EC527: High Performance Programming with Multicore and GPUS

GPU Implementation of

Histograms of Oriented Gradient Features

Project Report By:

Abhishek Gaur (abhigaur@bu.edu)
Ashutosh Sanan (asanan20@bu.edu)

Abstract

With the relentless innovation in computer vision owing to the vast applications in the fields of medical imaging, human computer interaction, self-driving cars, virtual reality etc., performance becomes critical. Computer Vision tasks are computationally intensive and repetitive, and they often exceed the real-time capabilities of the CPU, leaving little time for higher-level tasks. However, many computer vision operations map efficiently onto the modern GPU, whose programmability allows a wide variety of computer vision algorithms to be implemented. Histogram of oriented gradients (HOG) is one of the most popular descriptors used for feature extraction. Though it is one of the fastest descriptors available, the computation is still time consuming like most sliding window algorithms. This project presents the implementation of HOG algorithm using CUDA on NVidia GPU and also presents a comparative analysis with the CPU implementation of the same.

Introduction

Histogram of Oriented Gradient (HOG) is a method to obtain feature descriptor for interest points in an image. This method also uses gradient and defines the descriptor for each point based on histogram of gradients. This technique is based on a hypothesis that local shape appearance of an object is well described by edge directions irrespective of actual position of these edges.

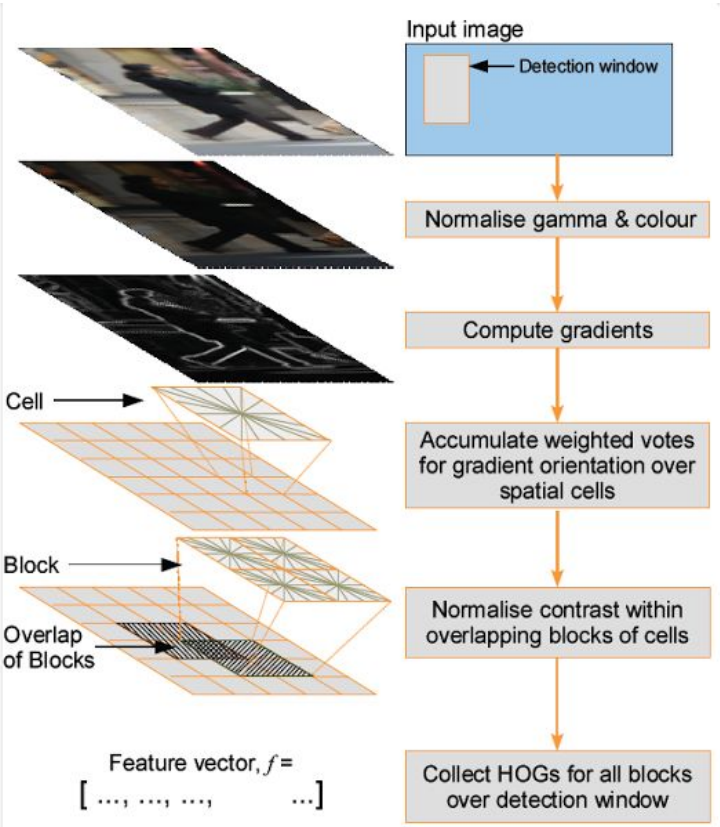

Figure 1: Steps involved in HOG Algorithm

(http://cs.brown.edu/courses/cs143/2011/results/proj4/hangsu/img/HOG.png)

The HOG descriptor has a few key advantages over other descriptors. Since it operates on local cells, it is invariant to geometric and photometric transformations, except for object orientation. The algorithm follows the following steps:

- Divide image into small rectangular or circular regions.
- Computer the gradient magnitude and orientation of the object.
- For each region find the histogram of edge directions.
- These histogram entries act as a feature vector describing the local features of the shape we are interested in.
- Normalize the cells across larger regions to provide better illumination invariance.

