

# **„Performance Measurements of Algorithms in Image Processing“**

## **Bachelor Thesis**

Department of Computer Science  
University of Applied Science Rapperswil

Spring Term 2011

Authors:	Tobias Binna, Markus Hofmann
Advisor:	Prof. Josef M. Joller
External Co-Examiner:	Dr. Hans J. Grossmann



# Abstract

How much can image processing algorithms be parallelized? In this project we implement image processing algorithms in a massively parallel manner using NVIDIA CUDA. Furthermore we analyze the resulting performance gains against current CPU implementations.

Image processing algorithms are often quite complex and can quickly become very compute intensive tasks. NVIDIA has evolved a technology called CUDA which is an extension to the C programming language and provides the opportunity of general purpose computing on NVIDIA graphics cards, using thousands of concurrent threads. Thus the use of CUDA in image processing is obvious and the potential performance benefits shall be investigated here in detail.

In this project we implemented two algorithms with CUDA, namely the Hough transform to extract lines from an image and a template matching algorithm to find a given pattern in a search image. We analyzed the performance of our implementations and compared the results with reference implementations from the OpenCV library.

Our results will not only focus on performance issues, but will also give information about the scalability of algorithms using different graphical processing units (GPU). Additionally we elaborate on hardware limitations and our experiences made during the CUDA implementations.



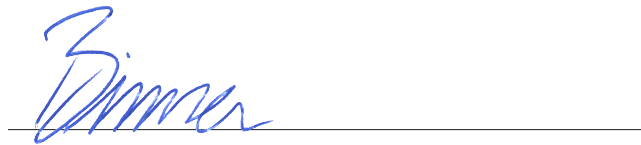


# Declaration

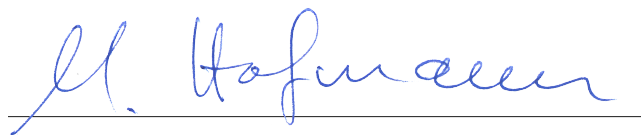
We, Tobias Binna and Markus Hofmann declare

- that this term project and the work presented in it is our own, original work.
- All the sources we consulted and cited are clearly attributed. We have acknowledged all main sources of help.

Rapperswil, June 16, 2011



Tobias Binna



Markus Hofmann



# Acknowledgments

First we would like to thank our supervisor, Prof. Josef M. Joller, for his support during our thesis. We appreciated to always have a free hand in realizing our ideas.

Many thanks also to our co-examiner Dr. Hans J. Grossmann who gave us valuable suggestions in the search of suitable algorithms.

A special thanks goes to Prof. Oliver Augenstein who provided us a helping hand and gave us useful explanations on mathematical and computational issues.

Last but not least we would like to thank our relatives and friends for their constant encouragements in this intensive time.



# Contents

1	Management Summary	1
1.1	Introduction . . . . .	1
1.2	Approach . . . . .	1
1.3	Results . . . . .	2
2	Technical Introduction	3
2.1	Problem Domain . . . . .	3
2.2	Image Processing Algorithms . . . . .	3
2.3	Implementing Algorithms with CUDA . . . . .	3
3	Implemented Algorithms	5
3.1	Hough Transform . . . . .	5
3.1.1	Problem Domain . . . . .	5
3.1.2	Method . . . . .	9
3.1.3	Results . . . . .	14
3.2	Template Matching . . . . .	22
3.2.1	Problem Domain . . . . .	22
3.2.2	Method . . . . .	26
3.2.3	Results . . . . .	29
4	Conclusions	35
A	Development Environment Setup	39
A.1	Hardware and Software Tools . . . . .	39
A.1.1	Hardware . . . . .	39
A.1.2	Software . . . . .	40
A.2	Setup CUDA . . . . .	40
A.3	Setup Eclipse CDT with CMake and CUDA . . . . .	40
A.3.1	Add File Type .cu . . . . .	41
A.3.2	Setup CMake and CMake Configuration in Eclipse . . . . .	41
	Bibliography	43



## List of Figures

3.1	Hessian Normal Form . . . . .	7
3.2	Hough Transformation with resulting Hough Space . . . . .	8
3.3	Illustration of the Additive Hough Transform . . . . .	9
3.4	Steps of the implemented Hough Transform Algorithm . . . . .	10
3.5	Room Edge Map with Thinning Factor 1 and 5 . . . . .	12
3.6	Calculation of Bin Sizes . . . . .	13
3.7	Room Image and its Edge Map . . . . .	15
3.8	Example of bad Edge Map . . . . .	15
3.9	Hough Transform Results with Thinning Factor 1 and 5 . . . . .	16
3.10	Hough Transform Results with Thinning Factor 10 and different Thresholds . . . . .	17
3.11	Parameter Spaces with detected Lines . . . . .	18
3.12	Comparison between Parameter Space and Output Image . . . . .	18
3.13	Runtime Measurements of the Standard Hough Transform . . . . .	19
3.14	Runtime Measurements of the Additive Hough Transform . . . . .	21
3.15	Conceptual Overview of Template Matching . . . . .	23
3.16	Computation of a Summed Area Table . . . . .	28
3.17	Template Matching Search Image and Reference Images . . . . .	30
3.18	Template Matching Output Images with small and large Templates . . . . .	31
3.19	Runtime Measurements of Template Matching . . . . .	33
3.20	Template Matching with Test Image to visualize numerical Artifacts . . . . .	34
4.1	CPU and CUDA Runtime with logarithmic Scale . . . . .	35
4.2	Scalable CUDA Implementations on different Graphics Devices . . . . .	36





## List of Tables

3.1	Runtime Example with Standard Hough Transform on the Room Image	18
3.2	Runtime Example with Additive Hough Transform on the Room Image	20
3.3	Runtimes of Template Matching Parts . . . . .	32
3.4	Template Matching Runtimes with small and large Templates . . . . .	32
A.1	Low-cost Project Hardware . . . . .	39
A.2	High-end Project Hardware . . . . .	39
A.3	Project Software . . . . .	40



# Listings

3.1	A trivial Template Matching Kernel . . . . .	26
A.1	Check for FindCUDA.cmake . . . . .	41
A.2	CMake CUDA Executable . . . . .	41



# 1 Management Summary

In this project, we analyze the performance gain of parallelized image processing algorithms using NVIDIA CUDA. The project was initiated by the Institute for Internet-Technologies and Applications at the University of Applied Sciences Rapperswil.

## 1.1 Introduction

The fact that the increased complexity of computable problems desires faster hardware is by far not new. Today's demand on computing power can no longer be satisfied by just increasing the clock speed of CPUs. Hence, the approach to parallelize things has become more and more popular these days. NVIDIA provides a technology called CUDA which is an extension to the C programming language. This technology allows programmers to do computations on a CUDA capable graphics card (almost all newer NVIDIA graphics cards are CUDA enabled) in a massively parallel way, using thousands of concurrent threads.

Image processing algorithms are often quite complex, because computers are not capable to logically interpret and extract information as seen by human beings and therefore often need a lot of steps to manage computer vision tasks. Due to this, image processing algorithms can become very compute intensive and time consuming.

Therefore the idea to use graphical processing units (GPU) to parallelize complex image processing algorithms should be investigated. Because in some situations CPU implementations can still be faster than a CUDA version, the performance results need to be analyzed in detail. It is important to get a good understanding when CUDA implementations make sense.

## 1.2 Approach

Initially we selected two different computer vision algorithms and tested their suitability for a parallel CUDA implementation. We decided to focus on the Hough transform, an algorithm to detect lines in an image and a template matching algorithm to find a given pattern in a search image. In a following step we made performance measurements of these algorithms on the CPU using reference implementations from the OpenCV library.

To make the transition from the sequential CPU implementation to a parallel GPU version we analyzed the algorithms and searched for parts which can possibly be parallelized. Afterwards we developed the algorithms in CUDA in an iterative process of implementation, performance analysis and performance tuning.

Finally we compared the performance measurements of the GPU implementations with their CPU references and interpreted the obtained outcomes.

### 1.3 Results

Our results show tremendous performance gains by using CUDA. The two implemented versions of Hough transform both performed better than their CPU counterparts. An even better result could be observed with the template matching implementation as we achieved a performance speedup of a factor of 31.

Besides the performance evaluation we made a lot of experiences on how to develop CUDA applications and learned about some important points that have to be observed when dealing with GPU programming. Specifically high performance CUDA implementations do not only depend on a clever realization, it is equally important to consider the properties and limitations of the underlying hardware.

## 2 Technical Introduction

### 2.1 Problem Domain

With the increased complexity of computing problems the desire of faster hardware increases as well. To be able to solve more and more complex problems, programmers must have options to get more power out of their machines. However this additional performance can no longer be achieved by just increasing the clock speed of today's processors, because of physical limits. As a result the industry is heading towards parallel programming concepts and multi-core architectures. Even if multi-core architectures are solutions to further increase computing power they involve new challenges in software development. Parallel implementations require new programming concepts and demand the developers to care about additional resource management and coordination.

### 2.2 Image Processing Algorithms

The demand on more computing power is especially present in the area of image processing, as images get larger or high frame rates desire faster processing. In addition, image processing algorithms are often very compute intensive by itself because of their complexity. This complexity is a result of the fact that computers are not capable to logically interpret and extract information as seen by human beings and therefore often need a lot of steps to manage computer vision tasks. Because of this circumstances we have to find ways to implement such algorithms in a parallel manner.

### 2.3 Implementing Algorithms with CUDA

In this thesis we will deal with the implementation of image processing algorithms, using NVIDIA's CUDA technology which provides the opportunity to do general purpose computations on a CUDA capable graphics card in an massively parallel way. Additional information about CUDA can be found in our term project "Massive Parallel Image Processing" from fall 2010 [1].

The goal of this project is to determine the performance gain when image processing algorithms are implemented in CUDA and have decided to implement two specific algorithms. On the one hand we chose the Hough transform, an algorithm to detect lines in an image while on the other hand we developed a template matching algorithm to find a given pattern in a search image. The comparison of the time measurements against CPU reference implementations should give information and experience about the benefit of CUDA in image processing algorithms.





## 3 Implemented Algorithms

### 3.1 Hough Transform

#### 3.1.1 Problem Domain

The Hough transform is a feature extraction technique used in computer vision. The aim of this technique is to find object instances with a certain shape. The classical Hough transform, patented by Paul Hough in 1962, was designed to identify lines in an image [11], but later the Hough transform has been extended to detect arbitrary shapes, most commonly circles or ellipses. The nowadays universally used Hough transform was invented by Richard Duda and Peter Hart in 1972, who called the algorithm generalized Hough transform [7].

Today there are a lot of variations and extensions of the Hough transform. To get an overview on the great variety of Hough transforms we studied multiple papers and web resources. From this research we can group the different attempts into two main categories: non-probabilistic and probabilistic algorithms. In the following two subsections we listed some interesting variants which we could find in [10], [8], [9], [12] and [19].

#### Non-probabilistic Algorithms

- *Standard Hough Transform (SHT)*
- *Backmapping Hough Transform (BHT)*
- *Fast Hough Transform (FHT)*
- *Hierarchical Hough Transform (HHT)*
- *Combinatorial Hough Transform (CHT)*
- *Curve Fitting Hough Transform (CFHT)*
- *Log Hough Transform (LHT)*

#### Probabilistic Algorithms

- *Randomized Hough Transform (RHT)*
- *Probabilistic Hough Transform (PHT)*
- *Progressive Probabilistic Hough Transform (PPHT)*

- *Monte Carlo Hough Transform (MCHT)*
- *Dynamic Combinatorial Hough Transform (DCHT)*

Non-probabilistic algorithms process all edge pixels of the input image. This usually leads to a robust algorithm with few false-positive matches for the price of more memory consumption. On the opposite probabilistic algorithms process only a randomly selected subset of edge pixels, which reduces memory usage and processing effort.

#### Implemented Variants

From the numerous objects which can be detected by the Hough transform we decided to focus on line detection. Furthermore we determined to implement two variants of the Hough transform on the GPU and compare them to their CPU counterparts.

Initially we realized the Standard Hough Transform (SHT) because it is a straight forward approach and we already had good results from the OpenCV library reference [14]. To try another solution we looked for a method to reduce the computational overhead of the Standard Hough Transform. During our search we came over the Additive Hough Transform as described in "Exploiting Inherent Parallelisms for Accelerating Linear Hough Transform" [17] which promises an enhancement against SHT.

In the following two sections we describe the two implemented approaches and explain their characteristics and differences.

#### Standard Hough Transform

The basic idea behind Hough transform is to convert an input image to a format that allows us to simply detect the lines from the transformation output. However, before we can start with the actual transformation the input image is processed to contain only the information we are interested in, in this case the edges. The edge map is the original image transformed to a binary image with the edges in white (see e. g. figure 3.5a). This can be produced by applying an edge detector such as the Canny edge filter [18].

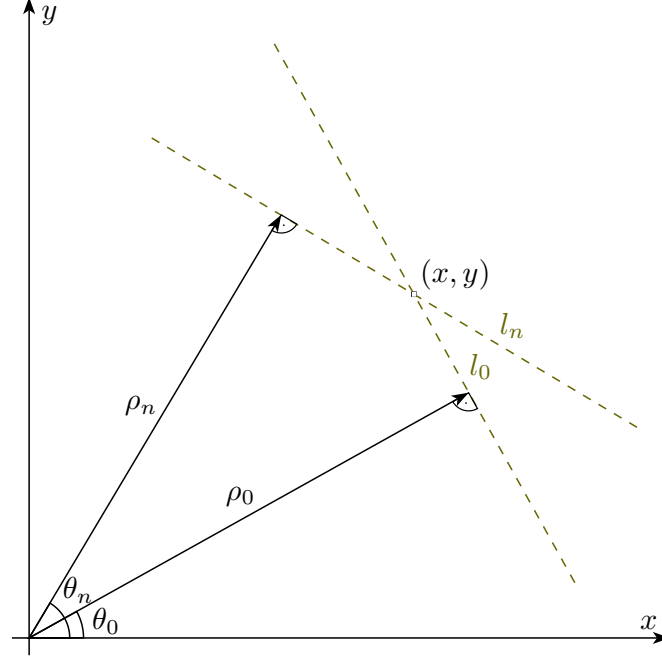
The Standard Hough Transform algorithm starts with a voting process. Given only an edge pixel  $(x, y)$ , as in figure 3.1, we do not know whether this pixel is part of a line or not. Even if we knew that pixel  $(x, y)$  lies on a line we would not know its orientation. The line could be at any angle between 0 and 179 degrees. As a result, the Hough transform examines all the lines passing through pixel  $(x, y)$  and makes a vote for each of these lines in the so called parameter space (also known as Hough space). The parameter space acts as an accumulator, summing up the votes of the different lines passing through an edge pixel. Accordingly, the parameter space has the dimension of  $\rho \times \theta$  where  $\rho \in [0, \pi]$  and  $\theta \in [-\frac{w+h}{2}, \frac{w+h}{2}]$  where  $w$  and  $h$  is the image width and height.

