**ECE499**

**NVIDIA GPU CUDA PROGRAMMING**

**C1C David Song USAF Academy**

**Department of Electrical and Computer Engineering**

**5 May 2022**

**EXECUTIVE SUMMARY**

Primary objectives of this independent study include learning how to implement image processing functions efficiently on NVIDIA GPUs using Compute Unified Device Architecture (CUDA) programming and providing a needed capability for the Counter-UAS capstone to be used this year and in future years. The first half of the semester was spent learning background knowledge. This included the bare minimum for python3 and many libraries that I would have to use both in this class and my capstone class (ECE464). The CUDA programming course that I took was NVIDIA’s "Fundamentals of Accelerated Computing with CUDA Python” which focused on matrix arithmetic and mathematical calculations. In addition to this, I spent some lessons learning what the CUDA architecture is and the programming model it uses to utilize that architecture. In the second half of the semester, I was tasked with using what I learned in the first half of the semester and apply parallel processing to the Counter Unmanned Aircraft Systems (CUAS) Capstone project that I was working on in parallel. Learning CUDA using Numba and integrating that with the CUAS system that heavily used OpenCV and Tensorflow did not translate well, leading to some more time devoted to figuring out how to use GPUs and CUDA cores with OpenCV and Tensorflow. Numerous compatibilities issues and manual configuration and building using both OpenCV and Tensorflow was more of a challenge than converting the CUAS image processing code for GPU support.

Preliminary testing on the CUAS blob detection was completed to set a baseline of shows bottlenecks at the read in of the image, which took almost double the actual image processing times on average. Testing was completed on a set of 33 images from a CUAS optimal offset test, but to increase the computational intensity, the 33 images were cycled through the algorithms 10, 100, and 1000 times over. More time and expertise is still needed to build OpenCV and Tensorflow for GPU support, but there is a .sh script made for Linux that builds OpenCV for GPU support with dependencies. Resources for learning the CUDA architecture and OpenCV GPU functions are centralized on a repository with this report.

**KEY BACKGROUND KNOWLEDGE**

**SECTION A: GPU**

Originally designed for speeding up graphics and optimized for video games, graphic processing units (GPU) were designed to hold large amounts of textures and polygon data that caches could not handle. GPUs are comprised of many cores (doubling every year), and each core runs slower than the CPU clock. They are not like CPUs in the sense that there are no interrupts, virtual memory, or means of addressing devices such as keyboards and mouse. They are also inefficient in tasks other than Single Instruction Multiple Data (SIMD). In fact, CPUs can outperform a GPU in calculating a hash for a string, but when computing several 1000’s hashes the GPU wins out. This is why developers prefer to outsource data-intensive tasks to a GPU. A general CPU and GPU architectural comparison can be seen in Figure 1.

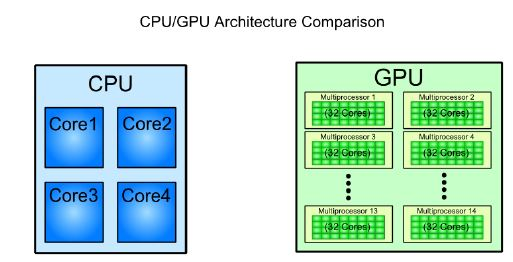
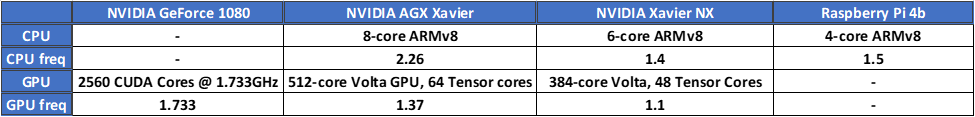


Figure 1: CPU/GPU Architecture Comparison [TutorialsPoint]

**SECTION B: CUDA**

NVIDIA and AMD are the leaders of the GPU industry, and the GPUs I had to work within this project were all NVIDIA products to use the CUDA (Compute Unified Device Architecture) computing programming model. The CUAS capstone team also had a Raspberry Pi 4b to use as well. All computers used in this project are listed in Table 1.

Table 1: Computers w/ CPU & GPU Specifications

CUDA is an extension of C/C++ programming, and this programming model allows the GPU to be used for tasks other than graphical calculations, such as processing matrices and other linear algebra operations that CPUs have trouble handling en mass. A CUDA-capable GPU is only connected by the Direct Memory Access from its host (CPU) as seen in Figure 2.

