University of Groningen
Faculty of Science and Engineering

BACHELOR'S THESIS

Sofie Lövdal

Efficient implementation of the COSFIRE algorithm with CUDA

Supervisors: Prof. Dr. Michael Biehl
Dr. Nicola Strisciuglio
Study programme: Computing Science

Groningen July 27, 2018

I would like to thank my supervisors for their competent guidance in completing this project, and especially my daily supervisor Nicola for his professional and enthusiastic advice.

Furthermore, I would like to thank my parents, together with all other persons that have come with valuable feedback on this document.

Title: Efficient implementation of the COSFIRE algorithm with CUDA

Author: Sofie Lövdal (s.s.lovdal@student.rug.nl)

Department: Computing Science

Supervisors of the bachelor thesis: Prof. Dr. Michael Biehl and Dr. Nicola Strisciuglio, Johann Bernoulli institute of Mathematics and Computing Science

Coordinator of B.Sc. Project: Prof. Dr. Alexandru C. Telea

Abstract: The thesis explores runtime performance improvement of the COSFIRE algorithm when parallelized with CUDA on GPUs. COSFIRE is a trainable filter for pattern recognition in images. The core algorithm consists of four steps which are applied a number of times with different parameters, creating several subresponses. The steps involve convolution with a Difference-of-Gaussians filter, max-blurring and shifting. The subresponses are gathered by geometric mean. In this algorithm both task parallelism and data parallelism can be identified. The CUDA implementation exploits the massively parallel architecture of a GPU by employing mainly data parallelism - in each step of the algorithm, one thread performs the calculations for one pixel. While the control flow of the CUDA implementation is kept in host code, the amount of data transfer between the CPU and the GPU is kept to a minimum and all key steps of the algorithm are executed on the GPU. On a GTX 1080 ti GPU with 3584 cores, a speedup of 28.3 has been achieved when applying the CUDA COSFIRE implementation on a retinal fundus image of resolution 999x960 pixels.

Keywords: COSFIRE, CUDA, GPU parallelization, Image processing

Contents

1 Introduction **2**

2 Background **4**
2.1 The COSFIRE algorithm 4
2.1.1 Configuration 4
2.1.2 Application 5
2.1.3 Tolerance to rotation 7
2.1.4 Result 7
2.2 General purpose computing on GPUs 8
2.3 CUDA 8

3 Problem analysis **10**
3.1 Identifying parallelism 10
3.2 Considerations 11

4 Implementation **13**
4.1 Design 13
4.2 Kernel design 15
4.3 Integration into Matlab code 15
4.4 Considerations 15

5 Results **16**
5.1 Accuracy 16
5.2 Performance 18

6 Discussion **23**
6.1 Analysis of results 23
6.2 Conclusion 24

List of Figures **27**

List of Tables **28**

Appendix **28**

Chapter 1

Introduction

COSFIRE is a trainable filter for pattern recognition and keypoint detection in the field of computer vision [2]. It can be used to extract certain features in an image based on a prototype pattern that is given to configure the filter. An example an of important application of the method is delineating blood vessels in retinal fundus images (see Figure 1.1). Other examples of applications are detecting traffic signs in urban areas, extracting cracks in a wall [12], or rivers in an aerial photograph.

Pattern recognition and keypoint detection is a classic problem in image processing, and the COSFIRE algorithm achieves highly accurate results while being faster than most other existing methods [3]. The method can be applied to a variety of problems, theoretically including real time image processing. However, for this purpose the run time of the algorithm needs to improve. As an example, the main processing step of a single retinal fundus image of dimensions 565x584 pixels takes around 1.2 seconds to execute with the existing Matlab implementation [11]. This would for example make the speed of a moving robot applying the algorithm to perform a task very slow.

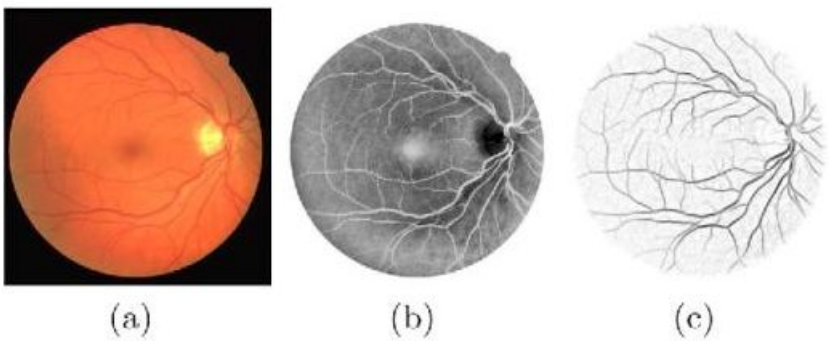

Figure 1.1: Example of blood vessel delineation in a retinal fundus image with COSFIRE. Input image (a), preprocessed input image (b) and rotation-invariant COSFIRE response (c) [3].

COSFIRE stands for Combination of Shifted FIlter REsponses. It is trainable, as its structure is determined by an automatic configuration process on a prototype pattern, rather than being fixed in the implementation [2]. The configuration results in a set of points that are determined to be vital to the structure of the prototype pattern. These points are used to configure a number of filters that are invividually applied to an input image. The algorithm finally obtains a response at a single pixel by combining the subresponses in a way that takes all distinct features of the prototype pattern into consideration. Due to the processing pipeline consisting of multiple stages of per pixel independent operiations applied to the image, the algorithm is very suitable to parallelisation.

The thesis explores how to efficiently implement the core algorithm with CUDA - performing the compute intensive tasks in parallel on a GPU - and the speedup that can be achieved hereby.

Chapter 2

Background

Here the COSFIRE algorithm is explained in detail, together with an overview of general purpose computing on GPUs and CUDA. This work is particularly oriented towards developing the CUDA application for bar-selective COSFIRE filters [3], and all examples in Section 2.1 refer to this.

2.1 The COSFIRE algorithm

The COSFIRE algorithm is a technique for pattern recognition. It consists of two phases: a configuration phase in which a prototype pattern is analysed and an application phase in which the pattern of interest is searched for in images. Note that before an input image is fed to the algorithm the image typically undergoes a preprocessing step, which is not covered in this work.

2.1.1 Configuration

A COSFIRE filter is automatically configured based on a given prototype pattern. A single center point is manually defined (point 1 in Figure 2.1) in the image containing the prototype pattern.

The first step of the configuration makes use of a Difference-of-Gaussians (DoG) filter. A DoG filter is a filter that can be used for detection of intensity changes in images [7], defined by

$$DoG_\sigma(x,y) = \frac{1}{2\pi\sigma^2} e^{\frac{-x^2+y^2}{2\sigma^2}} - \frac{1}{2\pi(0.5\sigma)^2} e^{\frac{-x^2+y^2}{2(0.5\sigma)^2}} \tag{2.1}$$

More particularly, the equation above describes a center-on DoG function used in [3], where the standard deviation of the inner Gaussian function is 0.5σ. Here σ denotes the standard deviation of the outer Gaussian function.

A response from processing an image with a DoG filter is obtained by convoluting the image with the kernel defined by the filter, according to

$$c_\sigma(x, y) = |I * DoG_\sigma|^+ \tag{2.2}$$

where $|\ |^+$ denotes half-wave rectification - all negative response values are set to zero.

A DoG filter with a given standard deviation is then applied to the prototype input pattern. Responses along a chosen number of concentric circles around the center point are inspected. The positions of the responses found that display a significant local maximum are chosen as points of interest. This can be interpreted such that these positions display a relevant intensity variation in the image and are a vital part of the pattern to be inspected.

These detected points are then described with regards to the center of the filter. Each point of interest is described by a tuple of three parameters σ_i, ρ_i and ϕ_i, where σ_i describes the standard deviation of the DoG filter that responded most strongly, and (ρ, ϕ) are the polar coordinates describing the position of the point of interest with regards to the center point (see Figure 2.1).

