GPU-vote: A Framework for Accelerating Voting Algorithms on GPU

Gert-Jan van den Braak, Cedric Nugteren, Bart Mesman, and Henk Corporaal
{g.j.w.v.d.braak, c.nugteren, b.mesman, h.corporaal}@tue.nl

Eindhoven University of Technology, The Netherlands

Abstract. Voting algorithms, such as histogram and Hough transforms, are frequently used algorithms in various domains, such as statistics and image processing. Algorithms in these domains may be accelerated using GPUs. Implementing voting algorithms efficiently on a GPU however is far from trivial due to irregularities and unpredictable memory accesses. Existing GPU implementations therefore target only specific voting algorithms while we propose in this work a methodology which targets voting algorithms in general.
This methodology is used in GPU-VOTE, a framework to accelerate current and future voting algorithms on a GPU without significant programming effort. We classify voting algorithms into four categories. We describe a transformation to merge categories which enables GPU-VOTE to have a single implementation for all voting algorithms. Despite the generality of GPU-VOTE, being able to handle various voting algorithms, its performance is not compromised. Compared to recently published GPU implementations of the Hough transform and the histogram algorithms, GPU-VOTE yields a 11% and 38% lower execution time respectively. Additionally, we give an accurate and intuitive performance prediction model for the generalized GPU voting algorithm. Our model can predict the execution time of GPU-VOTE within an average absolute error of 5%.

1 Introduction

Accelerating applications with GPUs (Graphical Processing Units) has become increasingly popular from 2006 on, when GPUs became programmable with the introduction of "CUDA" by NVIDIA and "Close to Metal" by AMD. Just on NVIDIA's website over 1000 applications are listed which use a GPU for acceleration. These applications originate from various domains, such as image and signal processing, finance, statistics and electronic design automation.

GPUs are used in consumer desktop computers and notebooks as well as in embedded systems and industrial machines such as professional printers. Together with the (power) efficiency and the off-the-shelf availability, GPUs are interesting for large companies as well as for small and medium-sized enterprises. This motivates programmers to spend time and effort on making libraries, tools and generic (skeleton) implementations.

A number of algorithms in the image processing domain, such as color space conversion and low-level (pixel) filtering operations, are fairly straight forward to implement on a GPU due to their inherent parallelism. Voting algorithms on the other hand are far from trivial to implement efficiently on a GPU due to irregularities and unpredictable memory accesses.

Existing GPU implementations only target a single specific voting algorithm, such as histogram [1–3] and 2-D Hough transform [4–7]. In this work we propose a generic methodology which targets voting algorithms in general. We also introduce a framework called GPU-VOTE which can be used to accelerate a large range of voting algorithms. With this framework the time consuming and cumbersome implementation and optimization of voting algorithms is a thing of the past. Measurements show that GPU-VOTE is not just more generic than previous dedicated implementations, but also gives a performance improvement. To predict the execution time of GPU-VOTE, we also give a model based on the parameters of the voting algorithm, such as input size and number of bins.

This paper is organized as follows. First related work and background information about the GPU architecture is presented in Section 2. In Section 3, voting algorithms are categorized, the generic methodology for voting algorithms and the implementation in GPU-VOTE is discussed. Section 4 evaluates the proposed methodology and describes performance results. A model to predict execution time is given in Section 5. Finally, conclusions and future work are presented in Section 6.

2 Background and related work

In this section histogram and Hough transform, two common voting algorithms, are described in more detail. Also related work on GPU implementations of these algorithms is discussed in Section 2.3, as well as details on the NVIDIA GPU architecture in Section 2.4.

2.1 Histogram

In the histogram algorithm a set of bins is filled according to the frequency of occurrence in the input data. For example, a 1-D histogram of an 8-bit gray-scale image usually has 256 bins (due to the 256 possible shades of gray in the image).

A histogram can be used to enhance the contrast of an image by applying histogram equalization. In [8] a 2-D 30×30 histogram of normalized red and green is used for locating a road in an image. In [9] larger 2D-histograms of 256×256 bins are used for image registration purposes. Histogram is also an important statistical tool for displaying and summarizing data [10].

2.2 Hough transform

The Hough transform is a popular technique to locate shapes in images, such as lines and circles, but also other arbitrary shapes. It is used in many computer vision and image processing applications, such as robot navigation [11], industrial inspection and object recognition [12].

