Generalized Hough Transform on GPU

Yijie Chen, Yiliu Xu
Carnegie Mellon University
yijieche@andrew.cmu.edu, yiliux@andrew.cmu.edu

1. Summary

In this project, we implemented one sequential and multiple parallel versions of Generalized Hough Detector. Three steps of the pipeline: image processing, R Table building, and accumulation are parallelized on CUDA separately. Running the parallel code on GHC machine (Intel i7-9700 CPU, RTX 2080 GPU) achieves a max overall speedup of 433x.

2. Background

2.1. Generalized Hough Transform

The Hough Transform is an image processing technique for curve detection that takes advantage of the duality between points on a curve and parameters of that curve. This method can be used to detect parametric shapes in grey level images, specifically lines and circles.

However, the classic Hough Transform algorithm is not applicable to the detection of non-analytic curves. The Generalized Hough Transform (GHT), proposed by Dana H. Ballard in 1981[1], is the modification of the Hough Transform based on the idea of template matching. By constructing the mapping of the edges between image space and Hough Transform space, GHT can be exploited to detect instances of an arbitrary template shape, even ones that cannot be described by equations. In addition, variations in the shape such as rotations, scale changes can also be detected using this mapping. Therefore, an efficient generalized Hough detector is valuable in a wide range of areas, such as medical image processing, and robot vision.

2.2. Sequential GHT Algorithm

The pipeline of implementing a Generalized Hough Detector is as follows (Figure 1):

- Apply derivative filter to both the template image and source image to detect all edge pixels
- Encode all edge points from the template shape with R-table representation
- Apply Generalized Hough Transform on the source image, accumulate the vote of each edge pixel into a 4D accumulator matrix (x_c, y_c, s, θ)
- Traverse the accumulator matrix to find the shape centers with high votes

Below we go through the key steps in implementing the sequential GHT algorithm.

2.2.1 Image Processing

The Image Processing step takes in an RGB image and outputs a binary image, with '1' indicating an edge pixel. We first convert the RGB image into a gray image by averaging the 3 color channels. Then we convolve the image with 3x3 Sobel filters in X and Y directions, which outputs two gradient images. Then we compute the magnitude and orientation for each pixel using its X and Y gradient values. In order to thin out the edges and increase the performance of the sequential code, we apply non-maximum suppression, which filters out the edge pixel with a small magnitude value if it has a neighbor pixel with a higher magnitude. Lastly, the result is fed into a threshold filter which outputs a binary image.

In the sequential implementation, we perform the image processing steps in order. For each step, we loop over the image and write the result to a new blank image. Therefore, this step by nature benefits from GPU implementation, as it computes value of each pixel independently which can be done in parallel. One thing to pay attention to is the memory allocation strategy. Instead of allocating a new memory block in every sub-step, we hope to reuse and overwrite existing memory as much as possible to further gain speedup.

2.2.2 R Table Computing

R Table is a data structure which encodes edge pixels by their orientation values. Edge pixels with the same orientation ϕ will be grouped into one R Table slice. Each edge pixel is converted to an R Table entry (r, α), where r is the length of vector pointing from the pixel to the image center, and α is the orientation of this vector. Therefore, each R

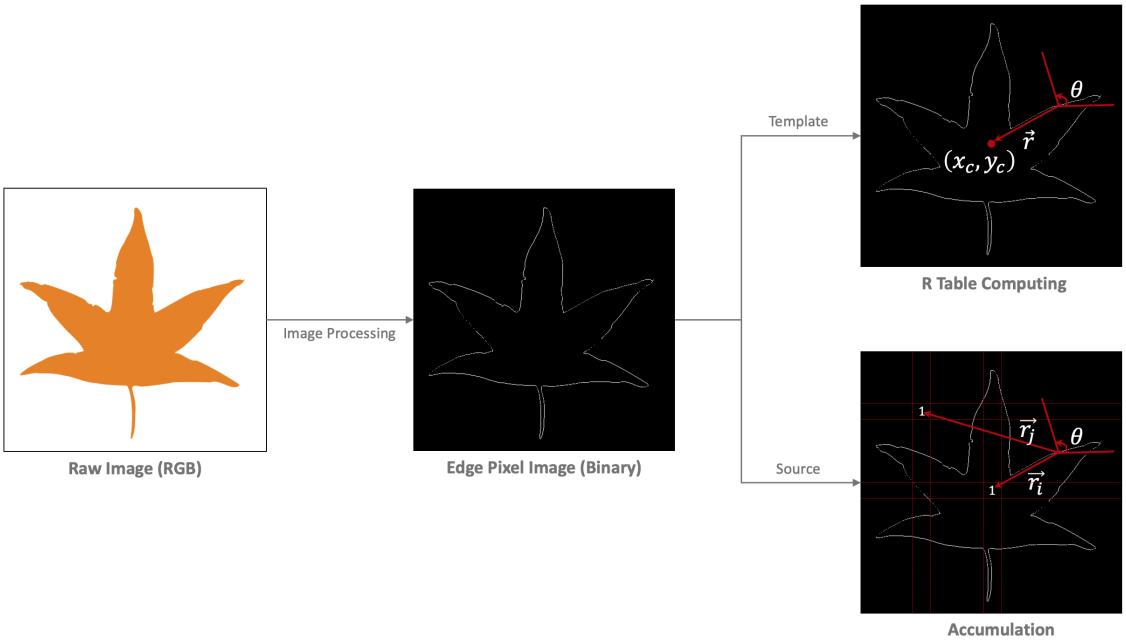

Figure 1. Algorithm Flow Chart with Sample Images

Table slice may contain multiple entries; each of them indicate a vector from the current pixel to the image center. When looking up the R Table, the user provides a ϕ value, and obtains all the entries with the orientation ϕ.