Essentially the Hough transform maps the image space to a parameter space using an appropriate mapping function. A trivial mapping function would be the representation of the lines as  $m \cdot x + q = y$ . The problem with this function is if the line is vertical,  $m$  goes to infinity. Because in practice it is impossible to deal with infinite large numbers

we have to find another approach. A commonly used solution is to use the Hessian normal form:

$$x \cdot \cos(\theta) + y \cdot \sin(\theta) = \rho \quad (3.1)$$

with the parameters  $\rho$  and  $\theta$  and a given point  $(x, y)$ . As indicated in figure 3.1 we can find an infinite set of lines passing through point  $(x, y)$  if we vary  $\rho$  and  $\theta$ .



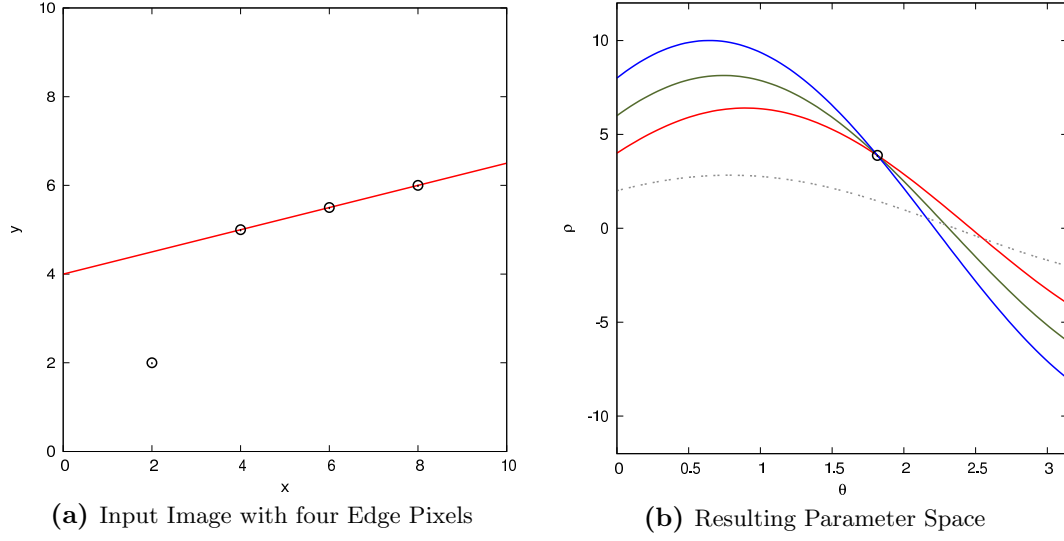
**Figure (3.1)** Hessian Normal Form

Because the transformation from image space to parameter space is essential to understand, we will try to visualize this procedure and its properties.

Reconsidering equation 3.1 we can recognize the transformation of a point in the edge map to a curve in parameter space. As illustrated in figure 3.2, each possible line through point (2,2) in figure 3.2a is mapped to the parameter space producing the dashed curve in figure 3.2b. The crucial property about this transformation is the common intersection point in parameter space (figure 3.2b) of points on the same line in the edge map (figure 3.2a). As a result, we can find  $(\rho, \theta)$  points in parameter space that show a remarkably higher accumulator value than the points around them.

Figure 3.2a shows three points lying on the same line and one point located off that line. If we transform these points with help of equation 3.1 into parameter space we get the resulting parameter space as shown in figure 3.2b. As we can see, the curves produced by the points located on the line, intersect at exactly the same position in parameter space, which results in an accumulator value of 3 at this position. On the other hand, the point that is located off the line does not pass through that point in parameter space. Hence it contributes non of its votes to the line in figure 3.2a.

The detection of a line is now trivial: To find the lines in the original image we only



**Figure (3.2)** Hough Transformation with resulting Hough Space

have to find local maxima in the parameter space. In the most primitive version this is done by simply thresholding the values.

#### Additive Hough Transform

Another approach to calculate the parameter space is called Additive Hough Transform (AHT). Notice however, the basic idea of the Hough transform algorithm remains unchanged with the Additive Hough Transform.

The AHT attempt is based on S. Suchitra Sathyanarayana's paper [17]. This method can be classified as a non-probabilistic method and has its main focus on the decomposition of the construction of the parameter space to compute it in parallel.

If we analyze the Hough transform we can recognize its additive property given by

$$HT(A, O) = HT(A, B) + HT(B, O) \quad (3.2)$$

where  $O$  is the origin of the image and  $HT(X, Y)$  represents the Hough transform of point  $X$  with respect to point  $Y$ . Applied to equation 3.2 this means, the Hough transform of some point  $A$  with respect to the origin  $O$  is the same as the Hough transform of point  $A$  with respect to some other point  $B$  plus the Hough transform of point  $B$  with respect to the origin  $O$ .

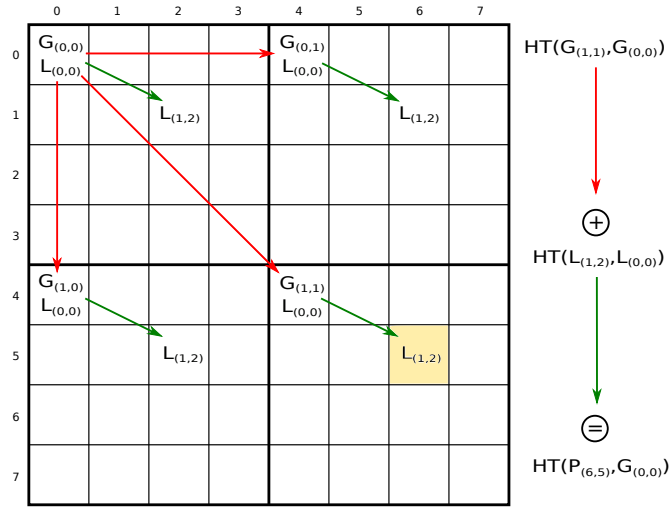
With the Additive Hough Transform a given input image with dimensions of  $m \times m$  pixels is divided into  $\frac{m}{k} \times \frac{m}{k}$  blocks (illustrated by the thick lines in figure 3.3) with  $k \times k$  pixels each. For each block the top left pixel is considered as a local origin  $(L_{(0,0)})$ . Notice that the so called local Hough transform  $(HT(L_{(x,y)}, L_{(0,0)}))$  of all blocks with respect to their local origin is the same and thus can be precomputed the same way as previously described in the Standard Hough Transform. To get the actual Hough transform of a certain block we have to add an offset to all pixels in that block. This

offset is referred to as the global Hough transform ( $HT(G_{(x,y)}, G_{(0,0)})$ ) which is effectively the Hough transform of the local origin of each block with respect to the global origin ( $G_{(0,0)}$ ).

As an example consider point  $L_{(1,2)}$  in the bottom right block in figure 3.3 with coordinates (6, 5). The Hough transform of this point is calculated as follows:

$$HT(P_{(6,5)}, G_{(0,0)}) = HT(L_{(1,2)}, L_{(0,0)}) + HT(G_{(1,1)}, G_{(0,0)}) \quad (3.3)$$

The first summand in equation 3.3 refers to the local Hough transform and can be found in the lookup table. The second summand is the global offset for all the pixels in the bottom right block.



**Figure (3.3)** Illustration of the Additive Hough Transform

The benefit of the Additive Hough Transform is the reduced number of computations. First we have to calculate a block only once and second we need only  $\frac{m}{k}$  global Hough transform values to compute.

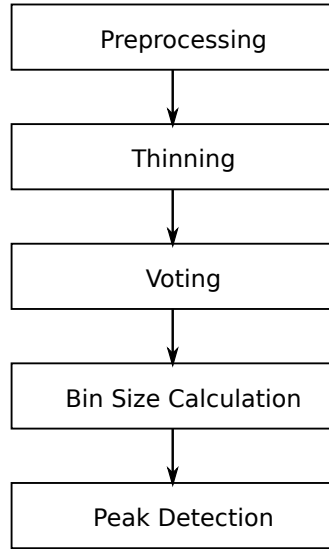
### 3.1.2 Method

In this section we focus on the implementation details of the two Hough transform algorithms described in section 3.1.1. To explain the program code details we do not distinguish between the two algorithm variants as they are very similar from an implementations point of view. Nevertheless we will explain special cases and differences between them in detail if necessary.

First of all it's important to see that a sequential implementation of Hough transform is straight forward as we do not have to care about thread synchronization or race conditions. However, with a massively parallel architecture such as CUDA we have to overcome some major difficulties in the implementation of the Hough transform. For instance, if we look at the voting procedure in a parallel implementation it is very likely

that two threads would like to vote for the same  $(\rho, \theta)$  cell in the accumulator (parameter space) at the same time. As a result the access would have to be synchronized, e.g. by an atomic add operation which unfortunately kills most of the performance gain by using the GPU. But this was not the only problem that came up during the implementation process, as we also had problems with too much memory consumption. So we had to find ways to overcome these problems which we will describe in the following paragraphs.

To get an overview of the code and a mental model of our implementation refer to figure 3.4. As illustrated we can identify five major steps in our code which will be discussed in detail on the following pages.



**Figure (3.4)** Steps of the implemented Hough Transform Algorithm

Theoretically it is possible to compute the Hough transform for any angle of  $\{\theta \in \mathbb{R} \mid 0 \leq \theta < \pi\}$ . Obviously it is not possible in practice to have an infinite resolution of angle  $\theta$  and thus we have to discretize this value. To decide for an adequate angle resolution we have to consider the memory consumption. If we chose the angle resolution too high, we will run into memory shortage pretty fast. Otherwise with a too coarse grained resolution we may miss some of the lines. Consequently we decided to process every angle at a resolution of one degree between  $0^\circ - 179^\circ$ , which proved to be a suitable value.

**Preprocessing:** In contrast to the Standard Hough Transform where this step to simply does some memory allocations, the Additive Hough Transform computes the global and local Hough transform arrays. These arrays act as lookup tables in the upcoming voting procedure of AHT. The computation of these lookup tables is done as described on page 8.

As the current NVIDIA graphics devices do not provide more than 64 KB of constant memory (see [3], "Features and Technical Specifications"), it was not possible to place the local and global lookup tables in constant memory. It would not be possible to

place the global Hough transform table in constant memory anyway, as the amount of constant memory has to be defined at compile time and the size of the global Hough transform table depends on the size of the input image. As a result we had to place these values in global memory. Because global memory read access is generally slow we had to find other possibilities to either reduce read access or improve the performance by faster types of memory. Since lookup tables have only read access we can use texture memory to cache data reads from global memory, which compensates for the performance loss of not using constant memory.

**Thinning:** We already mentioned in the introduction to this section that thread synchronization was not the only problem we had to deal with. Another problem that came up during the implementation was the amount of memory resources to store the data. In case of a CPU implementation the main memory is usually not a problem, as there are normally 2 GB or more available. Unfortunately, this might be a problem in the field of GPU implementations, as the graphics cards we tested on had between 512 MB and 1 GB of memory available, which is a limiting factor for our implementation.

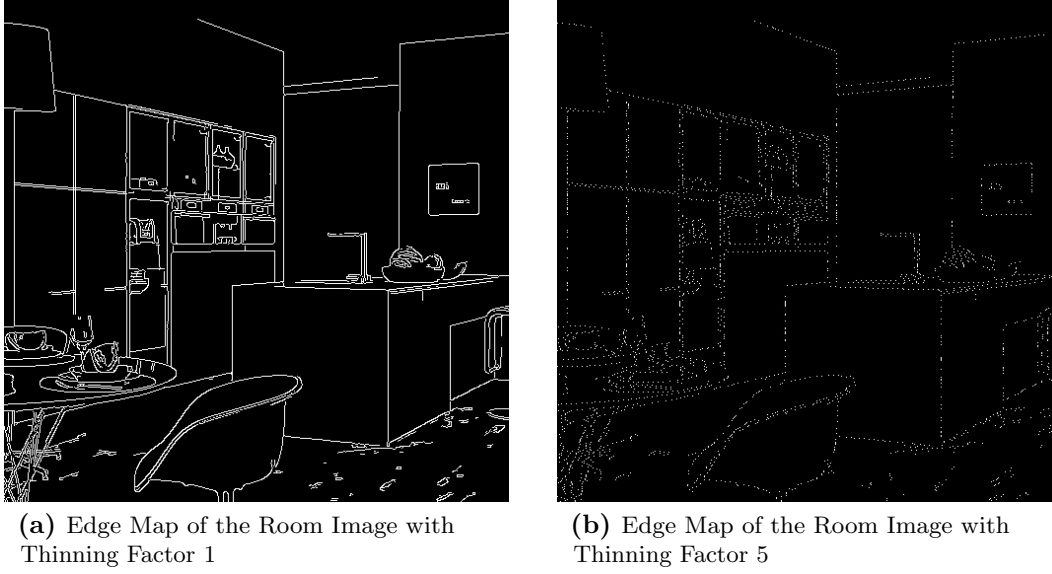
To mitigate the memory problems we came up with a solution inspired by the probabilistic algorithms. When we look at the parameter space after the transformation we detect high vote counts at the  $(\rho, \theta)$  cells which actually represent the lines in the original image. The votes on the other cells represent basically just noise. If we now process only a part of the edge pixels the results would not be much different. We will still get a strong response on the cells which represent a line, however the signal-to-noise ratio actually gets smaller.

