fastHOG - a real-time GPU implementation of HOG

Technical Report No. 2310/09

Victor Adrian Prisacariu
University of Oxford, Department of Engineering Science
Parks Road, Oxford, UK
victor@robots.ox.ac.uk

Ian Reid
University of Oxford, Department of Engineering Science
Parks Road, Oxford, UK
ian@robots.ox.ac.uk

January 18, 2012

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. We achieve speedups of over 67x from the standard sequential code, using a single video card. Furthermore it supports multiple video cards so speedups of 120x or more can be achieved. This allows us to achieve real-time performance, using the full HOG algorithm for the first time in the literature. All of this is done while keeping compatibility with the standard sequential implementation. Finally our implementation is available online and is open source.

1 Introduction

The histogram of oriented gradients descriptor is one of the best and most popular descriptors used for pedestrian detection. Unfortunately this technique suffers from one big problem: speed. Like most sliding window algorithms it is very slow, making it unsuitable for any real time application.

There have been attempts at making the detector faster, mostly based on early rejection of scales or positions. For example in [6] the authors use a cascade-of-rejectors approach to eliminate windows. These methods suffer from loss of performance.

In this paper we detail an implementation of the HOG based sliding window algorithm using the NVIDIA CUDA framework. We present improvements of over 67x from the standard sequential implementation of [1]. We are not the first to attempt such an implementation; notably [5]. We believe our implementation to be closer to [1], but, as far as we can ascertain, our implementation is faster than [5], bringing real time full frame HOG within reach for the first time. While [5] uses the cheaper and more easily implemented bilinear interpolation, we mirror [1] in using trilinear; this translates into significantly different implementation. More significantly we describe in detail our mapping to the GPU and provide open source code. Our full source code can be download from http://www.robots.ox.ac.uk/~victor/.

2 HOG Descriptor and Usage for Object Detection

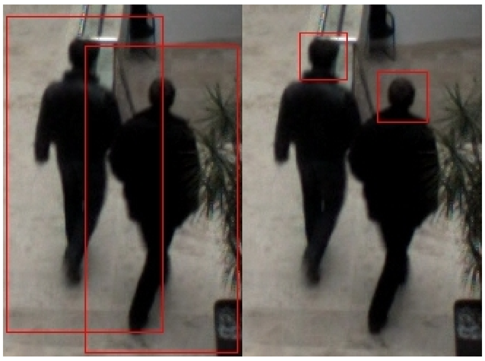

Figure 1: Example results after applying the HOG detector for pedestrians (left) and heads (right)

The histogram of oriented algorithm for object detection was introduced in [1]. An example of detection results for pedestrians and heads is visible in Figure 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 rectangular 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 $\sigma = 8$. 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.

3 GPGPU and the NVIDIA CUDA

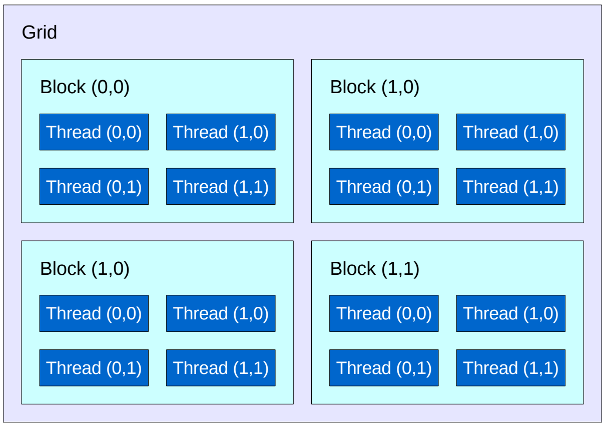

Figure 2: CUDA - thread architecture (after [4])