In the sequential implementation, the R Table is stored as a 2D vector. We loop over the 'orientation' image. If the current pixel belongs to an edge, we index into the R Table slice by the pixel's orientation ϕ, and append the corresponding R Table entry. This step is also $O(N)$ time complexity therefore is expected to benefit from parallelization. However, one challenge for this step is each R Table slice is a dynamic array whose size is unknown at compile time. Therefore, we need to carefully design how to represent the R Table in GPU memory.

2.2.3 Accumulation & Finding Candidates

Now this step is the actual step that will find the locations of the template shape. We introduce this algorithm starting from 2D accumulation.

If we don't consider the variation of scale and rotation, we will establish a 2D accumulator (x_c, y_c). The main concept is to examine and accumulate votes from every edge pixel in the source image. Specifically, for each edge pixel, we look up its orientation value ϕ_i in the R Table which leads us to all the (r, α) pairs. Each (r, α) will compute a 'candidate center point' and will contribute to one 'vote' to that candidate center point in accumulator matrix. The candidate center point is easily solved using following equations since we have the edge pixel coordinate (x, y) and a vector length, direction pair (r, α).

$$x_c = x + r * cos(\alpha)$$
$$y_c = y + r * sin(\alpha)$$

Finally, we search in the accumulator to find which points receive the most votes; they are most likely to be the centers of the shape to be detected.

Taking scale and rotation into consideration, we will instead establish a 4D accumulator (x_c, y_c, s, θ) which means that for each pixel, we should calculate all possible (x_c, y_c) using all possible values of scaling size and rotation angles. Besides, when looking up in R table, for each rotation angle θ, we need to look up new (r', α') pairs using new orientation value $\phi - \theta$. Here are the equations for calculating 'candidate center points':

$$x_c = x + r' * s * cos(\alpha' + \theta)$$
$$y_c = y + r' * s * sin(\alpha' + \theta)$$

In practice, we made some modifications to above algorithm. We realize that pixel-level voting creates clustered candidate points instead of ONE peak value, making it difficult to filter out the real center points. Therefore, we group nearby pixels into blocks (one block contains 10x10 pixels), and construct block accumulator. For each calculated pixel (x_c, y_c), we only vote on the block it belongs to. Experiments show that this block-level voting provides good enough accuracy.

In addition, instead of finding candidates after accumulation, we construct an array to record the point (x_c, y_c, s, θ)

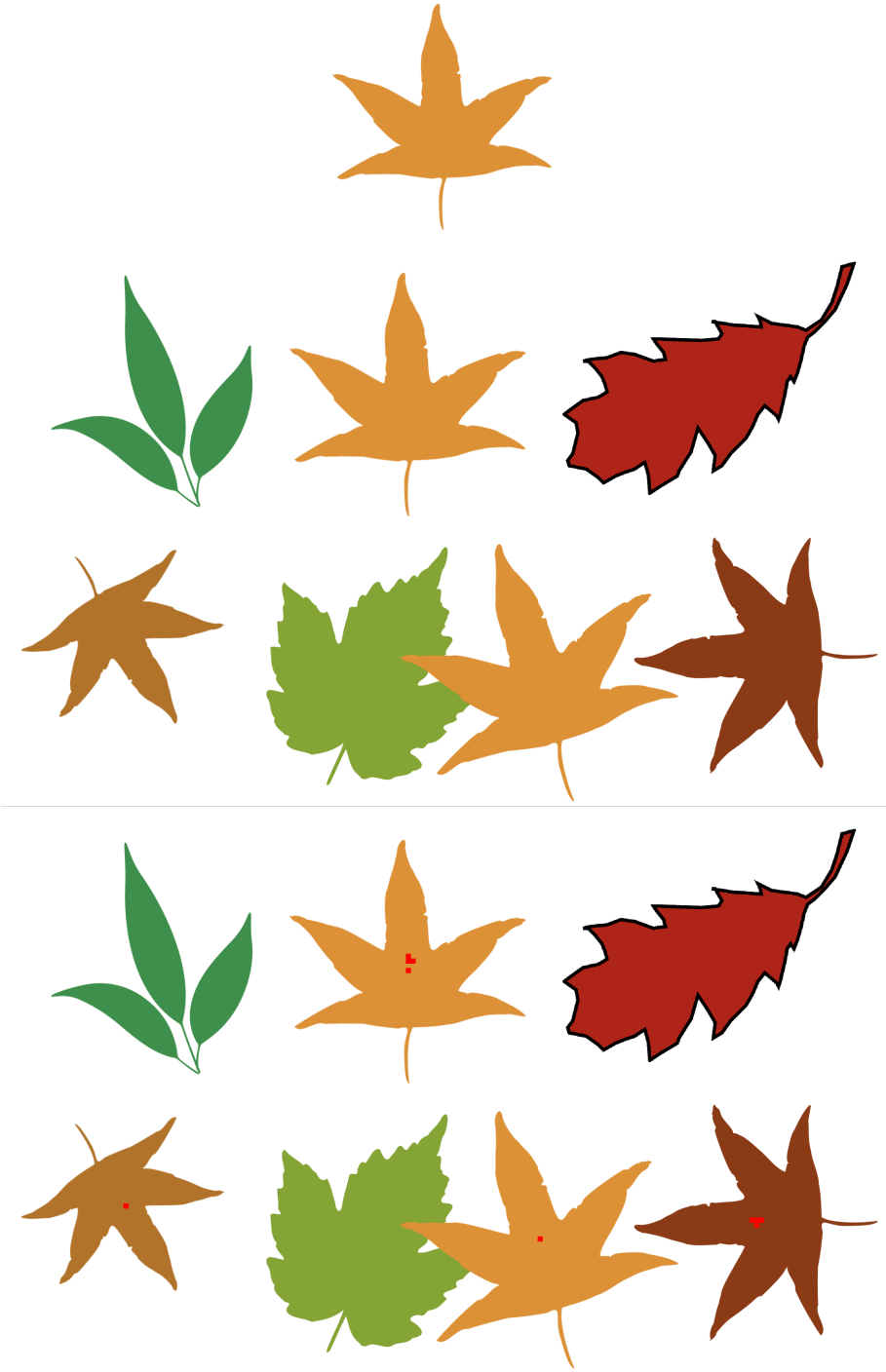

Figure 2. Input and Output of GHT algorithm (top: template image; middle: source image; bottom: output image).

