Automation using GPU Image Processing for Visual Feedback

Riley Kenyon

Independent Study - Spring 2019

Abstract

This report details the process of learning and developing algorithms for the NVIDIA Jetson line of embedded system on module (SoM) developer kits. The two products used throughout the independent study are the TX2 and the Nano to perform computer vision and automate the progression of a mobile phone application "Piano Tiles".

1 Purpose of Study

The purpose of this independent study is to build off a previous controls project to optimize the automation of a phone game utilizing a GPU. In the original construction of the automation unit, a Raspberry Pi 3B+ was used as the micro-computer to control the project. Although the Raspberry Pi is great for running simple python scripts and projects requiring access to peripherals, the on-board CPU is a Cortex-A53 (ARMv8) running at 1.4GHz with 1GB of LPDDR2 SDRAM. These performance specifications provided some barriers in terms of computational complexity of algorithms. In order to do more complex manipulation of pixel values and advanced image processing techniques, a more capable device was employed: the Nvidia Jetson TX2 and later the Jetson Nano. The end objective is to greatly succeed the record set by the previous device, and do so more reliably. This report is a document to reference for beginning CUDA programming and the process of developing the "Piano Tiles Project".

2 Installation of Jetpack SDK

2.1 Jetson TX2

The software development kit (SDK) that we will use for the Jetson TX2 is the JetPack SDK offered by NVIDIA. Use the JetPack installer to flash the Developer kit with the latest OS image, install developer tools, and install the libraries, samples, and documentation. The current release supporting the TX2 is the **Jetpack 3.3**. The SDK cannot revise a current platform, and is unfortunately not able to be directly downloaded with the existing version of ubuntu on the Jetson. To properly install all of the developer tools and libraries, a "host" computer flashes the TX2 with the development kit. This includes the OS L4T (Linux for Tegra). By definition flashing overwrites the existing firmware or data contained in the memory of the electronic device, and cannot be run directly on the device. For a full guide visit NVIDIA's website or reference the installation guide.

2.2 Jetson Nano

The installation of Jetpack is significantly simplified with the Nano. Similar to the installation of Raspbian for the Raspberry Pi, the Nano makes use of flashing an OS to a microSD card and booting from there. Downloading the image takes about 7 minutes, and approximately 10 minutes to flash to the SD card. The microSD card used was a 32GB, but a 64GB card works the same. The program used to flash the card is *Balena Etcher* and can be found here:

https://www.balena.io/etcher/

A detailed description of the installation process can be found on NVIDIA's website:
https://developer.download.nvidia.com/assets/embedded/secure/jetson/Nano/docs/Jetson_Nano_Developer_Kit_User_Guide.pdf?lpVe-nKD-HWt6Xhh2PUVcnBeRcSZwY7trGUybiD3hwbOfGdsz8HH4Vb_8XvW2fCVKP3Iu2gdPaiGkZR6khqBaMRS9iM3-g5S9vuFm02iqXNY5IdWBcPQjy5nUZoKCHJaF8tgv4EEGPKw9h7RxCIYLn6eBQAu

3 Getting started with CUDA

3.1 NVCC

Instead of using g++ to build, compile, and run sample scripts, nvcc (Nvidia Compilier) can be used. Is is in the best interest to use this compiler due to the fact it can compile .cpp, .c, and .cu code. The main application of this process will be making use of the GPU with it's parallel processing, and this avenue allows us to incorporate kernels later on but still remain constant with compiling. However nvcc is a little tricky to use, and referencing the programming guide for CUDA provides some more insight, found here:

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

Another excellent resource is Matthew Hanley's comparison of nvcc to cmake.

https://github.com/matthewdhanley/jetson-tx2/tree/master/cuda/7-nvcc-vs-make

3.1.1 Compilation on Nano

After installing Jetpack via the microSD card, the current version does not have a reference to the nvcc function call. In order to remedy this, it is necessary to place a couple of commands into the .bashrc script that is run on startup of the Jetson Nano. Edit the .bashrc file using the command below:

```
$ gedit ~/.bashrc
```

then add in the following lines at the end of the script.

```
$ export PATH=${PATH}:/usr/local/cuda/bin
$ export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:/usr/local/cuda/lib64
```

Now nvcc is linked to the correct location on the Nano where it is defined, and can be used to compile.

3.2 Format

Nvcc has a format where the function call requires the reference of the desired script and all of the associated libraries, as well as the object file that can be run afterwards. Any object file will show up as green in the current directory and can be executed with the action ./ preceding the filename. An example call of the compiler can be seen below

```
$ nvcc filename.cu -o objectFile -I. -I/usr/local/cuda/bin
```

breaking this down, the filename.cu is the script that is to be compiled using nvcc, the object file is objectFile and using the suffix -I. tells nvcc to look in the current directory for headers and libraries, as well as the location /usr/local/cuda/bin. After compilation, objectFile can be run using the command "./objectFile" . The ./ command is to run an executable.

3.3 Kernel and Configuring Threads

A kernel in CUDA programming is the function that is run on the device side of things (GPU), and is called from the host (CPU). The notation for using a kernel is

```
cudaFunction<<<numBlocks,numThreads>>>(arg1,arg2,...);
```

The three arrows on both sides specify the number of blocks and the number of threads that are used to determine how many times to run the function in parallel. The file extension used for any type of cuda programming is *.cu*, as can be seen in the format section above. In order to write the function *cudaFunction*, the format will appear similar to below:

```
__global__ cudaFunction(arg1,arg2,...){
    tid = threadId.x + blockId.x*blockDim.x
    //perform actions using thread ID
}
```

As a good place to start, it is useful to verify that the code can compile and run at least once. This is able to be done by setting the number of blocks and the number of threads equal to one for a single execution of the GPU kernel. In order to configure things further, it is necessary to look at the architecture of the board and the number of CUDA cores on the GPU. As a general rule of thumb, any multiple

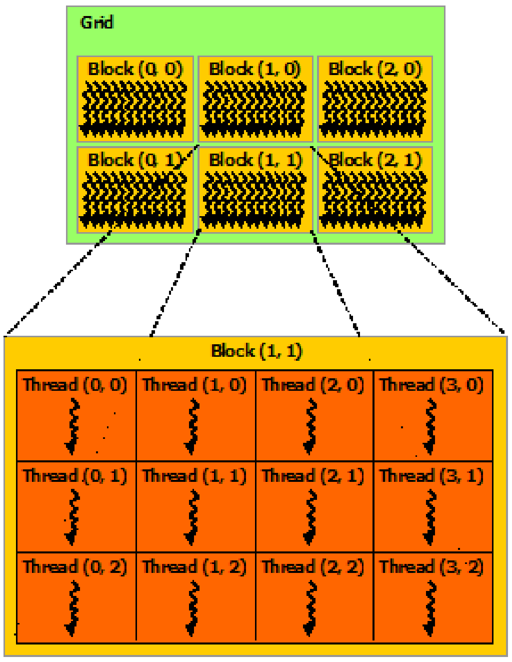

Figure 1: Hierarchy of threads within blocks on the GPU.

of 32 works well, and a baseline 512 is used in many of the project scripts. Function calls such as $cudaGetDeviceProperties(\&prop, 0)$, with $cudaDeviceProp\ prop$ declaration for the variable, allows for more detailed information of the GPU being used and what number of threads is most optimized for the system. For the Jetson Nano the maximum number of threads per block is 1024.

3.3.1 Thread ID

In order to determine what index the thread is overall, the thread ID is calculated based a three pieces of information. The thread index within a given block (threadIDx.x), the block ID (blockId.x), and the number of total threads within a block (blockDim.x). As seen in the function in the section above, the overall thread ID is given by threadId.x + blockId.x*blockDim.x. The same can be applied for determining IDs for multiple dimensions of threads, where there is an Id.y and an Id.z.

3.4 Synchronizing Threads

The unique thing about using GPU parallel programming is that all the threads do not execute in the same time or fashion. In order to compensate for this and synchronize a set of threads at a certain point in the script, use the function $cudaDeviceSynchronize()$. This function needs to be called after a kernel call, if the data from the kernel is needed to do further manipulations. If the threads need to be synchronized within the kernel before performing another operation, the command $\_\_syncThreads$ can be used to wait until all threads in the current block get to that line of code.

3.5 Memory Allocation

Memory allocation is needed due to the fact that the host and the device are required to read/write to the same data. For someone who is new to c/c++, array assignment cannot simply be put A = B, the memory has to be copied over to the other array from the actual memory position (using pointers). This is done by using the cudaMemcpy and cudaMallocManaged functions, which is used in the form below

```
unsigned char *dataDevice, *dataHost;
cudaMallocManaged(&datadevice,width*height*sizeof(unsigned char));
cudaMemcpy(dataDevice,dataHost,width*height*sizeof(unsigned char),
    cudaMemcpyHostToDevice);
```

The first line defines the pointer of type *unsigned char* and the other lines take care of allocating the memory properly. The variable dataDevice is a memory copy of the dataHost variable, which can be used on the GPU. After finishing using the variables the command cudaFree() is used to free whatever variables were allocated using cudaMallocManaged. If this is familiar, the practice is the same in C programming with the commands free() and malloc().

3.6 Adding Example

A simple example of using the processing capabilities of the GPU is with the addition of two arrays. The example can be detailed below. An image is a 2D array of pixel values, in order to avoid additional complexity associated with formatting multiple directions of a thread block and obtaining thread ID's, the 2D array can be flattened into a single dimension of indexes. Using the thread ID of each core on the GPU, the addition can be performed. An optimization technique used for larger arrays is to use strides, which are to cover more information for each thread of the GPU.

```
#include<stdlib.h>
#include<stdio.h>
#include<string.h>
#include<iostream>
#define NUM 307200 // Approximately the value of an image 640*480

//====================GPU Kernel======================
__global__ void gpu_add(int *matA, int *matB, int *matC){
  //thread ID
  int tid, stride;
  tid = blockIdx.x*blockDim.x + threadIdx.x;
  stride = blockDim.x*gridDim.x;
  for (int index = tid; index < NUM; index+= stride){
      matC[index] = matA[index]+ matB[index];
  }
}

//====================CPU Function=====================
void cpu_add(int *matA, int *matB, int *matC){
  for (int i = 0; i<NUM; i++){
    matC[i] = matA[i] + matB[i];
  }
}

// Display Matrix
void display(int *mat){
  for (int i = 0; i < NUM; i++){
    printf("%d \n",mat[i]);
  }
}
//Generating matrix based on number
void numGen(int *mat, int value){
  for (int i = 0; i<NUM; i++){
    mat[i] = value;
  }
}

//====================MAIN FUNCTION====================
int main(){
  int *matA, *matB, *matC; //pre-allocate matrices
  cudaMallocManaged((void **) &matA,NUM*sizeof(int));
  cudaMallocManaged((void **) &matB,NUM*sizeof(int));
  cudaMallocManaged((void **) &matC,NUM*sizeof(int));
  numGen(matA,1); //Populate A and B with ones
  numGen(matB,2);
  printf("MatrixA[0]: %i\n",matA[0]);
  printf("MatrixB[0]: %i\n",matB[0]);
  unsigned int numThreads, numBlocks; // configure blocks and threads for GPU
  numThreads = 1024;
  numBlocks = (NUM + numThreads - 1)/numThreads; //<1 additional block

  float calcTimer = 0; // time variable
  cudaEvent_t start, stop; // initialize timer events
  cudaEventCreate(&start);
  cudaEventCreate(&stop);

  float GPUtimer, CPUtimer; // variables for saving the average time
  for (int j = 0; j < 100; j++){ // Loop to average time(s) for GPU and CPU
```

```
    //=========================CPU=====================================
    cudaEventRecord(start);// start timer
    cpu_add(matA,matB,matC);// call CPU function
    cudaEventRecord(stop);// stop timer
    cudaEventSynchronize(stop);// save elapsed time for CPU
    cudaEventElapsedTime(&calcTimer,start, stop);
    CPUtimer = CPUtimer + calcTimer;
    calcTimer = 0;
    //=========================GPU=====================================
    cudaEventRecord(start);// start timer for GPU
    gpu_add<<<numBlocks,numThreads>>>(matA,matB,matC);
    cudaDeviceSynchronize();
    cudaEventRecord(stop); // stop timer for GPU
    cudaEventSynchronize(stop);// save elapsed time for GPU
    cudaEventElapsedTime(&calcTimer,start, stop);
    GPUtimer = GPUtimer + calcTimer;
    calcTimer = 0; // start timer for GPU
  }
  printf("MatrixC[74] on CPU: %i\n",matC[74]);
  printf("Time Passed for GPU: %f\n\n",GPUtimer/100);
  printf("Time Passed for CPU: %f\n\n",CPUtimer/100);

  //Free Memory
  cudaFree(matA);
  cudaFree(matB);
  cudaFree(matC);
  return 0;
}
```

Using this snippet of sample code, the GPU is compared to the CPU for determining the amount of time saved on the GPU. The important items to look at in the example is the function call of the GPU: gpu_add¡¡¡numBlocks,numThreads¿¿¿(matA,matB,matC). The number of threads is set to the maximum number on the GPU 1024, and the number of blocks is determined by taking the total number of operations, adding the number of threads less one, dividing by the number of threads. What this accomplishes is through floor rounding, the total number of blocks set to call will be less than one extra. Using that equation the number of blocks is optimized, and there are a limited number of unused threads.

