Parallelization of the classic GHT on GPU

Juan Lucena[⋆]
José María González-Linares[⋆]
[⋆]Dept. of Computer Architecture
University of Málaga

Juan Gómez-Luna[◇]
Nicolás Guil[⋆]
[◇]Dept. of Computer Architecture and Electronics
University of Córdoba

Abstract

In this paper we present an implementation of the Generalized Hough Transform (GHT), a technique for object detection in images, on a Graphics Processing Unit (GPU). The GHT is a technique that requests a high computational cost in CPU, but it can be parallelized in GPU obtaining a high performance, by taking advantage of the full programmability offered by CUDA. To achieve this performance, we have carefully designed our algorithms to expose substantial fine-grained parallelism and decompose the computation into independent tasks that perform minimal global communication. Finally, we present an experimental evaluation that proves the effectiveness of our implementation.

1 Introduction

Object detection in images is a complex problem with high computational requirements. One of the most popular algorithms for detecting shapes was devised by Hough, which is suitable for parametric shapes. Ballard [1] generalized the Hough transform in order to detect arbitrary shapes, represented by a template.

The Generalized Hough Transform (GHT) is a widely-used technique, consisting in discretizing all possible transformations between object and image, and testing them individually. In this regard, to obtain a reliable detection, a vast amount of object features must be considered, resulting in a huge number of candidate poses, which should be evaluated. This exhaustive search entails high memory and computational requirements, which involve unacceptable execution times for many applications.

Since the GHT is composed by many similar operations, its performance is improvable by a parallel implementation. In this way, Strzodka et al. [7] parallelized the GHT on a Graphics Processing Unit (GPU). Their implementation used numerous pre-computed poses of the template object, which were stored in the texture memory of the GPU. Then, a convolution between the edge image, previously obtained by the Canny algorithm [3], and the pre-computed template object edges was performed on the GPU. This work is nowadays out of date, due

to the spectacular recent evolution of GPUs and the launch of NVIDIA CUDA. For this reason, we have developed a CUDA version of the GHT, in order to analyze the potential benefit of many-core processors for image applications. GPUs offer a large amount of resources at cheap price, that can be exploited in image and video industries.

We have tackled several challenges while parallelizing the GHT:

- Since edge points are a reduce set of pixels, a compaction should be performed, in order to avoid uncoalesced memory accesses and divergent execution paths.
- An efficient workload distribution is carried out in the evaluation of the poses. Each GPU thread compares a contour point of one pose with every contour points in the image.
- Reduction operation is carried for maximum search in voting space.

The rest of the paper is organized as follows. Section 2 explains the GHT and shows the computing stages it is compound of. Section 3 presents the main issues involved in the parallelization of the GHT stages in a GPU. Section 4 shows the experimental evaluation of our implementation with four videos belonging to MPEG-7 Content Set. Finally, the main conclusions are drawn.

2 GHT algorithm

The Hough Transform is a widely used technique to detect objects in images. This technique uses the edge points of the image in order to find specific parameterized curves. Each image edge point votes in a parameter space that works as an accumulator. The positions of the maxima in the parameter space yield the value of the parameter for the curves. Thus, straight lines, circles, ellipses or any other parameterized shape can be located using this approach.

When a non-parameterizable shape, also called arbitrary shape, needs to be detected, a modified approach, called GHT, has to be applied [1]. Now, the shape of the object to be detected needs to be known in advance. This shape is called the object template. Then, the GHT uses a representation of the template to find similar objects in new images that can have a pose (orientation, scaling and location) different from that of the template. The new template representation is a table, called Reference Table (RT), where each template edge point, t, is going to be represented by a reference vector, r_t. Each reference vector is a vector that joins a template edge point with an arbitrary reference point, O, defined in the template. In this paper, without loss of generality, O is located in the center of the template.