The result of the configuration is the set S of all tuples

$$S = \{(\sigma_i, \rho_i, \phi_i) | i = 1, ..., n\} \tag{2.3}$$

where n denotes the number of considered point of interest. These will define the filters applied in the processing step and the interpretation of these for the final response.

2.1.2 Application

The tuples achieved in the configuration step are used to define parameters of the operations that are applied to the input image in the processing step. This can be divided into three distinct stages described below [3].

1) DoG filtering

For each σ_i in the set of tuples S a DoG filter is applied to the input image, creating a number of DoG responses. The standard deviation of this filter has the value of the corresponding σ_i.

2) Max-blurring

For each ρ_i in the set of tuples, a Gaussian max-blurring operation is performed on the corresponding response from step (1). Max-blurring or dilation is taking the maximum of the pointwise multiplications when applying a filter to an image, rather than the sum as with traditional convolution. The max-blurring is performed with a Gaussian 2D filter with standard deviation $\sigma' = \sigma'_0 + \alpha\rho_i$, where σ'_0 and ρ_i are constants. A response from this step can be defined as

$$m_{\sigma_i,\rho_i,\phi_i}(x, y) = max\{c_{\sigma_i}(x - x', y - y')G(x', y')\} \tag{2.4}$$

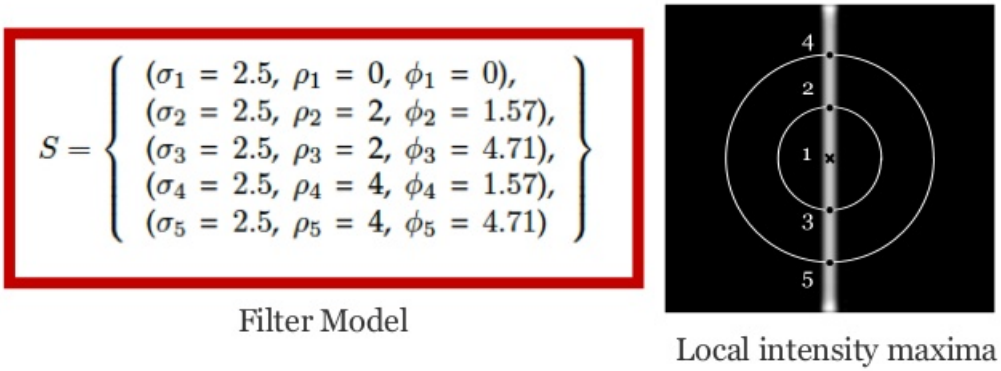

Figure 2.1: The COSFIRE filter is defined by a set S of tuples (left), corresponding to each point of interest in the prototype pattern (right). In this image, the filter is configured to be selective for a bar-like structure defined by the five points of interest that are marked in the prototype pattern. Each of these points is represented as a tuple with the second and third parameter corresponding to the polar coordinates of that point with respect to the center point (1). The first parameter describes the standard deviation of the DoG-filter that was applied to the prototype pattern in the first stage of the configuration process.

where c_σ is the output from the previous step. Each tuple has a corresponding max-blurred response, and the standard deviation associated with the Gaussian filter grows with ρ_i. This means that DoG responses representing points of interest further away from the center point are more blurred than points of interest closer to the center point. This is to achieve some tolerance in the position of the investigated points of interest, with a higher tolerance for points further away from the center point.

3) Shifting of responses

Each blurred response map undergoes a shifting operation in order for each investigated point of interest (tuple specific response) to be associated with a single pixel at the centre of the COSFIRE filter. The pixels of each response, corresponding to the tuple $(\sigma_i, \rho_i, \phi_i)$ are then shifted according to

$$\begin{aligned} \Delta x_i &= -\rho_i cos(\phi_i) \\ \Delta y_i &= -\rho_i sin(\phi_i) \end{aligned} \tag{2.5}$$

A response from this step can therefore be defined as

$$s_{\sigma_i,\rho_i,\phi_i}(x,y) = m_{\sigma_i,\rho_i,\phi_i}(x - \Delta x_i, y - \Delta y_i) \tag{2.6}$$

4) Response

The final response of the COSFIRE filter is achieved by taking the pixelwise geometric mean (alternatively, weighted geometric mean) of all subresponses:

$$r_S(x,y) = |\prod_{i=1}^{|S|} s_{(\sigma_i,\rho_i,\phi_i)(x,y)}|^{\frac{1}{|S|}} \tag{2.7}$$

where $|\ |$ denotes thresholding the response at a fraction of the maximum of the responses.

This means that the COSFIRE algorithm only gives a response when the all the concerned contributing filters give a response.

2.1.3 Tolerance to rotation

The above steps describe obtaining a COSFIRE response for a prototype pattern oriented in a specific direction. Tolerance to rotation can be obtained by running the algorithm for multiple directions of the prototype pattern, or alternatively by modifying the parameter ϕ of the tuples with a constant related to the rotation of the pattern. This creates the new, larger set of tuples

$$R_{\psi}(S) = \{(\sigma_i, \rho_i, \phi_i + \psi) | \forall (\sigma_i, \rho_i, \phi_i + \psi) \in S\} \tag{2.8}$$

The final output of the rotation-invariant COSFIRE algorithm is the pixelwise maximum of all rotation-specific responses.

2.1.4 Result

The output of the algorithm is an image where pixels that are determined to contain the prototype pattern are highlighted (see Figure 1.1). In the case of extracting bar-like structures from an image, responses from both a symmetric (for detecting bars) and an asymmetric filter (for detecting bar-endings) are combined to obtain an optimal response.

2.2 General purpose computing on GPUs

Graphics processing units (GPUs) were originally designed to improve the rendering of graphics to be displayed on a device. While a CPU normally has a small number of cores, a GPU has hundreds or even thousands of cores. Even though the GPU cores typically are simpler and slower than the CPU cores, they are very suitable for performing floating point operations [5]. Due to the inherently parallel nature of the computational stream in a GPU, the use of GPUs in scientific computing has become increasingly popular for speeding up the runtime of large jobs.

GPUs are especially suitable for SIMD-parallelizable problems, or problems that contain a high portion of data-parallelism. This is especially applicable to image processing, where each pixel often can be processed in parallel. In addition to that, image processing operations often involves relatively simple arithmetic calculations, which the GPU cores are designed for.

2.3 CUDA

Cuda is a platform for parallel computing and offers a C-syntax based programming language for writing GPU code [6]. It is developed by the NVIDIA corporation and is compatible with NVIDIA GPUs only. It is a programming model for heterogenenous computing, enabling interaction between the CPU (host) and GPU (device). The API offers structures of thread blocks and grids in order to map your program to GPU execution (see Figure 2.2). This structure is also related to hardware architecture, something that the API mostly allows to abstract from.

Functions written for the GPU are called kernels. Every kernel that is launched is queued and executed sequentially within each CUDA execution stream. If the GPU has enough capacity, multiple streams can be executed at once. The default number of streams is one and a single stream has also been used in this work.

The CUDA memory model can be viewed in Figure 2.3, picturing the multiple levels of memory space that is offered [6]. Out of these memory spaces, the host can communicate with global memory, constant memory and texture memory.

