A Fast and Accurate GHT Implementation on CUDA

Nikhil Jotwani, Sudhakar Sah
Center for Research in Engr Sciences and Technology-CREST
KPIT Cummins Infosystems Ltd., Pune, India
e-mail: {sudhakar.sah, nikhil.jotwani}@kpitcummins.com

Abstract

Generalized Hough Transform (GHT) is a well known but seldom used algorithm for object detection. The merit of this algorithm is its ability to detect object location and its pose accurately. However, this algorithm has a huge drawback of high memory and extensive computational requirement. As a result, usage of this algorithm for object detection is limited. In this paper, we are proposing the parallel implementation of GHT algorithm on GPU (Graphical Processing Unit) that is 80 times faster compared to its CPU (Central Processing Unit) version. We have also achieved comparable speed up with some of the best GHT implementations on GPU for limited number of poses. However, our parallel design performs better for large number of poses. The uniqueness of our parallel design is that the performance does not get affected by increasing number of poses. Increased number of poses identification at the same performance increases the resolution of scale and rotation that can be detected.

Keywords-CUDA; GPU; GHT parallelization; Object detection

1 INTRODUCTION

Object detection is an important problem in image processing and variety of methods are available in the literature. FFT based image registration is one of such method [4] [5]. The method is accurate but it has limitation for higher scaling. (Generalized Hough Transform) GHT is used to detect the object location and poses (scale and rotation) given the reference image with same object. Although, GHT is an accurate method of object detection, it suffers from a major drawback of huge memory requirement and extensive computational complexity. The memory requirement comes from storage of data for all possible poses into memory for voting to detect object position and its pose correctly. But the fact that these operations are performed repeatedly on different data elements makes this algorithm data parallel (with some modification). General purpose graphical processing unit (GPGPU) has become popular because of its massive parallel processing power that is suitable for data parallel algorithm. The GPGPU programming has also been made very easy and effective by NVIDIA CUDA (Compute unified device architecture) technology. CUDA provides top level APIs to ease parallel programming. Programmers can concentrate on the parallel algorithm design as most of the programming burden is reduced by CUDA. Because of the flexibility provided by CUDA, we can design or model our implementation according to the architecture of the GPU and achieve high speed up.

In this paper, we discuss the design and implementation of the computational and memory intensive but data parallel GHT algorithm. This implementation provides the dramatic improvement in performance compared to the CPU implementation. Compared to the other implementations [2],[3] in literature, our implementation performs better for higher number of object poses. This has been made possible by flexibility in parallel design that makes the GHT scalable without any performance penalty. We have computed up to 18000 poses of the object. We are able to get higher resolution for object detection in scaling and angle rotation parameters. The paper is organized as follows. Section 2 explains the related work done in this area. Section 3 talks about the GHT concepts in brief. Section 4 explains the CUDA concept. Section 5 explains the parallel implementation of GHT. Section 6 and 7 explains the results and conclusion respectively.

2 RELATED WORK

GHT implementation on GPGPU has already been proposed in the past. Strzodka et al. [2] proposed the parallel GPU implementation for the first time to detect object in less than one minute time. However, the implementation was done before introduction of CUDA and GPU was used for texture fetches for performance improvement. Juan et al. [3] proposed the GHT parallel design based on CUDA to achieve 44 times speed up over the CPU implementation and six times speed up over Strzodka implementation. This was remarkable speed up but the speed up was less because of limitation in the thread distribution. Each thread is assigned the job to compute a contour point of one pose with all the contour points in image. Hence, the load on each thread was too high to reduce the parallelization performance. Also, performance of the algorithm will reduce by increasing the number of poses (Reported poses – 990). This is a limitation the Strzodka’s implementation.

In this paper, we proposed an improved parallel design with following modifications

- Modification in the edge compaction algorithm
- Each thread is assigned to compare contour point of each pose with one contour point of image
• Reduction of memory usage by avoiding storage of pose table. Instead, voting process is done during calculation of pose.

Using above techniques, we have achieved better performance compared to the earlier implementation. We have also reduced the memory requirement by 40% that allows us to detect more poses and more edge points for the same image within same available memory. Also, because of the availability of large number of poses, we are able to provide very high resolution of scale and rotation.

