QuickHOG: Real time pedestrian detection

Jeet Kanjani

August 2017

1 Abstract

We introduce a parallel implementation of the histogram of oriented gradients algorithm for object detection. Our implementation uses the GPU and the NVIDIA CUDA framework. Our implementation is END to END which makes us able to get a running time improvement of 86X over the standard sequential implementation[1] and 1.2X over fastHOG[2]. The purpose is to implement it on OxSight SmartSpecs used by the visually impaired.

2 Introduction

There have been many papers on the HOG approach for linear classification before the introduction of deep neural networks. The significant papers which proposed a good running time improvements are [2] and [3]. While [3] uses the cheaper and more easily implemented bilinear interpolation, we mirror [1] by using trilinear interpolation for binning. While[2] uses trilinear interpolation, the algorithm flow makes an expensive step of copying data back to the CPU in between which is overcomed in our implementation.

3 HOG Descriptor

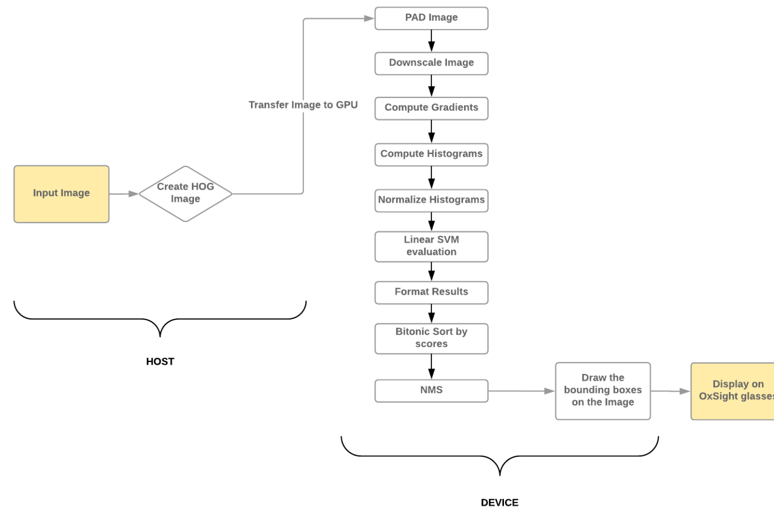
The histogram of oriented algorithm for object detection was introduced in [1]. The HOG detector is a sliding window algorithm. This means that for any given image a window is moved across at all locations and scales and a descriptor is computed. For that window a pretrained classifier is used to assign a matching score to the descriptor. The classifier used is a linear SVM classifier and the descriptor is based on histograms of gradient orientations. Firstly the image is padded and a gamma normalization is applied. Padding means that extra rows and columns of pixels are added to the image. Each new pixel gets the color of its closest pixel from the original image. This helps the algorithm deal with the case when a person is not fully contained inside the image. The gamma normalization has been proven to improve performance for pedestrian detection [1] but it may decrease performance for other object classes. To compute the gamma

correction the color for each channel is replaced by its square root. Gradient orientations and magnitude are obtained for each pixel from the preprocessed image. If a color image is used the gradient with the maximum magnitude (and its corresponding orientation) is chosen. This is done by convoluting the image with the 1-D centered kernel [-1 0 1] by rows and columns. Several other techniques have been tried (including 3x3 Sobel mask or 2x2 diagonal masks) but the simple 1-D centered kernel gave the best performance (for pedestrian detection). Each detection window is divided into several groups of pixels, called cells. For each cell a histogram of gradient orientations is computed. For pedestrian detection the cells are rectan- gular and the histogram bins are evenly spaced between 0 and 180 degrees. The contribution of each pixel to the cell histogram is weighted by a Gaussian centered in the middle of the block and then interpolated trilinearly in orientation and position. This has the effect of reducing aliasing. For more details on the interpolating see [1]. Multiple cells are grouped into blocks, with each cell being part of multiple blocks. Each block will therefore contain multiple cell histograms. Each block histogram is normalized. For pedestrian detection the L2-Hys norm [3] is used (the L2-norm followed by clipping to 0.2 and renormalizing. All the blocks inside a detection window form a HOG descriptor. This is used as input for a pretrained linear SVM classifier. For people detection the best results have been achieved using 16x16 pixel blocks containing 2x2 cells of 8x8 pixels. The block stride is 8 pixels (one cell) so the histogram of a cell is normalized over 4 blocks. Each histogram has 9 bins. The Gaussian used for weighting pixel values has = 8. The detection window is 64x128. The scale ratio is 1.05. Finally an algorithm based on mean shift is used to fuse the detections over the 3D position and scale space.