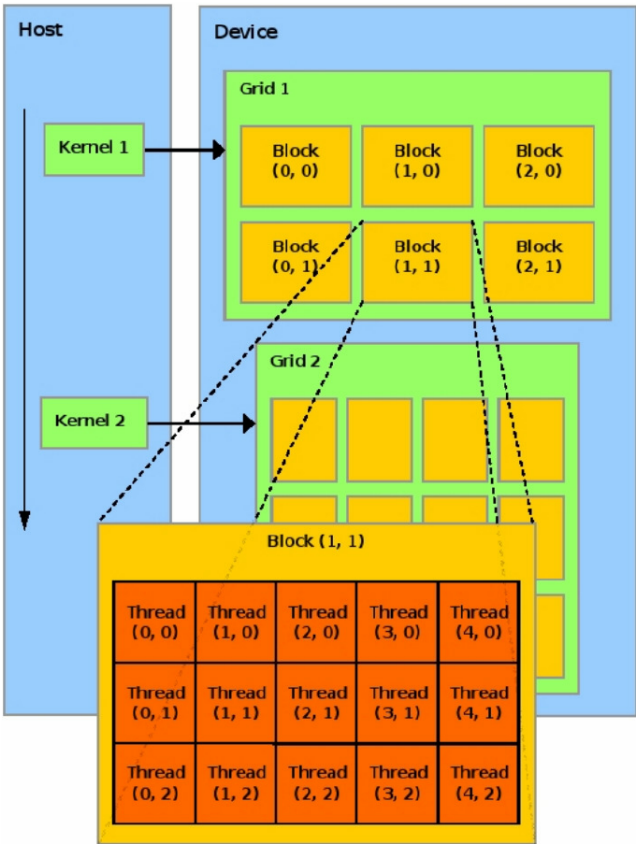

Figure 2.2: Example of flow of CUDA program. The host, corresponding to the CPU, executes the program code in serial. Each kernel call corresponds to a function to be executed on the device (GPU).

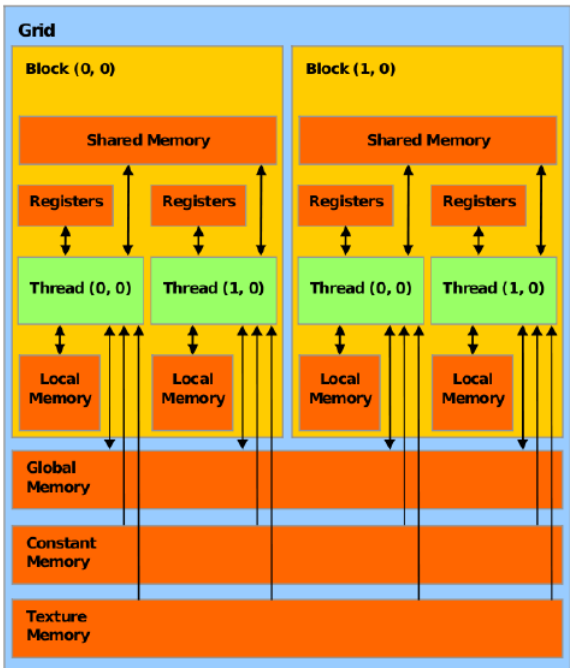

Figure 2.3: The CUDA memory model consists of multiple levels of memory spaces with increasing latency for data retrieval the further away from the execution threads that the memory space is located.

Chapter 3

Problem analysis

The configuration step is specific to each prototype pattern, and only needs to be done once for each specific pattern. All images containing the same pattern to be segmented can use the same configuration output in the application step. The configuration step can therefore be omitted from the parallel implementation. The input preprocessing step is also omitted, since the techniques used to preprocess the input image differ based on the particular application.

3.1 Identifying parallelism

The core algorithm consists of the steps described in 2.1. For one orientation, this becomes:

1. For each tuple, convolve the input image with a DoG-filter based on σ_i
2. For each tuple, perform max-blurring on the corresponding response from step (1.) based on ρ_i
3. For each tuple, shift the pixels according to (ρ_i, ϕ_i)
4. The COSFIRE output is the pixelwise geometric mean of the responses from (3.).

Tolerance to rotation is obtained by performing the steps above but modifying the parameter ϕ appropriately for each orientaton. The final response is then achieved by taking the pixelwise maximum of each orientation-specific response.

Considering the above, both potential task parallelism and data parallelism can be identified. It would be possible to compute the response of each contributing filter (i.e. tuple) separately in parallel, before gathering the result by geometric mean. Similarly, each orientation could be processed as an independent task, before gathering the result of all orientations by the operation of pixelwise maximum.

Massive data parallelism can be seen in the individual steps of the algorithm. The first step involves convolution, defined as

$$y(n_1, n_2) = \sum_{k_1=-r}^{r} \sum_{k_2=-r}^{r} I(n_1 - k_1, n_2 - k_2) f(k_1, k_2) \tag{3.1}$$

With other words, the response y for one pixel is obtained by fitting the center of the convolution kernel (filter) at the pixel under consideration, and then taking the sum of multiplying each pixel with the element in the kernel that is at the same position. To observe here is that each pixel can be processed in parallel. The same goes for step (2.) of the algorithm, which is a special case of convolution where instead of taking the sum of of the element-wise multiplication, you take the maximum.

Each pixel can also be individually processed in step (3.), as well as in (4.). It can be concluded that the algorithm contains both task parallelism as well as massive data parallelism.

3.2 Considerations

The goal of the implementation is to achieve a substantial speedup, i.e. reduce the runtime of the algorithm. Knowing that one bottleneck for general purpose computing is the bandwidth of the PCI bus between the CPU and GPU [5], it is desirable to keep the data transfers between the CPU and the GPU to a minimum.

Another thing to notice is that steps (1.) and (2.) of the core algorithm only need to be performed for each unique σ and each unique (σ, ρ)-pair respectively, since tuples that share these parameters can use the same response as input when proceeding to step (3.). In fact, the parameter σ is specific to each data set [3], so from here on we assume that σ is the same for each tuple. Thus, step (1.) only needs to be done once for each input image. Step (2.) needs to be done only as many times as there are unique parameters ρ in the set of tuples. This optimization is also implemented in the existing Matlab code [11].

The optimization above will reduce both the amount of computations and the memory usage. The latter is not the largest concern in this thesis, but it still needs to be economical enough in order for the GPU to have enough memory to accommodate for the intermediate response maps, considering that one response (double precision) of 565x584 pixels will take up around 2.6MB on the GPU.