3 GHT

The generalized Hough transform is mainly used for object detection. The main steps involved in the algorithm include

• Edge detection
• Edge compaction
• Pose table creation
• Voting in the parametric space
• Finding the max in voting space

3.1 Edge detection

There are two images required in GHT for object detection viz. the reference image (RF_I) and the template image (TP_I). Edges in these images are detected separately to know the contour of the objects. These contour points are required for the object detection. The edge detection is performed using standard Sobel operator. The Sobel kernels for horizontal and vertical edges are given in equation (1) and (2) respectively.

\[
S_H = \begin{bmatrix}
-1 & -2 & -1 \\
0 & 0 & 0 \\
1 & 2 & 1 \\
\end{bmatrix}
\]  

\[
S_V = \begin{bmatrix}
-1 & 0 & 1 \\
-2 & 0 & 2 \\
-1 & 0 & 1 \\
\end{bmatrix}
\]  

The edge detection for the RF_I and TP_I are performed as described by the equation (3) and (4) respectively. Let RF_Ei and TP_Ei be the edge images of RF_I and TP_I respectively. If G_X and G_Y indicate the gradient of the edge in the x and y direction respectively then,

\[
G_X(i,j) = RF_Ei(i,j) * S_H(k,l)
\]

\[
G_Y(i,j) = RF_Ei(i,j) * S_V(k,l)
\]  

Where i = 0 to width of RF_I

j = 0 to height of RF_I

k = 1 to 3

l = 1 to 3

\[
RF_Ei(i,j) = \sqrt{(G_X(i,j))^2 + (G_Y(i,j))^2}
\]  

The edge intensity for each pixel in the images is given by the equation (5). Edges for TP_I are computed in the similar manner.

3.2 Edge Compaction

Edge points provide important information about the image and it is sufficient for object detection using GHT. Edge points are the subset of complete image and hence most of the image points are not relevant during processing. [7] explains the importance and method of compaction. To reduce the computation cost and the time of execution, we store the information of the edge points from RF_Ei and TP_Ei in compacted arrays and operate on them instead of scanning the entire image. This method is called edge compaction. To determine the size of the compacted arrays for RF_I and TP_I we need to get the count of the number of edge points in each of the images. Once the count is known, the edge information is filled in the arrays. Figure 1 explains the edge compaction concept diagrammatically.

3.3 Pose table creation

The pose table is used to store the various possible poses of the object. The poses include change in rotation and scaling. Each edge point of the template image is subjected to all poses. So each edge point has scaling variations as well as angle variations. Let T be the magnitude of the reference vector, \(\phi\) be its angle. Reference vector along with the magnitude and angle is shown in figure 2. (X_O, Y_O) are the reference point in the template image. This reference point is usually taken as the centre of the image. (X_1, Y_1); (X_2, Y_2); (X_3, Y_3) are the edge points, the arrow from the edge point to the reference point shows the reference vector for
the particular edge point. Let $Tx'$ be the modified x value of the vector and $Ty'$ be the modified y value of the vector. Then we can compute the pose table by equation 6 and 7.

$$T_{x \text{'}} = Sc_{c} \times T \times \cos(\phi + \text{Ang}_{j}) \quad (6)$$

$$T_{y \text{'}} = Sc_{c} \times T \times \sin(\phi + \text{Ang}_{j}) \quad (7)$$

Where $Sc_{c}$ varies from min scale to max scale value and $\text{Ang}_{j}$ varies from min angle to max angle value.

**Template image**

![Template image](image)

Fig. 2 Reference vector of edge points in the template image

![Pose table](image)

Fig. 3 Pose table

Figure 3 depicts the dimensions of the pose table. It shows the way in which the poses for each edge point is stored. The X axis contains the TP$_{i}$ edge points, Y axis contains the variation in the angle and the Z axis stores the variation in the scaling values.

### 3.4 Voting in the parametric space

Voting in the parametric space for GHT is computationally very expensive. It involves lot of iterations to compute the possible centre of the object to be detected. RF$_{t}(i,j)$ are the edge points in the reference image. Using these edge points and the information from the pose table, possible center of the object is computed. Let the co-ordinates for the centers be denoted by $N_{X}$ and $N_{Y}$. The calculation of the new co-ordinates is given by equation (8) and (9). Here $N_{X}$ and $N_{Y}$ are the X and Y co-ordinate of the centre.

$$N_{x} = RF_{t}(x) + T_{x \text{'}} \quad (8)$$

$$N_{y} = RF_{t}(y) + T_{y \text{'}} \quad (9)$$

The value at the newly computed centre co-ordinates is incremented by 1 i.e. voting is done for center as (Nx,Ny).

![Voting array](image)

Fig. 4 voting array

The voting array is shown in figure 4. We can see that the dimensions of the voting array in the X and Y axis are same as the height and width of RF$_{t}$. The dimension of the voting array along the Z direction is the number of poses to be computed.

