointGrey camera we were loaned by Rockwell Collins. We had much difficulty working with the USB 3.0 port, due to the fact that the kernel support for it was 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 TX1, we continued development on the TK1 until the new, fully functional, TX1 had arrived. This was the point when development had completely shifted from the TK1 to the TX1.

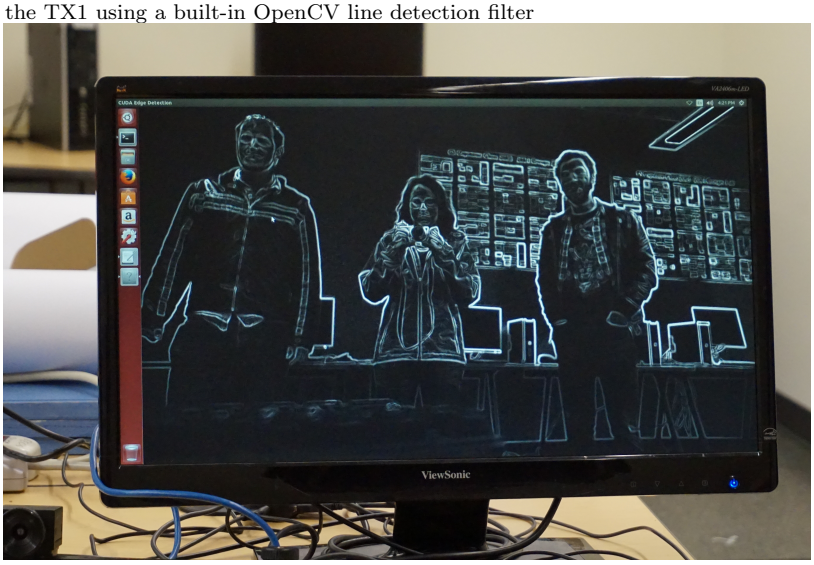
1) OpenCV Video Output: Our first step in exploring the Jetson TX1's capabilities, was to get a working video output from the on-board camera. We were able to display this image at 1080p with the option of either 30 or 60 frames per second. This video stream used embedded GStreamer code to transform the image buffer, generated from the camera, into a Video4Linux compatible format. 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 on-board camera, our next focus was on getting two cameras' video streams displayed to the monitor. This would allow us to be able to begin developing our software to handle processing of multiple video streams into one output. At the time we only currently had one USB 3.0 camera, but found that trying to combine the on-board camera with the PointGrey camera was going to be very difficult and would most likely have undesirable results. We decided that using two USB 3.0 cameras would be the best way to progress in order to accomplish our needed performance metrics. While we were waiting for a second USB 3.0 camera from Rockwell Collins, we were able to display the one PointGrey camera stream onto our monitor. Figure 6 is an unprocessed output stream using the EVS-3000 lens and figure 7 is using the standard LS-TP-08 lens. We were even able to implement two basic edge detection functions using several of OpenCV's built-in filters and apply it to our output video. Figure 8 below demonstrates one of our edge detection filters applied to our output video stream.

Fig. 6. Video output from the TX1 using a PointGrey camera with an EVS-3000 lens attached

2) OpenGL Video Output: We started using OpenCV because it was an easy framework to use. OpenCV could store the camera buffers, perform processing, and create a user interface that allowed us to see the video output stream. While this provided us with simplicity and easy developlment, the performance was well below out requirements. This was because OpenCV did not make use of the hardware on the Jetson. Part of this was because of OpenCV's poor implementation of copying the video buffer to the screen. In order to fully make use of the Jetson's hardware, we decided to switch to using an OpenGL framework, using GLUT, for creating the window and displaying the video. Our first implementation of an OpenGL system was still resulting in a slow image output. This was only because we were not exploiting the full capabilities that GLUT had to offer. GLUT has a zero copy function that allowed for us to not have to perform any memory copy operations. Once we had a better understanding of how to create and manipulate shared buffer space between the GPU and CPU, we were able to get much better output results. We then modified it to make better use of the hardware linking user buffers with the FlyCapture API to GPU textures. Our final product in this beta release ended up using an OpenGL framework

Fig. 7. Video output from the TX1 using a PointGrey camera with a LS-TP-08 lens attached

Fig. 8. Video output from the TX1 using a built-in OpenCV line detection filter



to produce our highest performing result.