With this insight our implementation requires an additional parameter which specifies a thinning factor. This factor allows us to control the amount of pixels to be processed by the algorithm. The thinning parameter represents the value on how many pixels should be kept for calculation. A thinning factor of e. g. 1 indicates that every pixel is processed, where a factor of 2 effects that only every second pixel is kept for calculation. In other words, a thinning factor of 5 reduces the number of pixels to be processed to 20%. To visualize the impact of this procedure, figure 3.5 displays the different edge maps of our input image with thinning factor 1 and 5, respectively.

**Voting:** Generally the voting step includes the transformation of the edge pixels into parameter space. We consider this process as a voting procedure where each edge pixel votes for the set of lines passing through it.

In a sequential implementation of the Hough transform the parameter space is built up directly from the edge map, where each edge pixel is being processed one after another and transformed into parameter space. If we look at this from a parallel implementations point of view we tend to do the same, but in parallel with one thread per pixel. This raises multiple problems:

- As only a small amount of pixels are actually edge pixels we launch a lot of threads that do no work at all.
- When multiple threads vote for the same line in parallel we get race conditions on the  $(\rho, \theta)$  cells in parameter space.



**Figure (3.5)** Room Edge Map with Thinning Factor 1 and 5

- If we try to solve the synchronization problems by allocating memory for each produced vote we immediately run out of memory.

These problems suggest that we have to find another way to build the parameter space in parallel. To do so, we initially analyze the edge map and save the coordinates of the edge pixels (combined with the thinning process as described in the previous paragraph). All edge pixel coordinates are coded in a one dimensional integer index given by

$$f: \mathbb{N}^2 \rightarrow \mathbb{N}, (x, y) \mapsto x + w \cdot y \quad \text{where } w \text{ is the image width.} \quad (3.4)$$

This mapping allows us to save memory as we have to store just one integer for the coordinates of an edge pixel instead of two. The actual voting process then consists of mapping the edge pixel to the cells in parameter space that represent the lines going through that point. In case of the Standard Hough Transform, we can extract the  $x$  and  $y$  coordinates from the linear index by

$$f^{-1}: \mathbb{N} \rightarrow \mathbb{N}^2, z \mapsto \left( z \bmod w, \frac{z}{w} \right) \quad \text{where } w \text{ is the image width,} \quad (3.5)$$

and solve equation 3.1 for all 180 angles. This results in 180 votes, one for each possible line going through the edge pixel at  $(x, y)$ . In case of the Additive Hough Transform the result of equation 3.1 is not calculated directly as we already have calculated some parts of the result. To get the actual voting coordinate we have to add the global Hough transform to the local Hough transform of edge pixel  $(x, y)$  as described in equation 3.2.



We get the corresponding global ( $G$ ) and local ( $L$ )  $x$  and  $y$  indexes by

$$G: \mathbb{N}^2 \rightarrow \mathbb{N}^2, (x, y) \mapsto \left(\frac{x}{k}, \frac{y}{k}\right) \quad \text{where } k \text{ is the block dimension,} \quad (3.6)$$

$$L: \mathbb{N}^2 \rightarrow \mathbb{N}^2, (x, y) \mapsto (x \bmod k, y \bmod k) \quad \text{where } k \text{ is the block dimension.} \quad (3.7)$$

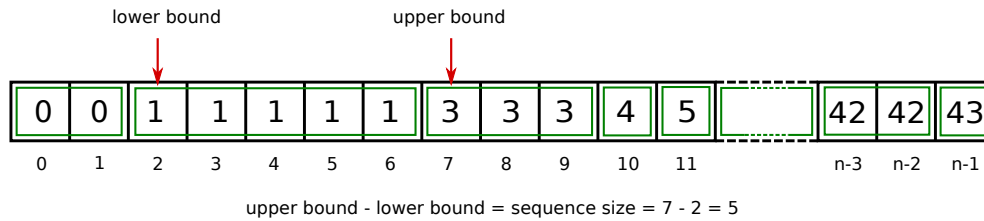
In the introduction to this paragraph we already mentioned the race condition occurring when multiple threads try to vote for the same  $(\rho, \theta)$ -cell at the same time. A simple solution to this problem would be to synchronize all write accesses to parameter space during the voting process. Unfortunately this simple solution is not suitable for a SIMT architecture such as CUDA, as it prevents a parallel execution path of the threads. To get around the synchronization problem we look at the voting process as the calculation of a 2D histogram, where the input data are the actual votes. The question is now, how to compute this histogram as fast as possible. To store the produced votes from this step we first allocate an array that is suitable in size to have a separate storage location for all the votes of each thread. Because we already analyzed the edge map and computed the number of edge pixels in the thinning step we know exactly how much memory to allocate for all votes.

In the voting process we store the votes the same way as in equation 3.4 and map the  $(\rho, \theta)$  indices to a linear index (bin index) by

$$g: \mathbb{N}^2 \rightarrow \mathbb{N}, (\rho, \theta) \mapsto \rho + n \cdot \theta \quad \text{where } n \text{ is the number of possible } \rho \text{ values.} \quad (3.8)$$

In the next paragraph we show how we implemented the histogram calculation and how we could avoid thread synchronization.

**Bin Size Calculation:** After storing the histogram data we calculate the bin  $((\rho, \theta)$ -coordinate) sizes of the 2D histogram. We first sort the votes in increasing order to get sequences of votes for the same  $(\rho, \theta)$ -cell, such as illustrated in figure 3.6. To determine the length of these sequences we search for the lower bound (start of sequence) and the upper bound (end of sequence) indices of the same value. From this two indices we can calculate the bin size (number of votes) by simply subtracting the upper bound index from the lower bound index.



**Figure (3.6)** Calculation of Bin Sizes

Now, that we have computed the bin sizes we actually computed the parameter space without having to synchronize any threads. The last step includes the detection of the

highest peaks in parameter space.

**Peak Detection:** Peak detection can be implemented in many different ways, where some are very compute intensive and others are not. We decided to not simply threshold the values in parameter space, moreover we reinforced the criteria for a cell to be detected as a line. Specifically, we look for local maxima which means a cell value has to be higher than the given threshold and higher than his neighbors on the top, bottom, left and right position. This way we can reduce the number of false-positive matches and increase the exactness of the detected lines.

When thinking about the implementation of this local maxima finder we naturally think of accessing the pixels in  $x$  and  $y$  coordinates and equally the neighbors by adding or subtracting 1 from the  $x$  or  $y$  coordinate index. Because this mental model perfectly fits to the concept of 2D texture memory we stored the calculated bin sizes of the previous step in a 2D texture array. Besides the simplified neighbor access scheme texture memory provides the benefit of having cached memory access which increases performance.

#### 3.1.3 Results

In this section, we show the results achieved by the two implementation variants and discuss them in terms of memory usage and runtime performance. Furthermore we look for the bottlenecks in our implementations and compare the results with reference CPU implementations.

Because the GPU runtime measurements have small fluctuations, we took the average runtime of multiple measurements.

#### Test Components and Conditions

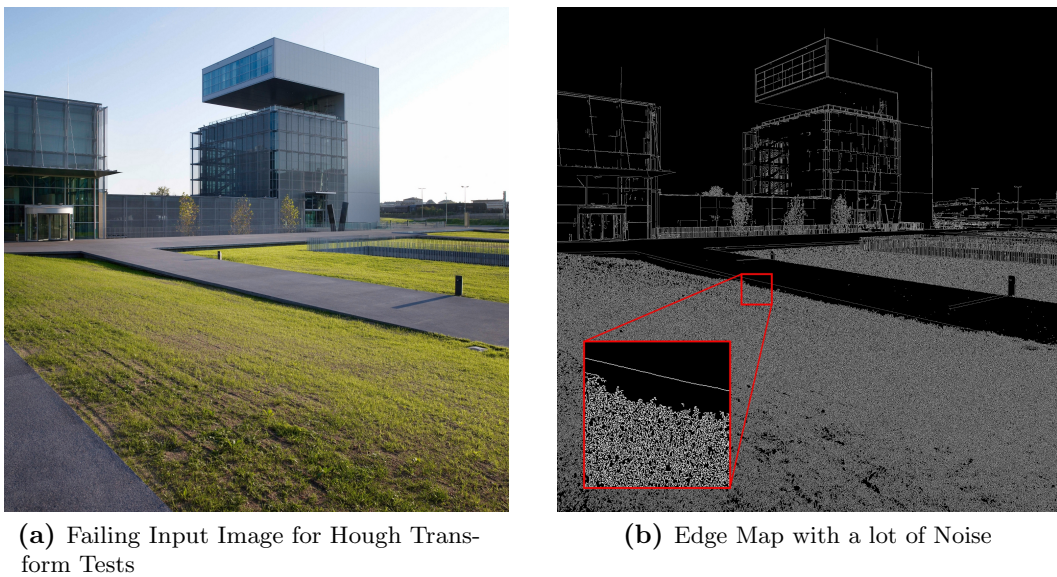
All our test results are based on the image shown in figure 3.7a and its corresponding edge map (3.7b). We generate the edge map from the original image by using the Canny edge detector from the OpenCV library.

We chose this image because its edge map has nice straight lines and the edge detector produces almost no noise. As we could see from tests with other images the Hough transform fails when the edge detector produces a lot of very small edge segments as shown in figure 3.8b. The grass parts in image 3.8a result in many little connected edge segments that add a lot of noise to the parameter space which then makes it difficult to distinguish between line or no line for the peak detector. This problem can be solved by configuring the edge detector with suitable threshold values. However, in our implementations we used a constant configuration to not further overload our algorithms parameter list.

Another important factor when testing CUDA algorithms is the underlying hardware. Before we start discussing our performance results we would like to mention that the following results stem from tests on a machine with an NVIDIA Quadro FX 3800 as described in the appendix in table A.2. Earlier tests with an NVIDIA Quadro FX 580 device in a machine described in table A.1 showed that it is essential to have a sufficient graphics card configuration to achieve enough parallelism and to satisfy the demand



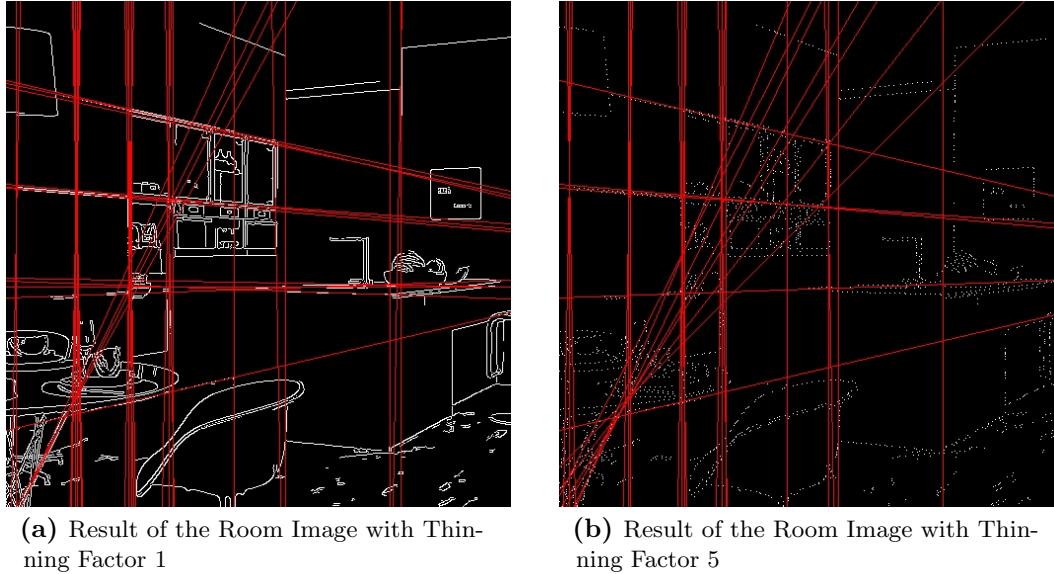
**Figure (3.7)** Room Image and its Edge Map



**Figure (3.8)** Example of bad Edge Map

on memory resources. Compared to the Quadro FX 3800 device, the Quadro FX 580 is much cheaper, has six times less GPU cores and holds only half of the amount of graphics memory. We documented the impact of different graphics card configurations on our implementations in chapter 4.

**Edge Pixel Thinning:** As already described on page 11 we implemented a thinning procedure which takes only a certain amount of pixels into account for the calculation (see figure 3.5). Because a line usually consists of a high number of pixels, its structure can still be recognized if we process only every fifth pixel. Figure 3.9 shows the output with a thinning factor of 1 (each pixel was taken for calculation) on the left hand side in contrast to a thinning factor of 5 (only every fifth pixel was taken for calculation) on the right hand side.



**Figure (3.9)** Hough Transform Results with Thinning Factor 1 and 5

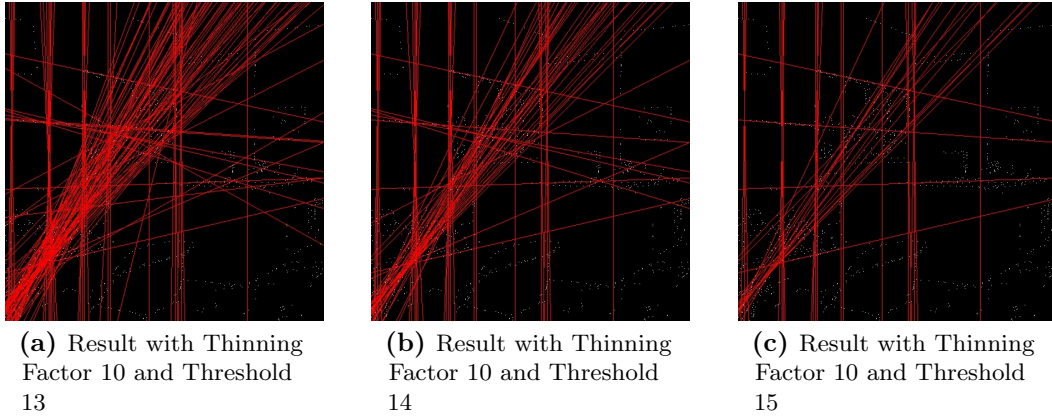
Even if the results in figure 3.9 seem to be the same, the thinning feature indeed has an effect to the algorithm. Namely the threshold becomes more sensitive as less edge pixels are taken for the calculation. Or in other words: A small change in the threshold value can have dramatic impact to the result when the thinning factor is too high.