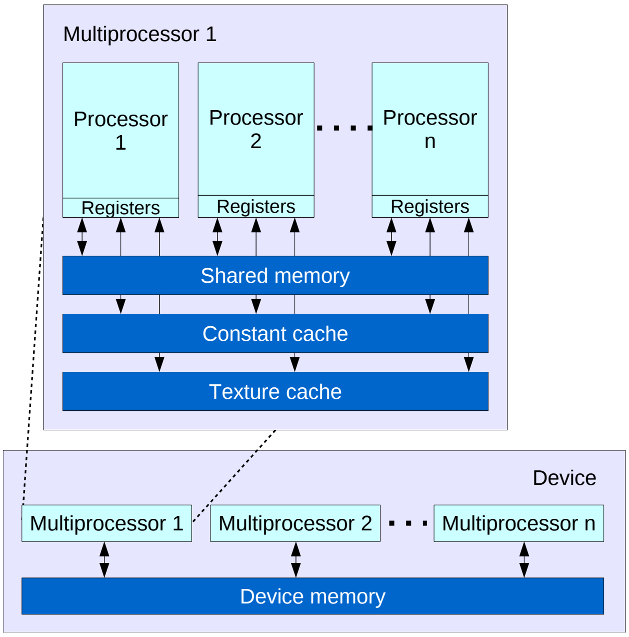

Figure 3: CUDA - memory architecture (after [4])

GPUs have traditionally been used to offload certain parts of the 3D graphics rendering pipeline from the main CPU. New-generation GPUs have a many-core architecture (hundreds of cores), support running thousands of threads in parallel and allow non-uniform access to memory (NUMA). A single graphics card can have a peak single precision float point performance of *1 TFLOPS/s* and a memory bandwidth of up to *102 GB/s*. GPGPU stands for "General-Purpose Computation on GPUs" and is a technique which allows GPUs to be used to accelerate non-graphics problems. NVIDIA CUDA is an SDK, software stack and compiler that allows for the implementation of parallel programs in C for execution on the GPU, therefore enabling direct access to the resources of the GPU, without the limitations of using a graphics API.

CUDA allows C functions to be executed multiple times, by multiple threads, on multiple GPUs. These functions are called *kernels*. The creation of a GPU kernels is virtually free and for complete utilization of the GPU thousands of threads have to be used. A thread can execute a single kernel at any given time. We use *3.749.760* threads to compute the linear SVM evaluation for a single 1920x1080 image. Multiple threads are grouped in blocks and multiple blocks are grouped in grids (Figure 2). Threads in a block can cooperate (they can share memory and can be synchronized) while blocks in a grid are independent. This programming model allows for a very high degree of scalability. Each thread block is executed on only one multiprocessor but a multiprocessor can execute several blocks at the same time.

A thread can access 6 different types of memory: register, local, shared, global, constant and texture memory (Figure 3). All threads can access global, constant and texture memory while only threads in the same block can access a shared memory. Each multiprocessor has its own registers and on-chip shared memory. Each thread has local memory and registers. Constant and texture memory are read-only while all the others are read-write. Global memory access is much slower than shared or register memory access. It takes between 400 and 600 clock cycles to issue a memory instruction for global memory but only 2 clock cycles for shared memory. Constant and texture memories are cached so access to these memories is also very fast. Consequently 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 higher. One final detail worth mentioning is that texture memory allows for virtually free hardware interpolation.

In the current hardware implementation a multiprocessor has 8192 registers and 16384 bytes of shared memory. There are 65536 bytes of constant memory. There cam be up to 512 threads in a single block. The numbers of blocks per grid is restricted to a 16 bit integer.

In our experiments we are using a GeForce GTX 285 video card. It has 32 multiprocessors, each with 8 cores. It is able to run *240 threads* at once.

4 Parallel Algorithms

Our algorithm can be split into 2 parts: host and GPU processing (as shown in Figure 4). 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 transferred back to the CPU, where they are formatted. A formatted result contains information about the position of the window (x, y, scale) and its score. The non-maximal suppression algorithm is executed (using the host CPU) on the formatted data, in order to obtain the final fused results.

The only differences between the standard sequential implementation of [1] are the image padding and resizing. For each scale the results produces by our implementation are virtually identical (up to the 5th or 6th decimal place) to the ones produced in [1]. These small differences are due to rounding errors. The the following sections detail our implementation.

4.1 Host-GPU transfers and Padding

One possbile bottleneck when working with the GPU is the memory copy operation itself, which is limited to PCI Express speeds, This translates to an average of *3GB/s* (2.7GB/s on our machine), which is very slow compared to device to device memory transfers which can running at over *100GB/s*. One way of increasing the speed is to mark the host allocated