### 3.5 Max in the voting array

The voting array contains the centre of the detected object. This centre has the maximum value in the entire voting array. The location of the centre in the voting array will also give us the information about the scale and the angular rotation that the object is subjected to. In the CPU implementation the maximum is found sequentially. As seen in figure 4 the voting space is divided in planes which contain different poses. In the sequential approach the maximum value is calculated from each plane and compared with the next plane. We update the value of the max value if a value greater than the present value is found. In this manner the maximum from the voting array is found. Now the exact position of the centre found along the X and Y axis gives us the location of the object in the image and the position of the centre along the Z axis gives us the pose i.e. the scale and angle information.
4 NVIDIA CUDA

GPU (Graphics processing unit) were earlier used to get high quality graphics in gaming applications. In 2007, NVIDIA introduced a new architecture called CUDA [9] (Compute Unified Device Architecture) to enable GPU usage for general purpose computation. CUDA provided top level APIs to make program development for GPGPUs easier in order to utilize massive parallelization powers of GPU.

Fig 5 shows the hardware architecture of GPGPU [8]. The architecture consists of array of multiprocessor called symmetric multiprocessor (SM), each having its own shared memory and stream processors (SP). NVIDIA GTX 480 has 15 SM and each SM contains 32 SPs. The code is dispatched from CPU to the thread execution manager which schedules these threads to cores. Hence, user is freed from the burden of writing code and scheduling it for load balancing. Each core can run multiple threads at the same time and hence can produce exceptional speed up required for high performance computation. Every SM can access large chunk of memory called global memory. Global memory size is huge but its performance is low compared to shared memory. Hence the memory management affects the GPU implementation throughput heavily. An application written using CUDA can be seen as a host program which runs on CPU in addition to a 'kernel' which is run by multiple threads at the same time on different cores of GPGPU. Threads are executed in groups, which are called as 'blocks'. Grid is the collection of blocks to be executed on the device. The kernel code is executed by multiple threads at the same time.

CUDA architecture provides different types of memories like global, shared, texture and constant. Performance of the CUDA code largely depends on how well the architecture of the GPU has been exploited. Some of these techniques will be discussed in the implementation section.

5 Parallel GHT

This section will explain the parallel implementation of GHT on CUDA.

5.1 Sobel edge detection

Sobel edge detection algorithm is explained in III (1) and is straightforward to implement on CUDA. The gradient in x and y direction for each pixel is to be computed and each of the computation is independent of each other. The image is divided into blocks and grids. We choose a block size of 16x16. Using this block size we divided RF and TF into grids. The Sobel operation is completely data parallel provides high speed up. We have used the shared memory with overlapping blocks for fast implementation of Sobel edge detection. Because of the use of shared memory, fetching the data is fast. Gradient computation for each pixel is scheduled on a separate thread along the X and Y direction.

![Overlapping blocks for Sobel edge detection](image)

Figure 6 explains the concept of overlapping blocks. Figure 7 explains the application of overlapping blocks for Sobel edge detection. The data from each block is stored in shared memory. The hashed lines show the overlapping between blocks. The overlap between blocks is because to compute gradient of edge pixels of each block we need the information from the neighboring block.
5.2 Edge Compaction

As already discussed before, we have to operate only on the edge points from both the images. Thus we have to perform edge compaction to reduce the computational cost and also the execution time. The size of array required for both the images is determined by the number of edge points present in the images. This count is stored during previous stage while performing Sobel edge detection. Also we have block wise count of the number of edge points (Using reduction method). Thus using this information from each block we can fill in the edge information in to the array. We use single thread for each block to fill the array. Thus we release threads equal to the number of blocks in the image.

Figure 8 explains the edge compaction concept. Every thread requires an offset in the compaction in the compaction array to start filling information from. This offset is provided by the count of edge points in each block. Thus the edge information from each block is filled in parallel.

![Fig. 8. Edge compaction](image)

5.3 Pose Computation and Voting

Original GHT implementation [3] requires pose table creation and then voting on pose table. In order to optimize the performance, we have integrated these two steps into one. By combining these two steps, we save huge memory. We are not preparing pose table and storing it in an array. Instead, we are calculating pose table entry for each template edge point and its corresponding image point and vote based on the obtained value. Hence we do not need storage for pose table. Figure 9 shows the kernel configuration for this stage. We are proposing significant modification as compared to Luan implementation where each thread is assigned the task of pose table. Let IE be the edge point from the compacted array for RF and TE is the edge point from the compacted array for TP. As we see in figure 9 each thread takes value from the compacted array of RF and TP, computes the possible centre of the object and votes in the voting array. The concept of voting is explained in III (4). As there is a possibility of multiple threads of voting at the same location in the voting array, we make use of atomic operations. The voting is performed by taking into consideration all the pose orientations. The advantage of our implementation over the other is that we have computed 18000 poses. We have 180 degree rotation with a resolution of 2 degrees. This means it covers 360 degree rotation of the object. Also we have considered 100 scaling factors with a resolution of 0.01.

