Using AdaBoost for Real-Time Object Detection on Programmable Graphics Hardware

Farooq Mela – CS229 Fall 2010

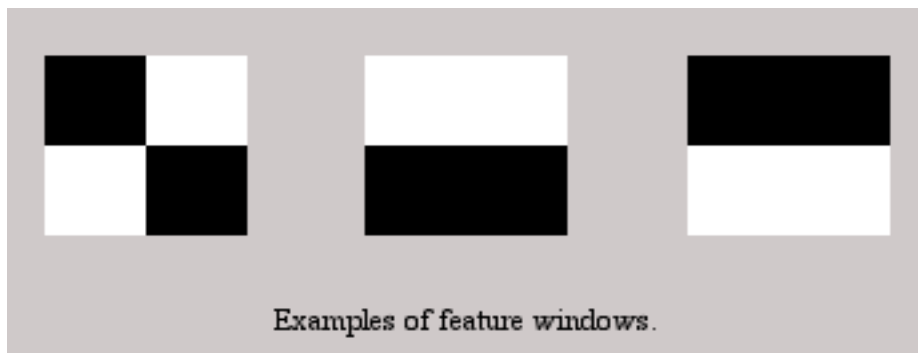
Introduction

An object detection algorithm is able to identify an object of a given type in a digital image. The use of machine learning methods for object detection has been well studied. One of these methods is based on a sliding-window approach, where the classifier is a detection window of fixed size and evaluates some criteria over all pixel positions and various resolutions of the input image. We apply machine learning to simple sliding-window classifiers to build a robust detector that can be trained to recognize various objects.

The rapid increase in both the performance and programmability of recent graphics hardware has made the discrete graphics processing unit (GPU) a compelling platform for computationally demanding tasks. Furthermore, this hardware is especially suited to both image processing and data-parallel computation. Our algorithm is inherently data-parallel, coherent-access, memory-intensive and image-oriented, so it is well suited to GPU implementation. In addition, media is ever increasing in both resolution and quantity. Methods that run in real-time for low-resolution images or video streams are inadequate for high definition photos and high definition, high frame-rate video streams. Thus, our present work is motivated by our desire to adapt a robust object detection technique to the processing demands of modern media.

Object Detection

The object detection method we employed was based on the work of Viola and Jones. Each sliding window is constructed of positive and negative regions. The value of a window at a particular position on an image is the sum of the pixels in the positive regions minus the sum of the pixels in the negative regions.



Examples of feature windows.

These windows are evaluated over all possible positions over the input image and scaled down versions of the input image. Each scaled down version of the image is half the size of the previous in each dimension, until a minimum image dimension is reached (in our case 24 pixels).

Viola and Jones propose accelerating the computation of sums of pixels in the evaluation of a window by a preprocessing step wherein we construct the integral image (also known as a summed-area table) of the input image. If we have an image with width W and height H, then let $im(x,y)$ be the value of the pixel at position (x,y) for $0 \leq x < W$ and $0 \leq y < H$. We can then define the integral image $ii(x,y)$ as

$$ii(x,y) = \sum_{x=0}^{i} \sum_{y=0}^{j} im(x,y)$$

The integral image can be computed dynamic programming-style by a simple recurrence relation:

$$ii(x,y) = im(x,y) + ii(x-1,y) + ii(x,y-1) - ii(x-1,y-1)$$

where $ii(x,y) = 0$ if either x or y is negative. Given the integral image, the sum of any rectangle of pixels in the original image can be computed with just four array references in the integral image at the corners of the rectangle:

$$\sum_{x=left}^{right} \sum_{y=bottom}^{top} im(x,y) = ii(right,top) - ii(left-1,top) - ii(right,bottom-1) + ii(left-1,bottom-1)$$

We can construct a weak classifier by choosing sliding windows at random, and testing them against our training data. We stop when we find one with acceptable false positive (when faces are falsely detected) and false negative (images in which faces are missed by the classifier) rates.