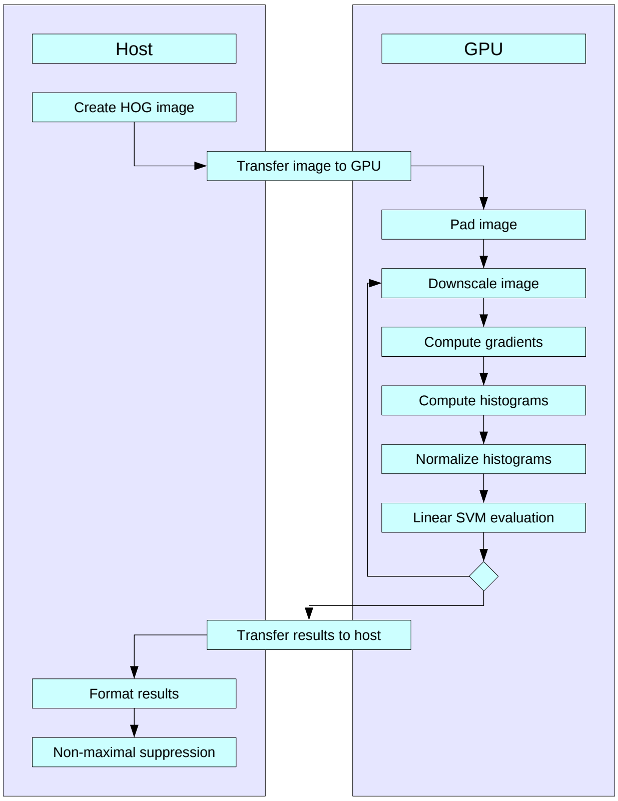

Figure 4: fastHOG steps

memory as unpagable. This means that the operating system will not page it out of main RAM. In latest CUDA SDK, NVIDIA introduced *zero-copy* memory, in which memory does not need to be explicitly copied to the GPU before it is accessed. Instead a device pointer can be mapped to host memory, and small chunks of memory are transferred when needed. This mode of operation has two disadvantages. Firstly the host can modify the image while the GPU is still using it, meaning that asynchronous operations are more difficult. Secondly this mode of operation only works on the newest generation of video cards. In our implementation we chose to use the first mode of operation (unpageable allocation and one memory copy operation per image) because we wanted to keep the usage of our library asynchronous, and to keep it backcompatbile to older video cards.

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.

4.2 Downscaling and Gamma Normalization

We use the hardware texturing unit to downscale the image. This is different from the standard sequential implementation, where the downscale is implemented in software. By copying the padded image inside a texture we can simply use the **tex2D** function to obtain a bilinearly interpolated value for each pixel. To minimize memory operations we also apply the gamma normalization at this stage.

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.

4.3 Color Gradients

To compute the color gradients we use two separable convolution kernels similar to the ones from the NVIDIA CUDA SDK. The first kernel convolutes the row with the centered 1-D mask, while the second kernel computes the column convolution, gradient orientations and magnitudes. According to the SDK thread block sizes are fixed to 145 and 128 threads for row and column convolutions, respectively.

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 host memory: gradient orientation and magnitude. In the color case the maximum magnitude (and corresponding orientation) is chosen.

4.4 Block Histograms

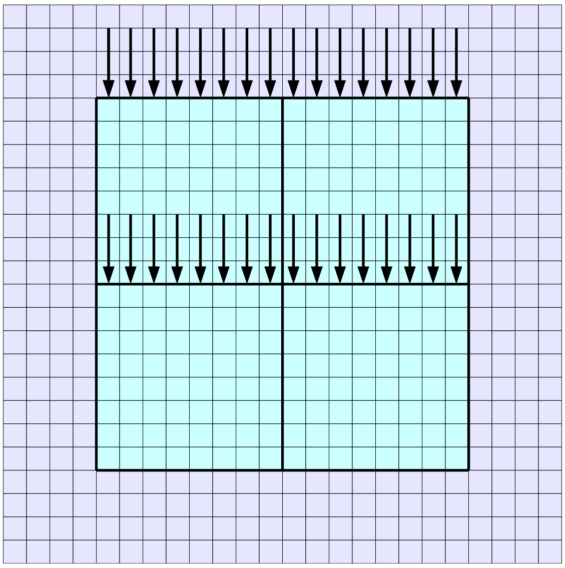

Figure 5: Thread configuration for the block histogram step. The dark area represents one block, split into cells. The vertical arrows represent threads and the direction they process the pixels. Each thread processes one column from once cell in a block.

In the histogram computation step a HOG pixel block is mapped to a CUDA thread block. In the case of pedestrian detection the block has 4 cells and each cell 8 columns of 8 pixels, so we will use 8x4=32 threads.