4 Accessing GPIO

4.1 Jetson TX2

When considering which GPIO pins to use on the Jetson TX2, there are a lot of options. However, a lot of the pins come with dedicated or suggested uses. For the purposes of this project, I will be avoiding using the pins with dedicated purposes (SFIO - special function input/output) and sticking with the unused GPIOs or MPIO (multi purpose input/output). In terms of type and location of pins, there are several to choose from, see Figure 2. The general types are: UART (Universal Asynchronous Receiver-Transmitter), SATA (Serial AT Attachment), JTAG (Joint Test Action Group), PCIe (Peripheral Component Interconnect express), and the GPIO expansion headers. Due to the lack of communication between the solenoids and the board, UART is likely overkill and may not be applicable. SATA is generally used for communicating with mass storage systems. JTAG is used for debugging, but likely could be used in this application. PCIe may also be a viable alternative. In general, the board layout has several expansion headers, but we will be focusing on J21 and J26 that manage the GPIO pins of the TX2. These have the most unused pins. The article found here

https://www.jetsonhacks.com/nvidia-jetson-tx2-j21-header-pinout/

describes the pinmux of the board, and details which pins are not currently being used by the device. This was used in combination with the excel sheet provided by NVIDIA for pinmux.

4.2 Jetson Nano

The nano features a familiar 40 pin GPIO header on the development board. For one, this is compatable with many of the hats produced for the Raspberry Pi but is also what is needed for many hobbyists or students. Althought the TX2 provides more options for GPIO access, the 40 pin header is all that is

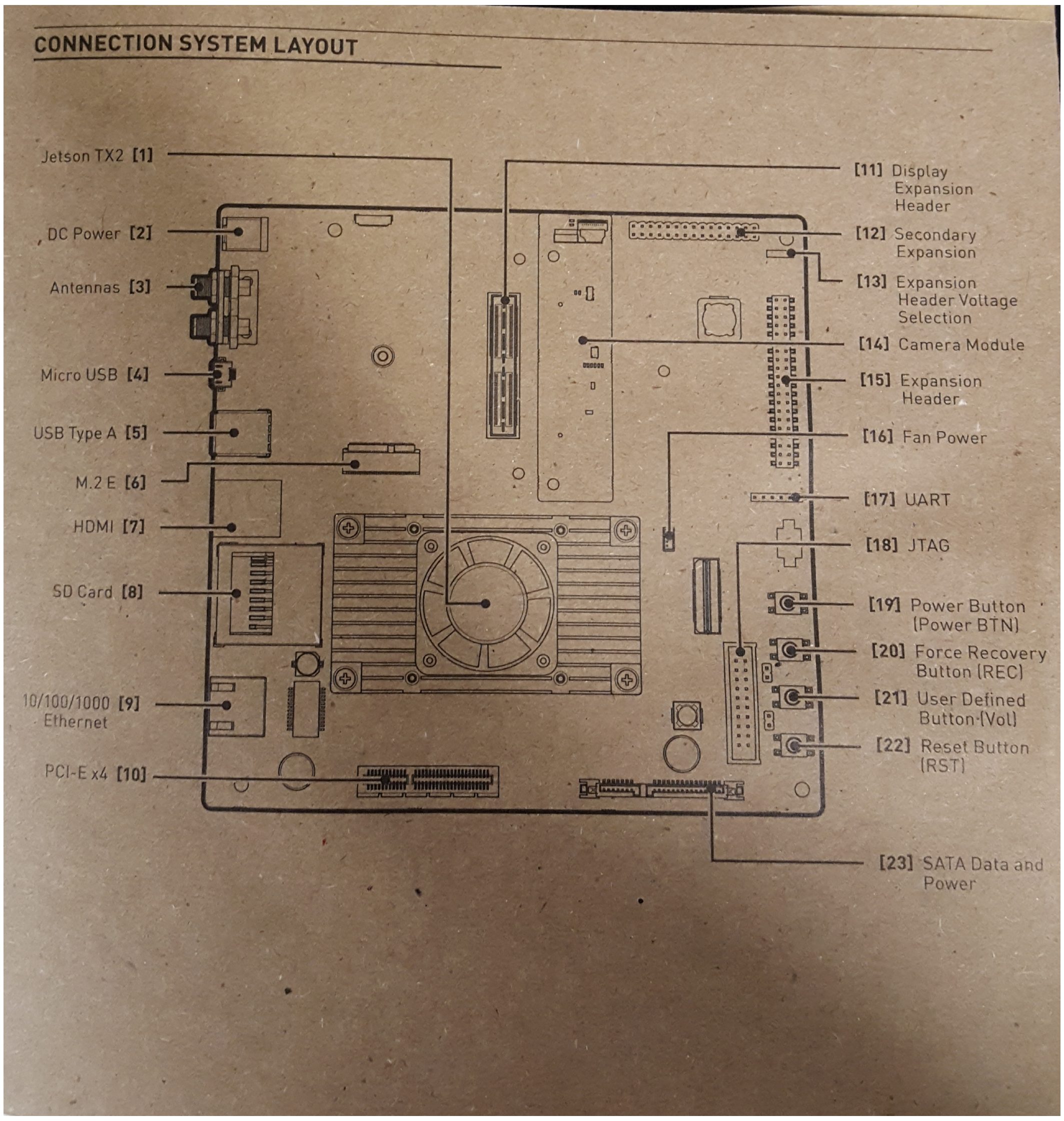

Figure 2: The NVIDIA Jetson TX2 Carrier Board Layout

necessary for the Piano Tiles Project. The pinout can also be found on JetsonHacks website at the link below.

https://www.jetsonhacks.com/nvidia-jetson-nano-j41-header-pinout/

4.3 Setup

The process is very similar accessing GPIO pins for both the TX2 and the Nano. A useful resource for all things related to the Jetson line of products is at the site : https://www.jetsonhacks.com/. Here there is a link to the github for a download of GPIO code for the Jetson TX1, which can adapted to work with the pin out of the Jetson TX2 and the Jetson Nano: https://www.jetsonhacks.com/2015/12/29/gpio-interfacing-nvidia-jetson-tx1/ Following the software installation from the link, the follow code is used to clone the repository, build the example and then compile.

```
$ git clone git://github.com/jetsonhacks/jetsonTX1GPIO.git
$ cd jetsonTX1GPIO
$ ./build.sh
$ sudo ./exampleGPIOApp
```

The sample was revised to reference pin 297, 393, 394, 395 on the TX2 and pins 12, 38, 200, 149 on the Jetson Nano, rather than the ones provided for the TX1. The main function was simplified to only switch on and off the pin rather than wait for an input from the button. Some issues arose with permissions for exporting and unexporting the pin, so the following code was implemented to change the owner and group of the operations.

```
$ sudo chown nvidia:nvidia export unexport
```

This allows the group nvidia to allow access to the export and unexport files. To test different variations of the script, these commands were ran to compile and run after building:

```
$ ./build.sh
$ sudo ./exampleGPIOApp
```

The build.sh file was only to perform the compilation rather than type out the whole command in the terminal, but to be verbose the command within the text file is

```
$ g++ -O2 -Wall exampleGPIOApp.cpp jetsonGPIO.c -o exampleGPIOApp
```

This is changed to use use nvcc with the command

```
$ mv exampleGPIOApp.cpp exampleGPIOApp.cu
$ mv jetsonGPIO.c jetsonGPIO.cu
$ nvcc exampleGPIOApp.cu jetsonGPIO.cu -o TESTRUN
```

The moving of files was to make use of the nvidia compiler that needed to have extensions of .cu. The object file that can be run later is TESTRUN.

4.4 Command Line Operation

In order to access the pins of the board, I used the expansion header J21 on the TX2, this header was better documented and easier to get up and running with. The code below details how the command line was used to manually turn on and off the pin. Initially I tried GPIO pin 388, but ran into some issues with writing values to the logic levels. I attempted the same thing with GPIO pin 297, and was able to get the pin shifting to HIGH and LOW through the command line.

```
$ cd /sys/class/gpio
$ sudo -s
$ sudo echo 297 > export
$ exit
$ cd gpio297
$ sudo -s
$ sudo echo out > direction && 1 > $ value
$ exit
```

This allows the pin 297 to be written a direction (input/output) and value (HIGH/LOW). The same procedure was done with pins 393, 394, and 395 to be used with the transistors to control the solenoids. This is done because of sysfs and how attributes are determined by the contents of files. The script version uses functions that write to theses files for direction, value, etc. Some advise for choosing which pins to read/write to are to pick pins that do not have dedicated purposes, for example anything that is not using SPI or I2C or any SFIO.

4.5 Within a script

If the objective is to access the GPIO pins within a script, it is necessary to define them within a CUDA file and the corresponding header. The *.cu* file extension is used in place of *.cpp* for compilation using nvcc. Specifically when using headers (.h), you need to specify the location of the library with -I and by using "-I." for using the current directory. A command for the current setup is:

```
$ nvcc jetsonGPIO.cu exampleGPIOApp.cu -o TESTRUN -I.
$ sudo ./TESTRUN
```

Something else I noticed was how using .cpp .c and .h files in combination resulted in an error compiling. Notice that in the previous commands, if you change the files to .cu then building and compiling works. Because of the simplicity of the files, and the standard libraries simply copying the files to .cu extensions work.(Note that the same works for .cpp)

```
$ mv jetsonGPIO.c jetsonGPIO.cu
$ mv exampleGPIOApp.cpp jetsonGPIO.cu
```

4.6 Speed Test

After the script was created to shift the pin HIGH and LOW, it was time to test the limitations of how quickly it could do so. The delay times were adjusted in increments and referenced to the same wave produced from a function generator. Starting at 5Hz, the waveform was increased reliable until 50Hz. The pulses can be seen below in Figure 3. A closer look reveals that the pulses have 3.3V logic, and are not affected by each other (a delay between them of approximately 60 μs).

4.7 Tap Test

After verifying that the TX2 could produce wave forms at the frequency needed to not inhibit the improvements made by the GPU in image processing. A test was performed to excite the solenoids and tap the phone screen. The pulses sent to the transistors were sent at a frequency of 25Hz, alternating between all four solenoids. This loop was cycled for 100 times (400 taps), and then the result was verified by a counting app on the phone. The test took approximately 16 seconds which corresponds to the frequency of pulses. The phone had to be placed in a specific orientation in order to correctly register with the correct surface area and pressure. See Figure 4 for a depiction of the setup. When using this assessment, some troubleshooting is necessary. Specifically, the solenoids will occasionally get stuck and will not extend when a signal is passed to them, by manually pressing down on them fixes the problem. Additionally, "sudo" privileges are needed when running the script to run the application. After determining a appropriate delay to wait between setting the value high/low, the delay of 35000 µs was used in the piano tiles project for tapping with the solenoids. The corresponding file on the nano is under $/Documents/TX2GPIO/exampleGPIOApp.cu$, where the pinouts can be changed to reflect the desired outputs from j41 header. The script is compiled using jetsonGPIO.cu using the nvcc command shown in the setup section.