with highest votes in every block during the process of accumulation. Finally, we traverse this array, those points with votes higher than a specific threshold would be counted as the candidate center points. The detailed algorithm of the accumulation and finding candidates is manifested in Algorithm 1.

The Accumulation step is expected to benefit the most from CUDA parallelization. The sequential approach runs deeply nested for-loops. When deploying the code to GPU, we will collect vote from each pixel in a parallel manner.

Algorithm 1 Calculating Accumulator Loop

for pixel (x, y) in image **do**
  if (x, y) is an edge pixel **then**
    ϕ = orientation(x, y)
    for θ in $[0, 2\pi)$ **do**
      entries = RTable$(\phi - \theta)$
      for entry in entries **do**
        r = entry.r, α = entry.α
        for s in [minScale, maxScale] **do**
          $x_c = x + r * s * cos(\alpha + \theta)$
          $y_c = y + r * s * sin(\alpha + \theta)$
          $b_x = x_c$/blockSize
          $b_y = y_c$/blockSize
          temp = ++accumulator(s, θ, b_x, b_y)
          if temp $>$ blockMaxima(b_x, b_y).hits **then**
            blockMaxima(b_x, b_y) = newPoint
          end if
          if temp $>$ maximum **then**
            maximum = temp
          end if
        end for
      end for
    end for
  end if
end for
maximaThres = maximum $*$ thresRatio
for block (i, j) in blocks **do**
  if blockMaxima(i, j).hits $>$ maximaThres **then**
    hitPoints.add(blockMaxima(i, j))
  end if
end for

2.3. Sample Input & Output

Figure 2 shows sample input and output images from the sequential program. All possible positions for the shape image are marked in red. Note that the algorithm is able to accurately detect scaled and rotated shapes.

3. Approach

The sequential version of GHT is implemented in pure C++. The parallel version is implemented using CUDA and Thrust library. Below are detailed discussions on the parallel strategies of each step in the GHT pipeline.

3.1. Image Processing

As mentioned above, in image processing phase, we used multiple functions to extract the boundaries of the template image and source image. Since the main implementation logic of these processing functions is to use two loops to traverse all pixels of the entire image, we adopted ONE PIXEL, ONE THREAD parallel approach for deployment on CUDA.

As shown in Figure 3, we have following functions to parallel: `convertToGray`, `convolve`, `magnitude`, `orientation`, `edgenms`, `threashold`, `createRTable`. To decrease the global memeory accesses on GPU, instead of writing kernel functions for each function, we implemented the parallelization of these processings in three kernel functions according to when synchronization is needed.

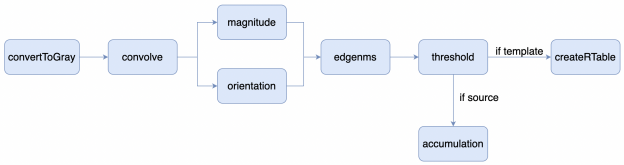

Figure 3. Image Processing Pipeline: Arrows represent dependencies between two steps

3.1.1 Synchronization

In general, when a step's processing requires not only current pixel's data, but also neighbor's pixels' data, it needs synchronization of ALL threads (not only threads in one block) before entering into this step.

Here we have two steps requiring neighbor's data: `convolve`, `edgenms`. In addition, before processing R table or calculating accumulator, synchronization is also necessary. As a result, the concrete timing of synchronization is shown in Figure 4. Overall, we could combine these steps in three kernel functions, first one realizes `convertToGray`, the second one calls `convolve`, `magnitude`, `orientation`, the third one implements `edgenms`, `threashold`, `createRTable`.

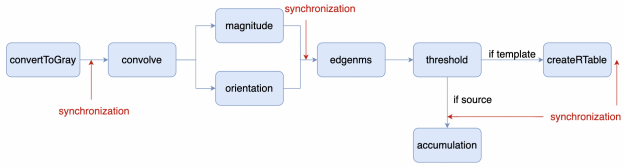

Figure 4. Image Processing Pipeline with Synchronization

3.1.2 Memory Access

In sequential algorithm, we could create a new gray image to store gray scale data after each step, however, it's highly expensive to do so on CUDA not only because of the overhead of allocating and freeing memory on GPU, but also the latency to access global memory on GPU. Since we have already combine some steps in one kernel functions, within one kernel function, we could leverage shared memory to store data if two steps have dependencies.

For example, steps `magnitude, orientation` need gradient data processed after `convolve`, and they don't require neighbor's pixel's data, so we used shared memory to store gradient data. In this way, the data `magnitude, orientation` access are not in global memory, which can decrease some latency due to global memory accesses. Similarly, the data processed after `edgenms` and `threshold` are also stored in shared memory to facilitate the following step's memory accesses. Figure 5 illustrates the details of shared memory use.