The Hough transform for lines [13] is a 2-D voting algorithm for which each feature (edge) point in an image votes for all possible lines passing through that point. All votes are stored in the so called Hough space, which size is determined by the input image and the accuracy for the parameterization of the lines.

The Cartesian coordinate system is not well suited for the Hough transform, therefore a polar representation of a line Eq. 1 is used in which a line is parameterized with ρ and θ [14]. Parameter θ represents the angle of a line normal to the line in the image and parameter ρ represents the shortest distance between the origin and the line in the image. The angle θ ranges from $0°$ to $180°$ and the radius ρ ranges from $-W$ to $\sqrt{W^2 + H^2}$, where W and H are the width and height of the image respectively.

$$\rho = x\cos(\theta) + y\sin(\theta) \tag{1}$$

By selecting a step size for the angle parameter θ, the number of sets of output bins (or independent vote spaces) is chosen; by selecting a value for N in Eq. 2, the number of bins in each independent vote space is set. With an input image size of 1920×1080 pixels, and a resolution for ρ and θ of one pixel and one degree, the total vote space consists of $(\sqrt{1920^2 + 1080^2} + 1920) \times 180 = 742{,}140$ bins.

$$\rho' = \frac{x\cos(\theta) + y\sin(\theta) \;\; + W}{\sqrt{W^2 + H^2} + W} N \tag{2}$$

2.3 Related work

Two implementations for histogramming are described by Podlozhnyuk in [1], one for 64-bin histograms and one for 256-bin histograms. Shams and Kennedy present two other histogramming methods in [2], which support a range of bin sizes. Nugteren et al. introduce two new histogramming implementations in [3] which are faster than the work of Podlozhnyuk and Shams and Kennedy. Their fastest implementation has a fixed processing time for a given input size.

The 2-D Hough transform has been implemented on a GPU in OpenGL and in CUDA. Two OpenGL implementations can be found in [4] and [5]. With the availability of CUDA nowadays, using OpenGL to program GPUs for general purpose computations is deprecated. A CUDA implementation of the Hough transform can be found in CuviLib [6], a proprietary computer vision library. Van den Braak et al. [7] introduced two new CUDA implementations of the Hough transform, one which focuses on minimizing processing time, while the other has an input data independent processing time.

2.4 GPU architecture

In NVIDIA's latest GPU architecture named Fermi [15], 32 CUDA cores are grouped into a processing cluster called a Streaming Multiprocessor (SM). Each SM has an on-chip memory of 48 kB. The GPU used for timing measurements in this paper is an NVIDIA GTX 470 which has 14 SMs and is connected to 1280 MB of off-chip memory.

The code executed on a GPU is called a kernel. Kernels run on the GPU as thousands or even millions of threads. Each thread executes the same kernel, but not necessarily the same instruction at the same time. Threads are organized into thread blocks. All threads in a thread block are executed on the same processing cluster (SM) and can communicate via its shared memory. Threads within a thread block are arranged in warps of (at most) 32 threads, with each thread in a warp executing the same instruction at the same clock cycle [16].

3 GPU voting methodology

To enable a generic methodology for voting algorithms on GPU, we classify voting algorithms. Two properties are distinguished which leads to a classification of four categories:

- The first property characterizes if either the element **value** or the **location** in the input data is used to determine in which bin the vote will be placed.
- The second property describes if a voting algorithm increases a bin in the vote space by one (**unity vote**) or by a calculated number (**number vote**).

An algorithm which uses the input element location can be converted to an algorithm which uses the input element value by applying a transformation on the input data. By building an array of input element locations that are used by the voting algorithm (e.g. less than 10% in the Hough transform [7]), a location based voting algorithm can be transformed into a value based voting algorithm.

An overview of this two stage generic GPU voting methodology can be found in Fig. 1. First the building of the array of input element locations is described in Section 3.1, and in Section 3.2 the final stage where votes are placed in the vote space is described. Improvements to this final stage are described in Section 3.3.