Let us explain this statement with the following example: The image 3.7a with a resolution of  $512 \times 512$  pixels and thinning factor 1 affords 16843 different potential lines with vote counts between 2 and 271. We generate the best output if we take a threshold of around 100. By taking a look at the vote counts we notice a difference of 48 votes with counts between 95 and 105. If we play with the threshold in the range of 99 and 101, we will have a total difference of 12 lines in our result.

The same picture processed with thinning factor 5 generates 19015 different lines with votes between 2 and 53. The best threshold we can take for this configuration lies somewhere around 23. Now we have a remarkable amount of 305 potential candidates with vote counts between 18 and 28 and still 90 candidates between 21 and 25 votes.

The situation gets even worse if we increase the thinning factor to 10. In the threshold range of 13 to 15 we will have a difference of 56 lines which is a clearly visible change as we can see in figure 3.10.

Finally, we think that the thinning feature is very useful in case of runtime performance



**Figure (3.10)** Hough Transform Results with Thinning Factor 10 and different Thresholds

and memory consumption, but it has to be used carefully.

#### Algorithm Output

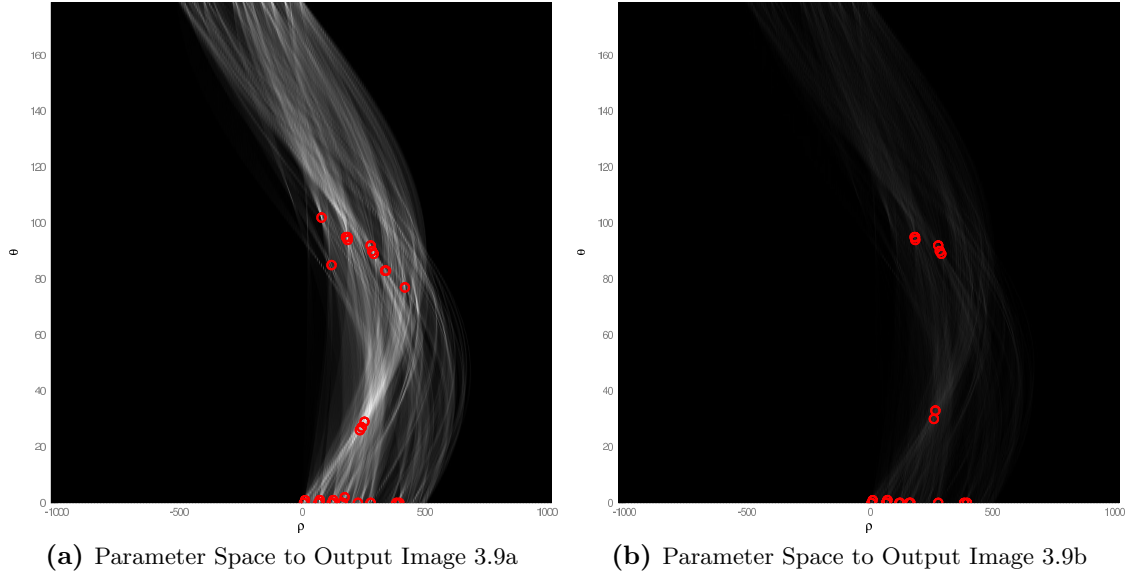
This section discusses the visual results produced by the algorithm. The visual results are namely an example of the parameter space and the actual output image with the detected lines. As the output image was already shown in figure 3.9 we here focus on the relationship between the output and parameter space.

The parameter space is an intermediate result of the computation and is usually not directly visible. To visualize the parameter space with its votes and peaks we rendered two versions in image 3.11a and 3.11b to their corresponding output images in figure 3.9a and 3.9b, respectively. If we ask ourselves how the parameter space is affected by the thinning factor, we can see in figure 3.11b that the illustration of the parameter space simply gets darker, as the vote count over the whole parameter space is lower.

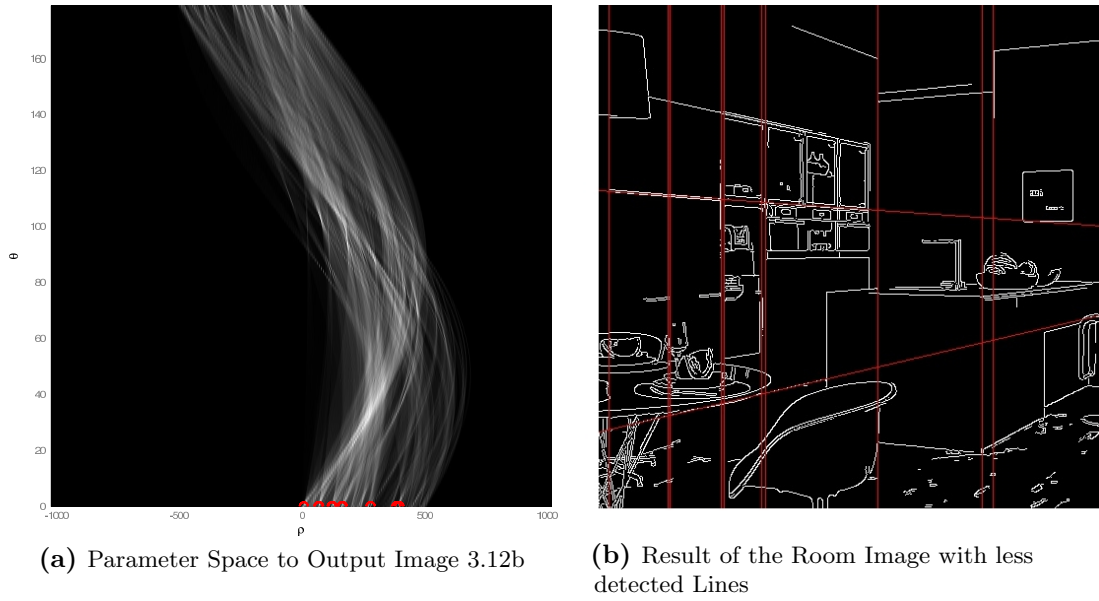
The circles in the parameter spaces mark the  $(\rho, \theta)$  coordinates of the detected lines. To roughly verify these circles we can compare these results with the found lines in the output image. To simplify this procedure we chose a higher threshold which makes the comparison more clearly because less lines are detected. In figure 3.12a we can see a lot of circles at  $\theta = 0^\circ$  in the parameter space, which means that the algorithm detected many vertical lines in our input image. If we check this result with the output image 3.12b we can see the matching vertical lines.

#### Runtime Measurement of Standard Hough Transform

The goal of the first test scenario is to test our Standard Hough Transform implementation against the OpenCV implementation. The resulting runtime measurements are listed in table 3.1.



**Figure (3.11)** Parameter Spaces with detected Lines



**Figure (3.12)** Comparison between Parameter Space and Output Image

Algorithm	$128 \times 128$	$256 \times 256$	$512 \times 512$	$1024 \times 1024$	$2048 \times 2048$	$4096 \times 4096$	$8192 \times 8192$
SHT CUDA	6.2 ms	9.8 ms	18.7 ms	49.6 ms	106.8 ms	351.3 ms	374.8 ms
SHT CUDA ( $\frac{1}{5}$ )	4.6 ms	6.1 ms	9.2 ms	16.8 ms	43.5 ms	89.8 ms	189.4 ms
SHT OpenCV	3.6 ms	9.6 ms	18.5 ms	55.2 ms	135.4 ms	303.5 ms	527.6 ms

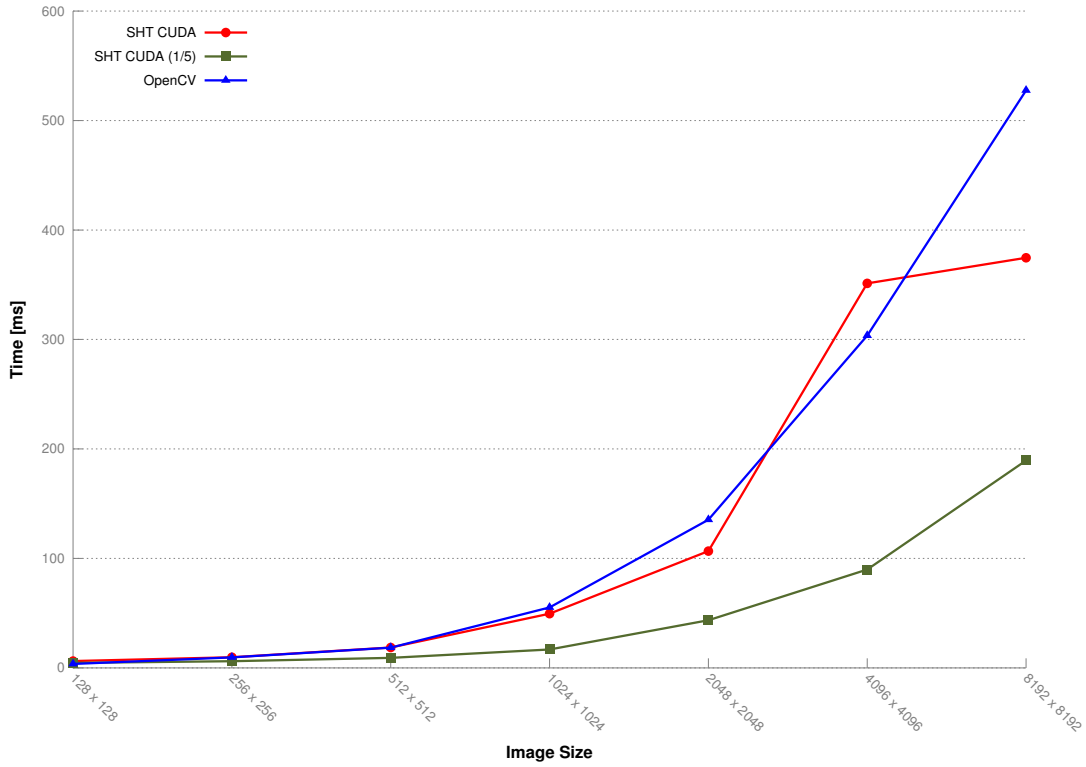
**Table (3.1)** Runtime Example with Standard Hough Transform on the Room Image



The first series of measurements is a results from processing each edge pixel. The numbers in the second row are generated by processing only every fifth pixel. Since the OpenCV implementation has no parameter to reduce the number of processed pixels, there is only one row of time measurements from the CPU. The data of table 3.1 is visualized in figure 3.13.

Even if not visible in figure 3.13, table 3.1 shows the typical effect of the graphics device, which is the slower runtimes on smaller images. Here, the OpenCV implementation is faster mainly due to the reduced memory transfer overhead, where on the other hand we can beat the OpenCV Hough transform if we process larger images.

A big enhancement in our implementation is the thinning procedure. If we process only every fifth pixel we can see that our algorithm performs better than the reference implementation. As already described in the previous section, one should be careful to not thin the pixels too much, as the resulting output could suffer from wrong matches. If, on the other hand the thinning factor and threshold is chosen wise, our implementation can beat the reference implementation of OpenCV up to a factor of 3.4 with the  $4096 \times 4096$  tests.



**Figure (3.13)** Runtime Measurements of the Standard Hough Transform

By taking a look at figure 3.13, we notice a gap between our theory and the actual curve of the Standard Hough Transform CUDA (SHT CUDA). We can see a sudden rise in runtime from  $2048 \times 2048$  to  $4096 \times 4096$  instead of an expected steadily increasing trend. Our measurements showed that this increase originates from the sort algorithm

we are using. To sort the vote array we rely on the Thrust implementation [16] which is based on CUDA Data Parallel Primitives Library (CUDPP) [13]. The CUDPP implementation uses different optimized sorting implementations based on the input size, we infer that the underlying CUDPP core switches to another implementation in this very moment. This assumption will be reinforced by the fact that after the sudden increase the slope of the curve evens out to a normal level.

A further analysis of the average runtime of the individual algorithm steps showed that the sorting of the vote array is the bottleneck in our implementation. The runtime of the sort increases faster in time as most of the other steps do. So, to get even better performance on larger inputs the sorting step would be the hot spot to care about.

#### Runtime Measurement of Additive Hough Transform

Now we compare a CPU version of the Additive Hough Transform with our CUDA implementation. Because we developed this CPU reference on our own, we have the opportunity to include the thinning step to reduce the amount of data to process, which allows us to compare the two versions with the same configuration. Notice that these test scenarios are limited to an image resolution of  $4096 \times 4096$  pixels as we ran out of GPU memory on larger inputs.

In table 3.2 we summarize the average runtime of the individual measurements. The tests are based on measurements where each pixel will be processed on the one hand and a test with a thinning factor of 5 on the other hand.