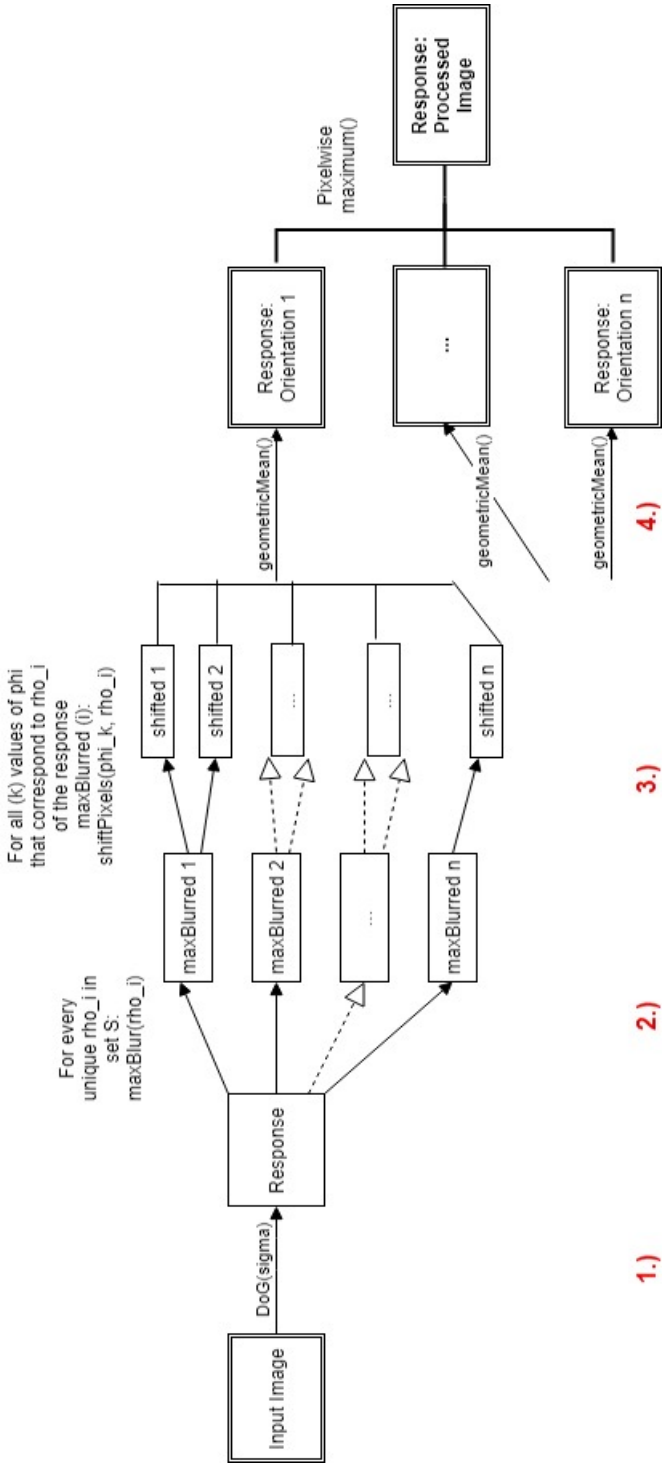

Figure 3.1: Diagram of optimized algorithm flow. The steps of the algorithm described in 2.1 are marked with red numbers

Chapter 4

Implementation

The implementation was performed incrementally by gradually integrating kernels into the Matlab code until the full program flow was implemented on the GPU. Initially, a version with both full task parallelism and data parallelism was attempted. This, however, became too memory expensive (exceeded memory capacity on a Quadro m620 GPU) as each stream (processing of tuple and orientation) needs separate copies in memory of each response. The implementation employed full dynamic parallelism in attempting this task. Due to the unreasonable strains on the hardware, it was decided to move the control flow back to the host code and launch the main kernels from there, focusing on data parallelism.

4.1 Design

The control of the algorithm was kept in the host code, with flow described in pseudocode by Algorithm 1. Each function call in this pseudo-code corresponds to a kernel being executed on the GPU, apart from computing the hashmap for mapping the responses from max-blurring to a corresponding tuple (line 8).

When applying multiple filters, typically a symmetric and a non-symmetric filter for detecting bar-like structures [3], the program was simply launched two times. The answers were then combined by pixelwise addition.

The program also has a minimal amount of data transfer between the CPU and the GPU: The input image is copied to the GPU in the beginning of the program, together with the tuples and necessary parameters. The full flow of the algorithm is performed without intermediate data transfer before retrieving the final response from the GPU.

Algorithm 1 Pseudo-code of CUDA implementation

procedure COSFIRE_CUDA(I, S, params) ▷ Image I, set of tuples S
    $I_{DoG} \leftarrow I \star DoG$
    $R \leftarrow uniqueRhos(S)$
    for `each` ρ_i `in` R **do**
        $G \leftarrow getGaussian(\rho_i)$
        $M_i \leftarrow I_{DoG} \star_{max} G$
    end for
    $hashmap \leftarrow mapToTuple(M, S)$
    for `each orientation` o_i `in params` **do**
        $rotstep \leftarrow i * rotationStep(params)$
        for `each tuple` t_j `in` S **do**
            $T_i \leftarrow shiftPixels(\rho_j, \psi_j + rotstep, M[hashmap[j]])$
        end for
        $O_i \leftarrow geometricMean(T)$
    end for
    $Resp \leftarrow pixelwiseMax(O)$
    return $Resp$ ▷ Rotation-invariant response
end procedure

4.2 Kernel design

All kernels employ full data parallelism, so one thread takes care of calculating the result for one pixel. In order to do this, two-dimensional CUDA thread blocks are designed that fits the processing of two-dimensional images. A block size of 32x32 is chosen, and as many blocks as needed to cover the whole image in both x- and y-direction is requested for each kernel launch.

In the kernels each thread calculates which pixel that they are assigned to by consulting runtime variables indicating the position of each thread in their block and position of each block in the grid respectively. This together with runtime information about the dimensions of the thread and grid blocks enables calculating a global ID for each thread.

The threads then execute the calculation for their specific pixel and writes it to the corresponding place in an output buffer. Many of the kernels launched need neighboring values of pixels for calculating the output, which is why in-place modifications are not used.

All responses are also represented as 1D instead of 2D arrays in order to speed up memory access [6].

4.3 Integration into Matlab code

Since the configuration and preprocessing is done in Matlab, a mex-function is needed in order to connect the Matlab program with the CUDA code. This is a wrapper that loads in variables from Matlab whereafter parsing they can be used as regular C or CUDA variables. Since Matlab stores all variables in column major order [8] and the kernels are written to expect row major order such as in standard C, the input image is transposed before being passed to the mex wrapper. Similarly, the output obtained from the mex wrapper is transposed before it is finally displayed.

4.4 Considerations

The implementation assumes that some specific minimum amount of memory is available to hold all the buffers for subresponses of the filters, since the code is not particularly optimized memory wise. This could be improved by adding automatic checking for hardware constraints or creating a memory-optimized version. Furthermore, there are some hard coded parameters related to buffer sizes in sub routines that need to be adapted in order to account for processing of input data of different size efficiently.

Chapter 5

Results

Matlab2018a was used together with CUDA Toolkit 9.0, Cuda driver version 9.2 and the experiments were ran on a PC with an Intel(R) Xeon(R) E3-1225 v5 @ 3.30GHz CPU (4 cores) for testing the sequential version. The experiments for the parallelized CUDA version were ran on a NVIDIA GTX 1080 ti GPU (3584 cores). Hardware specifications can be viewed in the appendix, as well as descriptions of the three data sets that were used.

5.1 Accuracy

The accuracy of the CUDA implementation is verified to ensure correctness of the implementation. The obtained results are compared to accuracies reported in [3], where measurements for accuracy (Acc), sensitivity (Se) and specificity (Sp) are reported according to

$$Acc = \frac{TP + TN}{N}, \quad Se = \frac{TP}{TP + FN}, \quad Sp = \frac{TN}{TN + FP} \tag{5.1}$$

TP (true positive) describes the amount of correctly classified vessel pixels, TN (true negative) the amount of correctly classified non-vessel pixels, FN (false negative) the amount of incorrectly classified vessel pixels and FP (false positive) the amount of incorrectly classified non-vessel pixels.

The results obtained when comparing test images from the DRIVE data set (after being processed with CUDA COSFIRE) with the corresponding ground truth image (extension manual1.gif) can be viewed in Table 5.2.

This can be compared to the corresponding metrics from [3], with average values on the same data set being $Acc = 0.9427, Se = 0.7526, Sp = 0.9707$ (see Table 5.1). The average values when running the CUDA version with the second ground truth images, (extension manual2.gif) are similar as to the first ground truth image with $Acc = 0.96466, Se = 0.79455$ and $Sp = 0.98041$. It can be concluded that the CUDA implementation is as accurate as the sequential version.

Accuracy	Sensitivity	Specificity
0.9442	0.7655	0.9704

Table 5.1: Accuracy of sequential implementation (avg measurements on DRIVE)

	Accuracy	Sensitivity	Specificity
01_test.tif	0.96025	0.85819	0.97025
03_test.tif	0.96475	0.80047	0.98349
03_test.tif	0.9517	0.81704	0.96661
04_test.tif	0.9652	0.71697	0.99035
05_test.tif	0.9616	0.65874	0.9929
06_test.tif	0.96002	0.7321	0.98459
07_test.tif	0.96158	0.72052	0.98583
08_test.tif	0.95893	0.71433	0.98196
09_test.tif	0.96427	0.79687	0.97903
10_test.tif	0.96458	0.75044	0.98378
11_test.tif	0.95816	0.77322	0.97635
12_test.tif	0.9557	0.53619	0.99535
13_test.tif	0.9593	0.74162	0.98289
14_test.tif	0.96188	0.83458	0.97308
15_test.tif	0.96339	0.77331	0.97804
16_test.tif	0.96405	0.79621	0.9807
17_test.tif	0.96267	0.7556	0.98176
18_test.tif	0.95894	0.86452	0.96707
19_test.tif	0.96991	0.85499	0.9803
20_test.tif	0.9648	0.84574	0.97425
Average	**0.96158**	**0.76708**	**0.98043**

Table 5.2: Accuracy metrics of CUDA implementation (images from DRIVE dataset). This is put in comparison with results from [3] (see Table 5.1) in order to verify the results of the parallelized version.