5.1 Maxima Calculation

The maxima calculation is done to know the position of the detected object and to know the pose the object is in i.e the angular rotation and the scale it is subjected to. The maxima calculation is performed using reduction technique. The voting array is a logical 3D array as shown in figure 4. Thus we find the max layer wise along the 3D voting array. Operating at a single layer, we find the max in each block. Then we store this block wise max in a temporary array. Now by using a single thread we find the max from the temporary array. This gives the max in a single layer. Using this value we update a global variable which would hold the max value in the entire voting array.

While finding block wise max, we store the max value in the first element of each block. Figure 10 shows the method of finding max from a single layer of the voting array. Once a max has been found from a single layer, we proceed in the similar manner for the remaining layers. As and when we find a value which is greater than the present max value, we update the global variable with it. In this manner we can efficiently find the max from the voting space. The final value of max gives us position of the required object, the angular orientation and the scaling to which it was subjected.

![Fig. 10. Maximum of single layer in voting array](image)
6 RESULTS

This section discusses the performance of our GHT implementation. The detected object for a template is shown with a red cross in the output image in Fig. 10- Fig. 12. We have used a system with Intel Core2Duo processor with 1 GB RAM as host and NVIDIA GTX-480 as device (GPU). Table 1 show that the three test cases have been tested by varying number of poses. In the case where 900 poses have been taken, we see that the minimum rotation resolution 2 degrees and scale resolution is 0.1. For test case 1, the actual rotation is 63 degrees. However the rotation detected by the algorithm is 64 degrees. Thus there is an error of 1 degree. Similarly the actual scaling of the image is 0.72. However, the scaling detected by the algorithm is 0.7. Thus the error is 0.02. Thus to minimize this error we need to increase the resolution for the poses. But, increasing number of poses increases the computational complexity and hence the detection will become slow. We have taken care of this problem by using a different parallelization approach. We have scheduled intersection of image point and template point to one of the thread. Computation of pose for this intersection point is done serial as the complexity of it is less. In the next stage, we have implemented the algorithm for angular resolution of 1 degree and scaling resolution of 0.1. Hence, we have taken care of the error in detecting angle accurately. As we see in the table 1, for test case 1 with 1800 poses, the actual rotation and the detected rotation is same. However, we still have an error in the scale. To remove this error we consider the third stage where we provide a resolution of 0.01 to the scale factor. As seen in the table for test case 1 with 18000 poses, the actual scaling and detected scaling is 0.72. Thus the error in scaling is minimized to approximately zero. We have also observed that the time of execution for a single test case almost remains constant for varying number of poses because of the reason explained earlier.

Fig 9. Pose creation and voting

<table>
<thead>
<tr>
<th>Test case no.</th>
<th>Image Size</th>
<th>No. of poses</th>
<th>No. of angles</th>
<th>Angle resolution</th>
<th>No. of scale factors</th>
<th>Scaling resolution</th>
<th>Actual rotation</th>
<th>Actual scaling</th>
<th>Detected rotation</th>
<th>Detected scaling</th>
<th>Execution time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>284x200</td>
<td>900</td>
<td>90</td>
<td>2</td>
<td>10</td>
<td>0.1</td>
<td>63</td>
<td>0.72</td>
<td>64</td>
<td>0.7</td>
<td>3010</td>
</tr>
<tr>
<td></td>
<td>1800</td>
<td>180</td>
<td>1</td>
<td>10</td>
<td>0.1</td>
<td>123</td>
<td>0.72</td>
<td>123</td>
<td>0.7</td>
<td>3016</td>
<td></td>
</tr>
<tr>
<td></td>
<td>18000</td>
<td>180</td>
<td>2</td>
<td>100</td>
<td>0.01</td>
<td>220</td>
<td>0.72</td>
<td>220</td>
<td>0.7</td>
<td>3020</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>300x300</td>
<td>900</td>
<td>90</td>
<td>2</td>
<td>10</td>
<td>0.1</td>
<td>110</td>
<td>0.85</td>
<td>110</td>
<td>0.8</td>
<td>910</td>
</tr>
<tr>
<td></td>
<td>1800</td>
<td>180</td>
<td>1</td>
<td>10</td>
<td>0.1</td>
<td>123</td>
<td>0.85</td>
<td>123</td>
<td>0.8</td>
<td>913</td>
<td></td>
</tr>
<tr>
<td></td>
<td>18000</td>
<td>180</td>
<td>2</td>
<td>100</td>
<td>0.01</td>
<td>220</td>
<td>0.85</td>
<td>220</td>
<td>0.85</td>
<td>914</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>256x256</td>
<td>900</td>
<td>90</td>
<td>2</td>
<td>10</td>
<td>0.1</td>
<td>126</td>
<td>0.96</td>
<td>126</td>
<td>0.9</td>
<td>630</td>
</tr>
<tr>
<td></td>
<td>1800</td>
<td>180</td>
<td>1</td>
<td>10</td>
<td>0.1</td>
<td>123</td>
<td>0.96</td>
<td>123</td>
<td>0.9</td>
<td>632</td>
<td></td>
</tr>
<tr>
<td></td>
<td>18000</td>
<td>180</td>
<td>2</td>
<td>100</td>
<td>0.01</td>
<td>220</td>
<td>0.96</td>
<td>220</td>
<td>0.96</td>
<td>636</td>
<td></td>
</tr>
</tbody>
</table>
We have performed object detection with 180 angular rotations in resolution of 2 degrees. So it would cover 360 degree range. Also we have computed 100 scale factors with a resolution of 0.01. Performance characterization of our algorithm for different image sizes and edge points is also discussed. Hence, GPGPU is going to be a promising platform for image processing application parallelization.