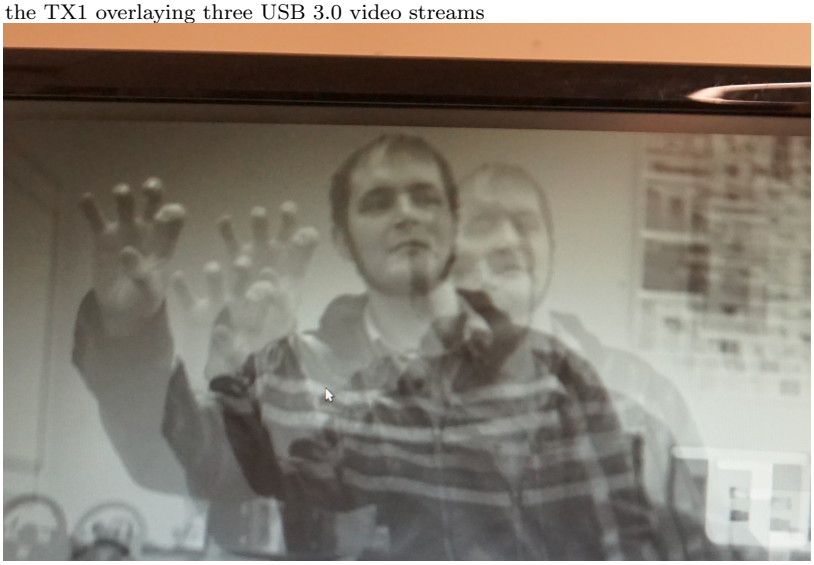
3) CUDA Library Functionality: After making use of the OpenGL framework to capture and display images we still needed a library to handle the processing of the images. We decided that NVIDIA's CUDA library would be adequate for the processing we needed to handle. We then installed the CUDA drivers and example code onto the Jetson TX1. CUDA includes a library and tool-chain that allows us to compile code that runs directly on the GPU called 'CUDA Kernels'. We can call these CUDA Kernels from our main thread in our software. This GPU compiled code can easily handle filters like line detection and much more. We are even planning to do a color space conversion using these CUDA Kernels. The CUDA package also came with some examples that were very useful in creating code spikes. The examples were very helpful and made our final code easier to write. We have rewritten and combined many of these examples to get the beta code working. Our results consisted in being able to overlay an outline detection image overtop of a separate camera's normal image.

4) Modular Software System: In order to create modularity and flexibility, 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 (displays). This module system allows for configuration without the need to recompile every time, which can be complex and time consuming. It also gives us the ability to implement different filter modules and quickly apply only the desired filters at a given time. 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 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 a 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. This image is then 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 for being able to assist in safe flight in almost all situations.

5) Development of the Modules: Development of this system is an iterative process. We first explored the modular system by outputting a static image to demonstrate that the system was working. We then created a module to handle input from the PointGrey camera by implementing its given software. This allowed us to be able to interact with multiple PointGrey cameras. When we received the USB 3.0 PCIe card, we were 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. The results of this output video stream is shown in figure 9 below. We have also added a line detection 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.

Fig. 9. Video output from the TX1 overlaying three USB 3.0 video streams



IV. Problems That Impeded Progress

A. Jetson TK1 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 original version of the kernel that was supplied with the TK1 had actually disabled support to the USB 3.0 port and was only allowing it to be used as a USB 2.0. The recompile of the kernel proved to be a difficult task, and we still had some connection and display issues with the video stream on the TK1. Despite these setbacks, we were able to get a little bit better video output, but still was not meeting the standards needed with the TK1. Because of the low performance, we decided to move from the Jetson TK1 to the Jetson TX1.