Since an image is made up of large number of pixels and due to sliding window nature of the HOG,

operations like gradient computation, histogram mapping and normalization are very computationally intensive giving us a massive room for improvement in terms of performance.

The GPU provides a streaming, data-parallel arithmetic architecture. This type of architecture carries out a similar set of calculations on an array of image data. The single-instruction, multiple-data (SIMD) capability of the GPU makes it suitable for running computer vision tasks, which often involve similar calculations operating on an entire image.

We implemented the HOG descriptor algorithm from scratch using OpenCV libraries on a CPU and then wrote the same implementation on an NVidia GPU using CUDA. The upcoming parts of the report will describe our baseline serial implementation along with some optimizations followed by GPU implementation. We will then present our results consisting of output images and comparison of CPU and GPU implementation.

Implementation

Baseline Sequential Implementation on CPU with loop unrolling

We used a MATLAB code as a reference to understand the working of the algorithm. We then wrote the C++ code using OpenCV libraries to implement the algorithm step-by-step.

Our CPU specifications were as follows:

Intel® Core™ i5-4570S CPU @2.90 GHz

Cores: 4

Threads: 4

Cache:

L1: 4 X 32KB 8-way set-associative (Individual Instruction and Data)

L2: 4 X 256KB 8-way set-associative (Shared)

L3: 6MB 12-way set-associative (Shared)

We started with a 512 X 512 image and calculated the gradient over whole image leading to 262,144 computations in total. The next step was to divide image into blocks of 16 x 16 with 50% overlap between adjacent blocks. Overlapping was done owing to the fact that changes in contrast are more likely to occur over small regions within the image. Each block was then futher divided into 2X2 cells. Next, we did binning of the image, which basically is just weighted polling of gradient orientation for each pixel in the cell. We got 9 bins ranging from 0-360 degrees with an interval of 40 degree each.

The final implementation looked like this:

Image Size: 512 X 512

HOG Block Size: 16 X 16

HOG Cell Size: 2 X 2

Number of Bins: 9

On making sure that the output was correct, we did loop unrolling in the various for loops that we used in each and every step. Then we ran our code for various image sizes ranging from 64 X 64 to 2048 x 2048 and noted the computation time for each of them.

GPU Implementation

Our GPU implementation approach can be seen in figure 2. We start with transferring our image from host to GPU for processing. After various steps on the GPU we get the HOG features on GPU and then transfer them back to CPU for visualization. The first step in our project was to determine parallel compute-intensive blocks in the complete algorithm. Therefore it was very vital for our project to first analyze the code prior to implementation and coding.