For the voting process a parameter space is required. During the voting process, the reference vectors of the RT are superimposed on each image edge point, i, and the value of the position of the parameter space aimed for each vector is increased by one. In order to detect object instances rotated or scaled

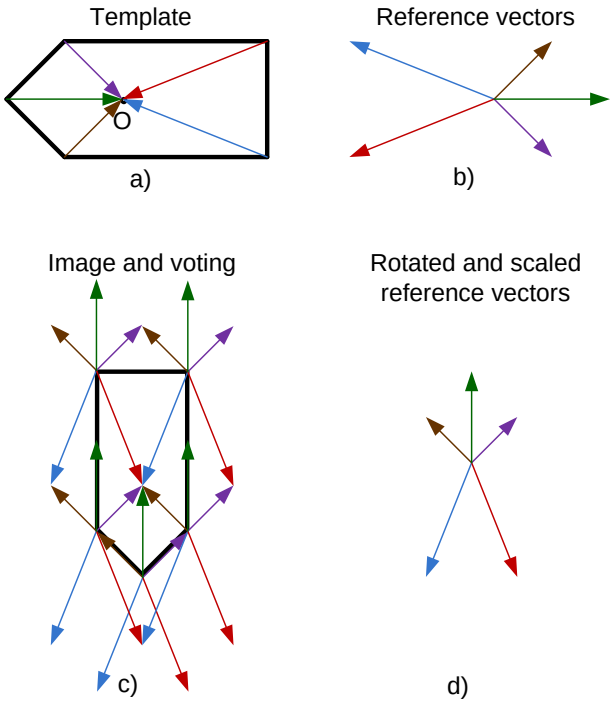

Figure 1: How GHT works. a) Template edges (black color) and calculation of the reference vector. b) Visual representation of the reference vectors as stored in the reference table. c) Image rotated and scaled with respect to the template with the d) rotated and scaled reference vectors superimposed. A maximum on the parameter space appears in the region pointed by the five reference vectors.

with respect to the template, the reference vectors of the RT are rotated (with respect to their origin) and scaled.

In Figure 1 an example of how GHT works is shown. For the sake of simplicity, in this example we have limited the number of edge points to those located at the polygon corners. In a real situation, all the edge points (contour points) collaborate to the object detection.

In GHT we can distinguish three stages, as shown in Figure 2, which are further described in the next sections.

2.1 Edge detection

Although edge detection is not really a part of the GHT algorithm, it must be implemented to extract the image and template edge points that GHT will use to detect objects. The detection of edge points is carried out by using the Canny edge detector [3]. This technique also computes the direction of the gradient vector associated to each edge point, which will be later used to save

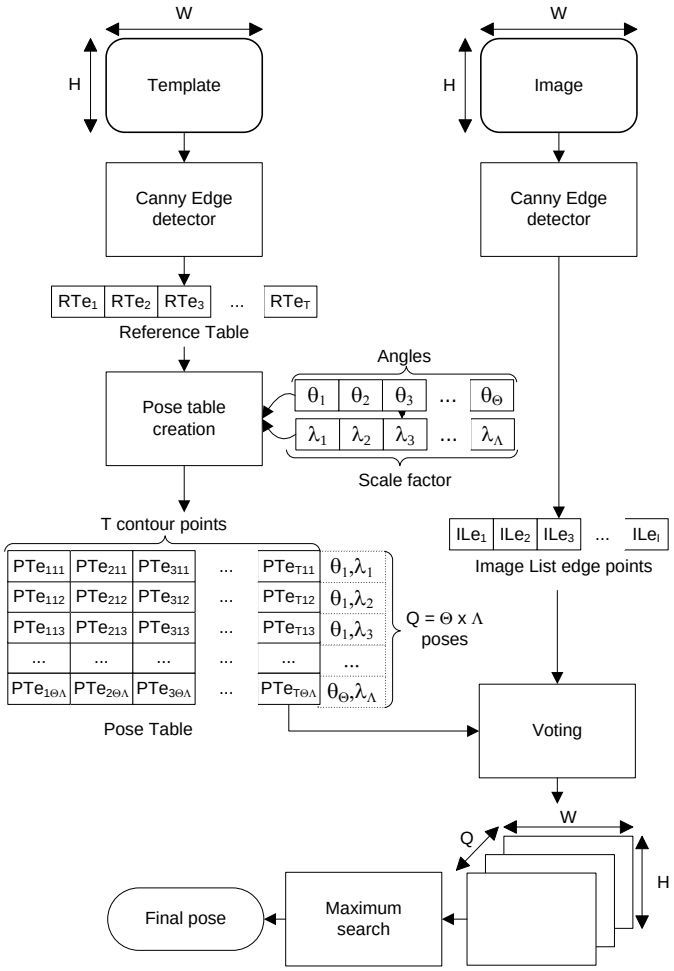

Figure 2: Workflow of the GHT technique.

computations. The computational complexity of this stage is given by $O(W \cdot H)$, where W and H are the width and height, in pixels, of the image, respectively.