B. Jetson TX1 Manufacturer Error

Development on the Jetson 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 Jetson TX1 we

currently had, and order a brand 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

After getting video output using the Jetson TX1's on-board camera, our next step was to interact with the USB 3.0 PointGrey camera. Once we got the PointGrey camera and the TX1 to interface with each other, we ran into the issue of getting the TX1 to register the camera and to display an image from it. 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 first needed to recompile the Linux kernel to increase the USB buffer size to accommodate for the higher resolution image capturing from the PointGrey camera, as the on-board camera had a much lower resolution. Secondly, we had to acquire the proper camera software needed from PointGrey directly to interface between the proprietary camera API and OpenCV. After all was said and done, we had generated an output video stream from the USB 3.0 camera.

D. OpenCV Limiting Frame Rate

After further development in OpenCV we found that our visual frame rate was not matching the frame rate that was being logged. The logs were saying we were outputting 30 fps, but the video output was obviously less than that showing significant lag. We determined that this lag was due to a bottleneck 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, and simply replaced the way we were display the images. We have changed development to two different libraries that are able to replace OpenCV. One is called GLUT and uses OpenGL to create a window and display images on the screen. The other is a low level API that allows us to directly manipulate the frames on the GPU. This API is called CUDA, which is NVIDIA's general purpose GPU API. This allowed us to use a feature called Zero Copy, which allows us to write directly into GPU memory from the host application. Zero copy 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. Performing image format conversions in-place on the GPU will improve performance further.

V. Beta Release Results

Our project requirement specification describes specific latency, frame rate, and resolution requirements in order for our system to be a viable option to replace the current FPGA system. Our end-to-end latency requirement specifies that it should take less than 100ms to capture video from the camera, process it, and display the output stream. 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. 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 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 with OpenCV	37 FPS average		
USB 2.0 with OpenCV	11 FPS average	10 FPS consistent	6 FPS
USB 3.0 with CUDA	40 FPS	35 FPS	32 FPS
USB 3.0 with CUDA and multithreaded	100-110 FPS	50-70 FPS	45 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.

After having poor initial results using OpenCV, we decided to completely implement our own vision processing platform that utilizes GPU optimized video processing with the CUDA library. Our new vision platform takes advantage of the shared memory of the CPU and GPU on the Jetson by using a CUDA feature called Zero Copy. This allows us to create buffers that are shared directly between the CPU and GPU, without having to copy the image every time it is used. We integrated this functionality directly into our camera input system by allocating a shared memory buffer and setting up PointGrey's colorspace conversion to output directly into this buffer. Once the image is in GPU memory, we then are able to perform extremely fast operations on it using the GPU. Finally, we map the CUDA buffer to an OpenGL texture, which is then drawn to the screen.

Using CUDA for our image processing system allowed us to meet high performance metrics, as we were able to add new filters with virtually no effect on frame rate or latency. Even a seemingly intensive edge detection algorithm causes a reduction of less than 1 frame per second. However, we were still limited on our frame rate by the input module.The 35-40fps we are now acquiring is acceptable for our requirements, but our cameras are capable of much greater frame rates: up to 166fps at 1080p. By inserting timing measurements into each step in the input module, we were able to determine that the main cause of slowdown is the PointGrey colorspace conversion, which can take up to 30ms to complete depending on the camera. This initially limited us to lower frame rates, but by threading the input module we were able to perform multiple conversions at the same time. This enabled us to achieve much higher frame rates, up to 110fps with a single camera. Our latency increased slightly to about 40ms, but this is still within our 100ms acceptable range.