5.2 Performance

The publicly available data sets DRIVE, CHASE_DB1 and HRF have been used for the experiments (see the Appendix). All data sets were processed with the filter parameters that were found optimal in [3], apart from HRF, for which the parameters were retrieved from [13]. The filter parameters can vary per data set e.g. in the number of tuples or number of orientations. This is an explanation to why the processing speed is not exactly the same per megapixel for each data set (see Figure 5.2). All experiments have been performed as a combination of symmetric (for detecting bars) and asymmetric (for detecting bar endings) filter responses, in order to achieve the most authentic setting. The runtimes for the individual sections of the program were achieved with the profiling tool nvprof. All measurements are reported as the average of 100 runs. For full output from the profiling tool, see the Appendix.

Dataset	Resolution	Cuda	Sequential	Speedup
DRIVE	565x584	0.0526 (0.0020)	1.2277 (0.0241)	23.34
CHASE_DB1	999x960	0.1880 (0.0027)	5.3216 (0.1247)	28.31
HRF	3504x2336	1.009 (0.0366)	26.5469 (0.6872)	26.31

Table 5.3: Average runtime in seconds, CUDA vs Sequential version. Standard deviation of 100 runs for each version is reported within brackets.

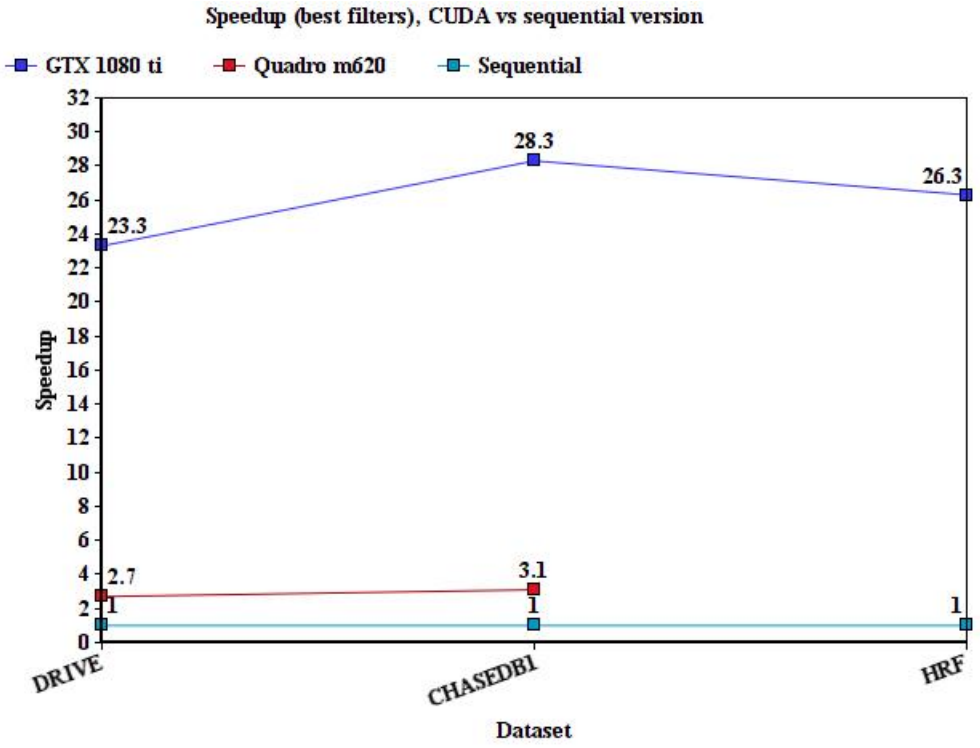

Figure 5.1: Speedup of CUDA versus sequential implementation with COSFIRE filter configured with parameters that were found most suitable to each data set.

Table 5.3 reports the results obtained when measuring the total run time of the algorithm for the three data sets respectively. This is put in comparison to the sequential version. The standard deviation of 100 runs is very small, varying between 0.002 for DRIVE and 0.03 for HRF, indicating that the run time is stable. The speedup varies between 23-28, which can also be seen in Figure 5.1.

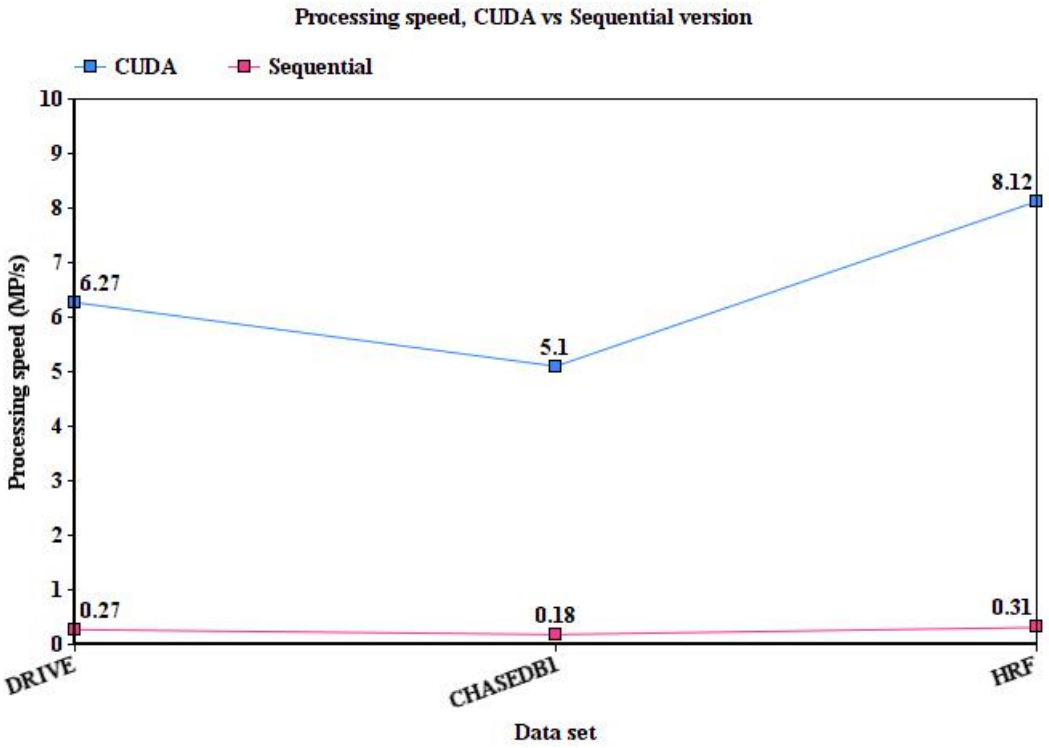

Figure 5.2: Comparison of processing speed in megapixels per second. The parameters denoting the COSFIRE filter vary somewhat for each data set, which is why the relative processing speed is not constant for each data set.

Dataset	getDoG(1.)	MaxBlur(2.)	Shift(3.)+ G. Mean(4.)	Max
DRIVE	0.08	4.70	32.16	0.28
CHASE_DB1	0.70	22.57	111.65	0.81
HRF	0.16	75.84	642.90	4.69

Table 5.4: Average runtime in milliseconds of individual steps of the algorithm (CUDA version)

Table 5.4 indicates the average run time for each individual part of the core algorithm, when processing one image. It can be seen that step (3.) shifting and geometric mean (4.) together by far consume the largest portion of the total run time. This is not surprising, since this is the part of the algorithm that needs to produce the largest amount of responses (see Figure 3.1). Similarly, step (1.)