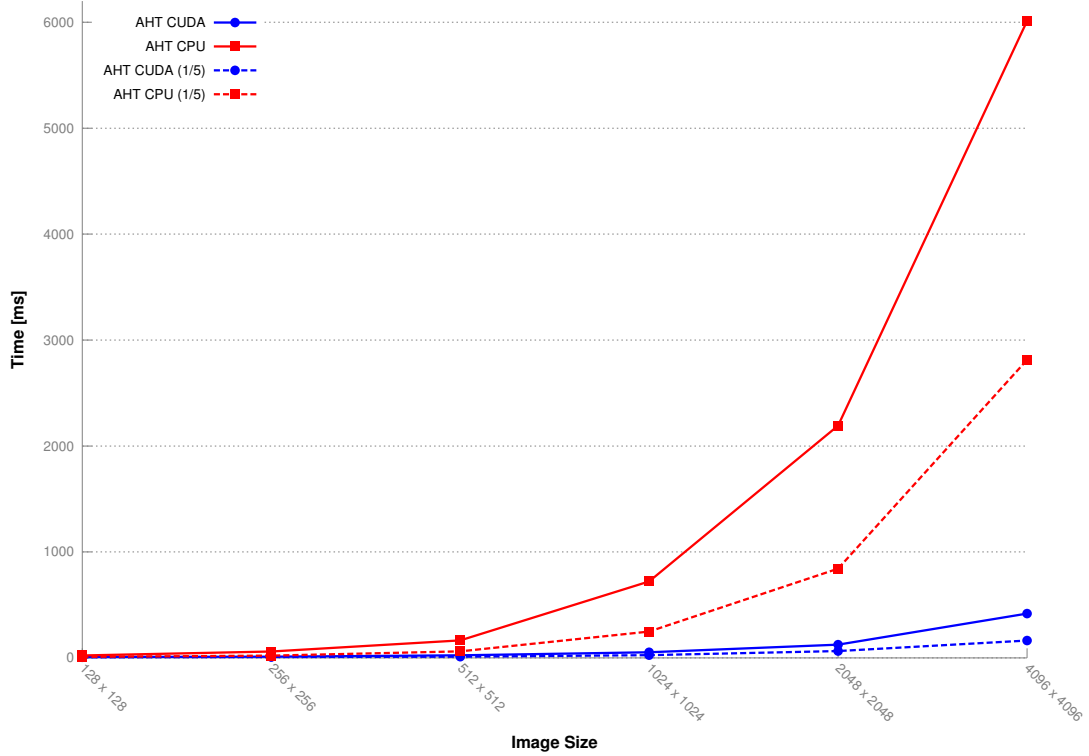
Algorithm	$128 \times 128$	$256 \times 256$	$512 \times 512$	$1024 \times 1024$	$2048 \times 2048$	$4096 \times 4096$
AHT CUDA	8.1 ms	11.6 ms	24.7 ms	53.11 ms	124.9 ms	418.9 ms
AHT CUDA ( $\frac{1}{5}$ )	6.4 ms	8.1 ms	11.9 ms	26.1 ms	64.8 ms	163.5 ms
AHT CPU	22.9 ms	60.1 ms	165.9 ms	722.0 ms	2192.0 ms	6011.2 ms
AHT CPU ( $\frac{1}{5}$ )	9.7 ms	21.5 ms	62.5 ms	248.7 ms	842.6 ms	2812.6 ms

**Table (3.2)** Runtime Example with Additive Hough Transform on the Room Image

The performance measurements of the Additive Hough Transform in this section show a much larger divergence between the CPU implementation and the CUDA approach. In contrast to the Standard Hough Transform implementation this algorithm has much more computational overhead. When the CPU version slows down very fast on an increasing number of input pixels, the GPU variant does not.

If we analyze the effect of the pixel thinning (dashed lines in figure 3.14) we can see a much stronger response on the CPU than on the GPU. This shows the higher computational power of GPU stream processing. Even if we can see the effect on the GPU as well, especially in figure 3.13 by comparing SHT CUDA and SHT CUDA ( $\frac{1}{5}$ ), the impact is not as striking as on the CPU. However, the insight that the GPU reacts less strongly on increasing input amounts holds only as long as we can keep the card busy. As soon as we have to synchronize threads or include too much memory transfers, the GPU curve will rise faster as well.





**Figure (3.14)** Runtime Measurements of the Additive Hough Transform

#### Comparison between SHT and AHT Runtimes

In addition to the previous runtime analysis we can see that the AHT on the GPU performs worse than its SHT GPU counterpart. The reason for this is that the computation of the lookup tables and the lookups itself take longer than a direct computation of the Hough transform. Generally the concept of pre-computation and lookup is successful in terms of performance optimization. However, in this case with the computing power of the GPU the direct computation is so fast that the pre-computation of a reusable lookup table brings no performance gain.

## 3.2 Template Matching

### 3.2.1 Problem Domain

To describe the following problem domain we use Wilhelm Burgers "Digital Image Processing" [2] as an aid.

Humans will be faced with the following main question when they compare two images: In which case are they the same or at least similar and how can similarity be measured? The most trivial approach might be to compare each pixel value and, if they are all the same the images are identical. Unfortunately this kind of definition is far too simple for practical use because of noise, small changes in lighting or quantization errors can cause numerical differences of pixel values. Obviously, human perception can handle this problem and will recognize image similarities even if the previously described approach would technically fail.

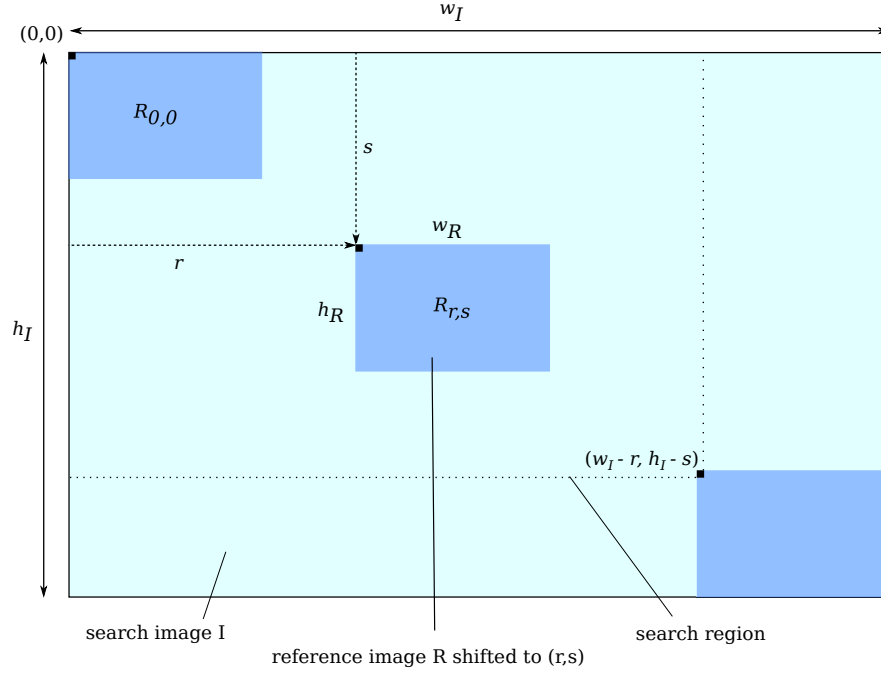
Burger deals with a simpler problem of image comparison in chapter 17 of his book; in particular, localizing a sub-image (so called "template") in a larger image. This task is frequently required to track certain patterns through an image sequence, or to find patches in stereo images, for example. The basic idea behind template matching is simple: measure the difference of a template to the current section of the search image while moving the template over it and record the positions where the highest similarity is obtained. Unfortunately this is not as simple as it initially sounds. Questions like: "What are suitable distance measures?", "What total difference is required for a match?" and "What happens when contrast or brightness of even the image or the template changes?" have to be considered.

We start with a conceptual overview of the algorithm with help of figure 3.15. The template image (also known as reference image)  $R$  with origin  $R_{r,s}$ , width  $w_R$  and height  $h_R$  will be shifted over the search image  $I$  with origin  $(0,0)$ , width  $w_I$  and height  $h_I$ . To cover the whole search image  $I$  we have to shift the reference image  $R$  line by line from  $(0,0)$  to  $(w_I - r, h_I - s)$ , where the distance function  $d(r,s)$  is evaluated at each position. We finally want to find the offset  $(r,s)$  where the similarity between the shifted reference image and the underlying sub-image of  $I$  is a maximum.

Now let's take a look to the problem of localizing a reference image  $R$  in a larger image  $I$ , where the template is a snippet of the search image  $I$  and therefore the contents of the reference image  $R$  are the same or at least similar to the corresponding sub-image of  $I$ . To successfully solve this task, we have to address several issues such as developing a good strategy for finding optimal displacements and determining a minimum similarity value to accept a match. In addition a suitable measurement must be found that is tolerant enough against contrast and intensity variations.

#### Distance Measures between Image Patterns

There is a variety of distance measures for two-dimensional intensity images. A carefully considered selection of an appropriate distance measure is essential to this algorithm, as it determines the quality of its output. Based on tests with the OpenCV implementation of template matching we decided to realize our algorithm using the correlation coefficient distance measure. On the following pages we start with a simple distance measure and



**Figure (3.15)** Conceptual Overview of Template Matching

repeatedly evolve it up to the desired correlation coefficient function. Notice that in all of the following distance functions  $(i,j) \in R$  is a short notation for  $\{(i,j) \mid 0 \leq i < w_R, 0 \leq j < h_R\}$ .

The principle of a distance function  $d(r,s)$  to compare the reference image  $R$  to a patch in the search image  $I$  is always the same: To get the distance value at a certain position  $(r,s)$  we iterate the pixels in  $I$  covered by the template  $R$ , compare them to their corresponding pixel in  $R$  and sum the resulting distance values up.

A trivial and commonly used distance measure is the sum of the squared differences also known as Euclidean distance:

$$d_E(r,s) = \sqrt{\sum_{(i,j) \in R} (I(r+i, s+j) - R(i,j))^2} \quad (3.9)$$

The distance measure 3.9 produces satisfying results when the intensity over an image is constant. On changing light conditions, however, this measure would fail.

### Cross Correlation

To find the best matching position between  $R$  and  $I$ , it is sufficient to find the minimum of the square of  $d_E$ , because a minimum in Euclidean distance remains a minimum even if we square this result. This insight will result in

$$d_E^2(r,s) = \sum_{(i,j) \in R} (I(r+i, s+j) - R(i,j))^2 \quad (3.10)$$

and by expanding equation 3.10, we will receive

$$d_E^2(r,s) = \sum_{(i,j) \in R} I^2(r+i,s+j) + \sum_{(i,j) \in R} R^2(i,j) - 2 \cdot \sum_{(i,j) \in R} I(r+i,s+j) \cdot R(i,j). \quad (3.11)$$

On closer consideration of equation 3.11 we can see that the second summand is simply the sum of the squared pixel values of the template image and the first summand the one of the corresponding sub-image. As a result, only the third summand provides any information about the correlation of the template image  $R$  with the corresponding sub-image  $I$  at the current offset  $(r,s)$ . This term is called linear cross correlation and is generally defined as

$$(I \star R)(r,s) = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} I(r+i,s+j) \cdot R(i,j). \quad (3.12)$$

Since we assume that  $R$  and  $I$  have zero values outside their boundaries, this is furthermore equivalent to

$$\sum_{i=0}^{w_R-1} \sum_{j=0}^{h_R-1} I(r+i,s+j) \cdot R(i,j) = \sum_{(i,j) \in R} I(r+i,s+j) \cdot R(i,j), \quad (3.13)$$

where  $w_R$  corresponds to the reference image width and  $h_R$  to its height, respectively. As we can see in equation 3.13, this is effectively the same as the third summand in equation 3.11.

If we assume that the intensity of image  $I$  is constant, we can ignore the first summand of equation 3.11 and interpret it as a constant. In this case, the minimum of  $d_E^2(r,s)$  can be found by calculating the maximum value of the correlation of  $I \star R$ .

### Normalization

Unfortunately, the assumption of a constant intensity of image  $I$  holds only for few images and therefore the results of the cross correlation varies because of intensity changes. With a normalization we can counteract this effect by considering the "energy" in the sub-image and the reference image. The so called normalized cross correlation is defined as

$$d_N(r,s) = \frac{\sum_{(i,j) \in R} I(r+i,s+j) \cdot R(i,j)}{\sqrt{\sum_{(i,j) \in R} I^2(r+i,s+j)}} \cdot \frac{1}{\sqrt{\sum_{(i,j) \in R} R^2(i,j)}} \quad (3.14)$$

With this calculation the result of  $d_N(r,s)$  is guaranteed to be in the range of  $[0,1]$  if all values in the reference and search image are positive (which is usually the case). This has the advantage that the decision about acceptance or rejection of a matching position can be made with ease by just using a suitable threshold between 0 and 1. Since the threshold values are between 0 and 1 a value of  $d_N(r,s) = 1$  indicates a perfect match whereas  $d_N(r,s) = 0$  means a complete mismatch.

### Correlation Coefficient

Even if there is an improvement with the normalized cross correlation (equation 3.14) in contrast to the cross correlation (equation 3.13) the calculation still has the problem that the result changes dramatically, when the overall intensity of image  $I$  varies.

A solution to overcome this problem is to compare the differences between  $R$  and  $I$  with respect to their average values, instead of comparing the original function values. With this modification we will receive following equation

$$d_C(r,s) = \frac{\sum_{(i,j) \in R} (I(r+i, s+j) - \bar{I}(r,s)) \cdot (R(i,j) - \bar{R})}{\sqrt{\sum_{(i,j) \in R} (I(r+i, s+j) - \bar{I}(r,s))^2} \cdot \sqrt{\sum_{(i,j) \in R} (R(i,j) - \bar{R})^2}} \quad (3.15)$$

$$= \frac{\sum_{(i,j) \in R} (I(r+i, s+j) \cdot R(i,j)) - |R| \cdot \bar{I}_{r,s} \cdot \bar{R}}{\sqrt{\sum_{(i,j) \in R} (I^2(r+i, s+j) - |R| \cdot \bar{I}_{r,s}^2)} \cdot \sqrt{\sum_{(i,j) \in R} (R(i,j) - \bar{R})^2}}, \quad (3.16)$$

where  $|R|$  is the number of pixels in the reference image and the average values  $\bar{I}(r,s)$  and  $\bar{R}$  are defined as

$$\bar{I}(r,s) = \frac{1}{|R|} \cdot \sum_{(i,j) \in R} I(r+i, s+j) \text{ and } \bar{R} = \frac{1}{|R|} \cdot \sum_{(i,j) \in R} R(i,j). \quad (3.17)$$

Equation (3.15) is well known in statistics as the correlation coefficient. In our case, the correlation coefficient  $d_C(r,s)$  describes the piecewise correlation between the template  $R$  and the actual sub-image at offset  $(r,s)$  of the search image  $I$ . The resulting values of  $d_C(r,s)$  will be in the range of  $[-1,1]$  regardless of the contents of  $R$  and  $I$ . As before, a value of 1 indicates a perfect match between the compared image patterns, whereas a maximum mismatch will now result in a value of  $-1$ . The value 0 means there is no correlation between  $R$  and  $I$ .