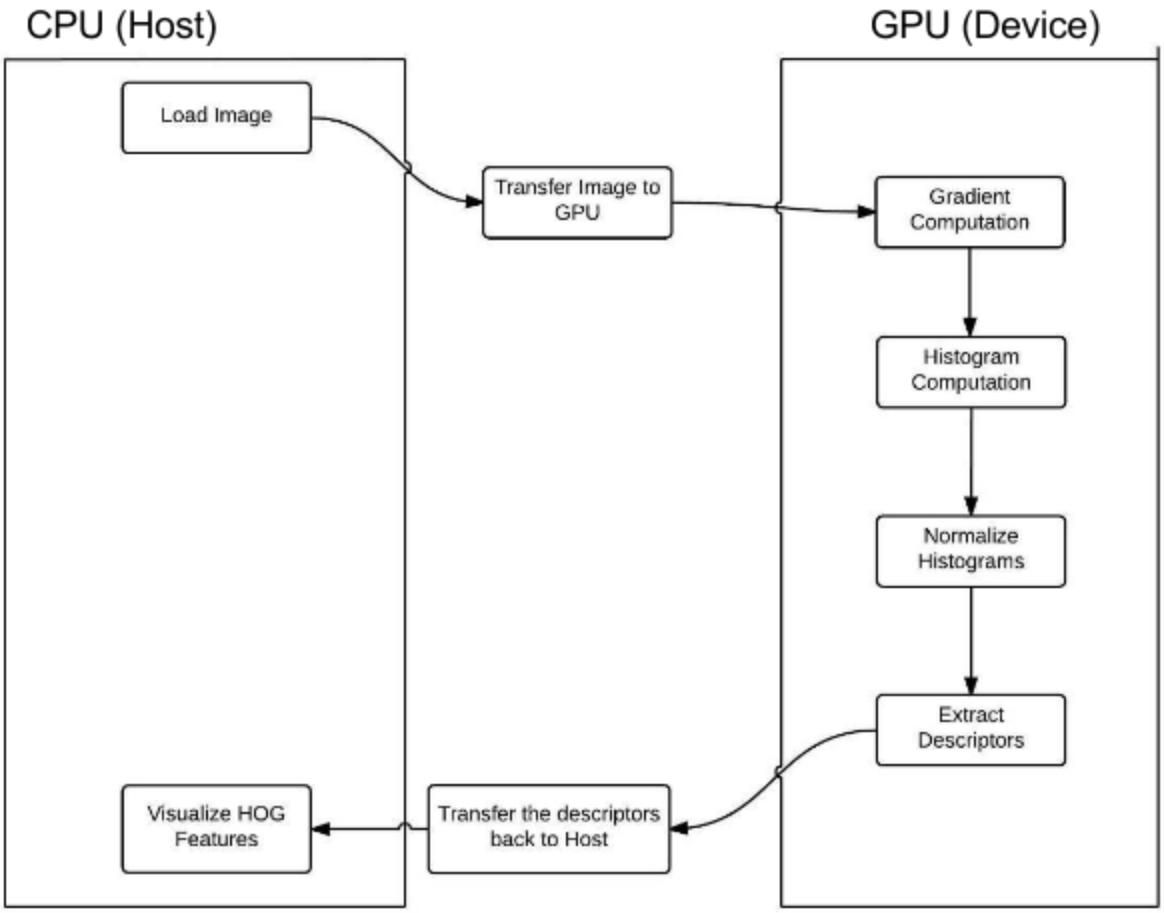

Figure 2: GPU Implementation for HOG feature extraction

Gradient Computation:

For the Gradient Computation we took 16*16 threads in each block to calculate that respective area of the input image. The number of blocks will be equal to (Height/16)*(Width/16). We have used shared memory for this computation. Using shared memory improved the performance by approximately 1.8x for image size 1024*1024.

For gradient purposes we are using 1-D mask which will calculate gradients for both x and y direction. Using that each thread will compute both magnitude and orientation.

Histogram Computation

For this step, each of the HOG Block corresponds to a CUDA block. Each block will have 32 threads (4*8 block). Each HOG block has 4 cells in which there are 8*8 pixels each. So we logically divide each CUDA block into four sub-blocks which consists of 8 threads to compute 8 pixels (one cell row).

We use shared memory for this step to speed up the memory reads and writes.

Parallelized Region	Number of Blocks	Number of Threads
Gradient Computation	(Width/16) * (Height/16)	256 (16 x 16 blocks)
Histogram Computation	Number of image HOG-blocks	32 (4 x 8 blocks)
Normalization of Histograms and Extraction of Features	Number of image HOG-blocks	32 (4 x 8 blocks)

Table 1: Number of Blocks and Threads for CUDA Implementation

Histogram Normalization and Extraction of Features

This uses same number of threads and blocks as used in the histogram computation step. Each thread is responsible for the normalization.

For feature extraction we use global memory and save the final descriptor to the same memory so that it can be transferred to the Host memory in the end.