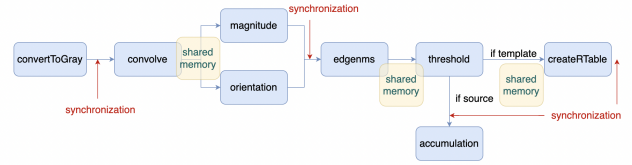

Figure 5. Image Processing Pipeline with Synchronization

3.2. R Table Processing

In `createRTable`, we record R entry (r, α) for each edge pixel. As CUDA is bad at dealing with dynamic-size array and concurrent 'append' operations, we instead add a processing step, storing all R entries in a global array `entries` on GPU which has the same size as the template image.

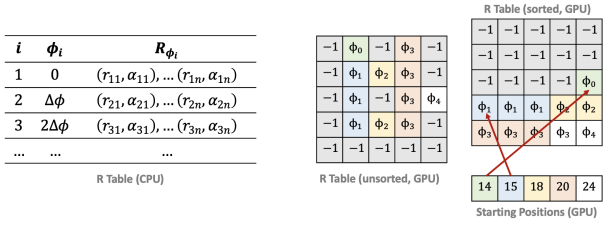

Figure 6. R Table Representations on CPU and GPU

As this R Table (`entries`) is out of order and contains a lot of non edge pixels, we use the *sort* function from Thrust library to sort these R entries based on each entry's orientation ϕ. Besides, in order to facilitate accesses to this array in the Accumulation step, we leverage another Thrust function *lowerBound* to perform lower bound binary searches for all orientation values ϕ, and record the starting position of each ϕ in `entries`. In this way, accumulation kernels can take advantage of starting positions to locate all R entries of a specific ϕ in R table. Figure 6 compares the R Table representations in CPU and GPU implementations.

3.3. Accumulation

The original serial accumulating algorithm is broken down into 3 phases: (1) *Filtering Edge Pixels*, (2) *Writing to Accumulator*, and (3) *Finding Candidates*. While (1) and (3) were implemented in a straightforward manner, a good amount of effort was devoted to improving the performance of phase (2).

3.3.1 Filtering Edge Pixels

The sequential algorithm first loops over all pixels, and starts accumulating only if the pixel is an edge pixel. We choose to add a pre-processing step to first filter out all the edge pixels. This ensures that all pixels processed by the GPU are edge pixels, thus reducing the divergent execution within a warp.

We leverage the *copy_if* function from the Thrust library, which works well as a stable and efficient filter on GPU. The input is a binary image, and output is a list of indices of the edge pixels. Both the input and output are stored in GPU memory.

3.3.2 Writing to Accumulator

For the step *Writing to Accumulator*, we experimented different parallel strategies to improve the performance. Below are 3 orthogonal directions we explored. By combining them, we generated a good amount of different parallel implementations, the results of which will be discussed in Section 4.

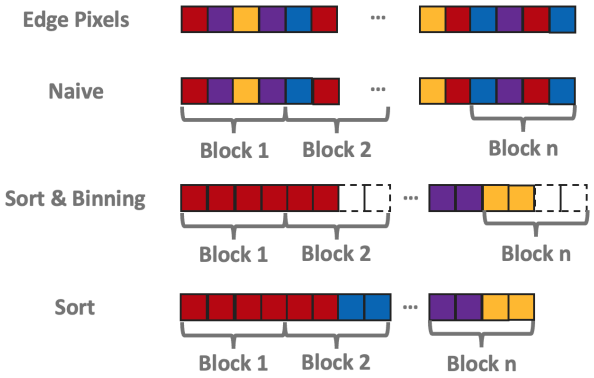

Figure 7. Strategies to Map Edge Pixels (Each color represents a unique edge pixel orientation; number of pixels per block is only for illustration purposes)

- **Mapping Edge Pixels to Blocks**
  The first, naïve approach we took is to directly map the edge pixel list to CUDA blocks (see Figure 7). Each block contains 32 threads; each thread performs the computation of one edge pixel, to find its corresponding orientation, loop through possible θ, and look up entries in the RTable using its orientation and θ value.

  Although this method is easy to implement, it suffers heavily from divergent execution flow. As the list of

edge pixels are filtered directly from the image array, the pixels within the same block are likely to have diverse orientations, making the warp SIMD execution inefficient.

To enforce coherent instruction stream, we added some pre-processing steps before running the accumulating kernel. We first implemented a binning approach, which involves sorting the edge pixel list by orientation, and assigning a subrange of edge pixels to a thread block. Each block still contains 32 threads, but only a subrange of them get assigned with an edge pixel and perform the computation.

This approach ensures all the edge pixels within a block share the same orientation, and therefore will always share the same execution flow. However, one drawback of this approach is that there are blocks handling the leftovers of each orientation bin, which means there are idle ALUs not being used for computation.

Realizing the flaw of the binning approach, we experimented with the second pre-processing approach, which sorts the edge pixels and then map them to each thread block, without performing the binning. This approach has higher ALU utilization, as only the last block will handle the leftover pixels. However, some warps will have divergent execution if they handles edge pixels with two or more orientations.

- **Kernel Dimension**
  Apart from parallelizing edge pixels (the outer loop), the sequential algorithm also loops over the θ and scale s. There's opportunity to launch a 3D kernel to accelerate the nested for-loops.

  Some slight modification is required to extend the CUDA kernel to 3D. In particular, the for-loop for scale s is moved one level out, making traversing all the entries the innermost loop. This makes it natural to add two more dimensions to the existing 1D kernel. The new grid dimension is <number of blocks for edge pixel $\times$ number of rotation slices $\times$ number of scale slices>. Each thread now handles one edge pixel at one θ and one s value.