4 GPGPU and the NVIDIA CUDA

We use 3,749,760 threads to compute the linear SVM evaluation for a single 1920x1080 image. The bottleneck in a CUDA application is often global memory access, rather than mathematical computations, which are essentially free. The speed of the memory access is also affected by the thread memory access pattern. If memory accesses are coalesced, that is if threads in the same multiprocessor access consecutive memory locations, fewer memory operation will be required, so speed will be considerably high. The algorithm being memory bandwidth limited, the link between GPU and CPU(PCIe or NVLink) plays a crucial role in determining the running time of the algorithm. The running times were tested on NVIDIA Geforce GTX 950M with 640 CUDA cores and PCI Express for CPU connectivity with a memory bandwidth of the order of 3 GB/s for our system. This bandwidth is less for the SmartSpecs and hence the improvements over [2] would be significantly High.

5 Algorithm



Our algorithm can be split into 2 parts: host and GPU processing (as shown in Figure). The image is acquired by the host, copied into GPU memory and then processed by the GPU. After all scales and windows are processed, the SVM scores are formatted to map them to their respective detection window. The detections are then sorted based on their scores which are passed through the NMS to obtain the final fused results.The final result will then be displayed on the OxSight Glasses in some format (like silhouette extraction).

5.1 Host-GPU transfers and Padding

We chose to pad the image with 0. While in some cases our padding might miss some detections, it is considerably faster. In our implementation the padding is done by the memory copy operation itself: the original image is copied from the host memory inside a new 0-padded image stored in GPU memory. This makes our padding virtually free.

5.2 Downscaling and Gamma Normalization

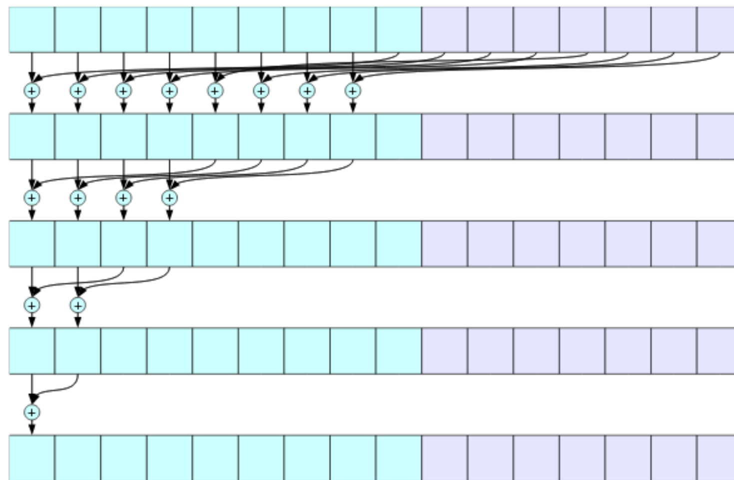
Our implementation supports two modes of operation: color and grayscale. If color is used the downscaled image is a gamma corrected RGBA image. In the grayscale case, the downscaled image is a gamma corrected grayscale image.

The source image is always a RGBA image. The gamma correction can be easily disabled. Each thread computes one thread per pixel and the thread block size is 16x16. We use trilinear interpolation so, in our cell/block configuration, each pixel will contribute to up to 4 histograms (one for each cell), and up to 2 bins per histogram.

5.3 Color Gradient

If color is used each thread computes three color gradients, three magnitudes and three orientations (one for each channel). The alpha channel is ignored. In the grayscale case only one gradient, magnitude and orientation need to be computed. Finally, for each pixel in the padded, downscaled image two values are written to global memory: gradient orientation and magnitude. In the color case the maximum magnitude (and corresponding orientation) is Chosen.