We logically divide each thread block into 4 sub-blocks, each processing one cell. Each sub-block consists of 8 threads, each thread processing 8 pixels (one cell column). The thread configuration is depicted in Figure 5. While this thread model might not fully utilize the GPU hardware in some cases, it does have the advantage of scaling to different block/cell configurations.

In normal histogramming each pixel would be counted in its nearest orientation bin. This causes an aliasing effect, which may lead to sudden changes in the computed feature vector

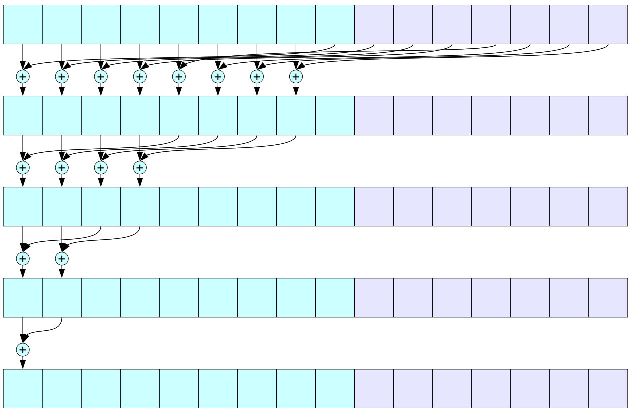

Figure 6: Parallel reduction algorithm, example for an array of 16 values. The array is stored in shared memory, and the numbers are added in *O(logn)* operations, where n is the size of the array. Using CUDA, in order to avoid shared memory conflicts, the first 8 numbers are added to the last 8 numbers (and so on), rather than adding the 1st number to the 2nd, the 3rd to the 4th, etc. When adding histograms, when we consider the 9 bin case, only the blocks in dark blue are added so in the first step of the reduction only one thread is actually used.

([1]). To avoid this, the authors in [1] use trilinear linear interpolation of the pixel weights into the orientation histograms.

1	1 and 3	1 and 3	3
1 and 2	1, 2, 3 and 4	1, 2, 3, and 4	3 and 4
1 and 2	1, 2, 3 and 4	1, 2, 3 and 4	3 and 4
2	2 and 4	2 and 4	4

Figure 7: A 2x2 block and the memory write pattern used for trilinear interpolation of the histogram bins. Each cell is split into 4 subcells. The pixels in the subcell marked with 1 contribute to the histogram of cell 1, the pixels in the subcell marked with 1 and 3 contribute to the histogram of cells 1 and 3, etc.

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, as depicted in Figure 7 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.

In contrast, in [5], the authors chose the bilinear interpolation technique, where each pixel will contribute to fewer bins, so less shared memory will be required and therefore a different thread block layout can be used. Also the weights do not need to be computed as it is more advantageous to read them from a texture. We chose to use trilinear interpolation because we wanted to keep our implementation as close to the original one as possbile and it improves the performance of the detector (according to [1]).

Finally these histograms are merged using a parallel reduction algorithm (Figure 6). The parallel reduction is similar to the one from the **reduction** example from the SDK.

The Gaussian weights are stored in a texture.

The values are written back to global memory in a 2D 18x2 grid.

4.5 Histogram Normalization

In the histogram normalization step a HOG pixel block is mapped to a CUDA thread block. In the current cell/block configuration each block consists of 4 histograms, each histogram having 9 bins. In our implementation each thread processes one pixel, so we are using 9x4=36 threads.

Each block is kept in shared memory for the duration of the processing and wrote back to global memory in the same grid like in the previous step.

We use the parallel reduction (Figure 6) 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.