Our new CUDA-based system is extremely fast, and enables us to work on more advanced image processing algorithms. Our current software includes multiple combination filters which blend two images together by averaging their color values, as well as a Sobel edge detection filter that was provided by NVIDIA in their CUDA sample code. Even with three camera inputs, two combination filters, and an edge detection filter, we still achieve roughly 35 frames per second without even coming close to our latency requirement.

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

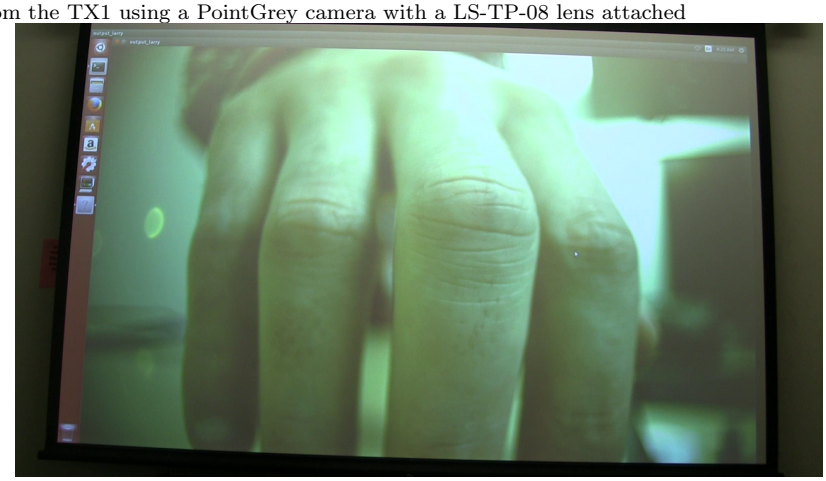


Fig. 12. Video output from the TX1 running at 30 fps

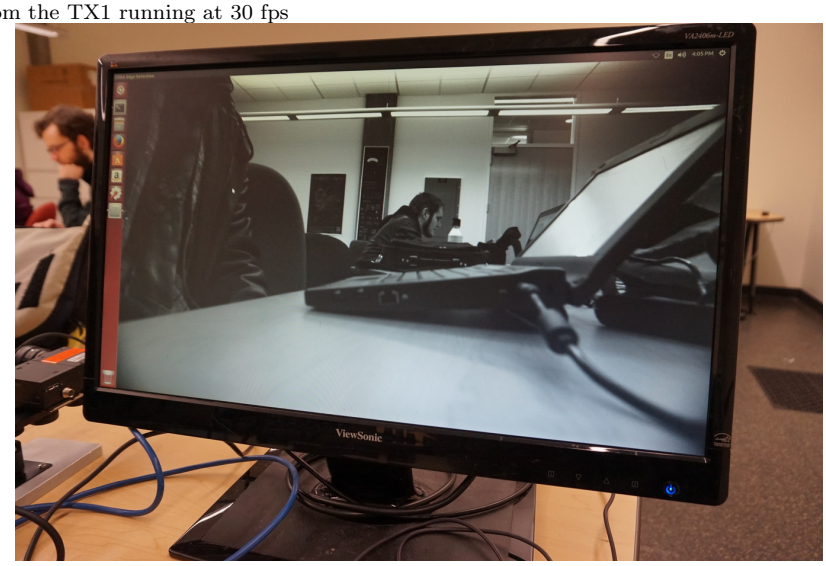
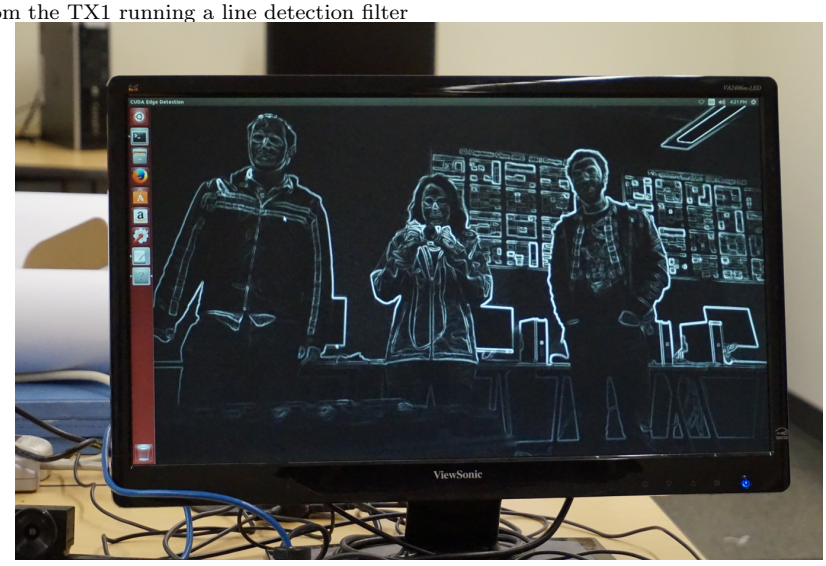


