Fast Parallel Object Tracking

Timothy Li

May 12, 2017

1 Summary

We will implement a machine learning algorithm called AdaBoost to track objects in video utilizing the parallel capabilities of graphics cards. In particular, we will use CUDA on the GHC machines to parallelize a boosting algorithm by Helmut Grabner, Michael Grabner, and Horst Bischof.

2 Background

The task of object tracking is the following: given a video and an identified object (usually given by a bounding box in the first frame), track the position of the object over the frames of the video. Difficulties in this process include occlusion or changes in orientation. The algorithm by Grabner, Grabner, and Bischof [3] is robust against these changes by using a large number of “weak classifiers” and “selectors” and is illustrated in Figure 1. The main building block of this system is the “weak classifier.” These are relatively simple objects that classify a group of pixels as the target object positively or negatively. The most important part is that they must do better than 50-50 guessing (a relatively loose condition). This heavy relaxation allows these types of classifiers to be computed very quickly.

Selectors have the job of boosting the accuracy far beyond 50-50 guessing. By choosing the most accurate weak classifiers, selectors are much more likely to accurately pinpoint the location of the object. Further, combining the outputs of the selectors creates a confidence map that accurately and precisely locates the object. The input to the algorithm is a new frame. This is then preprocessed and then passed to the weak classifiers to create individual confidence maps. These confidence maps are then chosen by the selectors based on error rate and linearly combined to create a confidence map for the strong classifier. This can then be translated to a coordinate which is the output of the algorithm. We then treat the coordinate to be the truth and update all weak classifiers to better recognize the object in the future.

I chose to use a weak classifier based on Haarlike features. In essence, these take the weighted sum of rectangles of pixels and use a threshold to determine classification. The upside of this type of classifier is that it can be computed
Figure 1: The AdaBoost Algorithm

very quickly using an integral image which is a table of summed areas. Both of these concepts can be seen in Figures 2 and 3.

There is a high level of parallelism throughout. For instance, the calculation of the integral image can be done in parallel. Next, the application of the weak classifiers can be done in parallel. Further, the creation of a confidence map of a particular weak classifier can be done in parallel across the pixels. Next, the combination of confidence maps can be done in parallel across the pixels. The computational dependencies will be shown in the next section.

Figure 2: Haarlike Features

Figure 3: Integral Images
3 Approach

The language used was CUDA targeting the NVIDIA GeForce GTX 1080 GPUs on the Gates machines. The overall approach was to parallelize at every possible step choosing good efficient algorithms to improve throughput (frames per second). The calculation of the integral image was done across blocks and threads using code from Bilgic, Horn, and Masaki [1]. The creation of weak classifier confidence maps was done in parallel with classifiers mapping to blocks and pixels mapping to threads. The combination of confidence maps across selectors was done in parallel mapping pixels to threads. This process can be seen in the pipeline shown in Figure 4. The updating of the weak classifiers was also done in parallel mapping classifiers to threads. Importantly, we have changed the algorithm as presented by Grabner, Grabner, and Bischof by dropping the “importance” variable $\lambda$. This is passed from selector to selector creating a sequential dependency. This is avoided in our variation by simply using the value $\lambda = 1$ across all selectors/classifiers.

A very important data structure that we used was the key-value pairs of classifier index to error rate. By sorting this structure by value (error rate), we were able to very quickly identify the best performing and worst performing classifiers at any given step. In other words, this sorted structure would give the most accurate $n$ classifiers at the beginning and the least accurate $m$ classifiers at the end. This essentially took the place of the selector by allowing parallel access the best classifiers. Further, we discarded the worst performing classifiers every $k$ steps. This could be done in parallel easily using the same structure.

4 Results

Testing was done with the following setup. We set the parameters like image dimension and number of classifiers. Then, an image is artificially created by drawing a design at a particular ($x, y$) coordinate. The object tracker then trains on this design. During testing, this artificial design is moved around the
image, and the object tracker tracks its location. The time required to track is then recorded over many iterations. This total time gives a frames per second measurement. We will present two types of results: performance vs. a sequential implementation and scaling ability.

4.1 Performance vs. Sequential

We use the following parameters in basic. Height:512, Width:1024, Bounding Height:15, Bounding Width:15, Search height:16, Search Width:16, Classifiers:250, Discard steps:10, Discard count:50, Selectors:50.

<table>
<thead>
<tr>
<th></th>
<th>FPS</th>
<th>Sequential FPS</th>
<th>Speedup over Sequential</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic</td>
<td>1281</td>
<td>141</td>
<td>9.085106383</td>
</tr>
<tr>
<td>Double Image Height/Width</td>
<td>522</td>
<td>58</td>
<td>9</td>
</tr>
<tr>
<td>Double Search Height/Width</td>
<td>1216</td>
<td>51</td>
<td>23.84313725</td>
</tr>
<tr>
<td>Double Classifiers</td>
<td>1247</td>
<td>91</td>
<td>13.7032967</td>
</tr>
<tr>
<td>Double Selectors</td>
<td>1264</td>
<td>137</td>
<td>9.226277372</td>
</tr>
</tbody>
</table>

We see a good throughput in all cases hovering around 1200 FPS in the 512 x 1024 image dimension cases. We see that this is good speedup over the sequential implementation. We see huge speedup in the double search height/width case. This is because this is perfectly parallelizable using CUDA, so there is no performance impact on the parallel version. On the other hand, the sequential implementation must spend 4× the time classifying each pixel for each weak classifier. Similarly, there is good speedup in the double classifier case because of perfect parallelizability in CUDA. The sequential implementation must spend 2× the time in the confidence map section. The fact that these speedup exists implies that we are not using the full compute ability of the GTX 1080s. On the other hand, given the pipeline above we are using all available parallelism.

4.2 Scaling

We use the following parameters. Height:1024, Width:2048, Bounding Height:15, Bounding Width:15, Search height:16, Search Width:16, Classifiers:250, Discard steps:10, Discard count:50, Selectors:50. Each type of scaling adjusts only the parameter tested. We see that there is significant slowdown as result of increasing the dimensions of the image. This can be explained by the bandwidth fraction graph. As the dimension grows larger, the fraction of time that copying the image to GPU memory increases. This means the bandwidth-bound section is very limiting for performance.

There is little slowdown when scaling in classifiers. This is because most of the classifier work can be done in parallel, and we are not reaching the maximum amount of compute. There may be an effect form the sorting of key-value pairs. There is a little slowdown when scaling in selectors. This is because the number of selectors is most applicable when combining confidence maps. This accounts
for relatively little of the runtime, so increasing selectors does not cause significant slowdown.

I believe most of the search box scaling graph is due to noise in measurement. Speedup was likely limited by the data transfer from host memory to device memory. This is supported by the bandwidth fraction graph. Further, every frame we are calculating the integral image across the entire image. This is not necessary, but makes the code cleaner. This bit of code is likely holding back speedup as well. The choice of machine target was sound: the GPU provided the compute capability needed to parallelize over many classifiers and over pixels. This was instrumental in achieving performant code.

5 Github Repo

https://github.com/Teemothy/fast-object-tracker-15418

References


Figure 6: Bandwidth fraction

Figure 7: Bandwidth fraction

Figure 8: Selector scaling
Figure 9: Search box scaling