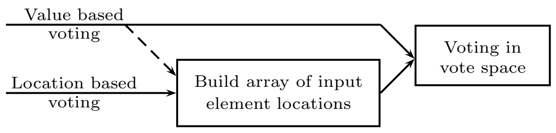


Fig. 1. Overview of the two stage generic GPU voting methodology

3.1 Building an array of locations

Building the arrays of input element locations and values is done similarly to [7], which is in turn inspired by the work in [3]. To build these arrays in a parallel way on the GPU, small arrays are created at warp-level granularity. The technique to make an array per warp is summarized in pseudo-code in Lst. 1. Note that all threads in a warp execute the same instruction at the same time in parallel, but some threads may be disabled due to branching conditions. More detailed information on the creation of the array can be found in [7].

```
input_value = input[x,y]
if(element_test(x,y,input_value)) {
  do {
    index++
    SMEM_index = index
    SMEM_array_L[index] = (x,y)
  } while(SMEM_array_L[index] != (x,y))
  SMEM_array_V[index] = input_value
}
index = SMEM_index
```

Listing 1. Building an array of input element locations (`SMEM_array_L`) and an array with the corresponding values (`SMEM_array_V`) in shared (on-chip) memory (`SMEM`).

3.2 Voting in vote space

After the arrays of element location and element value have been constructed for the location based voting algorithms, the votes can be placed in the vote space[1]. The voting process is executed on the on-chip memory of an SM using atomic memory operations. In case the vote space can be divided in independent vote spaces (e.g. Hough transform), each independent vote space is calculated by a thread block on an SM. The number of bins in a vote space is limited by the amount of on-chip memory in an SM. For the Fermi GPU architecture, the amount of on-chip memory per SM is 48 kB, resulting in maximum 12,288 32-bit bins (N_{max}). In case more than N_{max} bins are required, this second stage of the voting process is executed multiple times in order to calculate all bins.

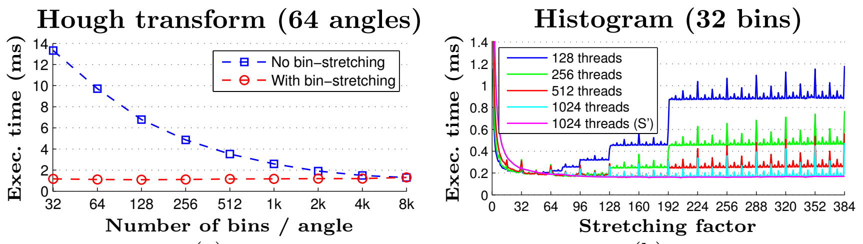
When the number of independent vote spaces is less than the number of SMs on the GPU (e.g. histogram algorithm, one independent vote space), the input data is split such that each SM will process an equal part. In a final step, the results of all parts are added together to make the final vote space.

3.3 Bin stretching

As described in Section 2.2, the Hough transform can be implemented with a parameterizable number of bins per independent vote space. The effect of varying this number on the execution time of the Hough transform implementation from [7] is shown in Fig. 2(a) with blue squares. The graph shows that the Hough transform can be calculated faster when more bins are used, which leads to the unintuitive conclusion that more accurate results can be calculated quicker than less precise results.

This effect is caused by contention in the on-chip memory of the GPU. According to Eq. 2, pixels that are close together in the input image, will also end-up in bins close together in the vote space. If a small number of bins is used, more votes are placed in the same bin (through atomic memory operations), causing contention in the memory.

[1] For value based voting algorithms, the input itself is directly used as location and value array. Only when not all elements in the input are used in the voting process, an array can be build first, indicated by the dashed arrow in Fig. 1.



(a) (b)

Fig. 2. a: Execution time measurements of the Hough transform algorithm with 64 angles for θ. Blue squares: no bin stretching, red circles with bin stretching. **b:** Execution time measurements of a 32-bin histogram for four different numbers of GPU threads per thread block. The last line shows the results of the updated stretching factor S'.

To solve this memory contention, and thus to improve performance, we introduce a technique called bin stretching. We explain the bin stretching technique and show the results based on two examples.