Code Snippets for GPU Code:

```
cudaMemcpy(img_d, img_h, rows*cols*sizeof(float), cudaMemcpyHostToDevice);

const int bl_size_x = 16;
const int bl_size_y = 16;

dim3 threads(bl_size_x, bl_size_y);
dim3 grid((int)ceil(rows/(float)bl_size_x), (int)ceil(cols/(float)bl_size_y));

gradient_kernel<<<grid,threads>>>(img_d, mag_d, theta_d, rows, cols);
```

The above snippet shows how we split our blocks and grid for computing gradient using CUDA. The name of the kernel is **“gradient_kernel”.**

The same is shown below for computing histogram and finding descriptor:

```
dim3 dimGrid;

dimGrid.x = (int)floor(cols/8.f)-1;
dimGrid.y = (int)floor(rows/8.f)-1;


dim3 dimBlock(4,8);
printf("Grid: %d\t%d\n", dimBlock.x, dimBlock.y);

d_compute_desc_kernel<<<dimGrid, dimBlock>>>(mag_d, theta_d, blocks_desc_d, rows, cols);
cudaMemcpy(hog_descriptor, blocks_desc_d, total_blocks_size, cudaMemcpyDeviceToHost);
cudaDeviceSynchronize();
cudaPeekAtLastError();
```

Above code shows how we spit our processing among threads in GPU. The CUDA kernel name is **“d_compute_desc_kernel”**.

Results:

Figure 3: Original Image

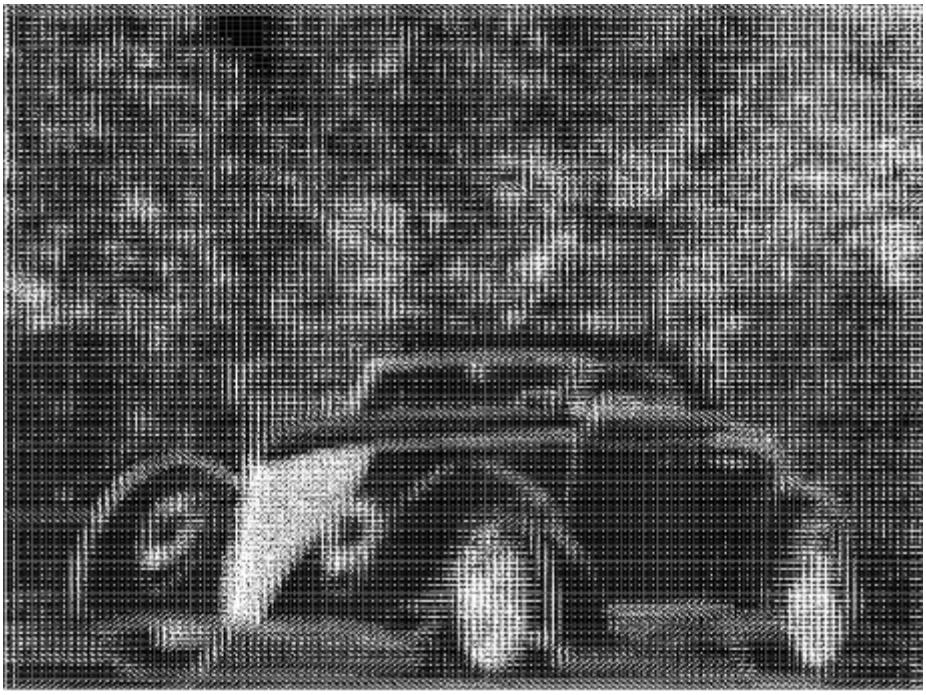

Figure 4: HOG Descriptor Visualization

Image Size	CPU Time (msec)	GPU Time (msec)	Speedup
64 x 64	230	120	1.91
128 x 128	228	103	2.21
256 x 256	256	143	1.79
512 x 512	447	129	3.46
1024 x 1024	1244	206	6.03
2048 x 2048	4222	526	8.02