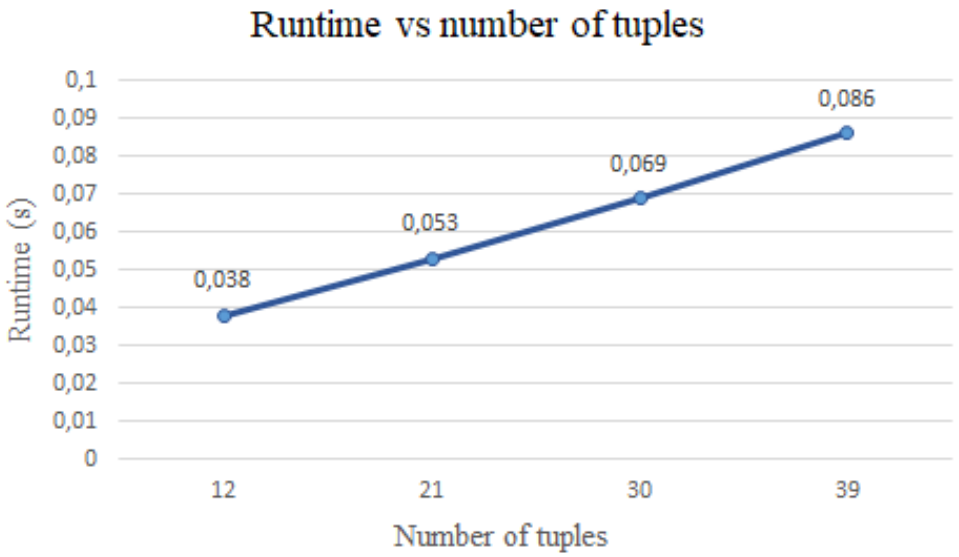

Figure 5.3: Average runtime (DRIVE) with a fixed number of orientations (12+24) and varying number of tuples

(image is convoluted with a difference-of-Gaussians filter) is always done once only, and hence this step becomes a very small fraction of the total execution time.

Next, it is investigated how the runtime is affected by changing the number of tuples and number of orientations respectively. These are parameters that will affect the runtime of the program, since with an increased number of tuples the number of computations for each specific orientation will increase, and with an increased number of orientations the program will have to do more iterations of the core algorithm. These experiments have been carried out only on the DRIVE data set.

No. tuples	No. orientations	runtime (avg)
12 (5+7)	12+24	0.038
21 (9+12)	12+24	0.053
30 (13+17)	12+24	0.069
39 (17+22)	12+24	0.086

Table 5.5: Average runtime for DRIVE with a varying number of tuples. The number of tuples and number of orientations are described as the sum of the parameters of the symmetric and asymmetric filter applied respectively.

Within the limits of the measured data points, the runtime of the program increases linearly when increasing the number of tuples (see Figure 5.3). This is hardly surprising, since adding a fixed amount of tuples will produce a fixed increase in responses produced in step (2.) and (3.) of the algorithm. This will

be the case for each orientation under consideration, while the parameter number of orientations was fixed in this experiment. The detailed output can be viewed in Table 5.5.

No. tuples	No. orientations	runtime(s)	shiftpixels vs total (%)
21 (9+12)	18 (6+12)	0.036	50,71
21 (9+12)	36 (12+24)	0.053	59,10
21 (9+12)	72 (24+48)	0.088	68,18
21 (9+12)	108 (36+72)	0.123	70,91
21 (9+12)	144 (48+96)	0.159	72,38

Table 5.6: Average runtime for DRIVE with a varying number of orientations

Increasing the number of orientations also leads to increasing the runtime in a seemingly linear manner (see Figure 5.4). This is expected, as each added orientation in total will produce an increase in responses from step (4.) equalling the number of tuples. It can also be seen that the portion of the operation for shifting pixels in comparison to the total execution time increases as the number of orientations are increased, however in a more converging manner. These numbers can be viewed in Table 5.6.

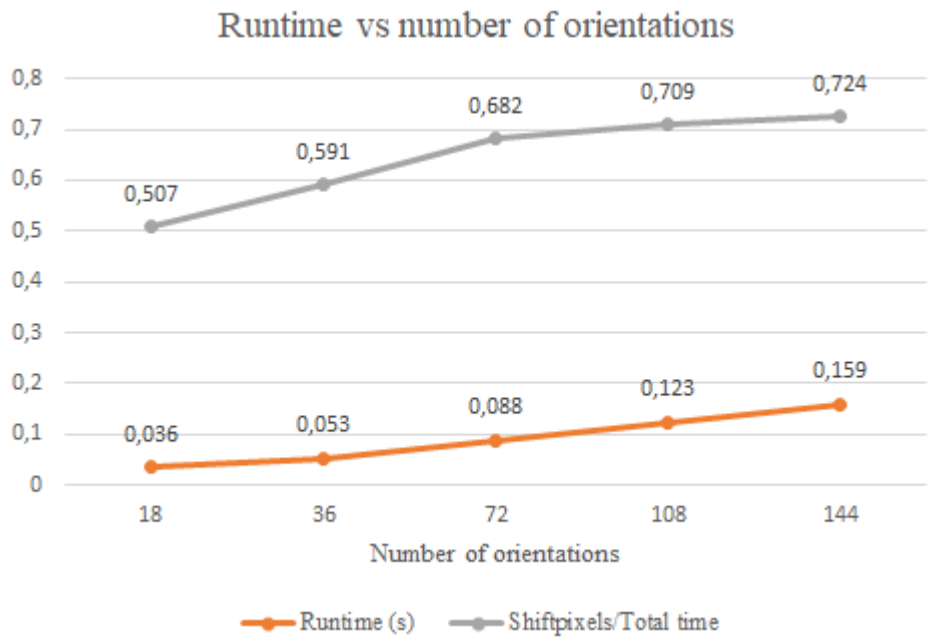

Figure 5.4: Average runtime (DRIVE) with a fixed number of tuples (9+12) and varying number of orientations. The ratio of runtime of step (4.), shifting versus total runtime is displayed in grey.

Figure 5.5: Response image from a bar-selective COSFIRE filter, using Figure 6.1 from DRIVE as input.

Chapter 6

Discussion

6.1 Analysis of results

The speedup achieved is substantial varying between 23-28 (see Table 5.3), even though the theoretical maximum achievable speedup is much higher. The latter is due to that the implementation processes each pixel in parallel for each step of the algorithm. This in comparison to a version where each pixel and each step is processed sequentially. The images in the data sets covered in this work contain in the order of 10^7 pixels. On an ideal system with an infinite amount of threads and no hardware latency, the theoretical speedup would be in the order of O(p) where p is the number of pixels. The sequential Matlab version is however internally not completely sequential, since Matlab uses multi-threading for many of its built in functions [8].

The most limiting factor for the measured speedup is likely to be the amount of cores on the GPU device that is used. This can also be observed in Figure 5.1, where two different GPUs were compared for speedup. On a Quadro m620 with 512 cores a speedup of 2.7 is measured on the DRIVE data set. The same speedup is measured to be 23.3 on a GTX 1080 ti with 3584 cores, i.e seven times more cores. The ratio between the two speedups is 8.6, which is noted to be even larger than the expected ratio of 7 which is proportional to the ratio of number of cores. Due to this relationship, we can expect the speedup to keep improving with my cores available, especially since the better performing GPU still has a number of cores that is far from the number of pixels in a single image.

Another thing that slows down the CUDA implementation is latency in different forms: transferring data to and from the GPU, latency in kernel launches and queuing thread blocks waiting to get executed [6]. The implementation also spends a large portion of time on the API call `cudaDeviceSynchronize` that blocks until the device has completed all preceding requested tasks. The implementation could potentially be further improved by optimizing the use of this call.