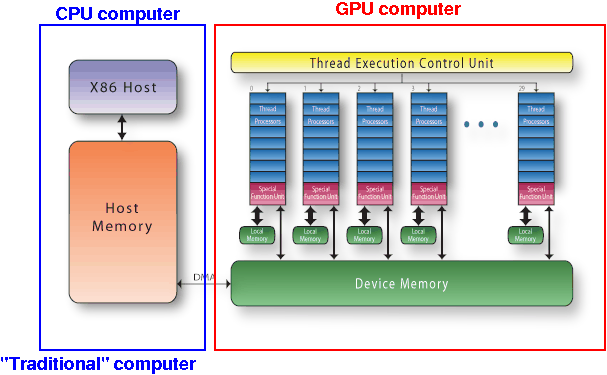


Figure 2: CUDA Architecture

It is important to note that the device (GPU) memory and the host memory (RAM) are not accessible to the other computer. The host controls the flow of data and all “normal instructions” that are executed in parallel. To access the GPU, the host computer must invoke the device by launching kernels, CUDA’s specific functions. This allows for heterogeneous programming, where the user has control of when to utilize the GPU and when to keep program execution in the CPU. Figure 3 shows how a C program would launch a kernel.

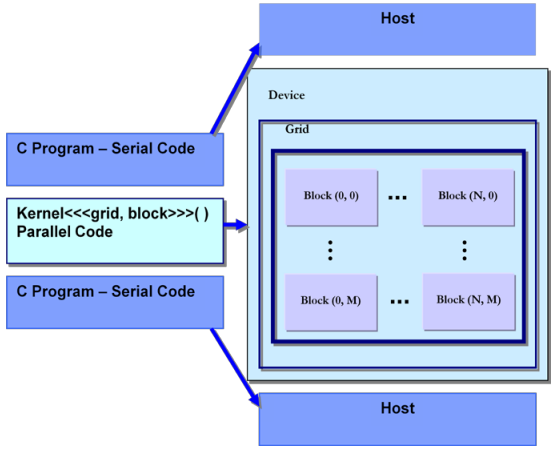


Figure 3: Heterogeneous programming

**METHODS**

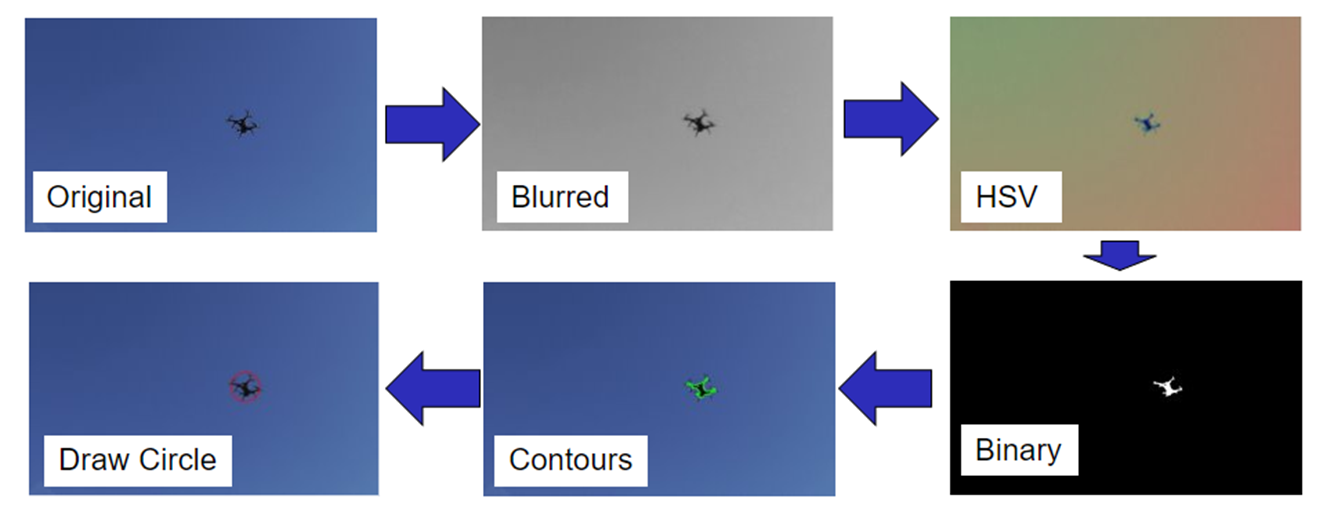
Setting a baseline for the image processing algorithms that the CUAS capstone is key for judging performance on CUDA cores/GPU systems. The approach that I took to test performance was to test the programs ability to process a set of images of varying sizes. The image set was set at 33 images from the CUAS optimal offset range test. To increase the volume of information, the same set of 33 images was cycled multiple times through the algorithms. The different data points were set at 10 cycles (330 “images”), 100 cycles (3300 “images”) and 1000 cycles (33000 “images”). The intent is for the tests to scale even more to 10,000 images and beyond, but time constraints prevented that advancement because those test points would take too long on CPU.

