Motion tracking using CUDA and OpenCV

Victor Amaral, Timothy Ng

May 13, 2015

Abstract

We analyze computer vision algorithms for motion tracking and compare their performances using both a serial implementation and a parallel implementation. We utilize libraries from OpenCV to perform motion tracking and use the relatively new, built in module written using CUDA, that allows parallelization with an Nvidia graphics card. We noticed that the GPU accelerates computation substantially.

1 Introduction

There are many applications for motion tracking ranging from security video surveillance to movie animation. We have explored a couple different algorithms to perform these computations and compare their performances using a serial and parallel implementation. We utilize OpenCV and CUDA to accomplish this.

OpenCV is an open source computer vision and machine learning software library with many optimized functions that can be used for image processing, object detection, motion tracking, and much more. It has interfaces for C++, C, Python, Java, and MATLAB, and supports Windows, Linux, Android, and Mac OS.

Modern GPU accelerators have become powerful and featured enough to be capable of performing general purpose computations. GPU's have been explored extensively by scientists, researchers and engineers that develop computationally intensive applications. Despite difficulties reimplementing algorithms on a GPU, many people are doing it to check on how much speed-up they can achieve. To support such efforts, a lot of advanced languages and tools have been available such as CUDA, OpenCL, C++ AMP, debuggers, profilers and so on[2].

A significant part of computer vision is image processing, the area that graphics accelerators were originally designed for. Therefore, its challenging but very rewarding to implement computer vision algorithms on the GPU to take advantage of its performance benefits. OpenCV includes a GPU module implemented using CUDA that contains libraries to interact with the GPU. The use can then control the data flow between the CPU and the GPU. It is supported by Nvidia and was initially released in 2011. The libraries are growing and also adapting to new GPU architectures[2].

We implemented and analyzed motion tracking algorithms using OpenCV 2.4.10 and CUDA 5.5. The hardware we used was the Nvidia Quadro NVS 4200M graphics card which contains 1024Mb of memory and 48 CUDA cores [1]. We assumed that most of the computation time is spent in the library calls to compute features in the optical flow algorithm and to perform background subtraction in the image subtraction algorithm hoping that we can achieve substantial speed-up by running these functions on the GPU.

2 Algorithms and Parallelization

We implemented two algorithms for motion detection: one based on optical flow and another based on image subtraction. In general, when performing computations on the GPU, an image is divided up into grids and the computation is carried out among each grid in parallel if the computation is associative.

2.1 Optical Flow

The first approach was to search for features that we could track, calculate the optical flow between frames for these features, and then filter out all features that did not have optical flow. Optical flow is the apparent motion of objects in a scene, measured by analyzing the current and previous frames in a video. The goal is to compute an approximation to the 2-D motion field from spatio-temporal patterns of image intensities [3]. We performed feature extraction using Shi and Tomasi's minimum eigenvalue method for detecting corners of objects. Since we are detecting corners, many stationary objects in the picture will also be detected. In order to help filter out the stationary points, we use the Lucas-Kanade method to calculate the optical flow between frames. We then draw the optical flow points calculated for the moving corners for each frame. This process is repeated for each pair of frames in the video file.

We first transferred our images in the forms of matrices from CPU memory to GPU memory. We then used a GPU feature detector to search for corners. In order to calculate corners, the program must inspect different windows of pixels of the image. Each window is compared to a close, neighboring window, which gives us an approximation, or a Harris matrix. We then determine the eigenvalues of this matrix and look at the minimum eigenvalue. If the minimum eigenvalue is above a certain threshold, then we can assume that the window we are inspecting holds a corner. Finally, we downloaded the coordinates for the corners found from GPU memory into a vector in CPU memory. We then drew circles for the features on the host.

2.2 Image Subtraction

Our second method was to use a simpler approach by subtracting subsequent image frames to look for moving objects. Images of a scene without the intruding objects exhibit some regular behavior that can be well described by a statistical model. If we have a statistical model of the scene, an intruding object can be detected by spotting the parts of the image that dont fit the model [4]. This process is usually known as background subtraction. This allows us to look for pixels that have moved, resulting in a motion detector. The background subtractor returns a mask of groups of pixels that have changed between frames for which we draw a contour around and display it on the video.

We transferred each image as a matrix from the CPU to the GPU. We then used a GPU background subtractor to create a binary image of any moving pixels by subtracting the intensity of each pixel in RGB space. The subtraction of intensity values can be parallelized because we have three different channels: red, green, and blue values to be analyzed. The computations for each channel run in parallel and a score is calculated that represents how different the image is from the previous. The scores are combined for each channel and a binary mask is drawn based on this value. The mask is then transferred from GPU memory to CPU memory where contours are determined for the mask. The contours are drawn on the image and displayed.