Bin stretching technique To reduce the memory contention, the number of bins (N in Eq. 2) can be increased to the maximum number of bins allowed by the hardware (N_{max}). Since the maximum number of bins and the number of bins in a voting algorithm are both known, a stretching factor (S) can be calculated as shown in Eq. 3. To make sure that consecutive threads use as many different memory locations as possible, the new bin-index is calculated as shown in Eq. 4. After all votes are placed in the $S \times N$ bins, the results need to be compacted back into the desired N bins by summing each group of S consecutive bins together.

$$S = \max\left(\left\lfloor \frac{N_{max}}{N} \right\rfloor, 1\right) \tag{3}$$

$$bin' = bin \cdot S + (thread\text{-}id \bmod S) \tag{4}$$

The effect of bin stretching by two examples Applying the bin stretching method on the Hough transform example from Fig. 2(a), the performance improvement between the original approach (blue squares) and the bin stretching approach (red circles) is clear. With bin stretching applied, the maximum amount of on-chip memory is used in the voting process for any number of bins per angle, which reduces memory contention to a minimum.

The effect of bin stretching on a 32-bin histogram algorithm is shown in Fig. 2(b). When using 1024 threads (which results in the overall lowest execution time compared to smaller numbers of threads per thread block), a stretching factor of 383 results in a speed-up of over 20× compared to an implementation without bin stretching.

In Fig. 2(b) the input is split into a number of parts to fill up all SMs as much as possible, as explained in Section 3.2. Each SM can process a few parts at once, given enough resources. The resource limitation can be either the number of resident threads per SM (maximum of 1536), or the number of resident thread blocks per SM (maximum of 8) or the on-chip shared memory per SM (48 kB) [16]. For example, when 128 threads are used (top blue line in Fig. 2(b)), and the stretching factor is 192, two parts can be processed by two thread blocks on one SM. But when the stretching factor is increased to 193, there is only enough on-chip memory for one part per SM, resulting in a significantly increased execution time.

One thing to notice in Fig. 2(b) are the peaks in execution time for certain stretching factors. These peaks occur at stretching factors which are a multiple of 32. This is due to the number of banks in the on-chip memory of an SM, which is also 32 [16]. Careful inspection of Fig. 2(b) shows that odd stretching factors give a lower execution time compared to an even stretching factor one value larger. But for stretching factors smaller than 64, the improvement in execution time due to the reduction in memory conflicts (caused by the stretching) is larger than the decline in execution time due to the even stretching factor.

Taking this into account, the optimal stretching factor is calculated with Eq. 5, and the corresponding bin-index with Eq. 6. This is in accordance with [16], where an odd step size is suggested for strided on-chip memory accesses.

$$S' = \begin{cases} S & \text{if } S < 64 \\ S - (1 - S \bmod 2) & \text{if } S \geq 64 \end{cases} \tag{5}$$

$$bin'' = bin \cdot S' + (thread\text{-}id \bmod S \bmod S') \tag{6}$$

The effects of this updated stretching factor on the execution time is shown in the last (purple) line in Fig. 2(b). By using only odd stretching factors with S', the peaks in execution time as shown before are removed.

```
#include "gpu_vote.h"
class Histogram : public CVoteFunction<uchar, uint> {
  public:
    __device__ uint vote_index(uint index, uchar value, uint nbins) {
      return value * (nbins / 256.0f);
    }
};
```

Listing 2. Implementation of a histogram algorithm using GPU-vote.

3.4 Implementation with GPU-vote

The generic GPU voting methodology is implemented in a framework named GPU-VOTE. To support a large range of voting algorithms, the variable types for the input and output and four functions have to be defined for each voting algorithm. One function is required for the first stage in the methodology (array building), the other three are used in the second stage (voting in the vote space).

One example implementation of a gray-scale histogram algorithm is shown in Lst. 2 On line 1 the GPU-VOTE framework is loaded. The variable types of the input and output (uchar and uint) are specified on line 2. Only the standard vote index function, which determines in which location a vote is placed, is overloaded on lines 4-6. The other functions' default implementation is sufficient.

4 Evaluation

To evaluate the proposed methodology, the execution time of GPU-VOTE is compared with state of the art implementations. In Fig. 3 the performance of various approaches to implement the 2-D Hough transform on a GPU is shown. The performance of [3] is indicated with blue circles. This approach is optimized for histogramming and can only handle up to 512 bins. The voting method from [7] focuses on the Hough transform and is indicated with green squares. The best performance for this algorithm is achieved with a large number of bins.