8 REFERENCES


Table 2 gives shows the performance of our GHT implementation for images with varying size and edge points. We see that execution time increases with the increase in number of edge points. Table 3 gives a comparison of our implementation against [6] that is so far the best implementation of GHT on CUDA available in the literature to the best of our knowledge. The performance achieved in [6] is significantly better than CPU implementations and one of the GPU implementations [2]. This performance of [6] is fast and comparable to our implementation but the performance degrades with increase in number of poses. Thus, to get real time performance, they have to compromise on the number of poses to be computed. However in our implementation time remains almost constant over the range of poses mentioned above, we get fast processing along with high resolution and accuracy.

7 CONCLUSION

We have discussed the GHT implementation on CUDA. We have proposed a new technique for parallelizing the algorithm. Due to this implementation, the execution time remains consistent for varying number of poses i.e. from 900 to 18000 poses. We have achieved a speed up of 80 times over the CPU version. As compared to [6] we have computed far more number of poses and achieved comparable time. However in [6], as the number of poses increases, the execution time also increases. We can achieve better accuracy by increasing number of poses. However, the limitation is that it takes a little more time to compute less number of poses as compared to [6]. However, the implementation is still fast and comparable for less number of poses. We have performed object detection with 180 angular rotations in resolution of 2 degrees. So it would cover 360 degree range. Also we have computed 100 scale factors with a resolution of 0.01. Performance characterization of our algorithm for different image sizes and edge points is also discussed. Hence, GPGPU is going to be a promising platform for image processing application parallelization.

Table 2. PERFORMANCE ANALYSIS OF GHT

<table>
<thead>
<tr>
<th>Sr no.</th>
<th>RF1 size</th>
<th>RF2(i,j)</th>
<th>TP3(i,j)</th>
<th>Time of execution (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>284x200</td>
<td>5521</td>
<td>511</td>
<td>3354</td>
</tr>
<tr>
<td>2</td>
<td>256x256</td>
<td>940</td>
<td>810</td>
<td>610</td>
</tr>
<tr>
<td>3</td>
<td>300x300</td>
<td>2222</td>
<td>416</td>
<td>920</td>
</tr>
<tr>
<td>4</td>
<td>300x300</td>
<td>2695</td>
<td>372</td>
<td>817</td>
</tr>
<tr>
<td>5</td>
<td>300x300</td>
<td>2720</td>
<td>1348</td>
<td>2897</td>
</tr>
</tbody>
</table>

Table 3. COMPARATIVE ANALYSIS OF GHT

<table>
<thead>
<tr>
<th>Sr no.</th>
<th>Method</th>
<th>Image size</th>
<th>Edge points</th>
<th>No. of poses</th>
<th>Execution time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Parallelization of the classic GHT on GPU</td>
<td>352x288</td>
<td>2684</td>
<td>990</td>
<td>818</td>
</tr>
<tr>
<td>2</td>
<td>A Fast and Accurate GHT Implementation on CUDA</td>
<td>300x300</td>
<td>2700</td>
<td>18000</td>
<td>920</td>
</tr>
</tbody>
</table>