The speedup could also be improved by more refined memory access strategies. Currently all input, responses and output are stored in global memory. This is far from optimal since it is one of the memory spaces with slowest access in the hierarchy [1].

As reported in Table 5.5 and Table 5.6, increasing the number of tuples or number of orientations will result in a proportional increase of the runtime. At the same time, these are parameters that denote the complexity of the prototype pattern and an increased precision when searching for rotations of the prototype pattern respectively. This means that that the CUDA version can also be used to increase the complexity or versatility of the prototype pattern compared to the sequential version, since you can increase the number of these parameters manyfold before the program will reach the same runtime as the sequential version.

6.2 Conclusion

The goal of this project was to improve the runtime performance by parallelizing the COSFIRE algorithm with CUDA. With an obtained speedup of 23, 25 and 28 on different datasets (see Figure 5.1) it can be concluded that a satisfactory speedup has been achieved, even though the speedup was proven to be dependant on hardware resources or number of cores available on the GPU.

There is room for improvement especially by going deeper into the details of memory access speed, optimizing details or even enabling the program flow for multiple devices. However, the achieved implementation has proved to produce a good solution to the general core part of the problem and it is always possible to continue the work from here.

Bibliography

[1] Carlos Aguilar Melchor et al. "High-Speed Private Information Retrieval Computation on GPU". In: (Sept. 2008), pp. 263–272.

[2] George Azzopardi and Nicolai Petkov. "Trainable COSFIRE filters for keypoint detection and pattern recognition". In: *IEEE Trans. Pattern Anal. Mach. Intell* 35.2 (2013), pp. 490–503.

[3] George Azzopardi et al. "Trainable COSFIRE filters for vessel delineation with application to retinal images". In: *Medical Image Analysis* 19.1 (2015), pp. 46–57. ISSN: 1361-8415. DOI: https://doi.org/10.1016/j.media.2014.08.002. URL: http://www.sciencedirect.com/science/article/pii/S1361841514001364.

[4] Attila Budai et al. "Robust Vessel Segmentation in Fundus Images". In: *International Journal of Biomedical Imaging* 2013 (2013). DOI: 10.1155/2013/154860. URL: http://www5.informatik.uni-erlangen.de/Forschung/Publikationen/2013/Budai13-RVS.pdf.

[5] Shuai Che et al. *A Performance Study of General-Purpose Applications on Graphics Processors Using CUDA*.

[6] NVIDIA corporation. *Cuda Toolkit Documentation - v9.2.88*. 2018. URL: https://docs.nvidia.com/cuda (visited on 06/07/2018).

[7] Ellen C Hildreth. "The detection of intensity changes by computer and biological vision systems". In: *Computer Vision, Graphics, and Image Processing* 22.1 (1983), pp. 1–27. ISSN: 0734-189X. DOI: https://doi.org/10.1016/0734-189X(83)90093-2. URL: http://www.sciencedirect.com/science/article/pii/0734189X83900932.

[8] Mathworks inc. *Matlab Documentation*. 2018. URL: https://se.mathworks.com/help (visited on 07/19/2018).

[9] Christopher G. Owen et al. "Measuring Retinal Vessel Tortuosity in 10-Year-Old Children: Validation of the Computer-Assisted Image Analysis of the Retina (CAIAR) Program". In: *Investigative Ophthalmology Visual Science* 50.5 (2009), p. 2004. DOI: 10.1167/iovs.08-3018. eprint: /data/journals/iovs/932959/z7g00509002004.pdf. URL: +%20http://dx.doi.org/10.1167/iovs.08-3018.

[10] J.J. Staal et al. "Ridge based vessel segmentation in color images of the retina". In: *IEEE Transactions on Medical Imaging* 23.4 (2004), pp. 501–509.

[11] Nicola Strisciuglio. *Matlab implementation of the B-COSFIRE filters for detection of curvilinear patterns in images*. 2018. URL: https://gitlab.com/nicstrisc/B-COSFIRE-MATLAB (visited on 05/15/2018).

[12] Nicola Strisciuglio and Nicolai Petkov. "Delineation of line patterns in images using B-COSFIRE filters". In: *2017 International Conference and Workshop on Bioinspired Intelligence (IWOBI)* (2017), pp. 1–6.

[13] J.M.R.S. Tavares and R.M.N. Jorge. *Computational Vision and Medical Image Processing V: Proceedings of the 5th Eccomas Thematic Conference on Computational Vision and Medical Image Processing (VipIMAGE 2015, Tenerife, Spain, October 19-21, 2015)*. CRC Press, 2015, pp. 149–155. ISBN: 9781315642796. URL: https://books.google.nl/books?id=1fEYCwAAQBAJ.

List of Figures

1.1 COSFIRE input and output 2

2.1 The tuples and their corresponding points of interest 6
2.2 CUDA flow 9
2.3 CUDA memory model 9

3.1 Flow of optimized algorithm 12

5.1 Speedup 18
5.2 Processing speed, best filters 19
5.3 Runtime, varying number of tuples 20
5.4 Runtime, varying number of orientations 22
5.5 Example of COSFIRE output 22

6.1 DRIVE - 01_test.tif 30
6.2 CHASEDB1 - Image_01L 30
6.3 HRF - 01_h.jpg 31

List of Tables

5.1 Accuracy, sequential implementation 17
5.2 Accuracy, CUDA implementation 17
5.3 Average runtime, CUDA vs Sequential 18
5.4 Runtime of individual steps 19
5.5 Runtime - varying number of tuples 20
5.6 Runtime - varying number of orientations 21

Appendix

Hardware specification

GPU Engine Specs - GTX 1080 ti
CUDA Cores 3584
Graphics Clock (MHz) 1480
Processor Clock (MHz) 1582
Memory Specs Standard Memory Config 11 GB GDDR5X
Memory Interface Width 352-bit
Memory Bandwidth (GB/sec) 11 Gbps

Code

All code is available from the Github repository https://github.com/SofieLovdal/COSFIRE_CUDA.

Data sets

For this work the publicly available data sets DRIVE [10], CHASEDB1 [9] and HRF[4] have been used. An example image from each data set is included below for reference, even though more images of the same resolution from each data set have been used.

Figure 6.1: Image 01_test.tif from DRIVE (584x565 pixels)

Figure 6.2: Image_01L.jpg from CHASEDB1 (999x960 pixels)

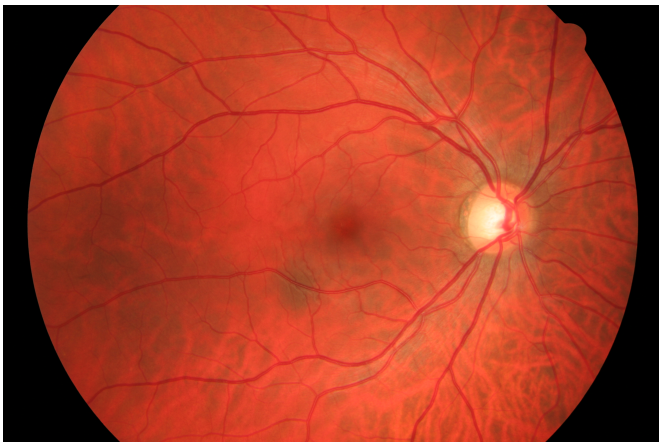

Figure 6.3: 01_h.jpg from CHASEDB1 (3504x2336 pixels)