5.4 Block Histogram



In our implementation each thread computes its own histogram and stores it in shared memory. Because of the trilinear interpolation the weight of a pixel is distributed into as many as 8 bins, from 4 different histograms. Each pixel will therefore contribute not only to its cell histogram but to one or more of the neighboring ones, and detailed in [1]. Because of this memory write pattern each thread must hold the complete block histogram (36 float values) in shared memory. This imposes certain restrictions on the block size and layout. Furthermore this means that it is not advantageous to keep the histogram weights in texture memory, as the memory read operations will be more costly than the actual computation of the weights. Finally these histograms are merged using

a parallel reduction algorithm (figure). The parallel reduction is similar to the one from the reduction.

5.5 Histogram Normalization

We use the parallel reduction (Figure above) once to compute the norm for each block, normalize each pixel value once, cap the value to 0.2, use the parallel reduction algorithm again and finally renormalize.

5.6 Linear SVM Evaluation

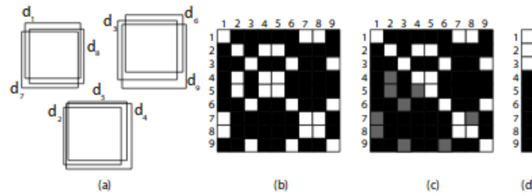
In this step each detection window is mapped to a CUDA thread block. In our case a detection window has 64x128 pixels, so it will be made of up 7x15 blocks. If we consider the grid layout used at the two previous steps, each detection window will be stored in global memory in a grid of 7x18 width and 15x2 height. We therefore chose to have, for each block/detection window, 7x18 threads, each processing 15x2 values. The thread configuration is depicted in the figure. The value of each histogram bin is multiplied with its SVM weight then parallel reduction is used to add the weighted values and finally the hyperplane bias is subtracted. The final scores are written back to global memory.

5.6.1 Computing Formatted Results and Sorting

After the SVM scores are evaluated, the scores are stored in one large array of floats of size = (NumberofwindowsX * NumberofwindowsY * ScaleCount) which is of the order of a million (1048576) for a a 1280x720 image. These scores are then mapped to other attributes such as x, y, width, height of the detection window. For this a Kernel with 3 dimensional grid is quede for which the x and y are the detection windows in x and y and z represents the scale. If we segment the grid by z, each segment would represent the detection results from one full convolution of the image with detection window of a single scale. Now as we the scores mapped to their respective detection, we perform sorting on the basis of scores. We use a bitonic sort(swaps in stages are completely independent of each other) to do the task which has a complexity of $O(n(\log n)^2)$.

5.6.2 Non Maximum Suppression

Scanning the classifier across all positions and scales in the image yields multiple detections for the same object at similar scales and positions. Multiple overlapping detections need to be fused together. Standard mean shift algorithm requires a lot of random memory reads and writes. So we have come up with a much simpler parallelizable algorithm which takes the window with the maximum score, and then reject the remaining candidate windows if they have an intersection over union (IoU) larger than a learned threshold. We use the concept of adopting a map/reduce parallelization pattern which uses a boolean matrix both to encode candidate object detections and to compute their cluster Representatives[4].



6 Conclusion

GPU implementation of the histogram of oriented gradients algorithm for pedestrian detection. We achieved a speedup of over 86x, with a single video card. All of this without compromising the performance of the algorithm

References

[1] N. Dalal, B. Triggs. Histograms of oriented gradients for human detection. Computer Vision and Pattern Recognition IEEE Computer Society Conference on, 1:886–893, 2005.

[2] V.Prisacariu, I. Reid. fastHOG - a real-time GPU implementation of HOG. Department of Engineering Science. Oxford University. 2310/09, 2009.

[3] C. Wojek, G. Dork o ,rA. e Schulz, B. Schiele Sliding-windows for rapid object class localization: A parallel technique 30th DAGM symposium on Pattern Recognition, pages 71–81, Berlin, Heidelberg, 2008.

[4] D. Oro, C. Fernandez, X. Martorell, J. Hernando Work-Efficient Parallel non-maximum suppression for embedded GPU architecture,Herta Security, Barcelona, Spain.