5 Accessing Cameras

5.1 Jetson TX2

The Jetson TX2 has a built in camera module that uses the camera serial interface (CSI) to process the camera. The specification is a 12x CSI2 D-PHY 1.1 lanes (2.5 Gbps/Lane) interface that uses the 5 MP Fixed Focus MIPI CSI Camera module. It is difficult to access the camera module without using additional pipelines, and that is not outlaid very well in the NVIDIA documentation. The following lines of code allows the on-board camera to be accessed from the command line.

```
$ gst-launch-1.0 nvarguscamerasrc ! 'video/x-raw(memory:NVMM),width=1920,
  height=1080,framerate=60/1,format=I420' ! nvvidconv !
  'video/x-raw(memory:NVMM), format=I420' ! nvoverlaysink -e
```

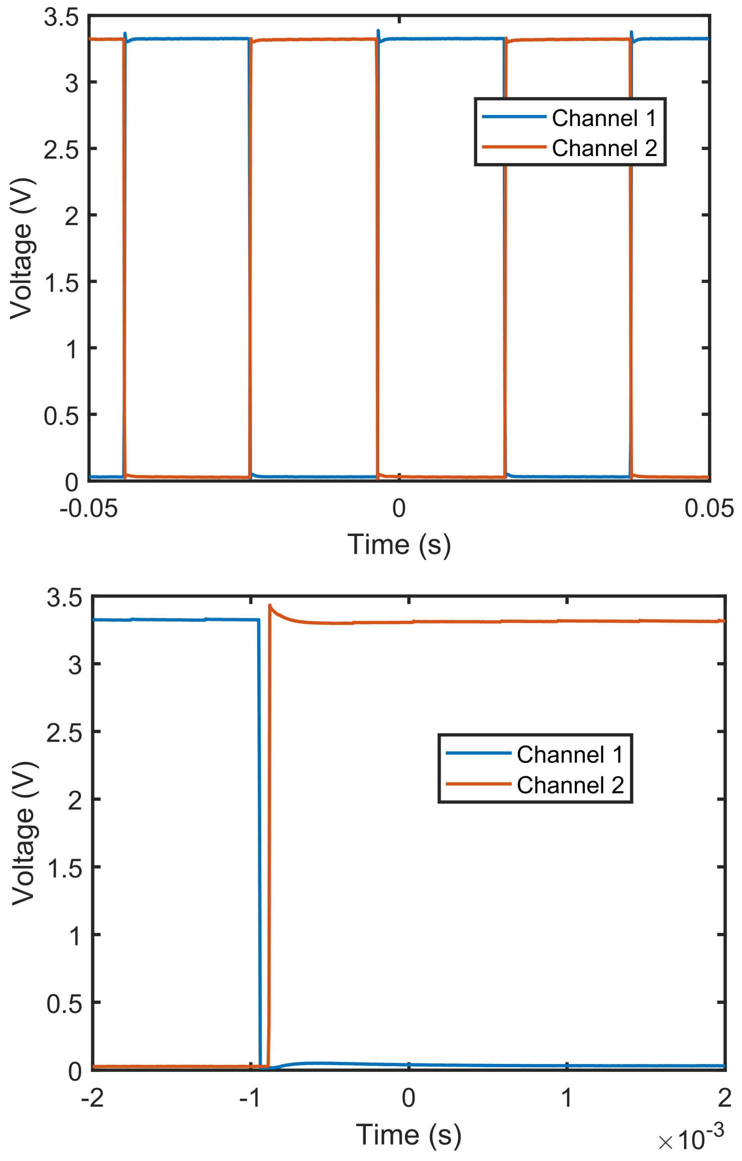

Figure 3: Two channels of pulses occurring right after each other at a frequency of 50Hz, used to determine the effect of swiching between solenoids and the minimum delay needed to actuate multiple times. The delay between the two is approximately 60 µs.

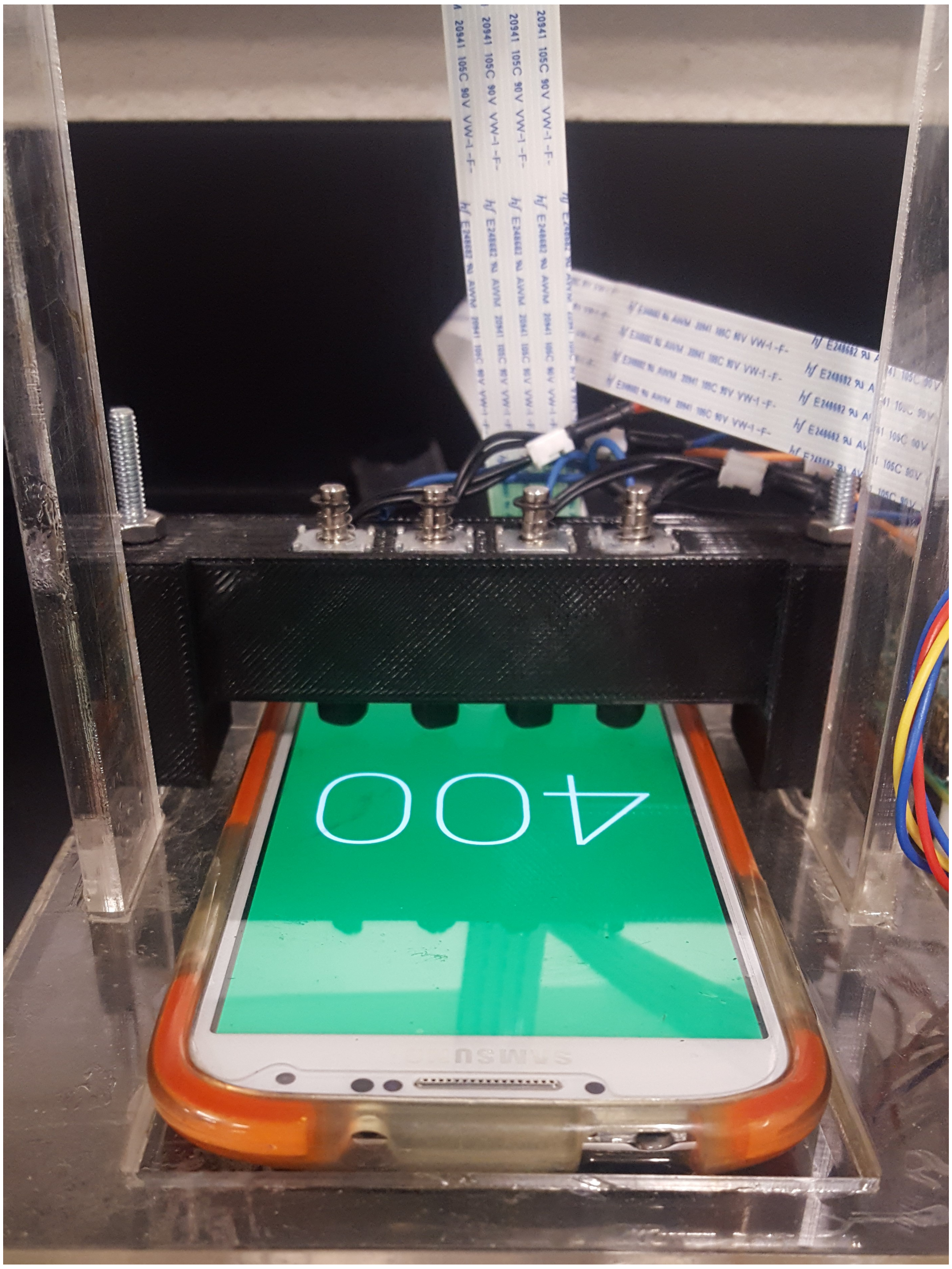

Figure 4: Tap test with counter for 400 pulses at a 25Hz, with the solenoid housing and phone used to run the Piano Tiles App.

5.2 Jetson Nano

The nano was built with a better camera interface, or at least easier to access. The module makes use of a flex ribbon cable to connect the CSI port to the desired camera. Luckily the layout is compatible with the Raspberry Pi camera and was easy to interface, just plug and play. The commmand below is a gstreamer pipeline that will test the camera from the command line.

```
$ gst-launch-1.0 nvarguscamerasrc ! 'video/x-raw(memory:NVMM),width=1280,
  height=720,framerate=120/1,format=NV12' ! nvvidconv flip-method=2 !
  'video/x-raw,width=960,height=616'! nvvidconv ! nvegltransform ! nveglgessink -e
```

Note the frame rate here is 120 fps, the camera has a variety of settings that can be configured through the gstreamer pipeline. Upon startup, there will be a variety of settings displayed in the terminal.

5.3 Gstreamer

A big part of accessing the cameras was with the usage of gstreamer and the nvargus libraries. Gstreamer is a pipeline that connects devices with the peripherals of devices. The nvarguscamerasrc allows the CSI cameras on the Jetson platforms to be communicated with gstreamer pipelines. The first arguments are relevant to the incoming stream, the height, width, and framerate variables set the configuration of the camera. The flip-method rotates the incoming feed, for example the a flip-method of 2 corresponds to a 180 degree rotation. After that is the display settings for the window, and the nvvidconv, nvegltransform, and nveglgessink configure the output to be compatible with the Jetson platform. The command seen above is for the Raspberry Pi camera 2 being used on the project which has multiple settings The configuration chosen to be used on the Piano Tiles project was the 120 fps at the 1280x720 resolution. This provided the maximum amount of pixels for a given physical area (the area of the screen) and needed to do less cropping from the initial image. More documentation on gstreamer pipelines can be found here:

https://gstreamer.freedesktop.org/documentation/application-development/introduction/basics.html

6 Image Processing and OpenCV

The project used libraries and utilities from openCV to form the display matrices that were used in acquiring pixels. All of the image processing was done by the GPU with a variety of kernels to manipulate the image. As the project progressed, openCV was also used to annotate the live video feed with the location of the tap from the solenoid, at the point where the action was triggered.

6.1 Camera Sources

Upon development on the Jetson TX2, the on-board camera was not friendly to work with. The gstreamer pipeline would not co-operate within the c scripts written to use openCV to access the camera streams. Possible factors were the version of openCV being used and the possible of using openCV linux for tegra (L4T). The resultant option was to use a Logitech USB camera to interface with the TX2. The operation of a USB camera within openCV is very simplistic, as the argument used to specify the camera is the camera index (1 in the case of the TX2). On the Jetson Nano, the CSI port is directly compatible with the Raspberry Pi camera (which is used on the previous iteration of the project). The gstreamer pipeline is identical to the source argument shown in the camera access section of this report for the Nano.

6.2 Displaying Images

After performing operations to an array, openCV has a built in object used for displaying matrices. The definition of a matrix was accomplished by the following:

```
cv::Mat matrixName(cv::Size(width,height),CV_8UC1,imageData);
cv::imShow("name_of_window", matrixName);
cv::waitKey(0);
```

The first argument of the definition is the size of the matrix to be constructed, the second argument is telling the function what information is being put into the matrix (8-bit unsigned char, one channel), and lastly is the flattened image array. The second line displays the matrix, and the third line is a wait key function. The wait key with input 0 waits until any key is hit, otherwise the function takes values of milliseconds and waits that amount of time before moving on with the rest of the script.

6.3 Saving Video

In order to get started with edge-finding and other image manipulations in real-time, recorded video was used and processed offline. This was accomplished by another openCV function:

```
cv::VideoWriter writer;
int codec = cv::VideoWriter::fourcc('M','J','P','G');
double fps = 30;
std::string filename = "./example.avi";
writer.open(filename,codec,fps,img.size(),1);
if (!writer.isOpened()){
  return -1;
}
```

This sets up the video writer for saving a .avi. Everytime an image is to be saved to the file, the command writer.write(videoFrame) is used. The variable videoFrame is a cv::Mat object. The fourcc code is a "four character code" which tells the writer what image codec to use when saving the images to the video feed. Saving a video requires the filename, frame rate, and image resolution. This was beneficial in prototyping the tapping of solenoids by displaying on-screen information about when a GPIO was set high without actually doing so (annotating screen). The playback of recorded video was also beneficial in determining the modes of failure for the project and adjusting the code accordingly.

7 Piano Tiles Project Overview

Piano Tiles is a mobile phone game were the user taps the screen to trigger the activation of a tile. Once a tile is triggered the screen begins moving at an accelerated rate. As the user continues tapping tiles, the rate increases until the user either misses a tile or selects the wrong one. As a benchmark of the previous iteration of this project, the max speed recorded was at 7.77 in/s. Typical experienced player can get up to approximately 6.8 in/s. However, in combination with fast frame rates, there are machines that have exceeded the 20 in/s mark.

https://www.youtube.com/watch?v=fq0W84ZTL7k

The current setup utilizes hardware from a previous independent study, this includes the structural design used in housing the Raspberry Pi camera, the four solenoids, and the Samsung Galaxy S4 that the application "Piano Tiles" is run off. The objective set forth is to exceed 10 in/s on the Jetson Nano, and do so with reliability. As of this writing the maximum speed set with the Jetson Nano was **9.220 in/s** with the use of delays to regulate the number of actuation per tile.

7.1 Functions Used in Algorithms

7.1.1 Gray-scale

When determining how to begin processing an image, two approaches were considered. A color image received by the Raspberry Pi camera can be broken down into the three color arrays RGB. The most simplistic approach would be to simply use one of the channels, for instance blue, for the main image array. This is what was done previously on the Raspberry Pi to save on computational complexity. The negative of this approach is that the image information is not preserved and this may ultimately lead to an erroneous tile triggering. The Jetson line of products are able to do image processing very well, and at low computational cost due to the number of cuda cores. For this reason, the gray-scale approach can be used. Gray-scale converts the magnitudes of the color arrays into a single value scaled between 0 (black) and 255 (white). There are two main methods for performing gray-scale, an average and a luminosity measurement. The average approach is simply taking the values $(R + G + B)/3$. The more sophisticated approach is to use luminosity, which is weighted to account for human perception. This equation

$$0.21R + 0.72G + 0.7B \tag{1}$$

is the most common, and the one that is implemented in this project. The GPU kernel of this operation can be found below.