Figure 4: Blob Detection image processing stages

The blob detection algorithm was divided to 8 stages: read image, 6 distinct image processing stages, and the write back. Each stage was timed using the time library. The time just prior to the first line and last of code in each stage were taken. The difference in time is then the time elapsed for that stage. The total time and every cycle time was also recorded and stored. Average times for each stage/cycle were also calculated. Those times for each stage were stored in lists for every cycle, and at the end of the program, that data would be converted to an .xlsx spreadsheet for further analysis. This program is highly modular thanks to the use of a main function and a separate function for the image processing stages shown in Figure 4. The goal is that the program can be used to leverage the heterogeneity of CUDA by mixing and matching GPU and CPU functions.

**RESULTS AND ANALYSIS**

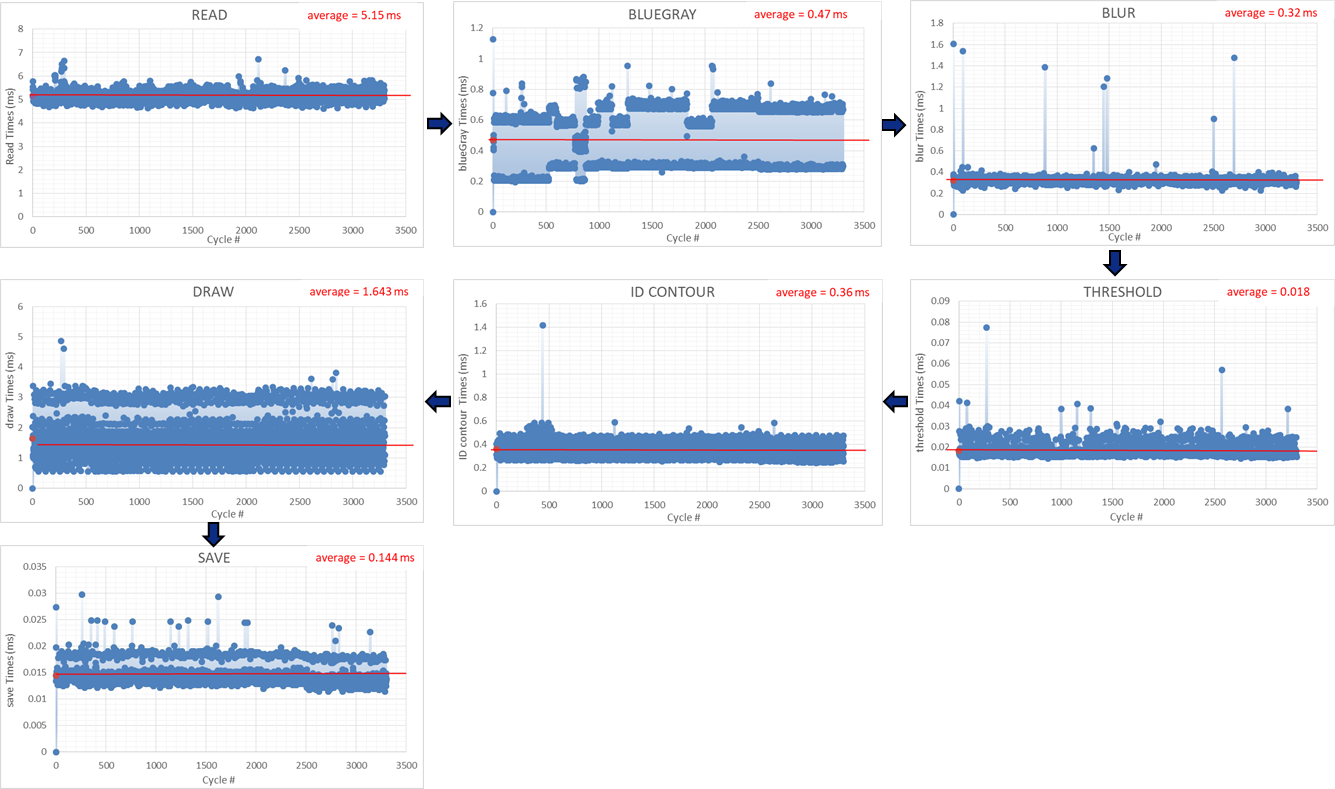


Figure 5: Average times for blob detection phases over 100 cycles of 33 images

Figure 5 shows a 100 cycle load test for the CPU version of BGR blob detection on an x86 system. We can see that the image processing is not cripplingly slow, but the longest operation seems to be the read image function at 5.15 milliseconds (ms). The total time for 1000 cycles of blob detection was 257 seconds (4:17 m:s). It is also interesting to see the occasional severe outliers in the image processing stages. The blue filter & grayscale filter stage seems to have 2 modes. I am not sure why the variation of blueGray is divided into two groups, but either way, the difference may not be significant enough to warrant further investigation.