If we further analyze equation 3.16 we can find the first part of the nominator already appeared as the last term in equation 3.11. We then identified this term as the linear cross correlation between  $I$  and  $R$ . As the linear cross correlation is basically the same as the convolution  $I * R$ , but without flipping the reference image over the diagonal, the linear cross correlation and the convolution have very similar properties. Consequently we can calculate the cross correlation  $I \star R$  the same way as the convolution, as a simple point wise complex conjugate multiplication of  $I$  and  $R$  in the frequency domain. This can be formulated as

$$(I \star R)(r,s) = \mathcal{F}^{-1}\{\mathcal{F}(I) \cdot \mathcal{F}(R)^*\} \quad (3.18)$$

where the asterisk indicates the complex conjugate. In the next section we explain why this mathematical property is interesting to speedup our implementation.

### 3.2.2 Method

To implement the template matching algorithm we followed again the standard scheme and started with the simplest and most trivial solution we could come up with. Besides that we used the OpenCV reference implementation as an orientation to help us with some algorithmic details. As you could expect from this approach, the first implementation is most likely not the best, however it helped us to get a good understanding of the difficulties and properties of the algorithm and gave us insights of where to start optimizing and modifying the implementation to speed it up.

We decided to implement the correlation coefficient version of the template matching algorithms described in the previous section. Even if the calculation of the correlation coefficients involves much more computation effort than other methods, it is also much more robust against intensity changes in the image in contrast to simpler versions. Accordingly this method is also much more relevant in practice when dealing with natural images where intensity changes are likely due to changing light.

#### A trivial Cross Correlation Template Matching Implementation

We started off with a trivial implementation of the template matching algorithm and identified two major steps: Firstly, we precomputed the constant terms of equation 3.16 followed by the actual distance computation using these constant values. But, what are the constant values in equation 3.16? Basically it is the second square root in the denominator which includes the computation of the average reference image intensity  $\bar{R}$ . This value is also used to calculate the nominator. Both of the two values,  $\bar{R}$  and the mentioned square root, are only dependent on the reference image and therefore constant for the computation of all correlation coefficients. The other parts of the correlation coefficient, especially the ones that are only dependent on the image  $I$  seem to be constant, however, as these values are only constant for the sub-image  $I(r+w_R, s+h_R)$  at a certain offset  $(r,s)$  with  $w_R$  and  $h_R$  being the reference image with and height, these values are not the same for all correlation coefficients. Due to this, we included the computation of these values in the second step, the actual computation of the correlation coefficients. Because each correlation coefficient at a certain position  $(r,s)$  can be computed independently from all the others we compute these values in parallel. The first part of the CUDA kernel for this trivial implementation is shown in listing 3.1.

```
1  __global__  
void match_template_kernel(config conf, char* d_img_out, int out_width  
    , int out_height)  
{  
    int r = blockIdx.x * blockDim.x + threadIdx.x;  
5    int s = blockIdx.y * blockDim.y + threadIdx.y;  
  
    if(r < out_width && s < out_height)  
    {  
10        unsigned int sum_img = 0;  
        unsigned int sum2_img = 0;  
        unsigned int sum_img_tmpl = 0;
```

```

15     for(int j = 0; j < conf.tmpl_height; j++) {
16         for(int i = 0; i < conf.tmpl_width; i++) {
17             unsigned int val_img = tex2D(tex_image, r + i, s + j);
18             unsigned int val_tmpl = tex2D(tex_template, i, j);
19
20             sum_img += val_img;
21             sum2_img += val_img * val_img;
22             sum_img_tmpl += val_img * val_tmpl;
23         }
24     }
25     // ...
26 }

```

**Listing (3.1)** A trivial Template Matching Kernel

Each thread executes the code shown in listing 3.1 and thereby computes:

- `sum_img` the sum of all pixel values in the current search image patch
- `sum2_img` the sum of all squared pixel values in the current search image patch
- `sum_img_tmpl` the cross correlation between the sub-image at offset  $(r,s)$  and the reference image

#### Identify potential Performance Improvement

An analysis of the trivial approach revealed great potential for performance improvement in the kernel in listing 3.1. Especially the nested loop to compute the three variables `sum_img`, `sum2_img` and `sum_img_tmpl` is proved to be slow. Consequently it would be best, if we could replace this computation with a more sophisticated approach and get rid of the whole loop.

**Summed Area Tables** Considering the execution path of each thread computing a coefficient in listing 3.1, we can observe that a lot of work is done multiple times. When a thread iterates over the pixels in the sub-image at offset  $(r,s)$  their values are accumulated to build the integral of the pixel values and of the squared pixel values, respectively. In this case the thread one to the right, at position  $(r+1,s)$ , iterates basically the same pixels except of column  $r$  and column  $r+1+w_R$ . So if we could simply subtract column  $r$  from the results at offset  $(r,s)$  and add the values at column  $r+1+w_R$ , we could calculate the values at the sub-image offset  $(r+1,s)$  given the values at  $(r,s)$  with a small effort. In fact this is almost what we do to eliminate the computations of `sum_img` and `sum2_img` from the loop. Before we call the kernel in listing 3.1 we compute a so-called Summed Area Table (SAT) or also called integral image, introduced by [6]. The structure of a SAT is fairly simple as the value at some point  $(x,y)$  in the table is the sum of all the values above and to the left of this position.

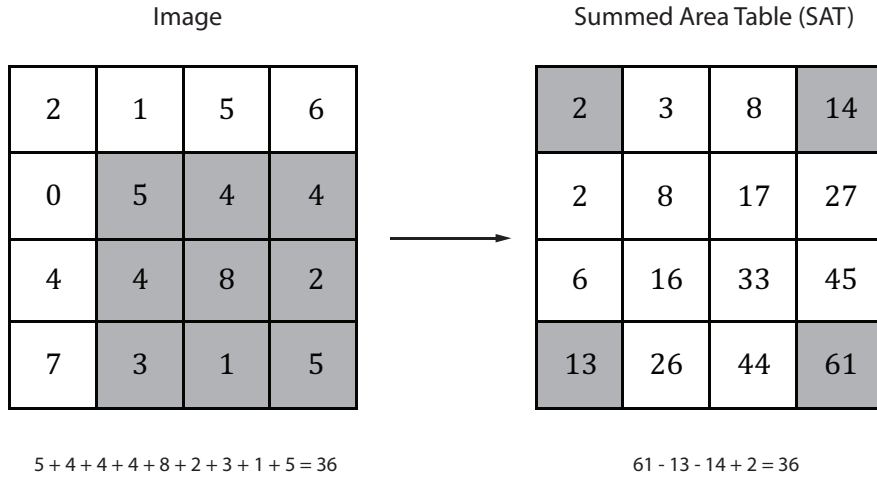
In figure 3.16 we give an example of how the SAT works: We can get the sum of the pixels in the gray area in the image on the left by simply reading the four marked values from the SAT and add them up as described in equation 3.19.

As you can imagine, with this concept we can get a large improvement the template matching kernel in listing 3.1 as we can pre-compute the whole SAT once and then each thread does simply four global memory reads and computes the sum of the desired area instead of iterating all the pixels in the area. Of course this concept works for the sum of the squared pixel values as well. In this case the sums stored in the SAT are the accumulated squared pixel values.

With the help of a SAT it is possible to compute the sum of the pixel values in any rectangular area in an image by sampling the SAT at the four corner pixels of the area. We can then compute the value for `sum_img` at a sub-image offset  $(r,s)$  in constant time by

$$s(r,s) = SAT(u+w_R,v+h_R) - SAT(u,v+h_R) - SAT(u+w_R,v) + SAT(u,v) \quad (3.19)$$

where  $u = r - 1$  and  $v = s - 1$ .



**Figure (3.16)** Computation of a Summed Area Table

Older CUDA architectures do not support double floating point or long integer data types which means we are limited to the bounds of representable numbers with these data types with the SAT approach. If we look at the SAT in figure 3.16 we can see very fast growing values towards the bottom right corner. If the SAT gets larger and larger we reach the limit of representable integers pretty fast. Specifically the integer has 32 bits and since we are dealing with intensity images with pixel values between  $[0, 255] = [0, 2^8 - 1]$ . Assuming we need 8 bits for the intensities we get the following equation for the maximum dimensions of a square sized image:

$$\log_2(a^2 \cdot 2^8) = 32 \Leftrightarrow a = \sqrt{\frac{2^{32}}{2^8}} \Leftrightarrow a = 2^{12} \quad (3.20)$$



As we can see from equation 3.20 the SAT size, which is equal to the input image size, should theoretically not be larger than  $2^{12} \times 2^{12} = 4096 \times 4096$  pixels. However because natural images are usually never all white this is only a theoretical value and the actual image size can be larger in practice. Of course this limitation gets even more significant for the squared integral image but in this case the use of floating point values provides an acceptable workaround for the price of lower precision on larger values.

We did not implement the SAT computation ourselves, instead we used the existing NVIDIA NPP [5] implementation. For a description on how to implement the SAT computation in CUDA refer to [15], Chapter 39 on Parallel Prefix Sum (Scan) with CUDA. The NVIDIA NPP implementation uses integers to store the standard SAT and floating point values for the squared SAT values. As a consequence of the use of the NPP implementation the image size in our case is, as previously described, theoretically limited to  $4096 \times 4096$  pixels.

**Cross Correlation in the Frequency Domain** Unfortunately the computation of the third variable `sum_img_tmpl` in listing 3.1 can not be eliminated the same way as the other two. This means, the nested loop has to remain in the kernel. However with the loop and the texture fetches still being present the speedup from the previously discussed summed area tables would be almost zero. Consequently we have to find another way to produce the same result for `sum_img_tmpl` but without doing all these iterations.

To the end of section 3.2.1 in the description of the correlation coefficient we mentioned the possibility to compute the cross correlation by transforming the image and the template into frequency domain, followed by a point wise complex conjugate multiplication and a final inverse transformation. Doing these steps allows us to pre-compute the cross correlation for all  $(r,s)$  offsets of the sub-image and therefore eliminate the nested loop in the kernel in listing 3.1.

The computation of the cross correlation through frequency domain starts with a 2D discrete Fourier transform. We use the NVIDIA CUDA Fast Fourier Transform (CUFFT) library [4] to transform image  $I$  and the reference image  $R$  into frequency domain. Because the point wise complex conjugate multiplication in frequency domain works best when  $I$  and  $R$  are the same size we previously had to pad the reference image  $R$  with zero values. To get the final result of the cross correlation of  $I$  and  $R$  we execute an inverse FFT.

With this optimization we eliminated the whole nested loop in 3.1 and the final matching kernel is now free of any loops. In the next section we discuss the impact of these optimizations and analyze how the final template matching algorithm performs.

### 3.2.3 Results

After optimizing the trivial implementation we analyzed the performance of the optimized version and compared the results with its OpenCV CPU counterpart. Additionally we wondered weather there is a difference in runtime between using a large template image and a small one.

### Test Components and Scenarios

The following runtime measurements are based on the test images shown in figure 3.17. Image 3.17a is the input image  $I$  in which we search for one of the template images  $R_1$  or  $R_2$  visualized in figure 3.17b. We have chosen this image because it contains many similar flowers of different sizes which makes it interesting to observe the effects of matching a template image that does not perfectly match.



**Figure (3.17)** Template Matching Search Image and Reference Images

All runtime measurements in this section come from tests on an NVIDIA Quadro FX3800 card in a machine described in section A.1 in table A.2. In the same way as with the Hough transform we can observe the scalability of the algorithm implementation on different graphics devices. The measurements on the smaller card, an NVIDIA Quadro FX580 in a machine described in table A.1, showed slower results. Nevertheless they were still much faster than the measured CPU runtimes.

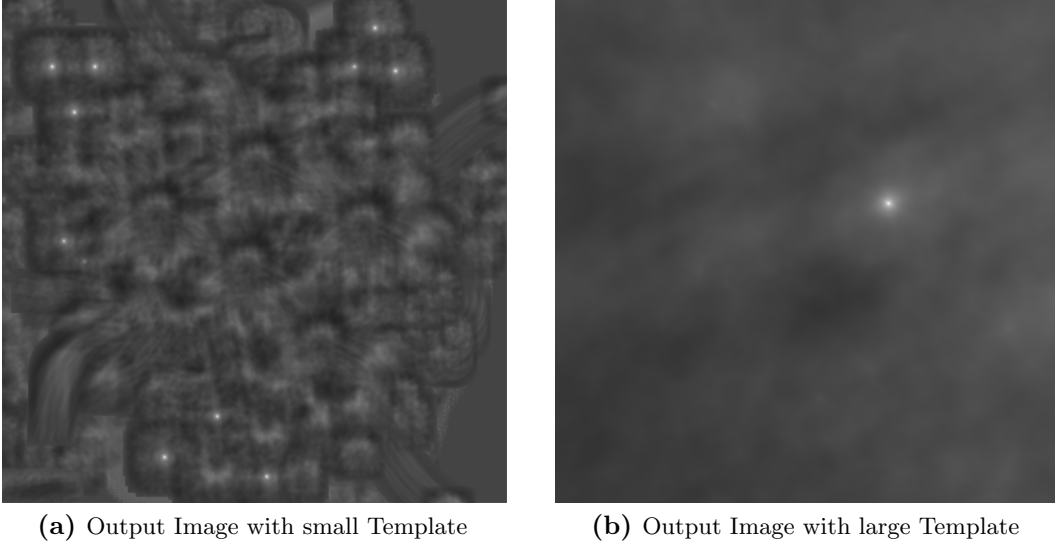
### Algorithm Output

In case our implementation works correctly we expect the resulting output image to indicate the template matches in image  $I$  with a bright area at the matching position of  $R_1$  and  $R_2$ . The properly place the template image  $R$  to the matching position we have to align its upper left corner to the center of the bright area. Places where the reference image  $R$  does not match perfectly, but still a vast majority of the pixels match, the location should appear only a bit brighter.