Profiling of kernel calls

DRIVE

```
==28899== Profiling application: ./cWrapper
==28899== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 61.09%  2.52743s     39996  63.192us  60.674us  826.87us  shiftPixels(double*,
double*, unsigned int, unsigned int, double, double)
 17.43%  721.13ms      3636  198.33us  192.14us  241.74us  geometricMean(double*,
double*, unsigned int, unsigned int, int, double)
 11.09%  458.76ms      1717  267.19us  73.570us  980.22us  maxBlur(double*,
double*, unsigned int, unsigned int, double*, unsigned int, unsigned
int)
  4.91%  203.33ms       202  1.0066ms  786.97us  1.5481ms  conv2(double*,
double*, unsigned int, unsigned int, double*, unsigned int, unsigned
int)
  3.02%  124.86ms       202  618.12us  592.76us  705.56us  [CUDA memcpy
DtoH]
  1.18%  48.622ms       404  120.35us     608ns  310.83us  [CUDA memcpy
HtoD]
  0.69%  28.582ms       202  141.50us  97.091us  195.59us  pixelwiseMax(double*,
double*, unsigned int, unsigned int, int)
  0.38%  15.736ms      1717  9.1640us  8.5440us  11.969us  getGaussian(double*,
double)
  0.20%  8.4660ms       202  41.910us  27.489us  72.674us  getDoG(double*,
double, double)

==28899== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 66.33%  3.60854s      7474  482.81us  4.9630us  1.6700ms  cudaDeviceSynchronize
 13.01%  707.87ms      1616  438.04us  96.121us  2.0887ms  cudaFree
  7.37%  400.86ms       606  661.49us  58.648us  1.8390ms  cudaMemcpy
  6.93%  377.22ms      1616  233.43us  108.21us  177.08ms  cudaMalloc
  5.23%  284.61ms     47672  5.9700us  5.3650us  388.47us  cudaLaunch
  0.89%  48.306ms    280275     172ns     144ns  373.46us  cudaSetupArgument
  0.21%  11.458ms     47672     240ns     181ns  3.5570us  cudaConfigureCall
  0.01%  477.49us      1616     295ns     185ns     936ns  cudaGetLastError
  0.01%  388.47us        91  4.2680us     139ns  171.09us  cuDeviceGetAttribute
  0.00%  170.06us         1  170.06us  170.06us  170.06us  cuDeviceTotalMem
  0.00%  43.878us         1  43.878us  43.878us  43.878us  cuDeviceGetName
  0.00%  1.9590us         3     653ns     151ns  1.4670us  cuDeviceGetCount
  0.00%  1.4540us         3     484ns     183ns     877ns  cuDeviceGet
```

CHASEDB1

```
==29638== NVPROF is profiling process 29638, command: ./cWrapper
programPart = 0, mean = 0.188008, std = 0.002659, max = 0.212434, min
= 0.186343
timings[0]: 0.636350
==29638== Profiling application: ./cWrapper
==29638== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 55.91%  9.10803s     52116  174.76us  168.49us  959.74us  shiftPixels(double*,
double*, unsigned int, unsigned int, double, double)
 14.00%  2.28051s      3434  664.10us  204.26us  1.6661ms  maxBlur(double*,
double*, unsigned int, unsigned int, double*, unsigned int, unsigned
int)
 13.31%  2.16866s      3636  596.44us  508.14us  1.0441ms  geometricMean(double*,
double*, unsigned int, unsigned int, int, double)
 10.77%  1.75397s       202  8.6830ms  8.2163ms  10.901ms  conv2(double*,
double*, unsigned int, unsigned int, double*, unsigned int, unsigned
int)
  3.71%  604.45ms       202  2.9923ms  2.7979ms  3.3326ms  [CUDA memcpy
DtoH]
  1.18%  191.44ms       404  473.86us     640ns  1.0957ms  [CUDA memcpy
HtoD]
  0.50%  81.587ms       202  403.90us  275.11us  550.38us  pixelwiseMax(double*,
double*, unsigned int, unsigned int, int)
  0.44%  71.644ms       202  354.67us  349.93us  443.25us  getDoG(double*,
double, double)
  0.19%  30.545ms      3434  8.8940us  8.2240us  11.872us  getGaussian(double*,
double)

==29638== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 77.78%  14.9658s     10908  1.3720ms  9.8350us  10.904ms  cudaDeviceSynchronize
 10.51%  2.02131s      1616  1.2508ms  7.1380us  3.3272ms  cudaFree
  5.99%  1.15179s       606  1.9006ms  11.767us  5.1613ms  cudaMemcpy
  3.61%  694.30ms      1616  429.64us  8.3350us  284.40ms  cudaMalloc
  1.72%  330.67ms     63226  5.2290us  4.7010us  377.82us  cudaLaunch
  0.33%  63.935ms    368448     173ns     142ns  383.58us  cudaSetupArgument
  0.06%  12.043ms     63226     190ns     157ns  369.59us  cudaConfigureCall
  0.00%  413.24us      1616     255ns     141ns  1.2600us  cudaGetLastError
  0.00%  361.81us        91  3.9750us     141ns  158.50us  cuDeviceGetAttribute
  0.00%  169.85us         1  169.85us  169.85us  169.85us  cuDeviceTotalMem
  0.00%  43.508us         1  43.508us  43.508us  43.508us  cuDeviceGetName
  0.00%  1.6420us         3     547ns     153ns  1.2510us  cuDeviceGetCount
  0.00%  1.2290us         3     409ns     156ns     582ns  cuDeviceGet
```

HRF

```
==30154== NVPROF is profiling process 30154, command: ./cWrapper
programPart = 0, mean = 1.009240, std = 0.036559, max = 1.197336, min
= 0.983324
timings[0]: 1.277089
==30154== Profiling application: ./cWrapper
==30154== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 59.10%  52.1809s     49692  1.0501ms  1.0029ms  1.9002ms  shiftPixels(double*,
double*, unsigned int, unsigned int, double, double)
 14.44%  12.7479s      3636  3.5060ms  3.2063ms  4.4430ms  geometricMean(double*,
double*, unsigned int, unsigned int, int, double)
  8.66%  7.64201s      2121  3.6030ms  1.2046ms  9.5814ms  maxBlur(double*,
double*, unsigned int, unsigned int, double*, unsigned int, unsigned
int)
  7.45%  6.57409s       202  32.545ms  28.379ms  138.31ms  [CUDA memcpy
DtoH]
  5.34%  4.71799s       202  23.356ms  20.839ms  32.222ms  conv2(double*,
double*, unsigned int, unsigned int, double*, unsigned int, unsigned
int)
  4.44%  3.92113s       404  9.7058ms     640ns  36.978ms  [CUDA memcpy
HtoD]
  0.54%  473.58ms       202  2.3445ms  1.6031ms  3.0925ms  pixelwiseMax(double*,
double*, unsigned int, unsigned int, int)
  0.02%  20.202ms      2121  9.5240us  8.8000us  12.288us  getGaussian(double*,
double)
  0.02%  16.175ms       202  80.073us  65.985us  123.33us  getDoG(double*,
double, double)

==30154== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 74.82%  76.3095s      8282  9.2139ms  4.5750us  32.395ms  cudaDeviceSynchronize
 11.93%  12.1706s       606  20.084ms  32.070us  156.61ms  cudaMemcpy
  7.88%  8.03642s      1616  4.9730ms  16.710us  169.77ms  cudaMalloc
  4.98%  5.07571s      1616  3.1409ms  8.1640us  27.194ms  cudaFree
  0.31%  312.58ms     58176  5.3730us  4.3030us  4.7022ms  cudaLaunch
  0.06%  60.965ms    342087     178ns     142ns  1.4687ms  cudaSetupArgument
  0.01%  14.696ms      1616  9.0940us     138ns  13.993ms  cudaGetLastError
  0.01%  12.792ms     58176     219ns     160ns  363.07us  cudaConfigureCall
  0.00%  362.43us        91  3.9820us     145ns  158.29us  cuDeviceGetAttribute
  0.00%  170.26us         1  170.26us  170.26us  170.26us  cuDeviceTotalMem
  0.00%  44.236us         1  44.236us  44.236us  44.236us  cuDeviceGetName
  0.00%  2.2570us         3     752ns     174ns  1.6970us  cuDeviceGetCount
  0.00%  1.1720us         3     390ns     185ns     559ns  cuDeviceGet
```