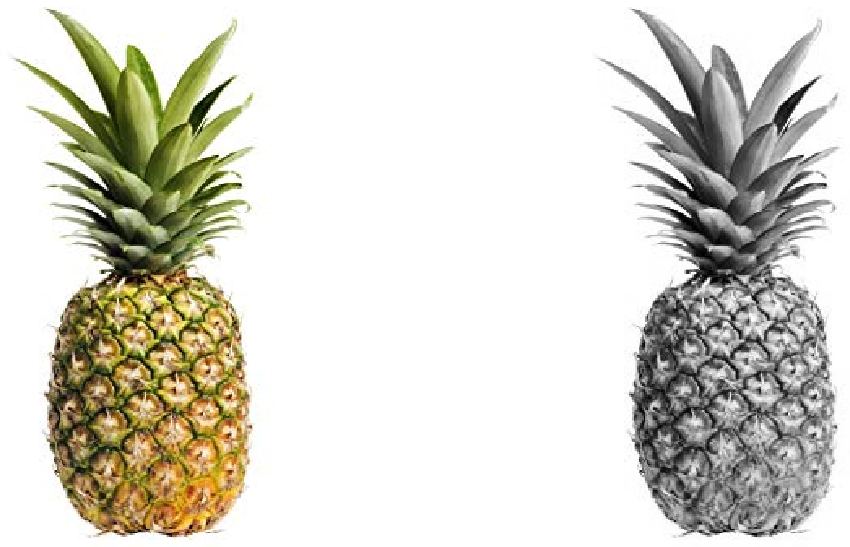

Figure 5: The comparison of the original BGR image (format used by openCV) and the gray-scale calculation of the image while maintaining all image intensity information.

```
__global__ void gpu_grayscale(unsigned char *matA,unsigned char *grayData, int width,
    int height){
  //Distance between array elements (i,j)[0] to (i,j)[1] is 1 not width*height
  int tid,stride;
  tid = blockIdx.x*blockDim.x + threadIdx.x;  //thread ID
  stride = blockDim.x*gridDim.x;  //stride length
  while (tid < width*height){
    grayData[tid] = matA[3*tid]*0.07 + matA[3*tid+1]*0.72 + matA[3*tid+2]*0.21;
    tid = tid + stride;
  }
}
```

Breaking down the code, the input arguments are the matrix A, the gray-scale matrix, and parameters describing the overall width and height of the image (to determine the appropriate maximum thread ID). The note at the beginning is important for interpreting the initial image array composed of BGR data. A common misconception is that the array elements would be organized by B, G, R, but the data being interpreted by openCV puts the information of an individual pixel back to back. For example, the first pixel has data B with index 0, G with index 1, and R with index 2. Strides are also used in the code to optimize the usage of threads and cover the span of the image. A stride is a value representing the overall length of threads, upon the kernel call in the main script the user specifies the number of threads per block and the number of blocks. In the kernel, these values are represented with **blockDim.x** for the number of threads per block, and **gridDim.x** for the number of blocks. Multiplying the two gives the overall number of threads. The while loop continues until the thread ID is the max length of the image, indexing each individual thread by the stride to cover the image in its entirety.

7.1.2 Allocate

Although making an image entirely gray-scale before cropping is not computationally optimized, this made the re-mapping of the image easier for cropping. The allocation was done to make edge finding easier and reduce the number of pixels dealt with for further manipulation. The cropping size was determined my using openCV region of interest (ROI), which allows for cropping to a different image size. This was not optimized on the GPU however and is mainly intended for static images, or post-processing. It was useful in determining the window size for the project, as the output had variables **r.x, r.y r.width, and r.height**. These values referred to the offset in the x direction, y direction, and the width and height from that pixel location. The mount created for the Piano Tiles project is fixed at a stationary location, with an inset for the placement of the phone. This was advantageous in being able to set a fixed cropping orientation for the image, as it was repeatable every iteration. The allocation kernel can be found below.

```
__global__ void screenAllocate(unsigned char *originalImage,unsigned char *screenImage
    , int *imageInfo, int *screenInfo){
  int imageWidth, imageHeight;
  imageWidth = imageInfo[0];
  imageHeight = imageInfo[1];

  int screenX, screenY, screenWidth, screenHeight;
  screenX = screenInfo[0];
  screenY = screenInfo[1];
  screenWidth = screenInfo[2];
  screenHeight = screenInfo[3];
  //printf("Size of image: %d, %d \n",imageWidth,imageHeight);
  //printf("Size of ROI: %d,%d,%d,%d \n",screenX,screenY,screenWidth,screenHeight);
  int tid,stride,index;
  tid = blockIdx.x*blockDim.x + threadIdx.x;
  stride = blockDim.x*gridDim.x;
  while (tid < screenWidth*screenHeight){
    index = imageWidth*(screenY + tid/screenWidth) + screenX + (tid - screenWidth*(tid
        /screenWidth));
    screenImage[tid] = originalImage[index];
    //printf("Screen Index: %d , Image Index %d\n",tid, index);
    tid = tid + stride;
  }
}
```

Breaking down the code, there are a number of intricacies involved in the remapping of desired pixels to a new image. The comments are used for helpful debugging, or in the interest of knowing the starting pixel location and any relevant information. The inputs of the function are the original image data, the cropped image data, image info (width, height), and screen info (starting position, width, height). The variable imageInfo is a two element array that consists of the width in index 0 and height in index 1. The screenInfo variable is a four element array that consists of the [starting x position, starting y position, width, height] in that order. The use of strides is again seen defined after thread ID, but is likely not used due to the cropped version being smaller than the initial image. The main calculation of the kernel occurs in the first line in the while loop. The image data are all flattened, meaning it is a single-dimension array. The calculation of the cropped position can therefore be seen as an offset of the y-index multiplied by the original image width, with the addition of the x-offset. Floor rounding is also used in the calculation to determine when to move on to the next row of the cropped image. The result is a cropped image based off the arguments given by the variable screenInfo.

7.1.3 Thresholding and Edge Finding

With a cropped image and less information to process, more computationally intensive operations can be used to create the foundation for the edge triggering. Initially, the edge-finding approach was used to actuate the solenoids that ultimately tap the screen but ended up being re-evaluated and replaced with the timing method of actuation. This edge finding technique is the foundation for the first algorithm. Edge finding is used in combination with thresholding to determine where the transition between one tile and the background occurs at. The method used is based off a gradient. If the pixel above is of the background and the current pixel value is of the tile, the pixel will signal that it is on the edge. The edge finding kernel can be found below.

```
__global__ void edgeFind(unsigned char *grayData, unsigned char *edge, int width, int
    height){
int tid, stride, threshold, offset;
threshold = 180;
offset = 50;
tid = blockIdx.x*blockDim.x + threadIdx.x;
stride = blockDim.x*gridDim.x;
while (tid < width*height){
  if (tid < width*(height-offset)){
    if (grayData[tid] < threshold && grayData[tid-width] > threshold && grayData[tid-
        width-1] > threshold && grayData[tid-width + 1] > threshold && grayData[tid-1]
         < threshold && grayData[tid-2] < threshold){
      edge[tid+offset*width] = 255; // set to white
    } else {
      edge[tid+offset*width] = 0;
    }
  }
tid = tid + stride;
```

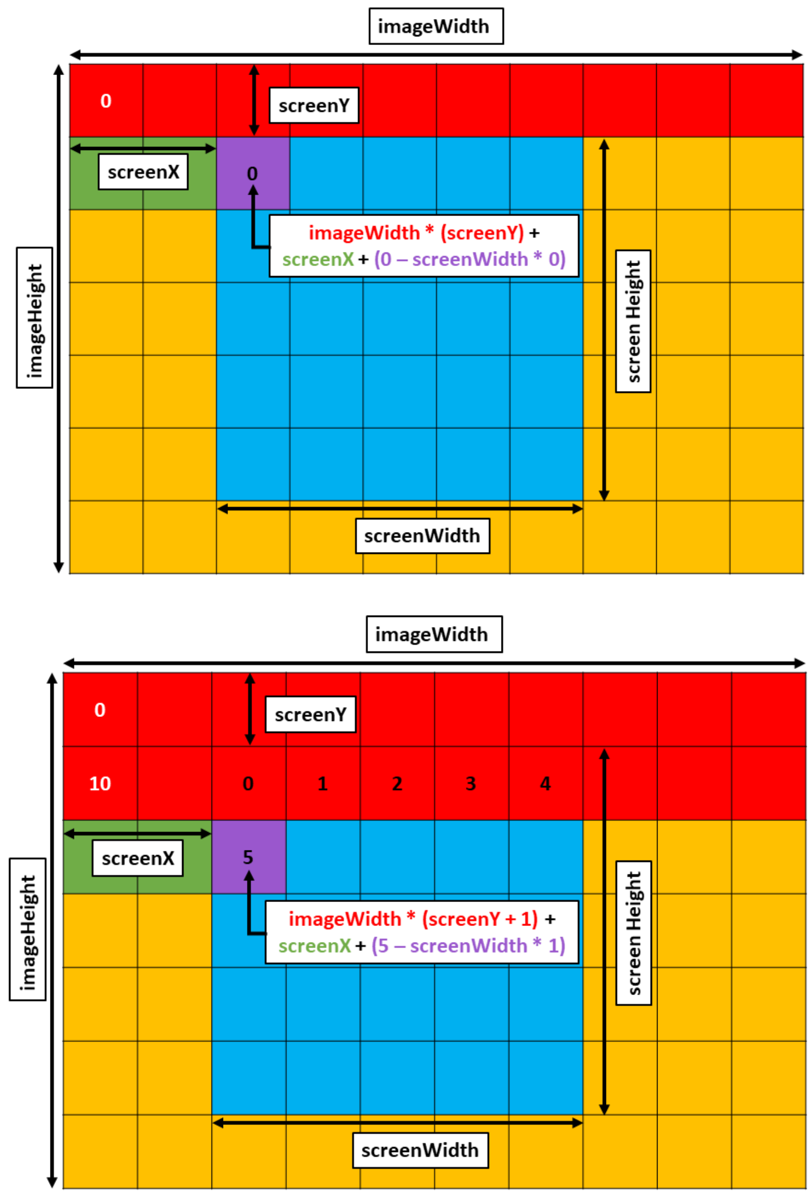

Figure 6: A visual representation of how the screenAllocate kernel crops an input image using thread IDs. The calculation of the desired index of the original image and the index of the cropped image are based of a y-offset (screenY), x-offset (screenX), and the cropped width (screenWidth) and height (screenHeight).

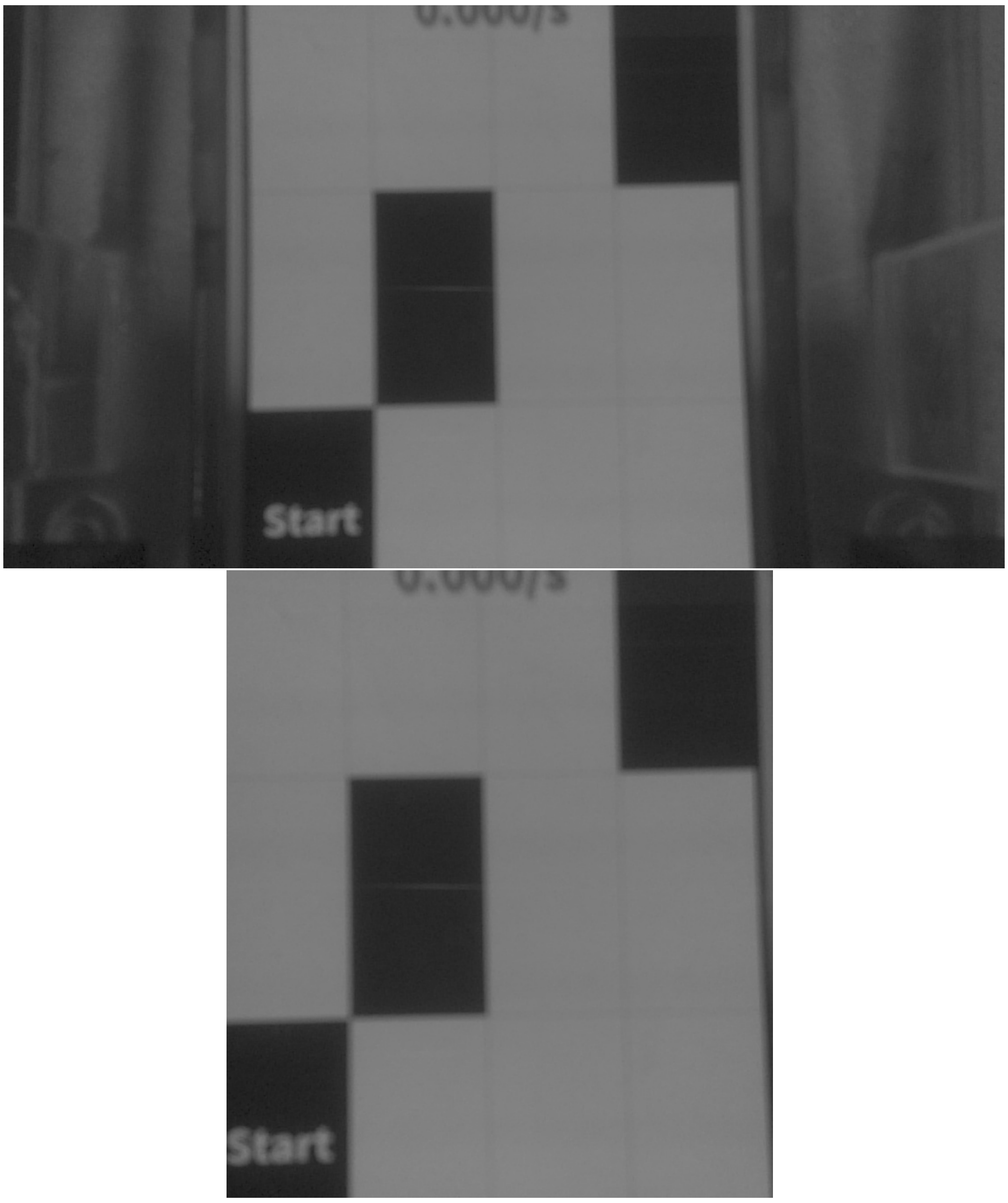

Figure 7: The results from the cropping operation of the gray scale image.

```
  }
}
```