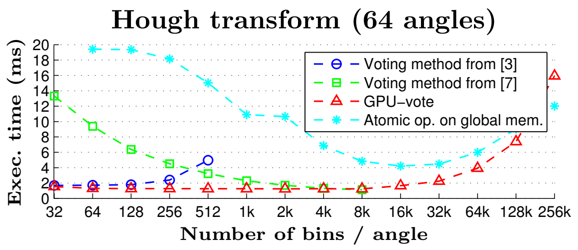


Fig. 3. Execution time of four different voting methods for a range of number of bins.

The proposed generic voting methodology is shown as red triangles in Fig. 3. It outperforms the previous two methods and supports a range of number of bins with equal performance. Only in case the required number of bins is larger than the maximum number of bins the implementation can support (N_{max}), the execution time increases. This increase is caused by the second stage of the voting methodology being executed multiple times (see also Section 3.2). To prevent the necessity of executing the second stage multiple times the on-chip memory size of the GPU has to be increased for future hardware, or taking a hierarchical approach for the voting algorithm could be investigated.

For comparison, a GPU voting method where all votes are placed directly in off-chip memory is also included in Fig. 3 (cyan stars). Although this method has a large performance penalty for a number of bins less than 8k, its execution time gets closer to the proposed generic voting methodology when the number of bins increases. For number of bins larger than 128k this approach even outperforms the proposed generic voting methodology.

The performance of the proposed generic voting algorithm is evaluated by implementing four algorithms with GPU-VOTE on an NVIDIA GTX 470. The input for all algorithms in this section are 1920 by 1080 pixels gray scale images.

For histogram and Hough transform, the results of GPU-VOTE are compared with the best known GPU implementation from [3] and [7] respectively. Calculating a 256-bin histogram with GPU-VOTE is 38% faster on average for 110 randomly selected test images compared to the the implementation in [3]. Calculating a 2-D Hough transform with a total of 64 angles and 4200 bins per angle takes 11% less time on average for the same 110 images compared to the implementation in [7]. The main improvement of GPU-VOTE over [7] is the bin-stretching technique introduced in Section 3.3.

Other algorithms, which may not be identified as voting algorithms directly, also fit our proposed framework. We show performance results to indicate the wide applicability of GPU-VOTE. For example, calculating the sum of all elements in a matrix can be seen as a voting algorithm with only a single bin. Compared to the heavily optimized implementation in the reduction example in the NVIDIA CUDA software development kit, the results of GPU-VOTE are 12% and 14% slower when summing 8-bit integer inputs and 32-bit float inputs respectively.

In image sub-sampling, the location of the input pixel determines in which bin the pixel value has to be added, which makes it a location-based number-voting algorithm. GPU-VOTE has a 2.6× higher execution time compared to a manual optimized GPU implementation. Since the structure of this algorithm is very regular, using a generalized voting method is clearly not the best approach in terms of execution time for implementing this algorithm. However the manual optimized implementation requires detailed knowledge about the GPU architecture. Using GPU-VOTE does not require this knowledge, and the implementation consist of only four lines of straight forward C-code, in contrast to the 20 lines of highly optimized CUDA kernel code for the manual optimized implementation.

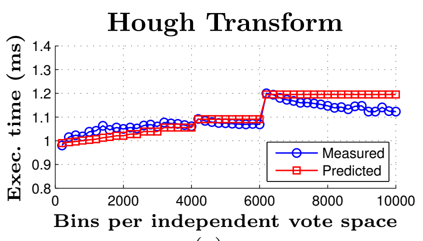
5 Performance prediction

To predict the performance of a voting algorithm, we introduce a model which is based on properties of the image (e.g. size), the voting algorithm (e.g. number of bins) and GPU parameters (e.g number of SMs). The prediction can be used to select a suitable GPU or to choose algorithm parameters for example. Both stages of the generalized GPU voting algorithm are modeled separately. We first explain how these models are derived. Following, we evaluate the quality of the performance prediction.