- **Leveraging Shared Memory**
  When voting in the 4D accumulator, there are design alternatives regarding memory access pattern. One straightforward way is to always write to the global memory. *AtomicAdd* is used to ensure the correctness under concurrent write.

  As writing to global memory has high latency, we also experimented with a shared memory version built upon the 3D kernel implementation. With the assigned θ and s value, each block only writes to a slice (2D) of the accumulator. This inspires us to create a shared copy of a 2D accumulator slice. Each thread first votes to the shared slice, and eventually the 2D slice will be written back to the global memory.

3.3.3 Finding Candidates

The parallel implementation is directly translated from the sequential algorithm. Firstly, a CUDA kernel is launched to reduce the 4D accumulator to 2D HitPoint array, each element is a max hit value at that image position along with the corresponding θ and s. This kernel maps each image position to a thread, and each thread loops over the 2D (θ and s) dimension to find the max hit value. Then the *max_element* and *copy_if* functions from the Thrust library are used to find the max value in the block maxima array, and to filter out the candidate center points with high hit values.

4. Results and Analysis

We benchmark the parallel version using the speedup with respect to the single-threaded sequential implementation on CPU. Specifically, each major step of the algorithm is timed and compared with the corresponding sequential program.

Test images are generated manually by stitching different shapes and converting to PPM format images. Two sets of test images are used:

- **Image size**
  A 4K-resolution test source image, down-sampled to 2K, 1080p, 720p, 360p. The corresponding template shape is also down-sampled.

- **Percentage of Edge Pixels**
  Six 4K-resolution test images with increasing shape density. The corresponding template shape remains the same.

To record time, we manually run the code on each test image.

4.1. Image Processing

Here we measure the performance for different test source images and analyze based on image size and the percentage of edge pixels.

Figure 8 presents the relationship between image size and the speedup based on the source image processing. Note that all source images' sizes are large, so this performance can achieve about 40$\sim$50x speedup. However, as image size gets larger, this speedup decreases and has a tendency to maintain at around 40x. This is because as image size grows up, the overhead for allocating and freeing memory on CUDA is much larger. Figure 9 verifies the above

hypothesis, this overhead takes up much of the time for image processing in parallel version. While on the other hand, there is far less time spending on memory allocation and free in sequential version, as a result, the speedup decreases with the increase of image size.

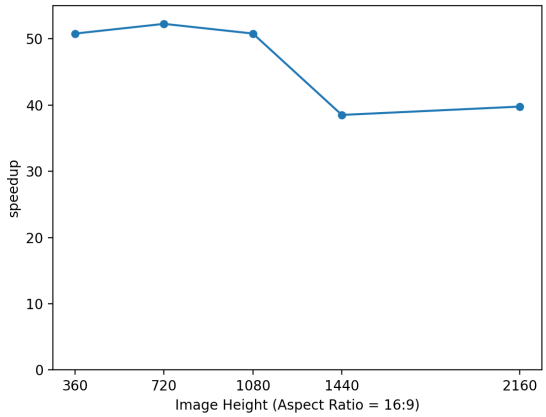

Figure 8. Speedup for image processing w.r.t image size

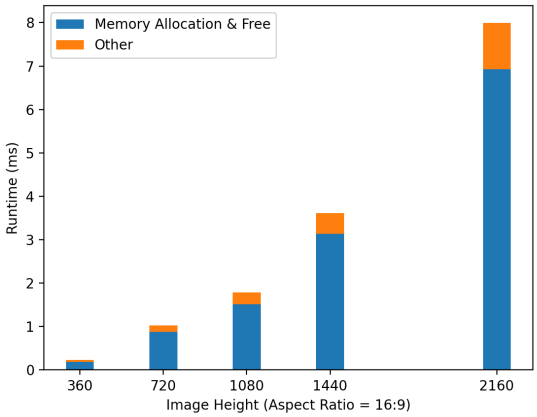

Figure 9. Breakdown runtime for image processing w.r.t image size

When considering percentage of edge pixels, here Figure 10 shows the tendency of speedup with increasing percentage of edge pixels. It turns out that there's not much relationship between two, the speedup for image processing maintains stable with the increase of edge pixels. This is reasonable since no matter how many edge pixels in an image, this algorithm always process all pixels, and the processing time for edge pixel and non-edge pixel are the same. Therefore, this curve can basically remain unchanged.

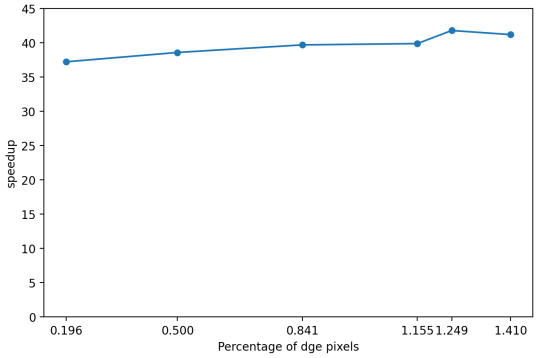

Figure 10. Speedup for image processing w.r.t percentage of edge pixels

4.2. R Table Processing

In parallel version, except for creating R table, we add a new step processing R table to facilitate the following accumulation step. In sequential version this step does not exist, so we only record R table processing runtime, as shown in Figure 11.