Fig. 13. Video output from the TX1 running a line detection filter



A. Implications of Beta Results

Now that our frame rate far exceeds our requirement, we can now focus on debugging, increasing stability, and implementing our stretch goals. Our current software works perfectly when it is running, but sometimes it will crash during exit because the GPU cleanup may occur before some of the threads operating on GPU memory have actually stop. Once we fix these issues, we will then focus on additional improvements. We intend to create a CUDA color space conversion to replace the existing one, which should enable us to maximize our frame rate for all cameras. Then we will focus on our stretch goal of implementing a runway detection filter.

VI. What is left to do

A. System and Code Bugs

We currently have a few code bugs in the system that are 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 reinitialization of the cameras. This can cause cameras to act strangely upon being 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 some other functionality that extends our requirements standard. We have developed a plan for some of these, and are considering them as stretch goals. A few of these features have already been implemented or are close to being meet:

- Improve the frame rate of our system (above 60fps)
- Display our video output stream on more than one monitor simultaneously
- Perform an object tracking filter
- Design a real-world model to test our cameras on, such as with miniatures (two of our cameras have micro 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, which would include re-implementing our code on different platforms and comparing their processing speed and hardware capabilities

VII. 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.

A. CUDA based modules

In our base program, 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. The HawkEye_CUDA_Module retains these functions, but provides a default run() method which calls an appropriate runKernel() CUDA function based on the number of inputs and outputs. This is the default functionality for cuda modules, but the run() function can be overloaded to provide different functionality if desired.

```
1     class MyModule: public HawkEye_CUDA_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
2   // CUDA kernel executed on the GPU
3     __global__ void runKernel(Pixel *output, Pixel *inputa, Pixel *inputb){
4       index = blockIdx.x*blockDim.x + threadIdx.x;
5     output[index]=inputa[index]/2 + inputb[index]/2;
6   }
```

This code describes a GPU accelerated module that would take two images and combine them together by a simple averaging method. The HawkEye_CUDA_Module class automatically allocates output buffers in shared CPU/GPU memory based on the number of inputs and outputs defined in the class constructor. This particular code sample is what is known as a CUDA kernel, which means that it is compiled by the NVIDIA compiler and executed on the GPU. The __global__ specifier tells the compiler that it is a CUDA section. The runKernel function is an overloaded virtual function that the HawkEye_CUDA_Module class sets up and runs in a manner completely transparent to the algorithm implementer. The only thing the implementer needs to know about are the blockIdx,blockDim, and threadIdx variables, which are part of the CUDA system. These variables define the different GPU threads' operating locations in the buffer.

Our system does not limit users to our built in CUDA kernel execution model. The user can choose to implement a more complex algorithm that includes a mixture of CUDA and CPU based operations. Also, we maintain compatibility with our OpenCV module system by automatically creating a OpenCV Mat object around our shared memory buffers. This allows the user to take advantage of OpenCV's built in filters and matrix operations, although we do not actually use them in our modules due to performance issues.

B. JSON configuration

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'':{
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 .