4.6 Linear SVM Evaluation

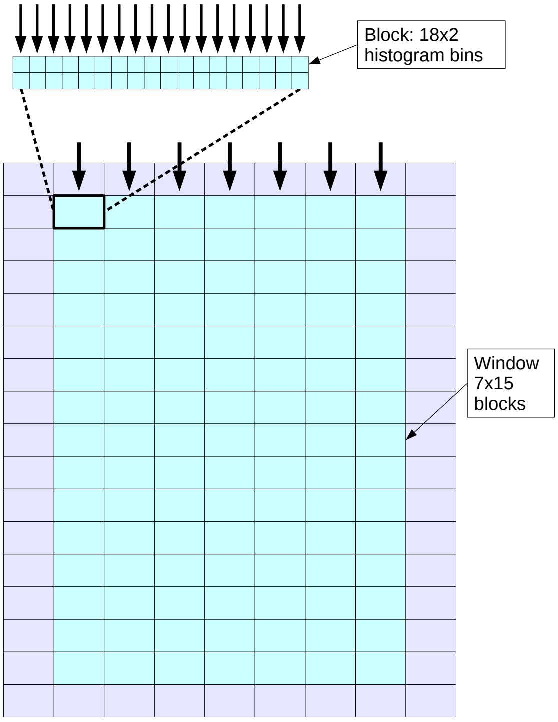

Figure 8: Thread configuration for the SVM evaluation step. The vertical arrows represent threads, and the direction they process the pixels. For our case, where a window has 64x128 pixels equal to 7x15 blocks, a thread block has 7x18 threads, each thread processing 15x2 values.

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 Figure 8.

The value of each histogram bin is multiplied with its SVM weight then parallel reduction (Figure 6) is used to add the weighted values and finally the hyperplane bias is subtracted. The final score is written back to global memory.

For each position and all scales the SVM scores are stored in global memory and then transferred back to host memory.

4.7 Non-Maximal Suppression

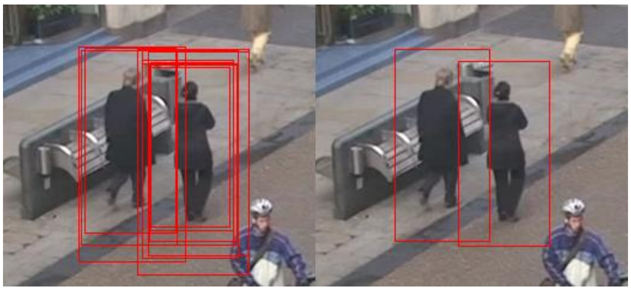

Figure 9: Non-maximal suppression. left - all detections, right - fused detections

Scanning the classifier across all positions and scales in the image yields multiple detections for the same object at similar scales and positions (Figure 9). Multiple overlapping detections need to be fused together. This is achieved using a mean shift algorithm in 3D position/scale space .

We do not run this part of the algorithm in the GPU because it requires a lot of random memory reads and writes. We are investigating an efficient parallel implementation.

After running the non-maximal suppression algorithm the result is an array of results, each with a 3D position (x, y and scale) and a SVN score.

4.8 Multi-GPU Support

The NVIDIA CUDA framework does not allow direct use of multiple GPUs: a host thread can execute device code on only one device at a time. It is impossible to define multiple grids (one for every device) and run a kernel on the multi-grid configuration. To be able to take advantage of multiple devices we create a separate thread for each device. The image is split into multiple regions of interest, and each region of interest is processed on a separate device. A 1920x1080 image for example will be split into two 960x1080 regions of interest and each will run on a separate device. The results are then be fused using the non-maximal suppression algorithm. Also, in our implementation, the user is able to define multiple regions of interest for a single image (when executing the algorithm). In this case each region is run on a separate device, using the Round-Robin load balancing algorithm.

5 Library Design and Usage

Because the source code for our implementation is available online we have decided to include in this technical report a brief description of the design and usage of our implementation.

5.1 Design

The following classes are used:

- **HOGEngine** – main class. It is implemented using the singleton design pattern (we only need one processing engine). It also could be thought as implementing the adapter pattern for the CUDA part of the code ([2]).

- **HOGImage** – image class. It encapsulates the array of bytes containing the image, its width and its height. Methods for reading the image from a file are also included. These are implemented using the FreeImage library.

- **HOGNMS** – the non-maximal suppression algorithm.
- **HOGResult** – result class. Encapsulates the 3D scale/space position of a results and its SVM matching score.
- **HOGROI** – region of interest class. Our implementation supports regions of interest defined across the 3D scale/position space. A region is defined by the minimum and maximum x and y positions and by the start and end scales.

CUDA uses C, not C++ so there are no classes in the CUDA part of the code. However we have have split this part of code into the following files:

- **HOGEngineDevice** – the counterpart of the **HOGEngine** class, for the CUDA part of the code. It controls the GPU part of the algorithm.
- **HOGPadding** – functions for padding the image (and implicitly for transferring the image to the GPU).
- **HOGConvolution** – functions for the row/column convolution, gradient magnitude and orientation computation.
- **HOGScaling** – functions for image scaling.
- **HOGHistograms** – the functions responsible with histogram computation and block normalization.
- **HOGSVMSlider** – the functions responsible with the linear SVM evaluation.
- **HOGUtils** – other utility functions, like conversion from a unsigned char image to a float image.