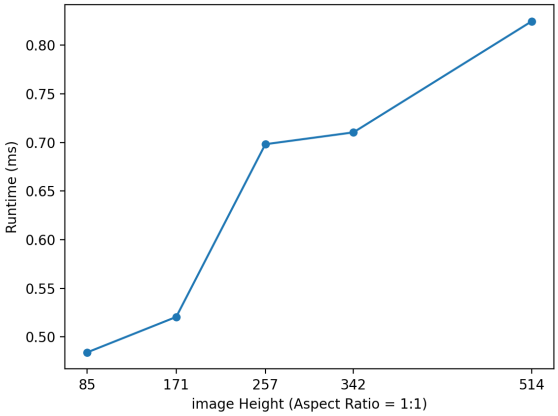

Figure 11. Runtime for handling R table w.r.t image size

As described in above section, in R table processing phase, we first sort for an array of length of n and then perform M times binary search, where n is the image size (the number of pixels) and M is a constant which represents there are how many rotation angles in R Table. Therefore, it's intuitive that as image size grows up, runtime for processing R Table grows up. Figure 11 basically meets this increasing tendency.

In order to have a more intuitive understanding of runtime for R Table processing, we counted the percentage of this step in the whole process of processing template, as shown in Figure 12. It turns out that this step takes longer time than the sum of all other steps in processing

template. And with image size getting larger, this percentage decreases slightly, which might be caused by the larger overhead of allocating memory.

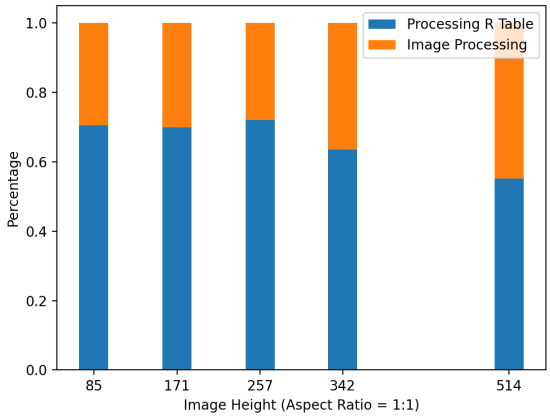

Figure 12. Percentage for handling R table during processing template

4.3. Accumulation

In total 6 versions of parallel code are used for benchmarking, which is a combination of 3 edge pixel mapping strategies (naïve, binning, sort) with 2 kernel dimensions (1D, 3D). The shared memory strategy will be discussed in Section 4.3.1.

Figure 13 measures the performance of 6 algorithms with different images resolutions. The 3D kernel implementations keep boosting performance as the image size grows, with the sort-3D version gives a best speedup of over 500x on a 4K image; the 1D kernel instead shows a speedup decrease with larger-size inputs. A possible reason is: for a 1D kernel, each block will always compute a deeply nested for-loop. As the image size grows, there are more entries stored in the R Table, making the block computation heavier. Although it's the same case for a 3D kernel, the average amount of computation increased for each CUDA block is bounded, as each block only loops over a particular slice of R Table entries.

Regarding the edge pixel mapping strategies, sorting consistently gives the best performance among the three. Although the binning approach ensures zero divergence within a block, it does not outperform the simple sorting algorithm.

In order to explain this behavior, we record the runtime for two critical steps in the accumulation function in Figure 14, which compares the sort-3D version with the binning-3D version. It turns out that the accumulator kernels of the two share very similar performance. However, the extra binning step in binning-3D takes almost the same amount of time as the accumulator kernel. This pre-processing step

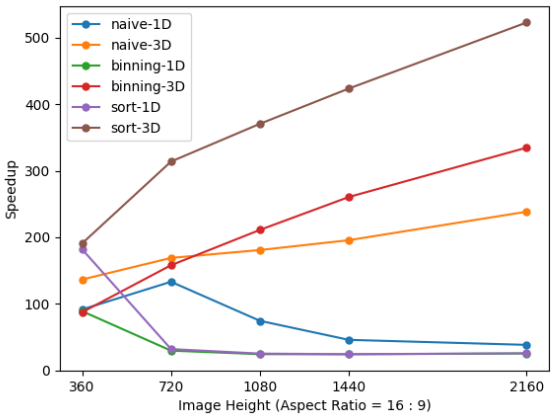

Figure 13. Speedup for accumulation w.r.t image size

reduces the performance of the binning-3D implementation approximately by half.

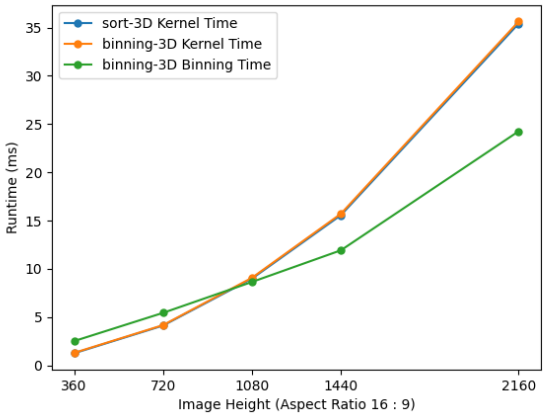

Figure 14. Runtime of critical step(s) in Accumulation

Another hypothesis for the poor performance of binning is that it needs more blocks to handle the leftover edge pixels belonging to each bin. To verify this, Table 1 lists the CUDA block count for each approach with increasing image sizes. It turns out that the difference in the CUDA block count is bounded, and thus is not a major reason for the performance gap. In fact, it's easy to prove that the maximum difference of CUDA blocks count is bounded by the number of distinct pixel orientation groups.

	360p	720p	1080p	2K	4K
Sort	313	675	1067	1439	2196
Binning	349	709	1098	1474	2229
Δ	36	34	31	35	33

Table 1. Comparison of block numbers of sort-3D and binning-3D

In addition, Figure 15 measures the speedup with different edge pixel percentage values. A higher edge pixel percentage indicates there are more shapes in the image. This shows a similar trend as Figure 13, with 3D kernels outperforming 1D versions, and the sort-3D strategy showing the best performance. Note that the percentage of edge pixels is overall small because of the non-maximum suppression step. All methods converge when more than 1.155% of the pixels are edge pixels.