After detecting the template edge points, r_t vectors are calculated and compacted in a reference table (RT). Each RT entry, RTe_t, stores r_t and the angle of the gradient vector associated to the t template edge point. Similarly, the image edge points are compacted forming a list, IL, with entries ILe_i containing the coordinates of the i-th image edge point, that is $\vec{i}$, and the angle of the gradient vector associated to this point.

2.2 Voting in the parameter space

This stage performs two main tasks: pose table creation and voting.

In the first task, each reference vector, r_t, of the RT is transformed according to the following expression:

$$\vec{r'_{tjk}} = \lambda_j * \vec{r_t} \cdot (cos\theta_k, sin\theta_k), \tag{1}$$

where λ_j is a possible value for the scaling and θ_k is a possible value for the rotation. The new $\vec{r'_{tjk}}$ vectors are stored in a pose table, $PoseT$, with T columns and $\Theta \cdot \Lambda$ rows, where T is the number of edge points in the template, Θ is the number of possible orientation values and Λ is the number of possible scaling values. The calculation of the $PoseT$ allows saving computations during the voting process due to data reuse. Algorithm 1 shows the pseudocode of this task.

Algorithm 1 Pose table creation

```
1: for each possible angle θ_k do
2:    for each possible scaleλ_j do
3:       for each reference vector r_t do
4:          PTe_ijk ← (r'_tjk, gradient(r_t) + θ_k)
5:       end for
6:    end for
7: end for
```

The second task of this stage carries out the voting in the parameter space. It takes the position of every image edge point, $\vec{i}$, from the compacted list of image edge points, LI, and performs the following calculation:

$$H(\vec{i} + \vec{r'_{tjk}}, \lambda_j, \theta_k) = H(\vec{i} + \vec{r'_{tjk}}, \lambda_j, \theta_k) + 1, \tag{2}$$

that is, all the reference vectors under a specific pose transformation are added to the position occupied by the image edge points. Then, the specific location of the parameter space given by the pose values and the new calculated vector is increased by one.

This task has a computational complexity given by the expression $O(T \cdot P \cdot \Theta \cdot \Lambda)$, where T and I are the number of edge points in the template and in the image, respectively, Θ is the number of possible orientation values and Λ is the number of possible scaling values.

2.3 Maximum search

If an object similar to the template object appears in the image the voting process will generate a clear maximum in the parameter space. The location of this maximum indicates the transformation (rotation, scaling and displacement) of the image object with respect to the template object.

As shown above, the GHT votes in a set of bidimensional parameter spaces. The number of parameter spaces is given by the number of possible values taken by the orientation and scaling. Typically, the maximum size of each parameter space is similar to the image size ($W \cdot H$) although it can be reduced to save memory at expenses of losing accuracy.

The number of rotations has to cover a range of 360°. The range for the scaling depends on the expected size of the objects to be detected. Typically, the values for orientation and scaling are discretized.

The final stage is in charge of searching maxima in the parameter space. Its complexity is given by the size of the parameter space, that is, $O(W \cdot H \cdot \Theta \cdot \Lambda)$.

3 GHT Parallelization

GHT exhibits a large amount of parallelism than can be exploited in many-core (GPU) platforms [7]. To achieve good occupancy in a GPU, thousands of threads needs to be active. This helps to hide long latencies in global memory accesses and keeps high the occupancy. Thus a finer computation granularity needs to be used on GPUs.

Next sections discuss the parallelization strategy we have employed in the three stages of the algorithm.

3.1 Edge detection

The GPU hardware allows for fast performance of pixelwise operations, threre for we assigned a single pixel of the image, and the corresponding computations, to each GPU thread. Many steps such as gradient finding, convolution, and non-maximum suppression can be performed in parallel on a pixel-wise level [4].