Table 2: Comparison of CPU vs GPU and Speedup

Conclusion

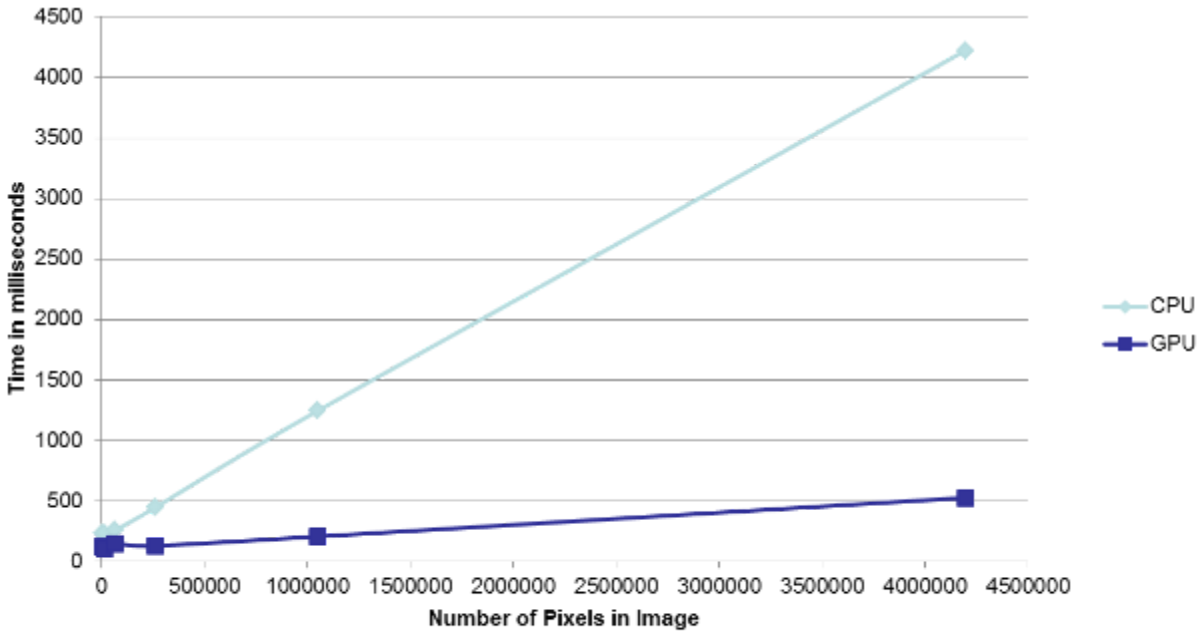

Figure 5: Performance Analysis of HOG Descriptor on GPU and CPU

The above graph shows the improvement in performance for calculating HOG descriptors using GPU. It is clearly evident that the GPU is successful in giving more than 6 times better performance for large images. But it can be seen that for smaller images there is not much differences in both the

performance. The reason for this is that there is a overhead to transfer the image from HOST machine to GPU, and for smaller images this overhead is much significant as compared to the processing improvement provided by the GPU, but for larger images the overhead doesn't make much of a difference.

So to conclude, we can say that GPU is a very efficient device for parallel computation but before starting the implementation and coding we need to make sure that it is worth making the code section parallel and the overhead for sending data to GPU doesn't affect the performance much.

References

- *"Histograms of Oriented Gradients for Human Detection"*

  Navneet Dalal and Bill Triggs, 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition.
- V A Prisacariu, I D Reid. fastHOG - a real-time GPU implementation of HOG.

  http://www.robots.ox.ac.uk/~lav/Papers/prisacariu_reid_tr2310_09/prisacariu_reid_tr2310_09.html, 2009.
- CUDA Programming Guide – NVIDIA Documentation

  http://docs.nvidia.com/cuda/cuda-c-programming-guide/#axzz46rMX1vuM
- OpenCV

  http://opencv.org
- *"Histogram of Oriented Gradients using MATLAB"*

  http://www.mathworks.com/matlabcentral/fileexchange/28689-hog-descriptor-for-matlab