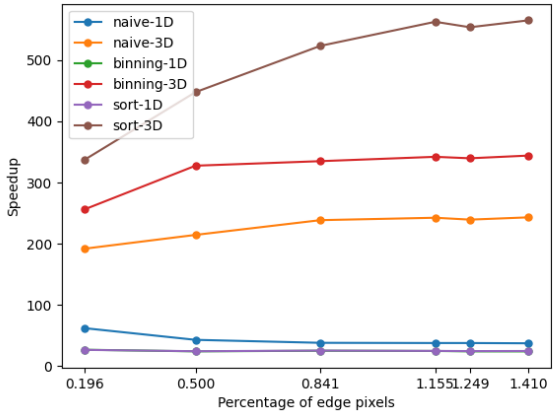

Figure 15. Speedup for accumulation w.r.t percentage of edge pixels

4.3.1 Speedup using Shared Memory

Lastly we discuss the performance using shared memory. One issue with this approach is the CUDA shared memory for each block is limited (48KB). As each 2D slice of the 4D accumulator is essentially a down-sampled image, the size of shared memory required by this algorithm is proportional to the image size. Therefore, the shared memory version is not scalable to a large workload.

Below we compare the performance of the shared-memory, binning-3D version with the global-memory binning-3D version. The template image remains the same, which guarantees the R Table always have the same size. The original source image is 1280x720 which requires 37KB shared memory on each thread block. It gets cropped to smaller size with 16 : 9 aspect ratio: 1024x576, 960x540, 854x480, and 640x360.

Figure 16 compares the accumulation kernel run time (binning excluded) of the global memory and shared memory version. Different block sizes are tested and charted. As each block contains more threads, the performance of the shared-memory implementation gets improved and closer to the global-memory version. However, it does not work as expected to further accelerate the program.

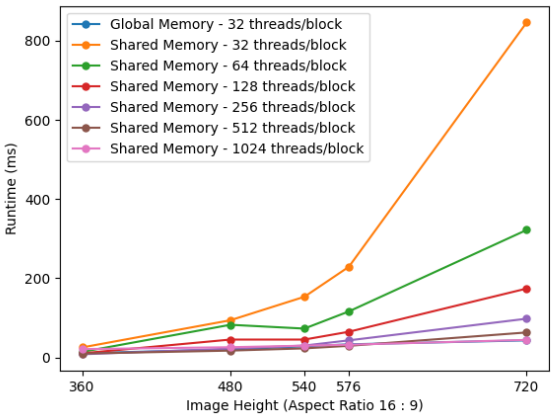

Figure 16. Accumulation Kernel Runtime

Note in Figure 16, the run time variation for 360p image is small, but we observer a huge gap for the 720p image with different block sizes. To explain this behavior, Figure 17 compares histograms of block vote count for 32 threads/block and 1024 threads/block versions on a 720p image. We randomly sampled blocks to count their total votes and plotted the histogram. The red vertical line indicates the size of the shared 2D accumulator slice. For the 32 threads/block case, all blocks vote much less than the threshold value, while for 1024 threads/block case, most blocks vote way more than the size of the shared accumulator. Note that the entire shared 2D accumulator slice needs to be initialized at the beginning of each thread block, and be written back to global memory at the end of block execution. Therefore when the average vote count is small, the overhead of initializing and copying the whole shared memory block outweighs the speedup it brings during voting.

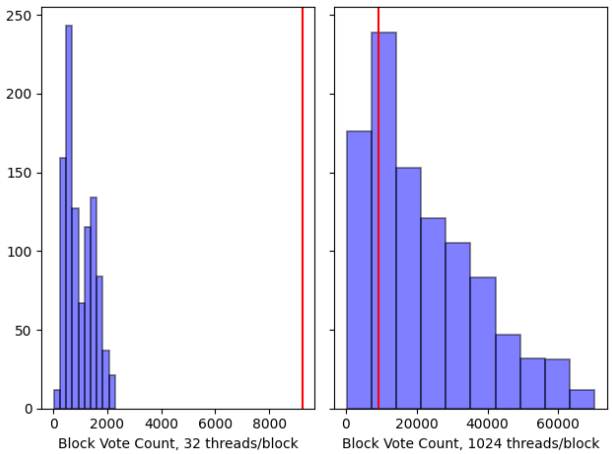

Figure 17. Histogram of Block Vote Count on 720p image

Although with a large block size, the shared memory

version shows as good performance as the global memory code, it does not achieve any further speedup. Some possible reason behind is that: (1) In the shared memory version, voting is still an atomic operation, as each thread may vote to the same location in the image. (2) It requires extra synchronization which may slow down the code.

We eventually choose not to use the shared memory version to measure the best speedup, as the size of input image is bounded by the memory allocation limit. It remains an interesting topic about how to represent the voting data in a more memory efficient way.

4.4. Overall Speedup

Figure 18 shows the best speedup of the overall algorithm we obtained, using the sort-3D, global memory strategy. On a 4K input image, it's able to finish the GHT pipeline with 433x speedup.