The RT and LI creation is a straightforward operation: r_i and p_i vectors calculated for the threads are concatenated. This operation is computed by many threads are available on GPU using a efficient compacting process [2].

3.2 Voting in the parameter space

In the parallelization of the pose table creation each GPU thread is in charge of computing a specific pose of all the possible poses ($\Theta \cdot \Lambda$) of a reference vector, that is, each thread calculates the value of a $PoseT$ cell.

Each thread calculates the vote generated by a specific image edge point position for a specific pose (λ_j, θ_k). This data distribution reduces the possible conflicts when accessing similar positions in the parameter space. In Figure 3 the computations carried out by each GPU thread are shown.

3.3 Maximum searching

Maxima search in the GPU is implemented by distributing the parameter space among the threads and applying a reduction operation explained in [5]. This

Task	Input	Output
Canny	$H \cdot W$	$H \cdot W$
RT Creation	$H \cdot W$	T
$PoseT$ Creation	T	$T \cdot \Theta \cdot \Lambda$
Voting	$P + T \cdot \Theta \cdot \Lambda$	$H \cdot W \cdot \Theta \cdot \Lambda$
Max. Search	$H \cdot W \cdot \Theta \cdot \Lambda$	1

Table 1: Number of elements at both the input and the output of each GHT task

reduction is used to remove unwanted elements from the output of a previous pass before sending it as input for the next pass.

3.4 Data transference

A determining factor which influences in the execution is the time involved in allocating the GPU device memory and the transference of data between the multicore and the GPU. Depending on the data size and the computation performed by each GHT task, those times may be significant. Table 1 shows the size of the input and output data for the different tasks in the GHT, expressed in number of elements.

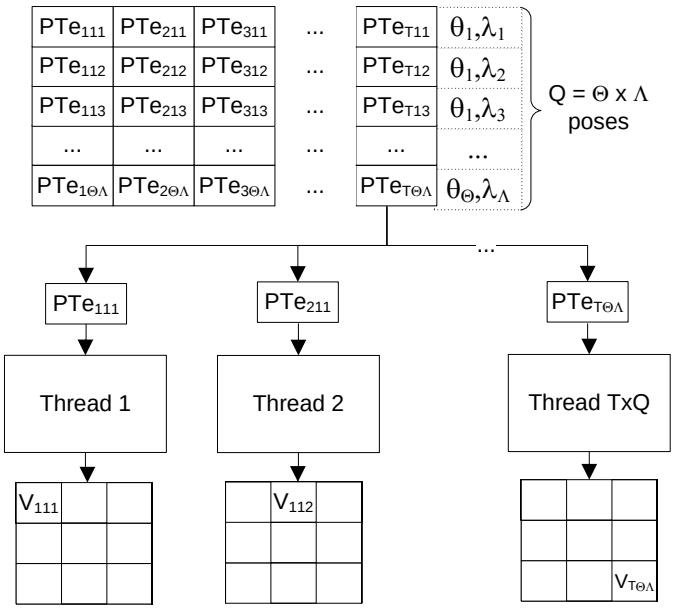

Figure 3: Example showing the computation of the voting stage carried out in the GPU architecture, when the number of available threads is much higher than the number of poses: each thread only computes a pose for an image point.

4 Experimental Evaluation

With the aim to carry out an exhaustive evaluation of the parallelized GHT technique, we have used four benchmark videos containing 4000 frames each one. The GHT is tested by implementing a application that detects edition effects in videos[6]. In this application, the detection is performed along all the video frames in a sequential order. Thus, when frame f is being evaluated, the frames f and $f+1$ take the role of template and image, respectively. However, due that the number of frames in a video can be very large, the overall computation time makes prohibitive the use of the GHT in a non-parallel architecture.

In this section we show how this algorithm can be used for video processing in a Intel Core 2 Quad @ 2.4 GHz processor and a NVIDIA GeForce GTX 280 GPU. CUDA is the programming model and the compiler employed for the GPU is nvcc. We use four benchmark videos from the MPEG-7 Content Set, whose characteristics are shown in Table 2. The size of each video frame is 352x288 pixels. Tests have considered 11 scale values (Λ) and 90 orientation angles (Θ). Therefore, 990 different poses are computed.