Breaking down the code, the inputs to the function are the input image (grayscale-cropped), the output image array (edge), and the parameters of the image (width, height). Note that these are representative of the cropped image, not the original one that was read-in by the camera. The threshold value of 180 is used in the current iteration, depending on the camera being used the value was adjusted to reflect the pixel intensity that was on the cusp of reading black (the tile intensity). Thresholding sets any pixel that is above that threshold to white (255) and anything below to black (0). The effect of light intensity plays a role in determining what the threshold cutoff is. For the lighting in the area the code was developed, the cutoff is at a value of 180 for the Raspberry Pi camera on the Jetson Nano. This can be adjusted in the edgeFind kernel of the main scripts (gpuBest.cu). Instead of making two kernels to do two operations (threshold and edge finding), they were combined to a single kernel. The pixel offset is set to 50 in the script due to move the edge lines down 50px to trigger the solenoids earlier to help account for actuation time. This was a trial and error approach to determine what would work with the solenoids. The edge-determination is based off several sections of logic. If the current pixel is black, the pixel one row directly above, above and to the left, above and to the right are all white. Then 50px below the current pixel is changed to white. Otherwise, if the logic is not satisfied, the pixel is set black. The edge finding can be shown in figure 8 and with the white pixels corresponding to the true statements in the edgeFind kernel representing the intersection of a transition between black and white.

7.1.4 Dilate

Another common technique commonly used in image processing is the usage of erosion and dilation. Dilation is the process of expanding a current section of pixels. For instance, if a single pixel is white and the surrounding pixels are black, dilation can be used to make the surrounding pixels white. The following is the kernel for dilating the edge-detection image and making the width of the lines more noticeable as well as filling in the gaps between pixels if the edge-finding kernel did not accurately find all the edges in the given row.

```
__global__ void dilate(unsigned char *image, int width, int height){
int tid, stride;
tid = blockIdx.x*blockDim.x + threadIdx.x;
stride = blockDim.x*gridDim.x;
while (tid < (width-1)*height && tid > width ){
  if (image[tid] == 255 && image[tid-width] == 0 && image[tid+width] == 0){
    image[tid-width] = 255; // set pixel above below left and right to white
    image[tid+width] = 255;
    image[tid-1] = 255;
    image[tid-2] = 255;
    image[tid+1] = 255;
    image[tid+2] = 255;
  }
  tid = tid + stride;
 }
}
```

Breaking down the code, the input arguments are only the image data, as as well as the image dimensions. This is because the output will just be an extension of the original image with thicker lines. The while loop includes checking to see if the threadID index corresponds to a white pixel (edge) and the pixels above and below are black. If this is true, then the surrounding pixels will be set to white, by one pixel above and below, and in two pixels looking forwards and backwards. The resultant image can be found to have thicker lines and is a more suitable surface to use for actuation.

7.1.5 Erosion

As mentioned in the previous section, the other component of dilation is erosion. This operation is the converse, and is usually used to make an image less noisy (remove floating pixel discolorations). If a pixel is surrounded by pixels of a different color, then the initial pixel will be changed to the surrounding pixel value. The kernel for erosion can be found below.

```
__global__ void erosion(unsigned char *image, int width, int height){
  int tid, stride;
  tid = blockIdx.x*blockDim.x + threadIdx.x;
  stride = blockDim.x*gridDim.x;
```

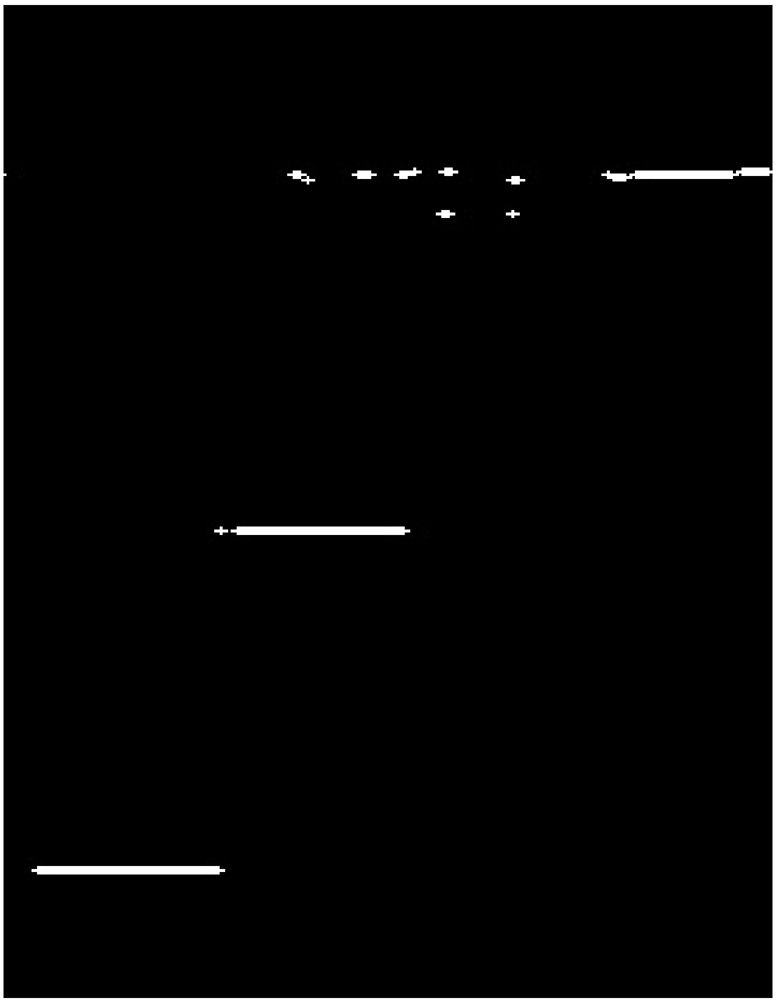

Figure 8: The usage of the erosion kernel with two rounds of dilation to remove noise from the edge-detection algorithm and thicken the lines for usage with edge-triggering.

```
    while (tid < width*(height-2) && tid > 2*width ){
      if (image[tid] == 255 && image[tid-width] == 0 && image[tid + width] == 0){
        image[tid] = 0;
        image[tid-width] = 0; // set pixel above below left and right to black
        image[tid+width] = 0;
        image[tid -1] = 0;
        image[tid+1] = 0;
      }
      tid = tid + stride;
    }
  }
```

This operation was used to rid of faulty edges after testing with solely the dilation script. The result was erroneous solenoid actuation due to amplifying noise with dilation. By utilizing erosion before dilation, the data was combed for noise and mitigated its effect. The only unfortunate consequence of using erosion is that it thins out the image, and in the case of the piano tiles project dilation was needed to be used multiple times to counteract the initial erosion which was more computationally intensive than initially planned. The result of the erosion script and two-runs of dilation can be found in figure 8.

7.1.6 Spacing

An effort was made to determine the exact distance each tile moved each frame, in order to predict when to tap the screen based on the previous position. The way this was achieved was through two kernel functions **addArr** and **spacing**. Adding would take the pixels of the previous edge-finding frame, and superimpose the values on the current frame. This results in an overlap with two lines visible on the screen for every edge. As the game progressed, the lines would grow further apart, visual indicative of the increase in tile speed. The second function, Spacing, would look below the current pixel in the kernel by more than 10px and determine if there was the other edge line, and would return the distance between the two. The average was taken of the spacing results and that value was the pixel velocity of the tiles. The adding function can be found below,

```
__global__ void addArr(unsigned char *arrA, unsigned char *arrB,unsigned char *output,
     int width, int height){
  int tid, stride;
  tid = blockIdx.x*blockDim.x + threadIdx.x;
  stride = blockDim.x*gridDim.x;
  while (tid < width*height){
    output[tid] = arrA[tid] + arrB[tid];
    tid = tid + stride;
  }
}

__global__ void spacing(unsigned char *pixelData, unsigned int *difference,unsigned
     int *count, unsigned int width, int height){
  //extern__shared__int difference[];
  //extern__shared__int count;
  int tid,i,stride;
  tid = blockIdx.x*blockDim.x + threadIdx.x;
  stride = blockDim.x*gridDim.x;
  if (tid < width*height){
    if (pixelData[tid] == 255){
      int init = tid;
      i = init + 10*width; // start looking 10 pixels down from current position
      while (i < width*height && i < init + width*50){
        if (pixelData[i] == 255){
          difference[tid] = (i - init)/width;
          count[tid] = 1;
          //printf("%d  ",difference[tid]);
          break;
        } else {
            difference[tid] = 0;
            count[tid] = 0;
        }
        i = i + width;
      }
    } else {
      difference[tid] = 0;
      count[tid] = 0;
    }
  }
}
```

Breaking down the code, the inputs to the kernel are the overlaid edge-detection image, the output spacing difference array, the count array, and the image width and height. This is not the most efficient way to determine the spacing between pixel, however it was able to determine a baseline estimate for how the the speed of the screen varied with respect to time. The code checks the input image at the threadID and if the value of that pixel is white, then the loop continues to scan down the image starting 10px below the original position and continues until 50px below the original position. The difference between the two is returned at the location of the initial threadID in the difference array, also the count array is set to 1 at that threadID. This is to later determine the average pixel distance between the two edge-detected lines. An example of the superposition can be found in figure 9.

7.2 Annotation

A useful tool to prototype actuation was to annotate the screen with openCV to show when certain logic was being executed. By annotating the screen with green circles, viewing the recorded video post-execution allowed for the review of how the solenoid actuation failed, or any faults in edge-detection.

```
void drawCircle(cv::Mat img, cv::Point center){
  cv::circle(img,center,10,cv::Scalar(0,255,0),cv::FILLED,cv::LINE_8);
}
```

The function is simplistic and takes arguments that are available using the cv namespace. The inputs of the function are the image to overlay the green-circles on, and the point at which they are centered

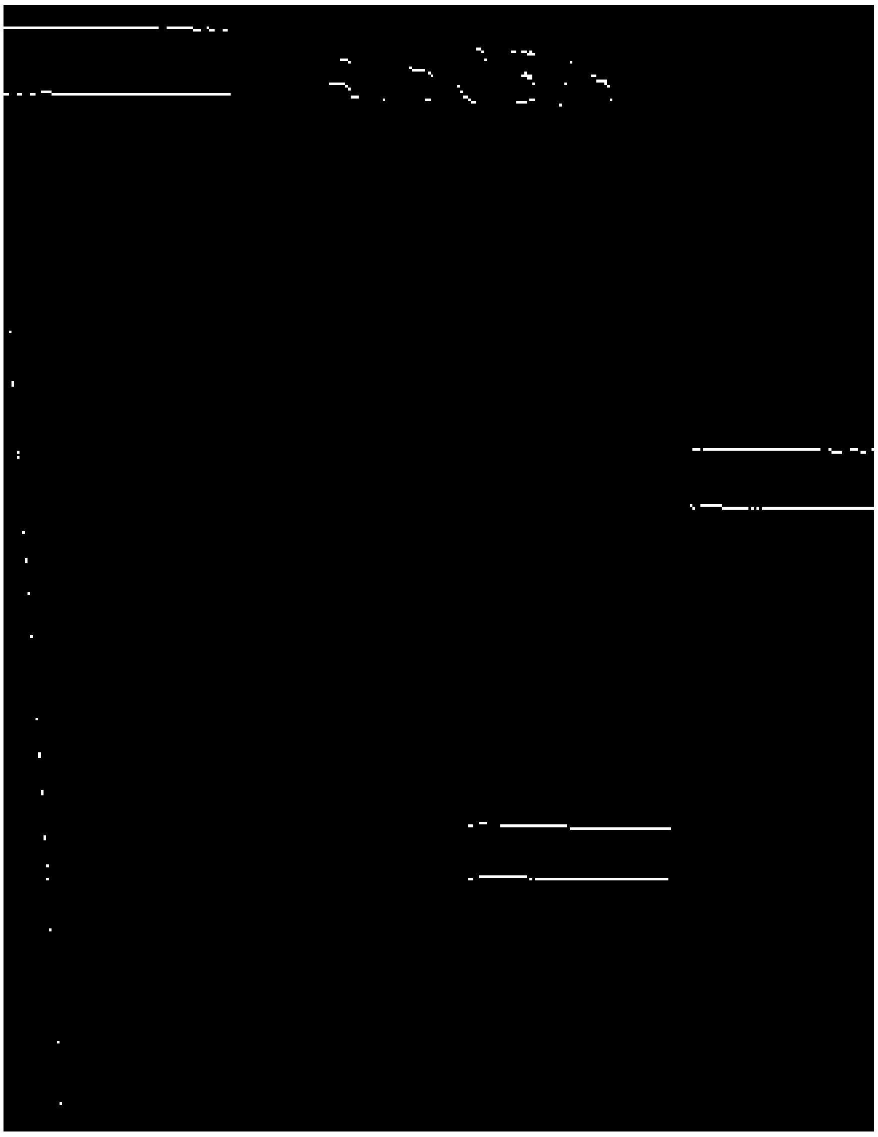

Figure 9: The super position of two frames, the previous edge-finding result, and the current frame edge-finding image. The spacing kernel looks at the difference between the two lines and outputs a matrix of the difference values.

around. The argument cv::Point is a ordered pair of the location, with respect to (x-offset, y-offset). The point is measured from the top left of the matrix. The other properties of the circle object is the radius, color (0, 255, 0) corresponds to BGR and is entirely green, and whether or not the circle is filled and the line-type. The annotated version of a frame is shown in figure 10.