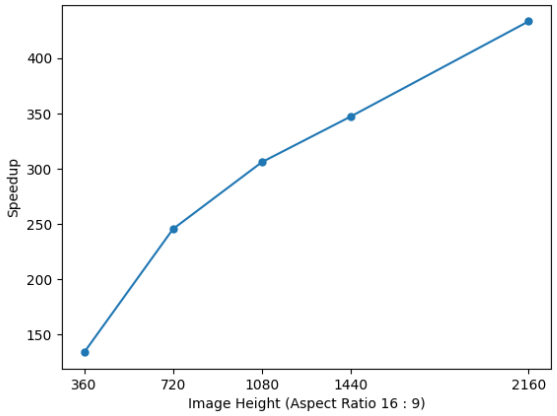

Figure 18. Overall speedup with the best parallel strategy

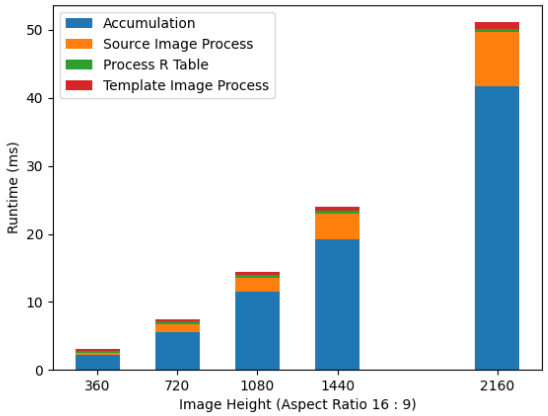

Figure 19. Runtime Rreakdown

Figure 19 breaks down the run time of the best speedup version into Accumulation, Source Image Processing, Processing R Table, and Template Image Processing. It further confirms that the accumulation step is the critical step of the whole pipeline, and by developing an efficient algorithm to deploy the accumulation on GPU, we are able to achieve a huge performance gain.

5. Distribution of Total Credit

Both authors contribute equally to this project.

References

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

A. Appendix: Test Images

Below we list all the test images used in Section 4. We resized all the images to fit in the page, but the relative image size is accurate.

Figure 19 - 28 show the test images described in the **Image Size** Section, 29 - 33 in the **Percentage of Edge Pixels** Section, and 34 - 39 used for Section 4.3.1.

Figure 20. Template - 360p

Figure 21. Source - 360p

Figure 22. Template - 720p

Figure 23. Source - 720p

Figure 24. Template - 1080p

Figure 25. Source - 1080p

Figure 26. Template - 2k

Figure 27. Source - 2k

Figure 28. Template - 4k

Figure 29. Source - 4k

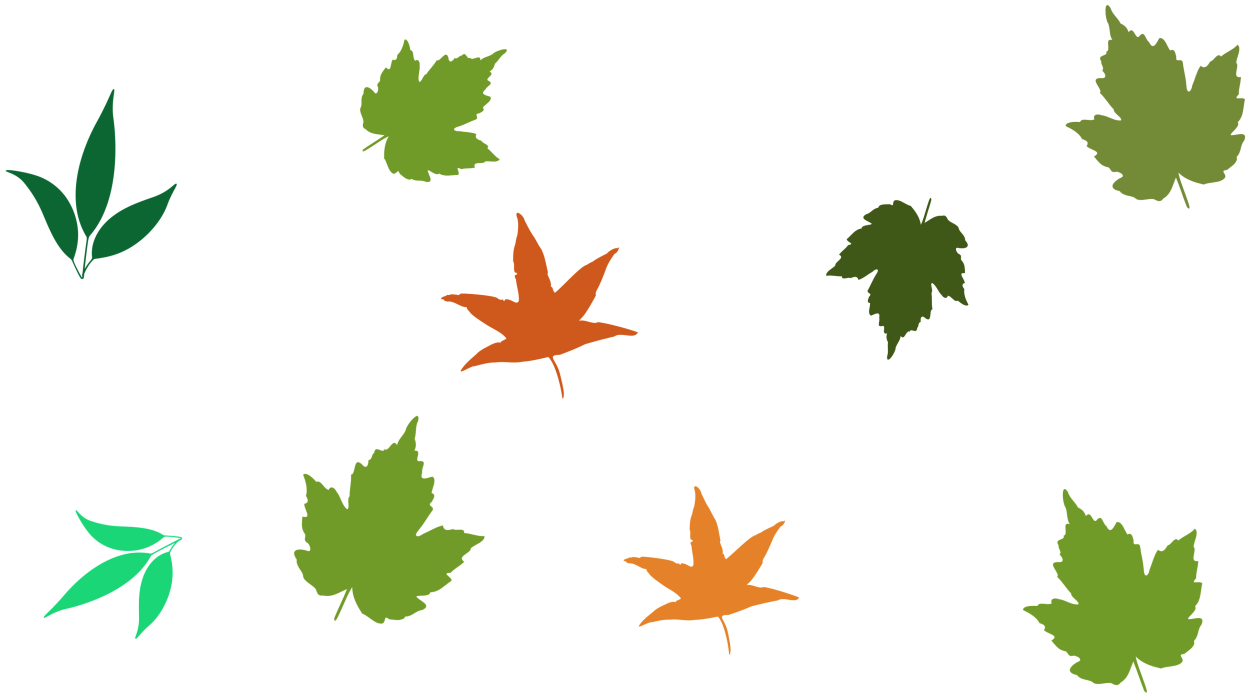

Figure 30. Source - 4k - 0.196% edge pixels

Figure 31. Source - 4k - 0.500% edge pixels

Figure 32. Source - 4k - 0.841% edge pixels

Figure 33. Source - 4k - 1.249% edge pixels

Figure 34. Source - 4k - 1.410% edge pixels

Figure 35. Template

Figure 36. Source - 360p

Figure 37. Source - 480p

Figure 38. Source - 540p

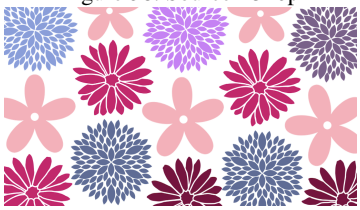

Figure 39. Source - 576p

Figure 40. Source - 720p