5.1 Prediction model

The first stage of the algorithm builds an array of input element locations. A part of the input elements (which pass the element test) are stored into two arrays, one of element locations and one of element values. The execution time of this stage of the algorithm is a combination of the total number of input elements which need to be evaluated (E) and the relative number of elements which have to be put in an array (ηE) as shown in Eq. 7.

$$T_1 = a \times E + b \times \eta E + c \tag{7}$$



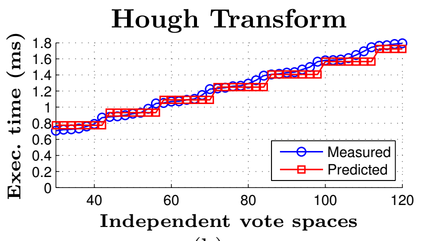


Fig. 4. Hough transform execution times for **a:** a range of bins per angle (N in Eq. 2 and Eq. 3) and **b:** a range of number of angles (G).

The second stage of the algorithm places votes in the vote space. The execution time of this stage of the algorithm depends on the relative number of elements used in this second stage (ηE) and the number of votes that need to be placed in the vote space ($G \times \eta E$), where G is the number of independent vote spaces. Since every independent vote space is calculated by one thread block, which is mapped to a single SM, G has to be rounded up to the nearest multiple of the available number of SMs (SM). In case the number of bins (N) in an independent vote space is larger than the maximum the hardware can supply (N_{max}), this second stage is executed multiple times and the product $G \times \lceil N/N_{max} \rceil$ has to be rounded up to the nearest multiple of SM.

When the product of the number of independent vote spaces and $\lceil N/N_{max} \rceil$ is less than the number of SMs, the input is split in parts (P in Eq. 8) to reduce the execution time of this second stage. The impact of G, N, N_{max} and SM on the execution time can be taken into account with the factor F in Eq. 9.

$$P = \max\left(\left\lfloor \frac{SM}{G \times \lceil N/N_{max} \rceil} \right\rfloor, 1\right) \tag{8}$$

$$F = \left\lceil \frac{G \times \lceil N/N_{max} \rceil}{SM} \right\rceil \frac{1}{P} \tag{9}$$

The execution time of the second stage also depends on the number of bins (N) through the stretching factor (S'), as explained in Section 3.3. The following equation can be composed to estimate the execution time of the second stage:

$$T_2 = d \times \frac{\eta E}{S'} \times F + e \times \eta E \times F + f \tag{10}$$

The parameters $a - f$ in the model are estimated by performing experiments in which training images and algorithm parameters in Eq. 7 and Eq. 10 are varied and a least-squares minimization on the predicted execution time is applied.

5.2 Evaluation of the prediction

The model of the prediction of the execution time is evaluated with two experiments. In each experiment the Hough transform is calculated on a 1920 × 1080

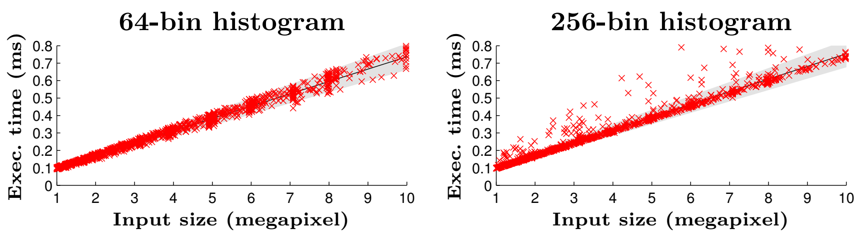


Fig. 5. Measured and predicted execution times for the 64-bin and 256-bin histogram algorithm executed on 1295 images.

input image (distinct from the training images), on which first edge-detection and thresholding are applied as described in [7]. Two aspects of the Hough transform are varied: the number of bins (N) and the number of independent vote spaces (G). As shown in Fig. 4, the model clearly follows the steps in execution time caused by the bin stretching factor S (Fig. 4(a)) and the steps caused by the rounding of $G \times \lceil N/N_{max} \rceil$ up to the nearest multiple of SM (Fig. 4(b)).

The model is also evaluated on two other experiments, a 64-bin and a 256-bin histogram on 1295 gray scale images, varying in size from 1 to 10 megapixel. The average absolute error of the predicted execution time compared to the real execution time is 5%. For 95% of the images the execution time is predicted within an error range of -10% to 10%, as shown with the gray marked area in Fig. 5. For the 256-bin histogram, the execution time for some images is underestimated significantly. These images contain large surfaces with a single color, causing memory collisions. These can be resolved by applying bin-stretching in the 64-bin histogram, but not in the 256-bin histogram, since the stretching factor is limited by the available on-chip memory and the number of bins used.