In Figure 4 the average execution time in milliseconds of GHT on the CPU and GPU for the benchmark videos are shown. As it can be observed, the execution time is related to the number of contour points. Compared to the CPU implementation, we observe an almost constant speedup of about 44x along the benchmark videos.

5 Related work

As we explained at the beginning of the document, Strzodka et al. [7] parallelized the GHT on an old NVIDIA GeForce 4 GPU. At that time, GPUs were specifically designed for graphics, so that they should suit texture operations to general purpose computations, in order to implement the convolution between

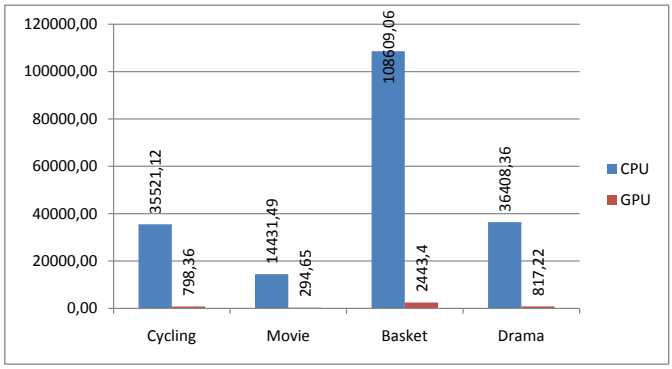

Figure 4: Average execution time in milliseconds of GHT on the CPU and GPU for four videos of 4000 frames.

Video	Description	Edge points
Cycling	A cyclist and people around him	2778
Movie	Beginning of a movie	1436
Basket	Basketball game	5061
Drama	A situation comedy	2684

Table 2: Test workloads characteristics. Videos have a resolution of 352x288 pixels. Number of edge points is the average value. Each video is consisting in 4000 frames

image edges and object contours. Their implementation performed an exhaustive search among many thousands of object poses and different object sizes in less than one minute. This represented a speedup of about 10 with respect to the original CPU implementation.

6 Conclusions

This work has presented the parallelization of the classic GHT on GPU. Since this algorithm presents a high computational cost, it takes advantage of the vast amount of resources that GPUs offer. The implementation exposes substantial fine-grained parallelism and decomposes the computation into independent tasks that perform minimal global communication.

An experimental evaluation has been presented in order to prove the effectiveness of our implementation. Our CUDA implementation has obtained an impressive speedup of about 44x, with respect to the CPU version. These results prove that the huge number of computing units of the GPUs can be efficiently applied to image processing.

References

[1] D.H. Ballard. Generalizing the Hough transform to detect arbitrary shapes. *Pattern Recognition*, 13(2):111–122, 1981.

[2] Markus Billeter, Ola Olsson, and Ulf Assarsson. Efficient stream compaction on wide simd many-core architectures. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009*, pages 159–166, New York, NY, USA, 2009. ACM.

[3] J. Canny. A computational approach to edge detection. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 8(6):679–698, 1986.

[4] Juan Gómez-Luna, José María González-Linares, José Ignacio Benavides, and Nicolás Guil. Parallelization of a video segmentation algorithm on CUDA–enabled graphics processing units. In *Proc. of the Int'l Euro-Par Conference on Parallel Processing (EuroPar'09)*, pages 924–935, 2009.

[5] D. Roger, U. Assarsson, and N. Holzschuch. Efficient stream reduction on the GPU. In David Kaeli and Miriam Leeser, editors, *Workshop on General Purpose Processing on Graphics Processing Units*, October 2007.

[6] E. Saez, J.I. Benavides, and N. Guil. Combining luminance and edge based metrics for robust temporal video segmentation. In *Proc. on the Int'l. Conf. on Image Processing (ICIP'04)*, volume 4, pages 2231–2234, October 2004.

[7] Robert Strzodka, Ivo Ihrke, and Marcus Magnor. A graphics hardware implementation of the Generalized Hough Transform for fast object recognition, scale, and 3D pose detection. In *Proc. of IEEE Int'l. Conf. on Image Analysis and Processing (ICIAP'03)*, pages 188–193, September 2003.