8 Implementation - Main Script

8.1 General Setup

The general setup includes the main headers and necessary libraries, setting up access of the GPIO pins, allocating memory, setting up the camera, configuring the number of threads and blocks, and initial declarations of variables. This general setup is the same for both of the algorithms.

```
#include <opencv2/opencv.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <iostream>
#include <cuda_runtime_api.h>
#include <cuda.h>
#include <string>

//Added on with tapping functionality
#include <string.h>
#include <vector>
#include <stdio.h>
#include <stdlib.h>
#include <termios.h>
#include <time.h>
#include <sys/time.h>
#include <unistd.h>
#include "jetsonGPIO.h"

#include <errno.h>
#include <fcntl.h>
#include <poll.h>

#define NUM 10000
#define Frames 120
#define LENGTH(x) (sizeof(x)/sizeof((x)[0]))
```

The libraries are all included due to some operation within the main script. As the code progressed, the libraries remained stationary, and likely not all of them are used in the final iteration. The function LENGTH() is to determine how long an array is and was used for prototyping. After the including statements at the top of the file, all the kernels described previously are placed underneath. Afterwards is the declaration of the main function.

```
int main()
{
  // Initialize timer settings
  float calcTimer = 0;
  float tap = 0;
  cudaEvent_t start, stop, tapInit, tapCurrent;
  cudaEventCreate(&start);
  cudaEventCreate(&stop);
  cudaEventCreate(&tapInit);
  cudaEventCreate(&tapCurrent);
  //float GPUtimer, CPUtimer;

  // Solenoid setup settings
  jetsonTX2GPIONumber pin1 = gpio12;//gpio393;
  jetsonTX2GPIONumber pin4 = gpio38;//gpio395;
  jetsonTX2GPIONumber pin2 = gpio200;//gpio297;
  jetsonTX2GPIONumber pin3 = gpio149;//gpio394;
  gpioExport(pin1) ;
  gpioSetDirection(pin1,outputPin) ;
  gpioExport(pin2) ;
  gpioSetDirection(pin2,outputPin) ;
  gpioExport(pin3) ;
  gpioSetDirection(pin3,outputPin) ;
  gpioExport(pin4) ;
  gpioSetDirection(pin4,outputPin) ;
```

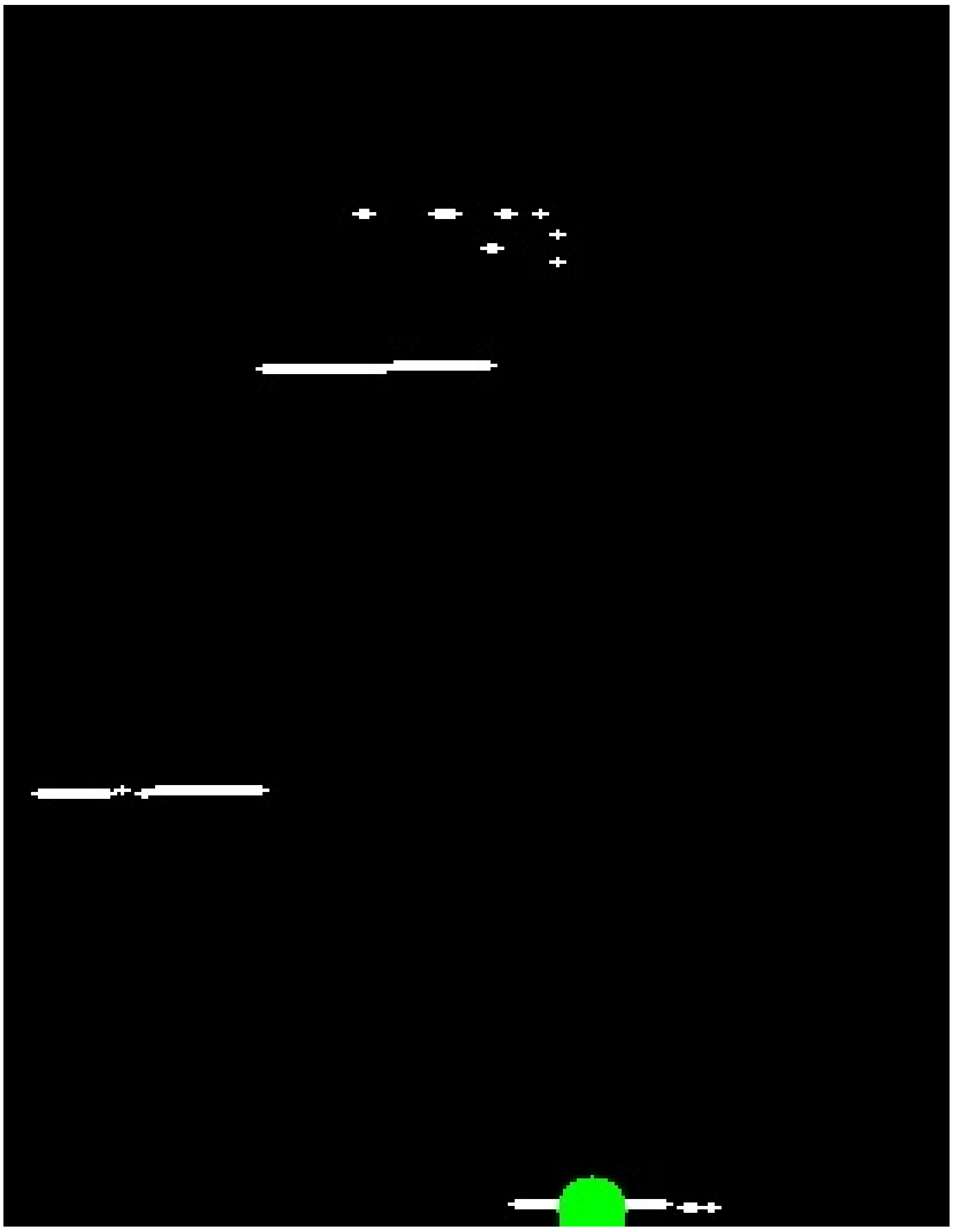
Figure 10: Example onscreen annotation with a green circle to indicate where the trigger would have acted for the solenoids.

This section initilizes settings and creates timing events and configures GPIO pins accordingly. The cuda events seen at the top are to track the timing of operations, two of the significant uses in this project are to track frames per second (FPS) to ensure the computational cost is not too high, and the time delay between signals sent to the solenoids (used in the second algorithm for timing).

```
// ORIGINAL IMAGE
//————————————————————————————————————————————————————————————————
  // Get initial image and print
  cv::Mat img;
  cv::VideoCapture cap("nvarguscamerasrc ! video/x-raw(memory:NVMM),width=1280,height
      =720,framerate=60/1,format=NV12 ! nvvidconv flip-method=2 ! nvvidconv ! video/x-
      raw, format=(string)BGRx ! videoconvert ! video/x-raw, format=(string)BGR ! 
      appsink",cv::CAP_GSTREAMER);
  if (!cap.isOpened()){
    printf("Error getting Stream \n");
  }
  cap >> img;
  int imageWidth = img.cols;
  int imageHeight = img.rows;
  printf("Resolution: %d x %d \n",imageWidth, imageHeight);
  int width = 624;
  int height = 717;
  unsigned int numThreads, numBlocksImage, numBlocksScreen;
  numThreads = 512; // good number for multiple of 32
  numBlocksImage = (imageWidth*imageHeight + numThreads - 1)/numThreads;
  numBlocksScreen = (width*height + numThreads - 1)/numThreads;
  printf("Number of Blocks Cropped: %d  ", numBlocksScreen);
```

This is the first image captured by the camera on setup to verify the correct position and settings. The gstreamer pipeline can be seen in the second line of code using nvarguscamerasrc. A width and height value are determined from the ROI openCV tool mentioned previously. Additionally, this is where the number of blocks and threads are configured for both the allocation to the cropped image and the initial grayscale calculation. Both numbers of blocks are configured as previously described to optimize the number of blocks used.

```
  // SETUP SETTINGS
//————————————————————————————————————————————————————————————————
  // Allocate device and host
  unsigned char *matA, *screenData, *grayData, *edge, *prevArr, *output;
  int *imageInfo, *screenInfo;
  unsigned int *difference, *count;
  cudaMallocManaged(&matA, sizeof(unsigned char)*imageWidth*imageHeight*3);
  cudaMallocManaged(&grayData, sizeof(unsigned char)*imageWidth*imageHeight);
  cudaMallocManaged(&screenData, sizeof(unsigned char)*width*height);
  cudaMallocManaged(&edge, sizeof(unsigned char)*width*height);
  cudaMallocManaged(&prevArr, sizeof(unsigned char)*width*height);
  cudaMallocManaged(&output, sizeof(unsigned char)*width*height);
  cudaMallocManaged(&imageInfo, sizeof(int)*2);
  cudaMallocManaged(&screenInfo, sizeof(int)*4);
  cudaMallocManaged(&difference, sizeof(unsigned int)*width*height);
  cudaMallocManaged(&count, sizeof(unsigned int)*width*height);
```

This section is where all the data management and allocation occurs. Variables for each array are declared as pointers and have memory allocated to them using the *cudaMallocManaged*() command. The size of each allocation varied, but the *sizeof*() command was used to appropriately determine the amount of memory to use.

```
// GPU CALCULATION
//——————————————————————————————————————————————————————————————
  // Initial assignment to previous arr
  int imageInfoHost[2] = {imageWidth,imageHeight};
  int screenInfoHost[4] = {334, 1, width, height}; // 202, 110
  cudaMemcpy(imageInfo, imageInfoHost, 2*sizeof(int), cudaMemcpyHostToDevice); // FOR
      COPYING ARRAY
  cudaMemcpy(screenInfo, screenInfoHost, 4*sizeof(int), cudaMemcpyHostToDevice); // FOR
      COPYING ARRAY
  printf("Size of image: %d, %d \n",imageInfo[0],imageInfo[1]);
  printf("Size of ROI: %d,%d,%d,%d \n",screenInfo[0],screenInfo[1],screenInfo[2],
      screenInfo[3]);
  cudaMemcpy(matA, img.data, imageWidth*imageHeight*3* sizeof(unsigned char),
      cudaMemcpyHostToDevice); // FOR COPYING ARRAY
  gpu_grayscale<<<numBlocksImage,numThreads>>>(matA,grayData,imageWidth,imageHeight);
      cudaDeviceSynchronize();
```

```
screenAllocate<<<numBlocksScreen,numThreads>>>(grayData,screenData,imageInfo,
    screenInfo); cudaDeviceSynchronize();
edgeFind<<<numBlocksScreen,numThreads>>>(screenData,prevArr,width,height);
    cudaDeviceSynchronize();
char c; // for waitkey
cv::Mat test(cv::Size(width,height),CV_8UC1,screenData);
```

This section is the configuration of the imageInfo and screenInfo that are used in many of the kernel function calls. The *cudaMemcpy*() command is used to copy the image from the capture to an array *matA*. The initial image processing is performed on the sample image, using the kernels defined in the previous sections of the report.

```
// Video Writer
cv::VideoWriter writer;
int codec = cv::VideoWriter::fourcc('M','J','P','G');
double fps = 15;
std::string filename = "./converted.avi";
writer.open(filename,codec,fps,test.size(),1); // boolean at end is color
if (!writer.isOpened()){
  printf("Unable to Open Video\n");
  return -1;
}
```

This section declares the video writer to save video files for later viewing, the current file being saved is converted.avi using the fourcc code (MJPG).