3 Results

We ran our algorithms both serially and in parallel and compared their results using a data set of 4 different traffic videos. Videos 1-3 are traffic cameras of highways and video 4 is a recording of a busy intersection with pedestrians as well. They are arranged based on complexity of their moving objects. We used the valgrind tool to profile our code and the kcachegrind tool to give the graphical analysis as displayed in figures 1 and 2.

Trial #	Serial Time	Parallel Time	Start-up costs
1	5.33	5.06	0.79
2	6.23	5.11	0.53
3	8.99	4.5	0.62
4	35.92	17.5	0.98

Table 1: Performance comparisons for four trials using the CPU vs the GPU for the optical flow algorithm. The time is displayed in milliseconds and each trial is performed on a separate video. This shows the average time taken per frame to perform the optical flow computation.

Trial #	Serial Time	Parallel Time	Start-up costs
1	7.49	3.51	1.59
2	8.28	4.35	1.21
3	11.78	7.78	1.45
4	46.85	71.73	3.54

Table 2: Performance comparisons for four trials using the CPU vs the GPU for the image subtraction algorithm. The time is displayed in milliseconds and each trial is performed on a separate video. This shows the average time taken per frame to perform the image subtraction computation.

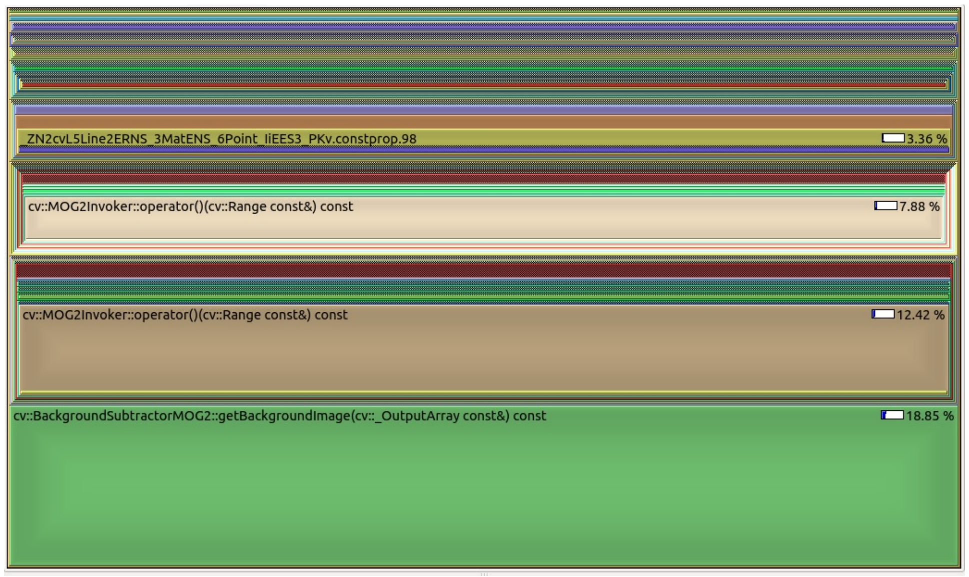

Figure 1: Diagram of the profiler output for the image subtraction serial algorithm.

4 Discussion

According to tables 1 and 2, we achieve nearly double speed-up by parallelizing our code and running it on the GPU. Keeping in mind that we used a low-end, laptop GPU, there is potential to do much better as we scale-up the hardware. We notice in table 1, however, that for the most simple video in trial 1, it may not be worth the start-up costs of memory transfers to perform the computation on the GPU since the total time is greater than that for the serial program. An anomally from the data trend is evident in trial 4 in table 2. Strangely enough, the parallel computation of image subtraction is about twice as slow as the serial implementation. This is probably due to the fact that many consecutive frames in the video are similar so characteristics of moving parts aren't as defined and many more contours are computed then there should be. After looking at the profiler, we can see that some of our assumptions about the operations in the processes that were parallelized were correct. When profiling our code for corner detection, we can see in figure 2 that the function cv::goodFeaturesToTrack() requires the most resources. We can also see functions that are related to matrix evaluation, eigenvalue