**Small Template Image  $R_1$**  We start the test scenarios with the search of the small template image  $R_1$  visualized in image 3.17b. Counting the bright points in the output image 3.18a the algorithm result suggests us to find 11 flowers of the same size and

orientation in search image 3.17a. If we cross-check this statement we can find indeed exactly these 11 flowers at the bright positions indicated in the output image.

Interesting to notice in output image 3.18a are the two flowers on the middle of the left border. The response from one of them is remarkably smaller than the signal of the other bright points. If we search for the reason of this effect we find the answer in input image 3.17a. The flower with the weak response is partly covered by its neighbours and therefore does not match pretty good. Fortunately the response is still good enough to produce a visible match.



**Figure (3.18)** Template Matching Output Images with small and large Templates

**Large Template Image  $R_2$**  In the second test we chose a larger template, namely the template  $R_2$  which is also illustrated in image 3.17b. Obviously this template exists only once in the search image, which is why we expect to receive only one bright point as result. Unsurprisingly we get the output in image 3.18b with the expected result in the upper right quadrant.

It is striking that the resulting image 3.18b looks like a blurred picture compared to image 3.18a. This effect arises from the larger size of the template and its structure, since there are a lot more values which contribute to the result of a certain pixel. Because the output value for a pixel is the mean of a larger area the pixel values over the whole image get averaged.

### Runtime Measurements of Template Matching

First of all we would like to determine the runtime differences between using a small and a large template. While the small template is approximately only 0.6% of size of the search image  $I$  and the large template counts around 17% their runtimes are almost equal. As we can see in figure 3.19 the difference in the CUDA implementation is so

small that the two graphs are almost congruent. We can find the answer to this in two algorithm parts, specifically the integral image computation of the template and the actual template matching procedure. On the one hand the computation of the integral image is very fast with a small template, whereas the template matching part takes longer. On the other hand, with a large template the situation is the other way round. Even if the runtime distribution between the two algorithm procedures is different, they surprisingly both sum up to the same total runtime, as illustrated in table 3.3. The remaining algorithm parts have only small fluctuations because they depend mostly on the size of the search image.

<b>Algorithm: TM CUDA (Image Size <math>1024 \times 1024</math>)</b>	<b>small template (<math>76 \times 76</math>)</b>	<b>large template (<math>413 \times 413</math>)</b>
Integral Image Procedure	0.9 ms	1.9 ms
Template Matching Procedure	3.2 ms	2.1 ms
<b>Total</b>	<b>4.1 ms</b>	<b>4.0 ms</b>

**Table (3.3)** Runtimes of Template Matching Parts

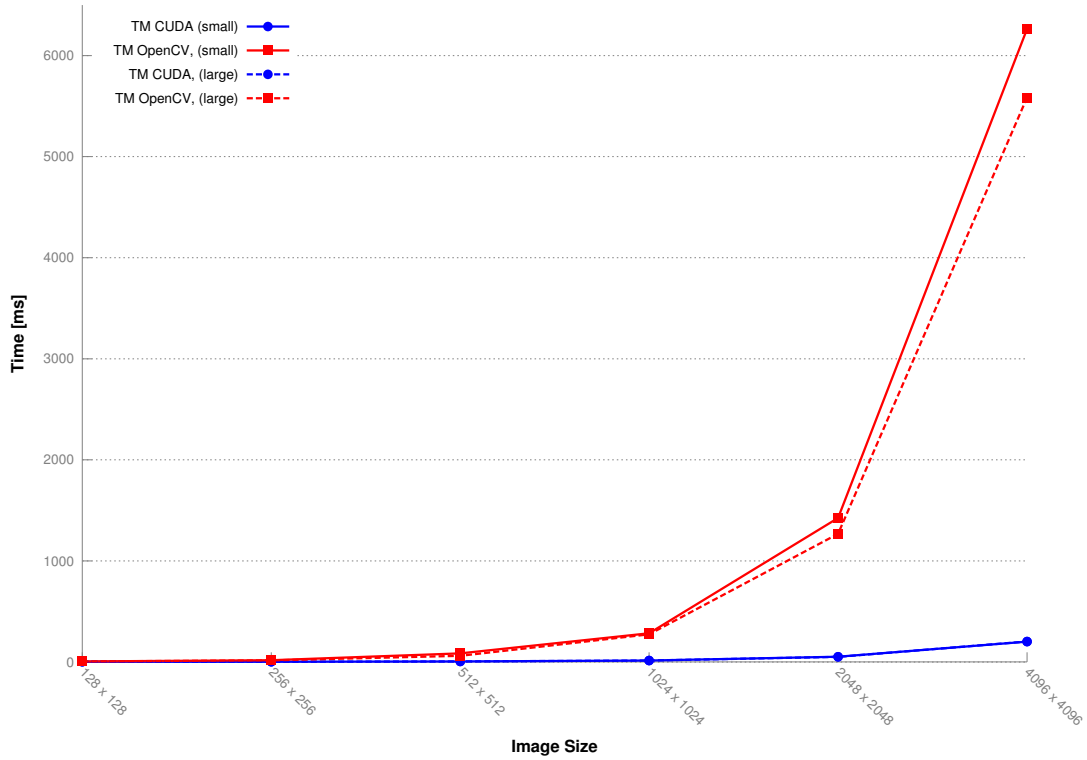
To conclude, it is generally unimportant in case of our CUDA implementation, whether we use a large template image or a small one, as the total runtime is approximately the same.

In contrast to the previous discussion, the measurements of the OpenCV implementation showed a stronger response in runtime on varying template sizes. If we count the steps to check the template against each offset  $(r,s)$  in the search image, we recognize that the algorithm needs to do more individual steps to complete with a smaller template. Even if the amount of calculations is higher on large templates, the CPU implementation is faster at that. Here we assume that the larger amount of individual iterations with smaller templates has a negative effect on the runtime.

By comparing the CUDA implementation with the OpenCV reference, we can see remarkably good results. As illustrated in figure 3.19 the graphs of the two implementations begin to drift apart rapidly when the image size rises. Again the benefit of the higher GPU computing power is evident when more data can be processed. To receive an impression how much faster the CUDA implementation is, we consider the runtime measurements in table 3.4 where we notice an performance gain of factor 31!

<b>Algorithm</b>	<b><math>128 \times 128</math></b>	<b><math>256 \times 256</math></b>	<b><math>512 \times 512</math></b>	<b><math>1024 \times 1024</math></b>	<b><math>2048 \times 2048</math></b>	<b><math>4096 \times 4096</math></b>
TM CUDA (small)	2.3 ms	3.1 ms	5.8 ms	14.3 ms	51.8 ms	201.9 ms
TM OpenCV (small)	6.1 ms	17.2 ms	84.8 ms	285.0 ms	1425.3 ms	6263.1 ms
TM CUDA (large)	2.1 ms	2.9 ms	5.6 ms	14.5 ms	52.0 ms	203.1 ms
TM OpenCV (large)	5.6 ms	15.7 ms	61.4 ms	273.7 ms	1268.8 ms	5584.7 ms

**Table (3.4)** Template Matching Runtimes with small and large Templates



**Figure (3.19)** Runtime Measurements of Template Matching

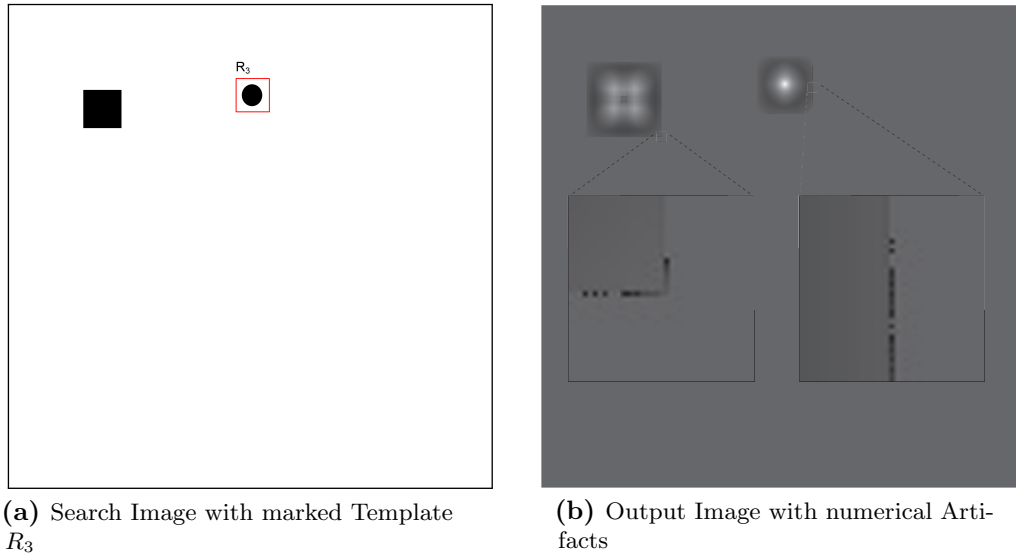
### Numeric Stability of Template Matching Implementation

Generally NVIDIA GPUs with CUDA compute capability 1.2 and lower do not support double precision floating point operations. Consequently all produced device code using double precision, is forced to use single precision on these older architectures. The limitation to single precision also influenced our implementations as we used an NVIDIA Quadro FX 580 device with compute capability 1.1. to test our algorithms.

In the paragraph about summed area tables on page 27 we discussed the numeric limitations of the SAT approach. When the table becomes too large, the values to the bottom right are likely to be bigger, then the representable numbers with a 32 bit integer. To solve this problem, we switched to single precision floating point values with the price of less precision.

In the context of our template matching algorithm the reduced accuracy of the floating point computations gets visible in numeric artifacts in the output image as shown in figure 3.20b. The limited numeric stability of single precision computations affects the result of our algorithm in the evaluation of the correlation coefficient with equation 3.16. If we do not do anything against these errors, the output image would contain false-positive matches. To counteract these artifacts we introduced some tricks in the template matching kernel to increase the stability and reduce the number of false-positives inspired by the OpenCV implementation. Specifically we check the plausibility of the data at neuralgic places and introduced fault tolerances in some

computations. In case the algorithm detects values that make no sense, they are set a defined value, that does not affect the result negatively.



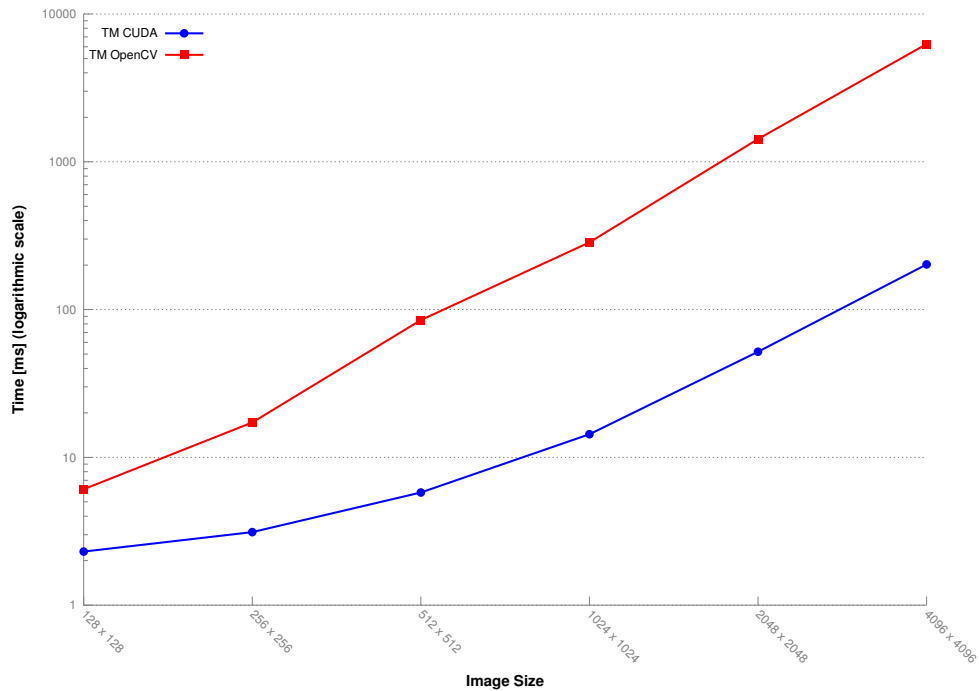
**Figure (3.20)** Template Matching with Test Image to visualize numerical Artifacts

An alternative to overcome the problems numeric stability causes, is to use double precision floating point values. However, devices with hardware support for double precision floating point numbers are not yet widespread.

## 4 Conclusions

In this chapter we summarize our most important insights we gained during this work. In general we can formulate the following main statements, where most of them are related to the implementation of algorithms in CUDA.

- *Asymtotic Behaviour:* Even if the graphs of our performance measurements look very nice, they deceive in their perception. When examining the curves in figure 3.19, one might think the CUDA implementation has almost linear runtime, which it has not. If we take a look at the CPU and GPU runtime measurements independently, we can see that the graphs of the CUDA implementations have similar progression in time as the CPU code. In fact, both implementations have exponential runtime behavior. We illustrate this insight by plotting the same curves in figure 3.19 but this time with a logarithmic scale of the time axis.