Learning with AdaBoost

We can combine a number of weak classifiers into a much stronger one using the AdaBoost method, which stands for adaptive boosting. AdaBoost constructs a series of weak classifiers h_i and a set of weights α_i for those classifiers and outputs a strong classifier. Given training data and labels $\{(x_1,y_1),...,(x_m,y_m)\}$, where $y_i \in \{-1,+1\}$, we define the weighted training error $E(D,h)$ of a classifier h according to the distribution D as:

$$E(D,h) = \sum_{i=1}^{m} D_i 1\{y_i \neq h(x_i)\}, \text{ where } \sum_{i=1}^{M} D_i = 1$$

AdaBoost begins by setting each $D_i = \frac{1}{M}$. Then, for $t = 1 \ldots T$, it chooses the classifier h_t that minimizes the error with respect to the distribution D: $\varepsilon = E(D, h_t)$. (Here, the algorithm terminates if it cannot find a classifier with $\varepsilon < 0.5$). It then sets the weight for the classifier $\alpha_t = \frac{1}{2}\ln(\frac{1-\varepsilon}{\varepsilon})$, which assigns a larger weight to classifiers with smaller error and smaller weight to classifiers with larger error. Finally, the distribution *D* is updated:

$D_i = D_i \exp(-\alpha_t y_i h_t(x_i))$ for $i = 1 \ldots m$

Then the distribution is renormalized to have unit sum. This update to *D* increases (boosts) the weights for misclassified training examples and decreases the weights for correctly classified training examples.

The output of AdaBoost is *T* classifiers and associated weights for each classifier. The composite "strong" classifier is then:

$$H(x) = \mathrm{sgn}(\sum_{i=1}^{T} \alpha_i h_i(x))$$

The advantage of using AdaBoost is that, even if we only have relatively weak classifiers, we can combine them to build a very accurate classifier. We would expect that, as we increase *T* (the number of weak classifiers), that *H* would suffer from over-fitting, so that even though we might be decreasing our training error as we increase *T*, our test error will increase. However, the surprising thing about AdaBoost is that, as we increase *T*, and even after our training error has gone to zero, the test error *continues* to decrease! There is still debate on why this is the case; some researchers have postulated that as *T* increases, AdaBoost is maximizing the classification margins.

Of course, there are caveats: the performance of AdaBoost depends on good training data and our ability to choose enough weak classifiers. AdaBoost can fail if the training data is noisy, if the weak classifiers are *too* weak, or if we cannot create enough weak classifiers.

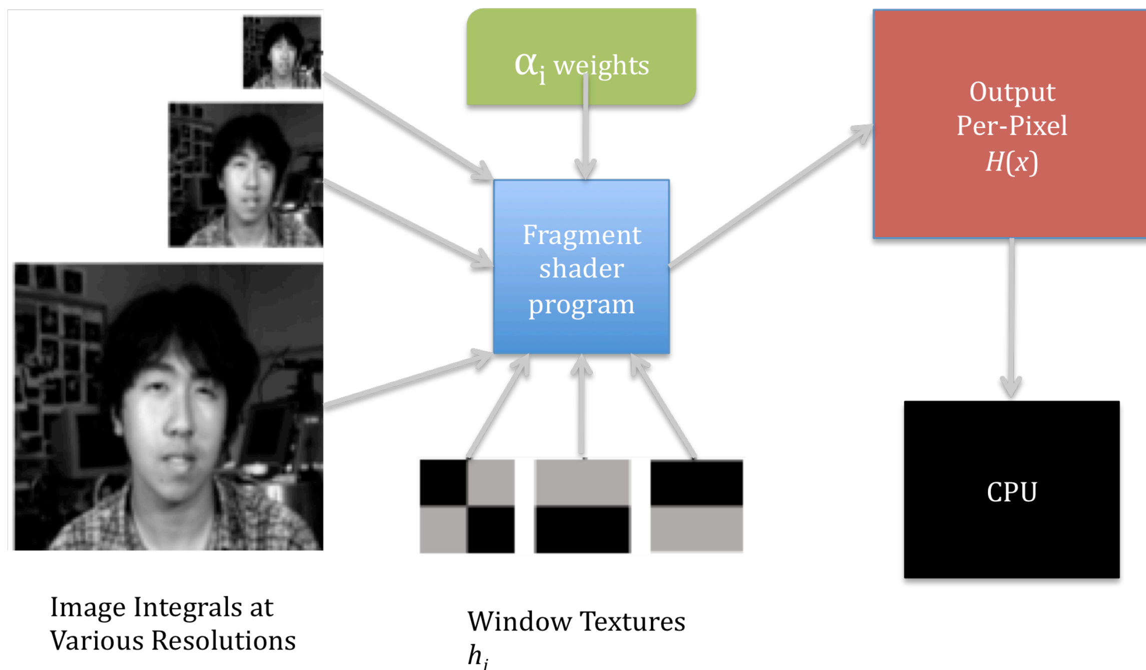
GPU Implementation

Now that we have described our algorithm, we will describe our GPU-accelerated implementation. We used a Linux machine with a quad-core Q6600 processor and an NVIDIA 8800GT GPU. After implementing the algorithm on the CPU, we used OpenGL 3.0 and programmable fragment programs to

implement the sliding window calculations and image down-sampling. The CPU must compute the integral image because of the data interdependence of the neighboring pixels in the integral image.

