Computer Vision w/ Cuda Acceleration

By Andrew Lung

Project Idea/Overview:

Image classification project that utilizes a camera module to detect objects in real time. OpenCV naturally runs its program on CPU, but allows support for GPU acceleration with cuda. I wanted to accelerate real-time image detection with cuda. Specifically, I wanted to make an image classifier that detects crosswalk buttons in real-time.

How is the GPU used to accelerate the application?

When we perform image classification on an image, we may need to perform multiple steps; in the Haar Cascade algorithm I used that is supported on OpenCV 2, I needed to read in a camera feed to take a frame as my initial image, convert it to grayscale, read all pixels in that image to score it against the Haar Cascade classifier in order to see if an image is found, and then blit the resultant image to screen – an image with no objects detected will appear "normal" and an image with an object detected will have a blue bounding box around it.

This is computationally extremely expensive; we have to convert a lot of pixels, read those pixels, then display an image. As a result, there is a delay from what the camera can capture in real-time and what it can display as an output. The GPU accelerates the process of comparing the image to the Haar Cascade classifier using OpenCV functions, decreasing the delay between real-time and what is displayed as an output.

Implementation Details:

First, I needed to set up my hardware. I used a Jetson Nano 2GB and bought all the required parts to run it. I installed OpenCV 2 4.5.1 afterwards, then tested my camera module. I troubleshooted issues with this step for approximately a full week.

Next, I needed to verify that I could actually detect objects with my camera module. I downloaded a Haar Cascade XML file generator GUI on Windows, prepared approximately 50 positive and 100 negative samples, then generated my first classifier. I was able to run my first classifier after referring to online guides for introductory computer vision, but I was detecting everything as a crosswalk button – there were way too many false positives. It was to the point where nearly everything in my room was marked as a crosswalk button when I tested it; I needed to generate a stronger classifier. From when I finished setting up, this took about a week.

I learned how the Haar Cascade algorithm works to generate more quality samples, took hundreds of more samples, tested approximately 20 different classifiers I manually generated with different numbers of training, and kept discarding classifiers that were either underfit – detecting too many false positives for crosswalk buttons – and too overfit – detecting nothing as a crosswalk button. From the last step, this took about 2 weeks.

After I generated a solid classifier, making the program utilize cuda acceleration was simple; OpenCV has built-in cuda support for accelerating the use of image classifiers, so porting to GPU was actually accomplishable in a few lines of code. I took note of results before cuda acceleration and after cuda acceleration, and the difference was actually pretty minimal. This took about a day.

Documentation on how to run code:

In order to run my code, you will need OpenCV 4.5.1 and you will need to use Python. Afterwards, you will need to download my classifier. Place the .py file and the classifier folder in the same file. If you run the .py file and have the matching hardware, the code should run.

Since the camera module code specifically works for the SainSmart IMX219 camera module on a Jetson Nano 2GB, you will need that specific camera module attached to your Jetson Nano 2GB via ribbon cable as well. If you have a Raspberry Pi Camera Module instead, which is officially supported, you can simply just attach to device 0 after attaching it to your Nano via ribbon cable.

Evaluations/Results:

With my current hardware, the difference can be as large as a 2 times speedup – around an approximate .15 second time to classify an image on a GPU down from a .25-.3 second time to classify an image on a CPU. This is a significant improvement, and delay on CPU will likely only increase with the greater the resolution of our image since we have to perform operations on much more pixels.

Problems faced:

First, I needed to set up my hardware. I used a Jetson Nano 2GB and bought all the required parts to run it. I accidentally formatted my C: Drive while trying to format a microSD card that the Nano uses, so I effectively bricked my computer, taking me a day to reinstall my software. After I wrote the device image to the Nano, I had to reinstall OpenCV 2; while the Jetson Nano 2GB comes with OpenCV 2, it comes with version 2.x.x. For cuda acceleration, I needed to use version 4.x.x. This meant I had to upgrade my version of OpenCV and this process took another full day, not just because unpacking OpenCV takes a long time on my hardware, but because the Jetson Nano 2GB has issues with an insufficiently low swap space so I had to configure my hardware.

Next, I needed to actually have a dataset to work with. This process was extremely time consuming; I took hundreds of photographs of crosswalk buttons and non-crosswalk buttons, cropped and edited them to 25x25 images for positive samples and 50x50 images for negative samples.

I ran my first classifier back on the Nano after finding some resources to help me write code to utilize the built in OpenCV functions for computer vision. Documentation regarding OpenCV functions is quite scarce for Python, so I mainly referred to existing code examples to read images in from a camera, converting the image to grayscale, etc. If I were to do a similar project in the future, I would use C++ for OpenCV. While Python is a wrapper for C++ code for OpenCV meaning that I should've gotten the simplicity of writing Python code with the time efficiency of C++ code, there is hardly any documentation for Python implementations for OpenCV. Furthermore, if I use native Python code – which I did when I imported the time library to time my implementation – I will get less value running C++ under the hood because I still have to run natively Python code. Nevertheless, after writing my first iteration of code that can detect objects using an image classifier, I ran my CPU-ran image detection program.

My first classifier was horribly inaccurate – it had approximately a 30% false negative rate and detected everything from my face as a crosswalk button to my bike as one too. I had to research online to learn how Haar Cascades work in order to generate more quality samples and to take more quality samples to generate stronger Haar features for the classifier to work with. I have generated at least 10 different classifiers using various sample sets with various training stages and requirements for Haar features detected; these training times ranged from 15 minute training times to 8 hours. I had to frequently toss classifiers after testing them because they were either too overfit or underfit to be useful as a predictive model. My final classifier had about ~100 positive samples and ~400 negative samples.

Detecting images is actually further complicated by the resolution at which the camera captures an image and how it can be resized; since Haar Cascade works based on detecting edges, a strange resolution can negatively impact the performance of the program because of how it may reshape the edge.

Table of tasks:

- Writing disc image to Jetson Nano 2GB - Andrew Lung
- Peripherals setup - Andrew Lung
- Installing OpenCV 2 4.5.1 - Andrew Lung
- Collecting samples - Andrew Lung
- Training models - Andrew Lung
- Writing code for image detection on CPU - Andrew Lung
- Writing code for image detection on GPU - Andrew Lung
- Verifying results - Andrew Lung
- Writing report - Andrew Lung