**Figure (4.1)** CPU and CUDA Runtime with logarithmic Scale

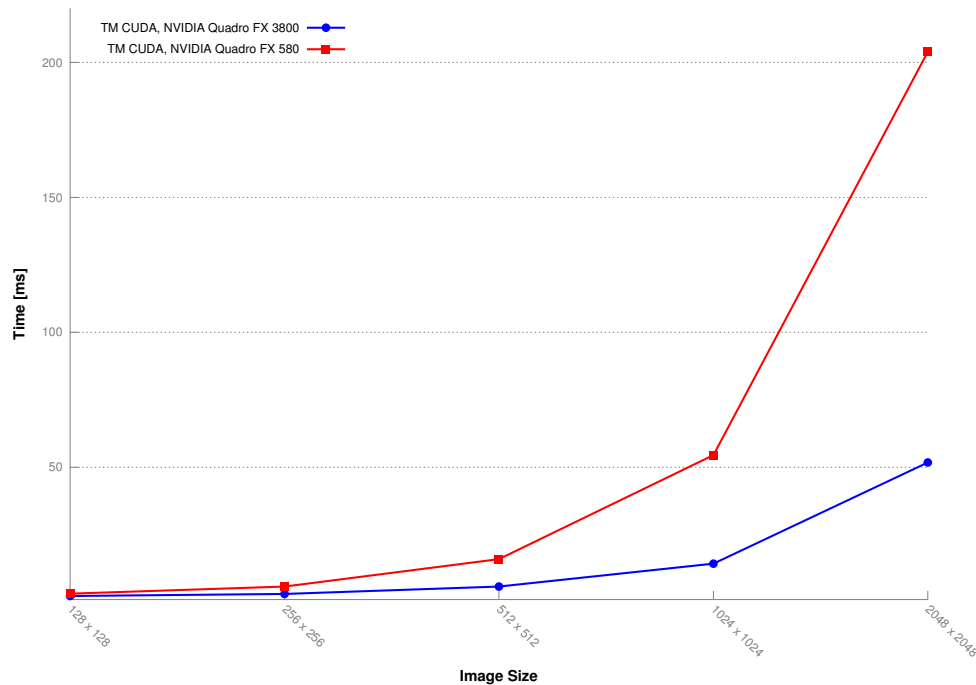
The logarithmic scale allows us to easily recognize a nearly linear progression of both curves in figure 4.1 and a roughly constant offset between the CPU and the GPU curve. With these observations we can verify that both implementations run in exponential time and accordingly their limiting behavior has to be the same.

We observe that the parallelization of some algorithm actually does not change the  $O$  notation of the algorithm itself, unless we implement it in a completely different way. To illustrate this fact consider the following example: Instead of executing some steps five times repeatedly a complete parallelization of this part results in a speedup of a factor of five. However the factor five is not relevant in an asymptotic analysis and hence the  $O$  notation remains unchanged when parallelizing. With CUDA we distribute the computation to multiple processors that compute in parallel and hence the asymptotic behavior remains unchanged as well, just the time the curve starts to increase heavily gets shifted to larger input sizes.

*The  $O$  notation of some CPU algorithm ported to the GPU remains unchanged unless we find a complete new attempt to implement it.*

- **Scalability:** It was nice to see how our algorithms performed better, only by changing the underlying hardware. The algorithms scaled to better hardware and made use of the increased number of resources, without changing anything in our implementations.

In fact we could compare the runtime performance of our algorithms on two different graphics cards which basically differed in the amount of global memory and multiprocessors. Not only we could work with images that had four times more pixel data with just twice the amount of memory, but also the computations were automatically scattered to more available GPU cores. This resulted into an enlarged parallelism and an enormous performance gain.



**Figure (4.2)** Scalable CUDA Implementations on different Graphics Devices



---

In figure 4.2 we illustrate of how an algorithm scales to better hardware. The graphs in this figure show the runtime measurements of the template matching algorithm executed on the NVIDIA Quadro FX 580 and the NVIDIA Qudaro FX 3800, respectively. As we realized that CUDA implementations usually are nicely scalable, it is important to always indicate the graphics card model with runtime measurements, otherwise they are meaningless.

*It is essential to have a sufficient graphics card configuration to achieve enough parallelism in CUDA applications.*

- *Intermediate Data Overhead:* When mapping data structures from CPU implementations to a "CUDA-suitable" format we often generated intermediate data that affected the memory consumption negatively. This means, that sometimes we introduced helper arrays with meta data. The problem with this data is that they additionally stress memory consumption and therefore can limit the variety of possible implementations.

*Meta data affect memory consumption negatively. Because this intermediate data cannot be avoided, it is important to keep their overhead small.*

- *Memory Limitations:* Memory limitations were one of the most serious problems we had to deal with. When implementing our algorithms we repeatedly encountered problems with not having enough memory. It was sometimes not easy to find ways to reduce the memory usage and required a lot of effort to come up with appropriate new solutions. The most simple solution to overcome memory issues is to replace the graphics device with another larger GPU with more main memory. Obviously it makes no sense to simply buy a better graphics device to get an application to work when the application code itself has a bad design. Even with a high performance GPU it is important to produce high quality code which carefully uses the provided resources.

*In CUDA implementations we sometimes have to deal with memory shortage. To counteract this problem programmers have to mind about good code design.*

- *Low-cost vs. High-end Graphics Cards:* With reference to the previous points Scalability and Memory Limitations we can say that a higher cost graphics card is definitively a worthwhile investment. Higher-priced cards in general have more resources to satisfy the demand on memory and computing power. While it was hard to beat the OpenCV implementation of Hough transform with the Quadro FX 580, the same algorithm on an NVIDIA Quadro FX 3800 surpassed the OpenCV Hough transform with ease.

Buying a high-end computing solution does not mean that you have to spend a lot of money. Todays high-end graphics devices cost a few thousand swiss francs, which is still cheap compared to a compute cluster with an equal amount of cores. Accordingly graphics devices have a much better cost-performance ratio.

*Spending a bit more money in a better graphics device is a worthwhile investment because high-end GPUs provide much more computing power.*

- *Optimization:* It should be avoided to try to implement an algorithm with a lot of optimizations from scratch. The best way from our experience to get well performing CUDA implementations can be described as follows:
  1. Start with a simple implementation and get it to work.
  2. Measure the runtime to get an overview where your algorithm spends most of the time.
  3. Optimize the algorithm on the basis of the previously collected data
  4. Iterate steps 2 and 3 until the the runtime performance becomes satisfying.

In case it is difficult to improve the performance after several iterations, it might be a solution to think of a completely different approach.

*Do not try to implement an optimized algorithm from scratch, without have a working implementation yet. Start with a simple solution and try to improve it.*

# Appendix A

## Development Environment Setup

### A.1 Hardware and Software Tools

Table A.1, A.2 and Table A.3 summarize the hardware and software tools we used for the development of this term project. We tested our algorithms on two different machines with different CPU's and different graphics devices.

#### A.1.1 Hardware

##### Low-cost Machine

Tool	Description
Processor	Intel Xeon CPU X3450, 8 × 2.67 GHz
Memory	8192 MB
Graphics Device	NVIDIA Quadro FX 580 - 32 CUDA Cores - 512 MB Memory - Compute Capability 1.1

**Table (A.1)** Low-cost Project Hardware

##### High-end Machine

Tool	Description
Processor	Intel Xeon CPU E5520, 8 × 2.27 GHz
Memory	4096 MB
Graphics Device	NVIDIA Quadro FX 3800 - 192 CUDA Cores - 1024 MB Memory - Compute Capability 1.3

**Table (A.2)** High-end Project Hardware

### A.1.2 Software

Tool	Description
Operating System	Ubuntu 10.10 - 64-bit
CUDA Driver Version	4.0
CUDA Runtime Version	4.0
Build System	CMake 2.8
Development Environment	Eclipse CDT (Helios)
Libraries	OpenCV, CUFFT, Thrust, NVIDIA NPP, CUDA Utility Library (cutil)

**Table (A.3)** Project Software

Writing CUDA code is basically programming C. Therefore we used the basic Eclipse CDT development environment with some small adjustments to support the build process and readability of the source code. To build our programs we used the CMake build system which made a good job on a "one-click" build process.

## A.2 Setup CUDA

The simplest way to setup the CUDA development tools is to just follow the "NVIDIA CUDA C Getting Started Guide" for the desired platform. This guide is available from the NVIDIA Developer Zone (<http://developer.nvidia.com>).

Note that in case you are working on hardware that has a built-in graphics chip and a dedicated graphics chip (in our case a MacBook Pro with a built-in Intel graphics chip and a external NVIDIA graphics processor) you may have to force the device to use the NVIDIA hardware to be able to run your programs. If a CUDA program is executed and the hardware is not running on a CUDA capable device one might get an error such as

```
cudaGetDeviceCount FAILED CUDA Driver and Runtime version may be mismatched
```

In this situation you have to force your hardware to switch to the NVIDIA capable device before you run the program.

If you are on a Mac you can install a nice free tool called "gfxCardStatus" (<http://codykrieger.com/gfxCardStatus/>) which indicates you with a little icon on which chip one is working and helps you to easily force your Mac to use the NVIDIA chip.

## A.3 Setup Eclipse CDT with CMake and CUDA

This section describes how to setup and configure the Eclipse installation to work with the build system CMake to build the CUDA programs.

### A.3.1 Add File Type .cu

Because Eclipse does not know the extension .cu you may configure Eclipse to treat these files as C or C++ source files. To do so do the following steps:

1. go to Window → Preferences
2. open C/C++ → File Types dialog
3. click New...
4. then write as Pattern: \*.cu and select Type: C++ Source File
5. finish the dialog with OK

Now all the .cu files should have code highlighting. The only thing that is not recognized by Eclipse are the kernel launches with the triple brackets (<<<...>>>). Repeat these steps if you would like to have highlighting for .cuh files as well. In this case select as type C++ Header File.

### A.3.2 Setup CMake and CMake Configuration in Eclipse

We assume that you have already downloaded and installed CMake on your system and you are aware of the concepts of CMake. Install the CMakeEd plugin for Eclipse (<http://cmakeed.sourceforge.net>) which supports you when editing CMakeLists files with code highlighting and code completion. Note, the following documentation lists just the parts of the CMake files needed to compile CUDA code, the complete files however can be found in the provided source material.

First of all, we check for the module FindCUDA.cmake (see listing A.1). This module should come with your CMake installation as a standard module.

```
1  # find cuda
   IF (INCLUDE_CUDA)
       FIND_PACKAGE (CUDA)
5
   IF (CUDA_FOUND)
       MESSAGE ("CUDA has been found")
   ELSE (CUDA_FOUND)
       MESSAGE (FATAL_ERROR "CUDA could not be found")
   ENDF (CUDA_FOUND)
10  ENDF (INCLUDE_CUDA)
```

**Listing (A.1)** Check for FindCUDA.cmake

Tell CMake to build a CUDA program by setting the compile instruction as shown in listing A.2, the link instruction is as usual.

```
1  CUDA_ADD_EXECUTABLE (${EXE_NAME} ${source} ${main})
   TARGET_LINK_LIBRARIES (${EXE_NAME} ${LIBS})
```

**Listing (A.2)** CMake CUDA Executable

These are the only special commands used to instruct CMake to build a CUDA program, nevertheless the FindCUDA module provides a lot more variables and options to customize the build and set NVCC compiler flags. All these variables can be found on the web in the official CMake documentation.

Now that we have covered the special CMake commands we have to configure the Eclipse project properties to work best with CMake. Follow the steps below to set up Eclipse with CMake.

1. start a new project with  
C++ Projekt → Makefile project → Empty Project
2. create a build folder for out-of-source builds in the project root  
File → New → Folder → Folder Name: build
3. adjust the project properties  
Project → Properties → C/C++ Build
  - set the Build location to the just created build folder
  - uncheck Generate Makefiles automatically
  - click OK to finish
4. edit the make targets On the right side of the editor window there should be a tab called Make Targets. Click this tab and select your project, then right click and select New...
  - set the target name to cmake
  - uncheck Same as the target name and Use builder settings
  - delete the text in the field Make target
  - write in Build command the text `cmake ..` (you have the possibility to set additional CMake command line arguments right after `cmake`)
5. Now you can start coding. Before you start the first build and after every change on the CMakeLists files you have to double click the cmake target created in the previous step. Afterwards compile your project as usual with the compile button.

# Bibliography

- [1] Tobias Binna and Markus Hofmann.  
Massive parallel image processing, 2010.
- [2] Wilhelm Burger and Mark James Burge.  
*Digital Image Processing, An Algorithmic Introduction using Java.*  
Springer Science + Business Media, LLC, 2008.
- [3] NVIDIA Corporation.  
*NVIDIA CUDA C Programming Guide*, 4.0 edition, March 2011.
- [4] NVIDIA Corporation.  
*NVIDIA CUFFT Library User Guide*, February 2011.
- [5] NVIDIA Corporation.  
*NVIDIA Performance Primitives Library User Guide*, 4.0 edition, February 2011.
- [6] Franklin C. Crow.  
Summed-area tables for texture mapping.  
*SIGGRAPH Comput. Graph.*, 1984.
- [7] Richard O. Duda and Peter E. Hart.  
Use of the hough transformation to detect lines and curves in pictures.  
*Commun. ACM*, 15:11–15, January 1972.
- [8] C. Galambos, J. Matas, and J. Kittler.  
Progressive probabilistic hough transform for line detection.  
1999.
- [9] G. Gerig.  
Linking image-space and accumulator-space.  
page 113, 1987.
- [10] B. Giesler, R. Graf, and R. Dillmann.  
Fast mapping using the log-hough transformation.  
1998.
- [11] Paul V. C. Hough.  
Method and means for recognizing complex patterns.  
Patent, December 1962.
- [12] H. Kälviäinen, P. Hirnoven, L. Xu, and Oja E.  
Probabilistic and non-probabilistic hough transforms.  
*Image and Vision Computing*, 13(4):239 – 252, May 1995.
- [13] Cuda Data Parallel Primitives Library.  
<http://code.google.com/p/cudpp/>.  
page last visited, June 2011.

- [14] OpenCV Library.  
<http://opencv.willowgarage.com/wiki/>.  
page last visited, June 2011.
- [15] Hubert Nguyen.  
*GPU Gems 3*.  
Addison-Wesley Professional, 2007.
- [16] Thrust project.  
<http://code.google.com/p/thrust/>.  
page last visited, May 2011.
- [17] S. Suchitra Sathyanarayana, R.K. Satzoda, and T. Srikanthan.  
Exploiting inherent parallelisms for accelerating linear hough transform.  
*IEEE Transactions on Image Processing*, 18(10):2255 – 2264, October 2009.
- [18] Canny Wikipedia.  
[http://en.wikipedia.org/wiki/Canny\\_edge\\_detector](http://en.wikipedia.org/wiki/Canny_edge_detector).  
page last visited, June 2011.
- [19] L. Xu and Oja E.  
Randomized hough transform.  
*Encyclopedia of Artificial Intelligence*, pages 1354 – 1361, 2009.