5.2 Usage

The engine needs to be initialized (only once), with the width and height of the image and SVM training data. The data can be read from a text file. Inside the engine different parameters are specified, like block size, cell size, etc. These parameters can be modified.

The algorithm is called trough 2 methods in the **HOGEngine** class: **BeginProcessing** and **EndProcessing**. The execution pattern for these two methods is asynchronous: the host is free after the **BeginProcessing** method has been called. When the **EndProcessing** method is called the host will wait for the GPU computation to finish and the results will be transferred to the host, where the non-maximal suppression will be run.

The **BeginProcessing** method has up to 3 parameters:

- **image** – the **HOGImage** to be processed.
- **ROIs**, **ROIsCount** – regions of interest. A Round-Robin load balancing algorithm is used to spread the regions across multiple video cards (see Section 4.8) . If no regions are defined the image will be split across all available video cards and the results will be fused in the non-maximal suppression step..

6 Benchmarks

Our implementation is designed to produce the same results as the normal CPU implementation of [1]. With the padding disabled and with our resize method, the results produces by our implementation and the standard sequential one of [1] are identical. We therefore believe that the tests done in [1] are valid for our implementation as well.

We tested our algorithm on a variety of images and video cards. The speed of the processing is obviously a function of the number of multiprocessors on the video card. Using a single GeForce GTX 285 video card we noticed about a 67x improvement over the normal sequential implementation. The processing time for one 640x480 image is between *74ms* and *84ms*, depending on the number of people in the image (because the speed of the non-maximal suppression is a function of the number of detections). With 2 GeForce GTX 295 video cards (in a Quad SLI configuration) our implementation should be able to run the full algorithm (all positions and scales) over images with 1920x1080 size at about 10 fps. Using the same configuration 640x480 images could be processed at about 40 fps. To our knowledge this is the first time when HOG performance can be usefully measured in frames per second and not seconds per frame. We expect these speeds to almost double with the next generation of NVIDIA video cards and therefore allow for real time 1920x1080 HOG tracking. We chose to keep the parameters (block size, cell size, etc.) reasonably flexible at the expense of some speed. Still we do believe that if the algorithms are adapted to a more specific configuration greater speed will be achieved.

In Table 1 we show a comparison between the processing time required for an image at three size, when using our implementation, compared to the parallel implementation of [5]. We could not run a side-by-side comparison since we do not have access to their implementation.

We ran our implementation on the INRIA dataset.([1]), testing our implementation with all 288 positive images. In Table 2 we compare the processing time for our implementation (both color and grayscale modes) with the [5] GPU and CPU implementations. When comparing GPU implementations, ours is about twice as fast in color mode and about three times as fast in grayscale mode. We achieve a 67x speedup in color mode and a 95x speedup in grayscale mode, over their CPU implementation. This is achieved using a single video card. With two video cards we believe the speedups would almost double.

Implementation	320x240	640x480	1280x960
Ours (grayscale)	19ms	64ms	261ms
Ours (color)	22ms	80ms	353ms
Wojek et. al [5]	29ms	99ms	385ms

Table 1: Example processing times for different image sizes

Implementation	Processing time
Ours (grayscale)	22s
Ours (color)	31s
Wojek et. al ([5]) GPU	1m 9s
Wojek et. al ([5]) CPU	35m 1s

Table 2: Processing time for the INRIA Person test set

There are two ways of measuring CUDA code performance: *memory throughput* and *occupancy*. The *overall memory throughput* of a CUDA kernel measures how fast the kernel reads, processes and writes from global memory. This measure is affected by the memory read and write pattern and by the processing done inside the thread. *Occupancy* measures how much a multiprocessor is used by a kernel function. Higher occupancy leads to better performance by hiding the latency of global memory access. Occupancy has a maximum value of 100%. It is determined by the number of registers, amount of shared memory used and thread block configuration.

The most costly step in our implementation is the computation of the cell histograms. This step takes about 50% of the total processing time and is done with an average bandwidth of *11GB/s* (using our video card). The linear SVM evaluation step for example has an average bandwidth of more than *96GB/s*. See Table 3.