8.2 Algorithm 1 - Edge Triggering

This method makes use of the edge finding techniques as well as erosion and dilation to improve the quality of the resultant image. The only problem with this technique was the variability in edge finding. For example, at higher velocities (approximately 5 in/s), the edge finding becomes more intermittent and the ultimate failure is based off a missed actuation rather than a wrong actuation. The record with this setting was approximately 5.5 in/s.

```
for (;;){
    //capture and calculations
    cap >> img;
    cudaMemcpy(matA, img.data, imageWidth*imageHeight*3* sizeof(unsigned char),
        cudaMemcpyHostToDevice); // FOR COPYING ARRAY
    gpu_grayscale<<<numBlocksImage,numThreads>>>(matA,grayData,imageWidth,imageHeight)
        ; cudaDeviceSynchronize(); // sync threads and cpy mem
    screenAllocate<<<numBlocksScreen,numThreads>>>(grayData,screenData,imageInfo,
        screenInfo);
    cudaDeviceSynchronize();
    edgeFind<<<numBlocksScreen,numThreads>>>(screenData,edge,width,height);
        cudaDeviceSynchronize();
    //Thickening of lines for trigger
    erosion<<<numBlocksScreen,numThreads>>>(edge,width,height);
    cudaDeviceSynchronize();
    dilate<<<numBlocksScreen,numThreads>>>(edge,width,height);
    cudaDeviceSynchronize();
    //Triggering
    cudaEventRecord(start);
    trigger<<<1,numThreads>>>(edge,width,height,value,1);cudaDeviceSynchronize();
    trigger<<<1,numThreads>>>(edge,width,height,value,2);cudaDeviceSynchronize();
    trigger<<<1,numThreads>>>(edge,width,height,value,3);cudaDeviceSynchronize();
    trigger<<<1,numThreads>>>(edge,width,height,value,4);cudaDeviceSynchronize();
    cudaEventRecord(tapCurrent);
    cudaEventSynchronize(tapCurrent);
    cudaEventElapsedTime(&tap,tapInit, tapCurrent);
    if (tap > 260-velocity){ //260 was chosen by trial and error
      for (int i =0; i<4; i++){
            if (value[i] == 1){
              gpioSetValue(pinList[i],on);
              usleep(35000);
              gpioSetValue(pinList[i],off);
              value[0] = 0; value[1] = 0; value[2] = 0; value[3] = 0;
              break;
            }
      }
      cudaEventRecord(tapInit);
```

```
        } else {
          value[0] = 0; value[1] = 0; value[2] = 0; value[3] = 0;
        }
        cudaEventRecord(stop);
        cudaEventSynchronize(stop);
        cudaEventElapsedTime(&calcTimer, start, stop);
        //printf("Loop Timer: %f \n",calcTimer);
        calcTimer = 0;

        //saving
        cv::Mat build(cv::Size(width,height),CV_8UC1,edge);
        cv::Mat videoFrameGray;
        cv::cvtColor(build,videoFrameGray,CV_GRAY2BGR); //3 array from grayscale
        writer.write(videoFrameGray);
        if (initial =='d'){ // keypress of d at beggining displays annotated live-video
          if (velocity < 260){ // Do not want to index higher than max value
            velocity += 0.18;
          }
          cv::imshow("GPU",videoFrameGray);
          c = cv::waitKey(1);
          if (c =='p') // press 'p' to printout a frame during the loop
            cv::imwrite("differenceProgress.png",build);
          if (c ==' '){
                break;
          }
        }else{ // otherwise computationally faster
            if (velocity < 260)
              velocity += 0.101;
        }
        if (c ==' ') //stop script
          break;
    }
```

The benefits of using this method is that the majority of calculations are performed on the GPU side, and were able to be sped up considerably compared to pixel calculations on the CPU for the image array. The variables to change for further iteration of a timed loop would be to change the value of *velocity* which controls the delay between loop executions, a larger value indicates a larger decrease in loop duration every loop.

8.2.1 Edge Determination

In this algorithm, the majority of the processing is performed on the GPU, however in order to have the triggering on the device side of processing another kernel was built using shared variables to determine if an edge was in a given column. The operation was repeated for each lane, and if the resultant calculation returned high, then the solenoid in that lane is set to actuate.

```
  __global__ void trigger(unsigned char *image, int width, int height, int *isTriggered,
      int lane){
    __shared__ int value[4];
    int tid, stride, offset, quarter;
    offset = width/8; //quarter of screen
    quarter = width/4; //want centered in middle of tile (1/8 offset)
    tid = blockIdx.x*blockDim.x + threadIdx.x;
    stride = blockDim.x*gridDim.x;
    while (tid < 150){ //look 150 px above bottom of frame
      if (image[tid*width + offset + quarter*(lane-1) + (width*(height-150))] == 255 ){
      value[lane-1] = 1;
      }
      tid = tid + stride;
    }
    __syncthreads();
    if (value[lane-1] == 1){
      isTriggered[lane-1] = 1;
    }
  }
```

The syncthreads functionality and the shared variable is what makes the triggering able to be done on the GPU side. The __shared__ declaration allows all threads within a given block to share the same variable/array. The value is an array of four, to store information about each tile lane (column). During the kernel call, the number of blocks is specified to be one because the total number of pixels being looked at is 150px, and simplifies the linking of shared variables. The __syncthreads() command is used to see if there was an edge in any of the 150px being looked at and sets them all to 1. This sets the input *isTriggered* to 1 regardless of which thread is the last one to write to it, the lane argument allows for the kernel to be called on all four lanes independently. For a different range to look in, the value of 150 in the while loop and the index of the term in the subsequent if statement could both change.

8.3 Algorithm 2 - Time Triggering

The second algorithm is more based off the previous project iteration on the Raspberry Pi, where a small number of pixel values are interpreted to determine the likely-hood of a tile in that area. The script used for triggering used the value of the pixel 10px above the bottom the the camera, and the pixel 90px above that. If both are below a certain threshold then the solenoid in that column is triggered. In addition to the interpretation of individual pixel values, a timing aspect was implemented to restrict the number of actuation per tile length. Without the timer, the solenoid would continuously actuate until the end of the tile. This was proven to be problematic as tile velocity increased, due to the fact the solenoid could still maintain contact with the screen after the tile had passed (causing the end of a run due to a faulty tile press).

8.3.1 Timing

A parallel approach to the edge-triggering method was to use a more robust image (no edge-detection), but with timing to control how often the solenoids were allowed to actuate. The timing was accomplished using cudaEvents and the accompanying timer. The timing was not accomplished through a kernel, as parallel timing would not be beneficial in this instance and was instead placed into the main loop of the script. The following code snippet shows how the timing operation was executed.

```
//Trigger Parameters
int quarter = width/4;
int offset = quarter/2;
int prevVelocity = 0;
int trigger[4] = {offset, offset + quarter, offset + quarter*2, offset + quarter*3};
```

The trigger parameters are used for determining where to annotate the screen and look for edges. This is used exclusively for the timing case, as the other version has a kernel named *trigger*.

```
// Converter Loop
//------------------------------------------------------------
float velocity = 0;
printf("Press key 'd' to display feed\n");
char initial;
cv::imshow("initial frame",test);
initial = cv::waitKey(0);
cudaEventRecord(tapInit);
//Startup tap
gpioSetValue(pin1,on); usleep(35000);
gpioSetValue(pin1,off); usleep(10000);
gpioSetValue(pin2,on); usleep(35000);
gpioSetValue(pin2,off); usleep(10000);
gpioSetValue(pin3,on); usleep(35000);
gpioSetValue(pin3,off); usleep(10000);
gpioSetValue(pin4,on); usleep(35000);
gpioSetValue(pin4,off);
cudaDeviceProp prop;
cudaGetDeviceProperties(&prop,0);
printf("MaxThreads: %d\n",prop.maxThreadsPerBlock);
```

This is the initial tapping sequence that is used to actuate the first tile. The solenoid housing is placed in such a manner where the rapid sequential tapping of all four solenoids only registers on the correct tile. Afterwards the algorithm takes place and continues in the main section of the loop. The last few lines are to determine the maximum number of threads per block, as was mentioned in a previous section of the report.

```
for (;;){
  cudaEventRecord(start);
  for (int j = 0; j< Frames; j++){
  //capture and calculations
  cap >> img;
  cudaMemcpy(matA, img.data, imageWidth*imageHeight*3* sizeof(unsigned char),
      cudaMemcpyHostToDevice); // FOR COPYING ARRAY
gpu_grayscale<<<numBlocksImage,numThreads>>>(matA,grayData,imageWidth,imageHeight);
    cudaDeviceSynchronize(); // sync threads and cpy mem
screenAllocate<<<numBlocksScreen,numThreads>>>(grayData,screenData,imageInfo,
    screenInfo); cudaDeviceSynchronize();
edgeFind<<<numBlocksScreen,numThreads>>>(screenData,edge,width,height);
    cudaDeviceSynchronize();
//saving
cv::Mat build(cv::Size(width,height),CV_8UC1,edge);
cv::Mat videoFrameGray;
cv::cvtColor(build,videoFrameGray,CV_GRAY2BGR); // 3 array of grayscale for saving to
    file
for (int i = 0; i< LENGTH(trigger); i++){
  int pixelCount = 10;
      int index;
      int sum = 0;
      index = width*height - trigger[i] - (width*pixelCount);
      pixelCount+=1;
  if (edge[index] == 255 && edge[index-90*width] == 255){ // if two points in the
      first tile are both the trigger value (255)
    cudaEventRecord(tapCurrent);
    cudaEventSynchronize(tapCurrent);
    cudaEventElapsedTime(&tap,tapInit, tapCurrent);
        if (edge[index] == 255 && tap > 225-velocity){  //check for the amount of time
            elapsed between the last tap and now
          cudaEventRecord(tapInit);
          //printf("Between Taps: %f \n",tap);
          tap = 0;
          //drawCircle(videoFrameGray,cv::Point(trigger[3-i],height-pixelCount)); //
              annotation
          if (i == 3){
            gpioSetValue(pin1,on);
            usleep(35000);
            gpioSetValue(pin1,off);
          }
          if (i == 2){
            gpioSetValue(pin2,on);
            usleep(35000);
            gpioSetValue(pin2,off);
          }
          if (i == 1){
            gpioSetValue(pin3,on);
            usleep(35000);
            gpioSetValue(pin3,off);
          }
          if (i == 0){
            gpioSetValue(pin4,on);
            usleep(35000);
            gpioSetValue(pin4,off);
          }
        }
      }
    }
    writer.write(videoFrameGray);  // write to video file - commented for speed
        purposes
    if (initial =='d'){
      if (velocity < 225){
        velocity += 0.07; //experimentally found speed up of the game for the current
            script configuration
      }
      cv::imshow("GPU",videoFrameGray);
      c = cv::waitKey(1);
      if (c =='p') //printout the current frame
        cv::imwrite("differenceProgress.png",build);
      if (c ==' '){
        break;
      }
    }else{ // if no display is needed
      if (velocity < 225) // do not want to surpass max velocity value
        velocity += 0.0504; //experimentally found speed up of the game for the
            current script configuration
    }
  }
  if (c ==' ') //exit loop, terminate script
```

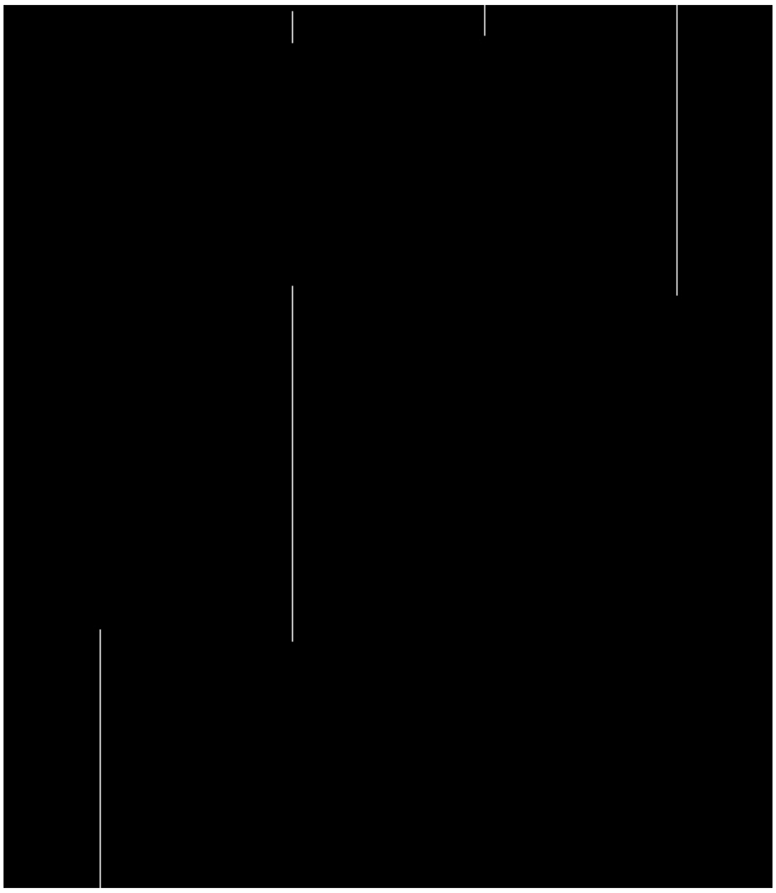

Figure 11: The information being sent to the trigger, and acted upon on the basis of a timer.

```
        break;
        cudaEventRecord(stop);
        cudaEventSynchronize(stop);
        cudaEventElapsedTime(&calcTimer, start, stop);
        //printf("FPS GPU: %f \n",Frames/calcTimer*1000);
        calcTimer = 0;
    }
```

To break down the code, the event timers run on milliseconds and were also beneficial in determining FPS of the incoming video feed (often hovered around 30 fps). The if statement evaluates if an adequate amount of time has passed since the last solenoid actuation, and if true then allows for another actuation. The velocity variable is indexed by a fixed value, this was a trial and error estimate that was implemented assuming a constant screen acceleration. It should be noted that even if the if-statement was not executed, the velocity is continuously indexed. This trigger was optimized to only tap once per tile, and was hard-coded in a a fixed rate to account for the gradual increase in screen velocity. The time triggering configuration yielded the maximum speed observed for this project, at a speed of 9.220 in/s. This record surpassed the previous record on the Raspberry Pi using a different algorithm but similar concept. For future implementations of this method, changing the variable that is in charge of tap timing (velocity) could improve the overall max-speed using.