We represent our various sliding windows as textures, where the texel value 0 indicates a negative pixel, and a texel value of 255 indicates a positive pixel. We store the sliding-window primitives as textures on the onboard GPU memory, which is generally faster than main memory, has a very wide bus interface, and is well-suited to data-parallel access. We will use these sliding-window primitives in our fragment program, which is executed for each pixel and each resolution of the output texture. If our input image is 128x128, we run our sliding-window operation at the 128x128, 64x64, and 32x32 resolutions in a single pass. GPU hardware can compute all downsampled versions of our image very quickly; furthermore it represents each image in an optimized format known as a mipmap, giving us better texture read performance and lower memory usage. Each integral image is loaded into GPU memory as a texture using *glTexImage* and then downsampled using *glGenerateMipMaps*.

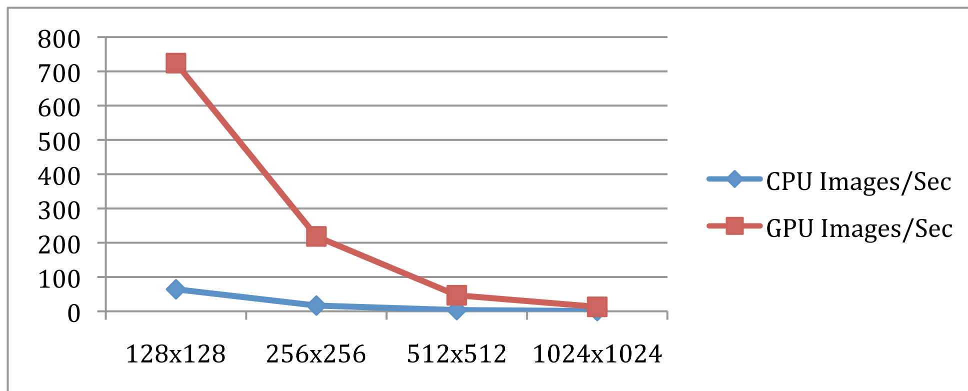
The fragment shader is at the heart of the speedup. Once the mipmapped integral images are loaded into GPU memory, the fragment shader is invoked to render an output texture that is the size of the original image. The fragment shader program executes independently and in parallel at each pixel location, where it computes $H(x)$ at that pixel, to be stored in the output texture.



For each image resolution, the fragment shader computes the α-weighted sum of the sliding window evaluated on the integral image at the pixel location. If any of the sums are positive, the fragment shader outputs one, otherwise it outputs zero. Thus, we can compute $H(x)$ for an image x at every pixel position for every resolution *in a single pass*. The CPU can then read back the output texture, and wherever there is a one, the object detection algorithm has predicted an object (in our case, a face).

Experimental Results

We trained our object detector using the UCI Machine Learning face image database (the source of the image in the diagram above) for positive training examples and non-face images downloaded from Google Images as negative training examples. Each image was converted to 8-bit grayscale. By randomly choosing 70% of the data for training and 30% for testing, and only choosing weak classifiers with both less than 15% false positives and less than 15% false negatives, we were able to achieve a test error rate of less than 5% with T=30. However, our CPU implementation ran at less than real-time rates; just to bring the number of images classified per second to more than 10, we rescaled all of our input images to 256x256 before computing the integral images or performing any other processing. After implementing the same algorithm on the GPU, we achieved approximately an 11x speed-up at the same resolution. These results have been achieved on ca. 2007 consumer-level hardware. We have not yet gone to any lengths to optimize our GPU implementation, so it is a certainty that much higher performance can be achieved.



In conclusion, we find that GPUs provide ample opportunity for the acceleration of image processing algorithms, and that our implementation of real-time object detection would be suitable for the kinds of high-resolution images or high-framerate video streams that have become commonplace.