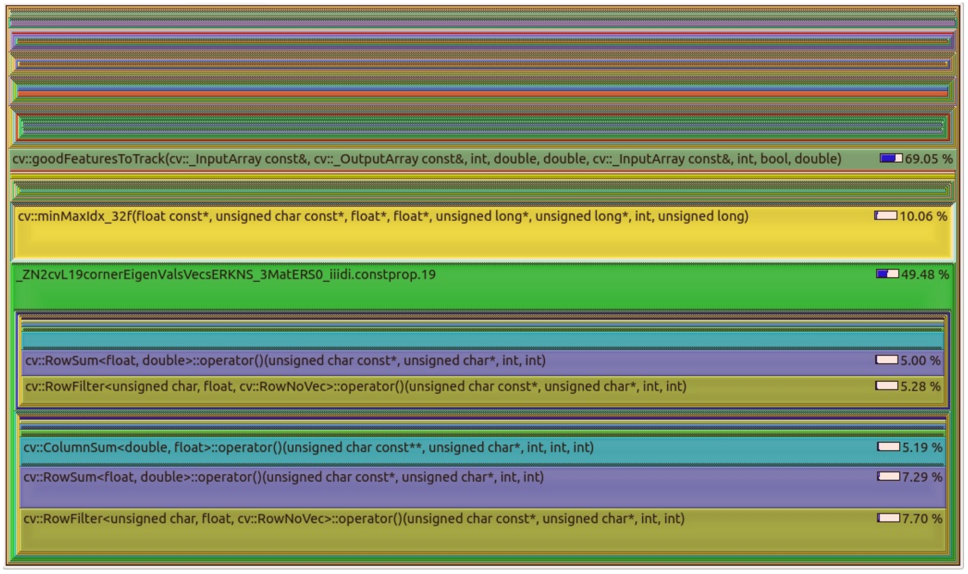

Figure 2: Diagram of the profiler output for the optical flow serial algorithm.

evaluation, and minimum and maximum comparisons taking up a large part of computation time. After profiling our code for background subtraction, we can see in figure 1 that the background subtraction process required the most resources since extracting a background image and comparing the foreground were the most prominent calls. All of the method calls with "MOG" have to do with the main calculations in the image subtraction algorithm.

5 Conclusions

We found that the speed-up is significant if we use CUDA to implement the computer vision motion tracking algorithms we used. It would be nice, to experiment on a more powerful graphics processor to measure scale-up as well. The work was divided relatively equally among the team members. Victor focused more on installing and configuring OpenCV with CUDA, implementing the serial versions of the algorithms, analyzing data, and explaining background information. Tim focused more on parallelizing the serial versions, and writing about how the algorithms work. This was overall a rewarding project as we got to see how much better GPU computation can perform as

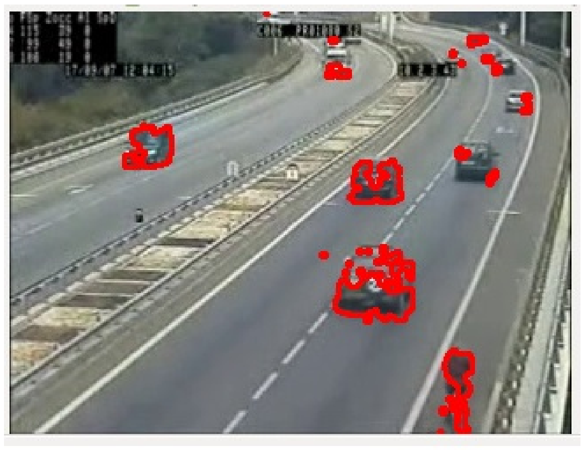

Figure 3: Image of a video frame as the image subtraction algorithm is running

opposed to the CPU.

References

[1] Nvidia Corporation. Tech specs, 2015. http://www.nvidia.com/object/nvstechspecs.html.

[2] itseez.com. Cuda|opencv, 2015. http://opencv.org/platforms/cuda.html.

[3] Alessandro Verri and Tomaso Poggio. Motion field and optical flow: Qualitative properties. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 11(5):490–498, 1989.

[4] Zoran Zivkovic. Improved adaptive gaussian mixture model for background subtraction. In *Pattern Recognition, 2004. ICPR 2004. Proceedings of the 17th International Conference on*, volume 2, pages 28–31. IEEE, 2004.

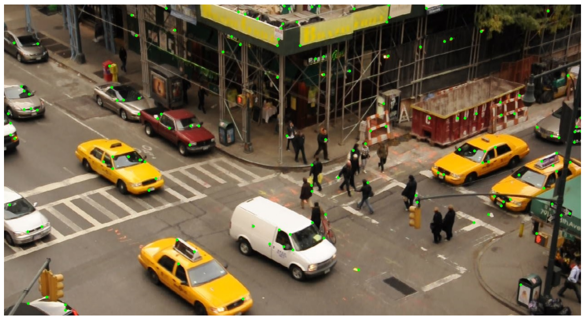

Figure 4: Image of a video frame as the optical flow algorithm is running