Processing step	Color	Grayscale
Block histogram computation	11 GB/s	11 GB/s
Block histogram normalization	19 GB/s	19 GB/s
Column-wise convolution	43 GB/s	91 GB/s
Bilinear downscaling	43 GB/s	22 GB/s
Row-wise convolution	72 GB/s	79 GB/s
Linear SVM evaluation	96 GB/s	96 GB/s

Table 3: Overall memory throughput for each step in the algorithm, for color and grayscale modes. Bilinear downscaling is slower in the grayscale mode because the image is converted to grayscale. Row-wise convolution is faster in grayscale mode because fewer values are read from memory. Column wise convolution is considerably faster (accounting for the 20% overall speedup when using grayscale) becase fewer values are read from memory and the maximum of the magnitude does not need to be computed.

A full outline of the processing time required by each step is detailed in Figure 10. We chose not to use bilinear interpolation (or no interpolation at all) in order to keep our implementation as close as possible to the standard sequential one ([1]). According to [1] this helps improve performance considerably.

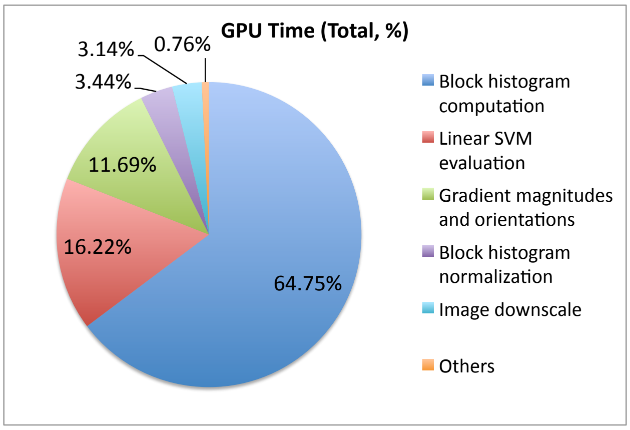

Figure 10: GPU time for one image, all positions and scales processed.

In Table 4 we detailed the occupancy of each kernel used by our algorithm, in both color and grayscale modes. Occupancy is lowest in two kernels: column-wise convolution and block histogram computation. In the first case we could get occupancy up by separating the color channels after the downscaling kernel, running the convolutions on each channel separately and having another kernel chose the channel with the maximum gradient magnitude. We have tested this method and it lead to poorer performance in the downscaling kernel and in the row-wise convolution kernel, which caused an overall loss in performance. Occupancy is worst with the block histogram computation kernel. This is because, in order to do the trilinear interpolation, each thread must hold in shared memory the complete block histogram. If we used no (or bilinear) interpolation fewer the amount of shared memory needed would have been smaller, and occupancy would have been higher and therefore speed would have been higher. This however would have lead to lower performance (as stated in [1]).

Processing step	Color	Grayscale
Linear SVM evaluation	100%	100%
Bilinear downscaling	100%	100%
Row-wise convolution	94%	94%
Block histogram normalization	50%	50%
Column-wise convolution	17%	50%
Block histogram computation	9.4%	9.4%

Table 4: Occupancy for each step in the algorithm.

7 Conclusions

In this paper we have described a GPU implementation of the histogram of oriented gradients algorithm for pedestrian detection. We achieved a speedup of over 67x, with a single video card. All of this without compromising the performance of the algorithm. With small decrease in performance a speedup of over 95x can be achieved. Our implementation supports multiple video cards, so greater speedups are possible. Furthermore, our implementation is permissive with respect to changes in algorithm parameters (cell/block/window size, etc). Finally all the source code is available online.

References

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

[2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Professional, 1995.

[3] David G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60:91–110, 2004.

[4] NVIDIA. *NVIDIA CUDA Programming Guide 2.0.* 2008.

[5] Christian Wojek, Gyuri Dorkó, André Schulz, and Bernt Schiele. Sliding-windows for rapid object class localization: A parallel technique. In *Proceedings of the 30th DAGM symposium on Pattern Recognition*, pages 71–81, Berlin, Heidelberg, 2008. Springer-Verlag.

[6] Qiang Zhu, Shai Avidan, Mei C. Yeh, and Kwang T. Cheng. Fast human detection using a cascade of histograms of oriented gradients. In *CVPR*, pages 1491–1498. IEEE Computer Society, 2006.