6 Conclusions

In this work we have introduced a generic methodology for implementing voting algorithms on a GPU. A classification of voting algorithms is presented, which arranges voting algorithms into four groups. We also gave a transformation on the input data of a voting algorithm to be able to merge categories. This enables the development of a methodology to solve voting problems in all categories with a single unified solution. The proposed bin stretching technique forms the base of the performance improvements of the generic methodology.

The generic methodology is implemented in GPU-VOTE, a framework which can be used to accelerate a range of voting algorithms on a GPU. With GPU-VOTE two examples, histogram and Hough transform have been implemented on a GPU. As shown in the evaluation section, GPU-VOTE is not just generic but even yields a lower execution time compared to previously published GPU implementations which targeted only a single voting algorithm. The performance of histogram and Hough transform is improved by 38% and 11% respectively. To show the wide range of applicability, algorithms such as sum reduction and image sub-sampling have been implemented with GPU-VOTE. Although the performance of these implementations cannot match the reference implementations, the programming effort to implement such algorithms is reduced significantly.

To estimate the execution time of a voting algorithm a model is is given based on parameters of the input, voting algorithm and GPU used. Results show that in 95% of the cases the model predicts the execution time within a 10% range.

As part of future work the proposed methodology can be implemented in OpenCL to enable its use on other architectures, such as AMD GPUs and multi-core CPUs, making the methodology even more useful. We also plan to to improve performance by making GPU-VOTE multi-GPU enabled and to develop a hierarchical approach for voting algorithms to support a larger number of bins.

References

1. Podlozhnyuk, V.: Histogram Calculation in CUDA. (2007)
2. Shams, R., Kennedy, R.A.: Efficient Histogram Algorithms for NVIDIA CUDA Compatible Devices. In: International Conference on Signal Processing and Communications Systems. (2007)
3. Nugteren, C., Van den Braak, G.J., Corporaal, H., Mesman, B.: High Performance Predictable Histogramming on GPUs: Exploring and Evaluating Algorithm Trade-offs. GPGPU 4 (2011)
4. Fung, J., Mann, S.: OpenVIDIA: Parallel GPU Computer Vision. In: 13th ACM international conference on Multimedia. (2005)
5. Ujaldón, M., Ruiz, A., Guil, N.: On the computation of the Circle Hough Transform by a GPU rasterizer. Pattern Recognition Letters (2008)
6. TunaCode (Ltd): CUDA Vision and Imaging Library `http://www.cuvilib.com/`.
7. Van den Braak, G.J., Nugteren, C., Mesman, B., Corporaal, H.: Fast Hough Transform on GPUs: Exploration of Algorithm Trade-Offs. In: ACIVS. Volume 6915 of Lecture Notes in Computer Science., Springer Berlin (2011)
8. Tan, C., Hong, T., Chang, T., Shneier, M.: Color Model-Based Real-Time Learning for Road Following. In: Intelligent Transportation Systems Conference. (2006)
9. Maes, F., Collignon, A., Vandermeulen, D., Marchal, G., Suetens, P.: Multimodality Image Registration by Maximization of Mutual Information. IEEE Transactions on Medical Imaging (1997)
10. Scott, D.W.: On Optimal and Data-Based Histograms. Biometrika (1979)
11. Forsberg, J., Larsson, U., Wernersson, A.: Mobile Robot Navigation using the Range-Weighted Hough Transform. Robotics Automation Magazine, IEEE (1995)
12. Wang, Y., Shi, M., Wu, T.: A Method of Fast and Robust for Traffic Sign Recognition. In: Fifth International Conference on Image and Graphics. (2009)
13. Hough, P.: Method and Means for Recognising Complex Patterns. US Patent No. 3,069,654 (1962)
14. Duda, R.O., Hart, P.E.: Use of the Hough Transformation to Detect Lines and Curves in Pictures. Commun. ACM **15** (1972)
15. NVIDIA Corporation: NVIDIA's Next Generation CUDA Compute Architecture: Fermi (2009)
16. NVIDIA Corporation: NVIDIA CUDA C Programming Guide - Version 4.0 (2011)