On the GeForce 1080 integrated desktop, the MobileNet algorithm took 1379.6 sec (~23 min) to get through 1000 cycles. This equates to a frame rate of 23 fps in video specifications. This was possible to measure because Tensorflow automatically uses a GPU if available.

An issue is that not every OpenCV function has a GPU equivalent version. However, there are some portions of blob detection that do have a GPU function within OpenCV. cv.cudacodec.createVideoReader is a GPU accelerated video reader that could be useful for blob detection if a live video is necessary for continuous tracking. It’s partner function is cv.cudacodec.createVideoWriter, GPU accelerated video writer. In the image processing side, there is cv.cuda.createGaussianFilter, a GPU accelerated Gaussion blur, and there is cv.cuda.createCannyEdgeDetector, an edge detector that is crucial in identifying drone contours.

**CONCLUSION**

It is important to note that blob detection was only tested on CPU. Building OpenCV for GPU support on ARM architecture and x86 were both unsuccessful, restricting my comparison. However, OpenCV\_CUDAON.sh is a script that I have been working on to properly build OpenCV with GPU support on Linux and the .zip files in the proper site\_package folder. The suspected reason why this script did not work is that there were conflicting versions of OpenCV. However, based on the baseline testing of blob detection done on CPU, GPU functions may not help this basic algorithm that may not have the potential for improvement by using parallelization. The time for data transfer from host to device may outweigh the diminished return on boosted performance for such a simple program.

For those reasons, I have set some possible future work for future students interested in parallel programming, the CUAS capstone, and Neural Networks:

1. More complicated blob detection that better use GPUs
   1. KCF (Kernel Correlation Filter)
   2. SIFT/SURF
2. MobileNet implementation
   1. Solving aarch64 tensorflow installation difficulties for Jetson boards (wheel)
3. Training new neural network for Discovery Drone
   1. Lower res training set (drone < 800 px)
   2. GPU supported neural network training
4. Learning to properly build OpenCV for GPU support
   1. Continue testing performance on GPU vs. CPU for different algorithms
   2. Not learning CUDA in Numba unless creating custom image processing and large scale math operations.

**REFERENCES**

**Introductions to CUDA/GPU architecture and key concepts**

* Figure 1:<https://www.tutorialspoint.com/cuda/cuda_introduction_to_the_gpu.htm>
* Figure 2: <https://www.diva-portal.org/smash/get/diva2:447977/FULLTEXT01.pdf>
* Figure 3:  
  <http://www.mathcs.emory.edu/~cheung/Courses/355/Syllabus/94-CUDA/CUDA-arch.html>

**CUAS Capstone (Image Processing)**

* **Repository:** <https://github.com/lukeegorham/C_UAS>

**OpenCV documentation**

* [**https://docs.opencv.org/4.x/d6/d00/tutorial\_py\_root.html**](https://docs.opencv.org/4.x/d6/d00/tutorial_py_root.html)
* **Free Course:** [**https://opencv.org/opencv-free-course/**](https://opencv.org/opencv-free-course/)
* **CUDA Image Processing:**[**https://docs.opencv.org/4.x/d0/d05/group\_\_cudaimgproc.html**](https://docs.opencv.org/4.x/d0/d05/group__cudaimgproc.html)
* **Canny Edge Detector in OpenCV GPU**[**https://medium.com/@heeduugar/canny-edge-detector-in-python-and-opencv-cpu-vs-gpu-with-cuda-229e7bafc5e9**](https://medium.com/@heeduugar/canny-edge-detector-in-python-and-opencv-cpu-vs-gpu-with-cuda-229e7bafc5e9)
* **OpenCV GPU Module**[**https://learnopencv.com/getting-started-opencv-cuda-module/#:~:text=By%20default%2C%20each%20of%20the,distribute%20the%20work%20between%20GPUs**](https://learnopencv.com/getting-started-opencv-cuda-module/#:~:text=By%20default%2C%20each%20of%20the,distribute%20the%20work%20between%20GPUs)**.**

**Free Access to GPU (useful for when remote)**

* [**https://colab.research.google.com/**](https://colab.research.google.com/)