8.4 Algorithm 3 - Bare-Bones

In a final effort, I wanted to try another method of edge detection. The initial usage of dilation and erosion were helpful in creating a thicker line but the edge-finding was not consistent. A new simple edge detection algorithm was implemented

```
__global__ void edgeFind(unsigned char *grayData, unsigned char *edge, int width, int
    height){
  int tid, stride, threshold, offset;
  threshold = 120; //180
```

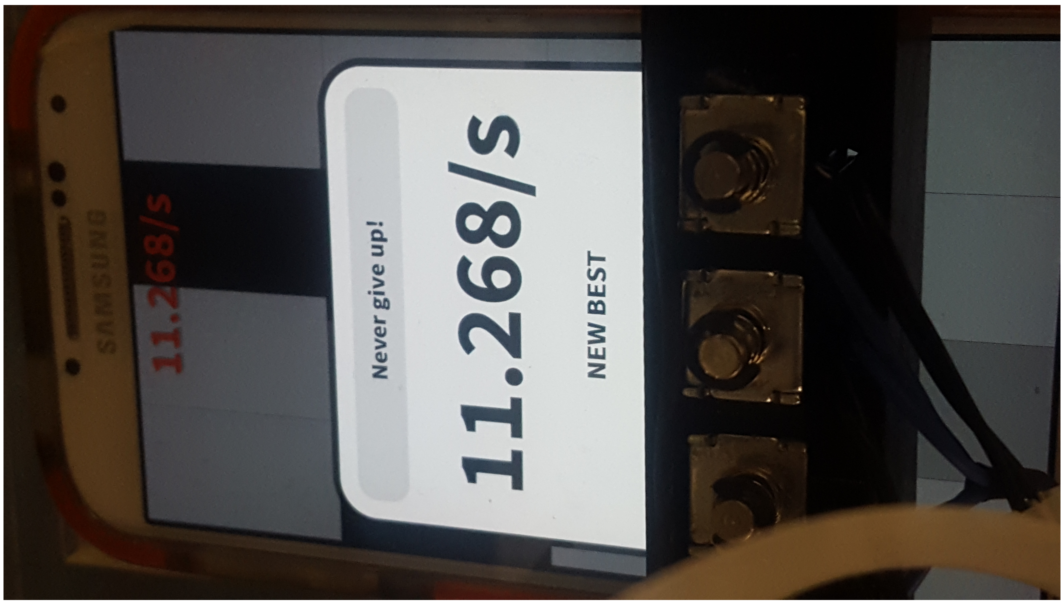

Figure 12: The Record held by the timer triggering method of actuation, at a final screen speed of 9.220 in/s.

```
  offset = width/8; //50
  int quarter = width/4;
  tid = blockIdx.x*blockDim.x + threadIdx.x;
  stride = blockDim.x*gridDim.x;
  while (tid < 4*height && tid > 24){
      if (grayData[tid*quarter + offset] < threshold && grayData[(tid-4)*quarter +
          offset] < threshold && grayData[(tid-24)*quarter + offset] > threshold){
        edge[tid*quarter+offset] = 255; // set to white
      } else {
        edge[tid*quarter+offset] = 0;
      }
      tid = tid + stride;
  }
}
```

The kernel looks at the thread id in each quarter, looks at the pixel above, and six pixels above. This edge-detection is for black to white easier to get conclusive results due to not dealing with the faded coloring that appears after a tile has been tapped. If the pixel and the one above it are both black, and six pixels above is white, then the output array is set to white at that threadID. For variations of the algorithm, the variables in the if statement could be varied to look at different pixels, and the threshold value could change dependent on the lighting situation. By using this, dilation and erosion were avoided and reduced computation time. Additionally, the triggering kernel was modified to the following:

```
  __global__ void trigger(unsigned char *image, int width, int height, int *isTriggered,
       int lane){
  __shared__ int value[4];
  int tid, stride, offset, quarter;
  offset = width/8; //50
  quarter = width/4;
  tid = blockIdx.x*blockDim.x + threadIdx.x;
  stride = blockDim.x*gridDim.x;
  while (tid < 130){
    if (image[tid*width + offset + quarter*(lane-1) + (width*(height-410))] == 255 ){
      value[lane-1] = 1;
    }
    tid = tid + stride;
  }
  __syncthreads();
  if (value[lane-1] == 1){
    isTriggered[lane-1] = 1;
    //printf("isTriggered %i: %i\n",lane,isTriggered);
  }
}
```

This shifts the range of triggering up to about halfway up the screen (410 pixels), and looks down 130 pixels down from there. If an edge is found anywhere between The delay between solenoid taps was set to be 50 ms, and ended up producing the fastest speed of 11.268 in/s without using a timer. The algorithm is usually able to get around 8 in/s once it passes 5 in/s. This method is completely based off the image pixel locations, and most likely could be sped up further. The reason is because each solenoid can be tapped around 25hz and each tile is being tapped multiple times even at failure. The main section of the code was adjusted to remove as much as possible including dilation, erosion, and the saving of video to increase speed.

```
gpioSetValue(pin1,on);
usleep(35000);
gpioSetValue(pin1,off);
usleep(10000);
gpioSetValue(pin2,on);
usleep(35000);
gpioSetValue(pin2,off);
usleep(10000);
gpioSetValue(pin3,on);
usleep(35000);
gpioSetValue(pin3,off);
usleep(10000);
gpioSetValue(pin4,on);
usleep(35000);
gpioSetValue(pin4,off);
for (;;){
    cap >> img; // Capture Frame
    cudaMemcpy(matA, img.data, imageWidth*imageHeight*3* sizeof(unsigned char),
        cudaMemcpyHostToDevice); // Copy Array
    gpu_grayscale<<<numBlocksImage,numThreads>>>(matA,grayData,imageWidth,imageHeight)
        ; cudaDeviceSynchronize();
    screenAllocate<<<numBlocksScreen,numThreads>>>(grayData,screenData,imageInfo,
        screenInfo); cudaDeviceSynchronize();
    edgeFind<<<numBlocksScreen,numThreads>>>(screenData,edge,width,height);
        cudaDeviceSynchronize();
    cv::Mat build(cv::Size(width,height),CV_8UC1,edge);

    cv::Mat videoFrameGray;
    cv::cvtColor(build,videoFrameGray,CV_GRAY2BGR); // 3 array of grayscale for
        annotating

    trigger<<<1,numThreads>>>(edge,width,height,value,1);cudaDeviceSynchronize();
    trigger<<<1,numThreads>>>(edge,width,height,value,2);cudaDeviceSynchronize();
    trigger<<<1,numThreads>>>(edge,width,height,value,3);cudaDeviceSynchronize();
    trigger<<<1,numThreads>>>(edge,width,height,value,4);cudaDeviceSynchronize();
    for (int i =0; i<4; i++){
      if (value[i] == 1){
        gpioSetValue(pinList[i],on);
        usleep(50000);
        gpioSetValue(pinList[i],off);
        value[0] = 0;
        value[1] = 0;
        value[2] = 0;
        value[3] = 0;
        //usleep(1000);
        break;
      }
    }
    value[0] = 0;
    value[1] = 0;
    value[2] = 0;
    value[3] = 0;
    }
}
```

Otherwise the code is practically identical to the original gpu-based algorithm of triggering.

8.5 Releasing Memory - Resetting

The last step in both approaches is the releasing of memory using the cudaFree commands and setting the GPIO pins low and un-exporting them. This essentially frees up the resources that were being used during the script and then exits the loop, terminating the program.

```
// Free memory
cudaFree(matA);
cudaFree(grayData);
cudaFree(edge);
cudaFree(prevArr);
cudaFree(output);
// Turn off the Pins
gpioSetValue(pin1,low) ;
gpioSetValue(pin2,low) ;
gpioSetValue(pin3,low) ;
gpioSetValue(pin4,low) ;
gpioUnexport(pin1);     // unexport the pin
gpioUnexport(pin2);     // unexport the pin
gpioUnexport(pin3);     // unexport the pin
gpioUnexport(pin4);     // unexport the pin
```

8.6 Future Improvements

The future improvements on the project could use a more advanced edge finding technique such as canny edge detection, or performing more intensive operations such as image convolution. The edge detection currently is reliable and the next steps would be optimizing the parameters for not tapping the solenoids excessively per tile. The Nano or TX2 may be fast enough to do more intensive operations but in this exercise simple gradients were used to determine edge position. The usage of memory pre-fetching and streams could increase the efficiency of the code and would be interesting to implement. Another promising method would be to simply use a photo resistor that could be used in combination with an Arduino or Micro-controller. This would provide a faster reaction time, and more reliable solenoid actuation. Although not as interesting, this implementation could likely surpass the current record using computer vision due to the simplicity of the approach. Additionally, rebuilding the test stand to better house the solenoids and Nano would be beneficial. The unit, while functional, is not easily carried around.

9 References

Links:

https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/
https://developer.nvidia.com/embedded/buy/jetson-tx2-devkit
https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
https://www.johndcook.com/blog/2009/08/24/algorithms-convert-color-grayscale/
https://github.com/matthewdhanley/jetson-tx2

10 Appendix

10.1 Script Glossary

Location	Name of Script	Description
camera	cameraCommand.sh	Displays video from Raspberry Pi camera on Nano
GPIOTest	helloWorld.cu	Prints Hello World using GPU
GPIOTest	01-adding.cu	Initial try at matrix addition
GPIOTest	02-IO.cu	IO using bash commands within a script
Images/Adding	matrix_add.cu	Takes two arrays of size nx1 and adds the two together
Images/CV	command.txt	Commands used for compiling, includes command for TX2 camera
Images/CV	hello.cpp	Uses openCV to display ”hello” on image
Images/CV	manipulateImage.cpp	Displays pineapple.jpeg in blue and one channel with openCV
Images/CV	videoTest.cpp	Used on Jetson TX2 for the USB camer and with gstreamer
Images/CV	rgbTest.cpp	Displays gray-scale video from webcam with openCV functions
Images/grayScaleCalc	grayscale.cu	Performs grayscale calculations using GPU
Images/grayScaleCalc	video.cu	displays grayscale video from USB webcam
TX2GPIO	exampleGPIOApp.cu	Example GPIO app given by JetsonHacks
TX2GPIO	jetsonGPIO.h	header used for controlling GPIO
TX2GPIO	jetsonGPIO.cu	corresponding file for using GPIO with header file
Video	calibrate.cu	Takes cross-sections of current image and saves data to a.txt file
Video	cpu_edge.cu	Used for edge finding with a webcam on the CPU
Video	getData.cu	Used to do get ROI data and initial cropping
Video	gpu_edge.cu	Used for edge finding with GPU
Video	middle.cu	Used for overlapping of frames and annotating recorded test0.avi
Video	pixel.cu	Hit/Miss edge finding with openCV
Video	recordBGR.cu	Records Video from Raspberry Pi camera
Video	trigger.cu	Outputs velocity estimate based off overlay difference
Video	velocity.cu	Extension/Prototype of trigger.cu
Prototype	actuatorTest.cu	First attempt at tapping the screen based off video
Prototype	actuatorTest2.cu	Improvement off actuatorTest.cu - different edge detection
Prototype	lines.cu	Different try at edge detection and triggering using 4 lines
Prototype	lines2.cu	Variation of lines.cu
Prototype	fpsTest	Initially FPS testing, extension of gpuTrigger.cu
Prototype	**gpuTrigger.cu**	Main GPU Trigger script using shared variables between threads
Prototype	**inProgress.cu**	Version used to prototype new edge techniques
Prototype	**current.cu**	Current iteration of Piano Tiles Project
Prototype	**best.cu**	Best script with timing- achieved 9.220 in/s
Prototype	**gpuBest.cu**	Best script with CV - achieved 11.268 in/s

** Directories stem from home directory of Documents*