Självständigt arbete på grundnivå

Independent degree project - first cycle

Datateknik
Computer Engineering

Correspondence-based pairwise depth estimation with parallel acceleration

Nadine Bartosch

Mittuniversitetet
MID SWEDEN UNIVERSITY

Campus Härnösand Universitetsbacken 1, SE-871 88. **Campus Sundsvall** Holmgatan 10, SE-851 70 Sundsvall.
Campus Östersund Kunskapens väg 8, SE-831 25 Östersund.
Phone: +46 (0)771 97 50 00, Fax: +46 (0)771 97 50 01.

MID SWEDEN UNIVERSITY
Department of Information Systems and Technology (IST)

Examiner: Ulf Jennehag, ulf.jennehag@miun.se
1. Supervisor: Elijs Dima, elijs.dima@miun.se
2. Supervisor: Martin Kjellqvist, martin.kjellqvist@miun.se
Author: Nadine Bartosch, naba1700@student.miun.se
Degree programme: Computer Engineering, 180 credits
Main field of study: Thesis Project DT099G
Semester, year: Spring, 2018

Abstract

This report covers the implementation and evaluation of a stereo vision correspondence-based depth estimation algorithm on a GPU. The results and feedback are used for a Multi-view camera system in combination with Jetson TK1 devices for parallelized image processing and the aim of this system is to estimate the depth of the scenery in front of it. The performance of the algorithm plays the key role. Alongside the implementation, the objective of this study is to investigate the advantages of parallel acceleration inter alia the differences to the execution on a CPU which are significant for all the function, the imposed overheads particular for a GPU application like memory transfer from the CPU to the GPU and vice versa as well as the challenges for real-time and concurrent execution. The study has been conducted with the aid of CUDA on three NVIDIA GPUs with different characteristics and with the aid of knowledge gained through extensive literature study about different depth estimation algorithms but also stereo vision and correspondence as well as CUDA in general. Using the full set of components of the algorithm and expecting (near) real-time execution is utopic in this setup and implementation, the slowing factors are inter alia the semi-global matching. Investigating alternatives shows that results for disparity maps of a certain accuracy are also achieved by local methods like the Hamming Distance alone and by a filter that refines the results. Furthermore, it is demonstrated that the kernel launch configuration and the usage of GPU memory types like shared memory is crucial for GPU implementations and has an impact on the performance of the algorithm. Just concurrency proves to be a more complicated task, especially in the desired way of realization. For the future work and refinement of the algorithm it is therefore recommended to invest more time into further optimization possibilities in regards of shared memory and into integrating the algorithm into the actual pipeline.

Keywords: Depth estimation, disparity, stereo vision, stereo correspondence, NVIDIA, GPU, CUDA, parallelization

Acknowledgements

Foremost, I want to express my sincere gratitude to my two supervisors Elijs Dima and Martin Kjellqvist who both supported me throughout my study, introduced me to the work methods of the research team at the university, provided me with important and useful knowledge, offered constructive and helpful feedback and just gave me the impression to be always welcome when having questions or difficulties during my study.

Furthermore, I would like to thank Mårten Sjöström who gave me the chance to work on this interesting and challenging topic and who provided me with encouragement since the beginning as well as insightful comments and ideas.

Finally, I would like to thank my family and friends for supporting me in various ways during my study but also the whole study program and for always encouraging me to work hard and to show resilience in my actions.

Table of Contents

Terminology

Acronyms/Abbreviations

CPU	Central processing unit
GPU	Graphics processing unit
CUDA	Compute unified device architecture architecture
2D	Two-dimensional
2.5D	Two-point-five-dimensional
3D	Three-dimensional
PCIe	Peripheral component interconnect express
CTA	Cooperative thread array
SAD	Sum of absolute differences
DSI	Disparity space image
WTA	Winner takes all
ASW	Adaptive support weights
SGM	Semi-global matching
MPS	Multi-process service
MP	Multi-processor
CT	Census Transform
MC	Matching cost
HD	Hamming distance
B	Bytes
MB	Megabytes
MB/s	Megabytes per second
secs	seconds
ms	milliseconds

Mathematical notation

I	Intensity value of the pixel
d	Current disparity: horizontal offset between pixel in left and pixel in right image
$width$	Width of original input image
$height$	Height of original input image
x	$x=[0, width-1]$, x-position value in image of current pixel
y	$y=[0, height-1]$, y-position value in image of current pixel
$CensusTransform(I(x, y))$	Result for Census Transform for the Intensity of a certain pixel (x,y) in the image
$MC_{width, height, max.disparity}$	Result for Matching cost calculation, results in a 3D-cost structure with *width,height* from the images as one plane and the max. disparity range as the depth of the structure
$ABS.()$	Calculation of absolute differences
dir	Direction for scanline in SGM, either y-pos, x-pos, x-neg or y-neg through image
$\text{3DCostpath}_{width, height, max.disparity}(x, y, d)$	Result of scanline calculation in a certain direction with the image width and height for one pane and the disparity range as depth
$dir.x$	Calculation of previous pixel in horizontal scanline direction either for x-pos, x-neg. Possible values for dir.x are described as follows: $dir.x=0,\ \{dir \mid dir=\text{y-pos, y-neg}\}$ $dir.x=1,\ \{dir \mid dir=\text{x-pos}\}$ $dir.x=\text{-}1,\ \{dir \mid dir=\text{x-neg}\}$
$dir.y$	Calculation of previous pixel in vertical scanline direction either for y-pos, y-neg. Possible values for dir.y are described as follows: $dir.y=0,\ \{dir \mid dir=\text{x-pos, x-neg}\}$ $dir.y=1,\ \{dir \mid dir=\text{y-pos}\}$ $dir.y=\text{-}1,\ \{dir \mid dir=\text{y-neg}\}$
$\min_i$	Selection of minimum disparity value of previous pixel in SGM where $i=[0, Max.disparity-1]$, excluding disparities equal or with a difference of 1 to the disparity of the current pixel

$\min_{k}$	Selection of minimum disparity value of previous pixel in SGM where $k=[0, Max.disparity-1]$, to ensure result stays inside bounds
$\min_{d}$	Selection of minimum disparity value in WTA for disparity image, $d=[0, Max.disparity-1]$
$3DCost_{width, height, disparity}$	Result of summation of $3DCostpath_{width, height, max.disparity}$ and $MC_{width, height, max.disparity}$
$Disparity_{width, height}$	Final result for disparity image with original image dimensions

1 Introduction

The human perception allows based on a numerous number of stimuli to observe the environment in a spatial manner which makes interactions possible. At the same time the eyes have often the biggest influence on this process as a sense organ. Moreover, gained knowledge as well as subjective perception about the environment are influential. The images of a scene are obtained by the eyes and forwarded to the brain. The brain interprets these which finally results in an inner model of the environment and is called the spatial perception. The whole process is fairly complex which makes a 3D reconstruction of a recorded or observed scene for machines more challenging.

Therefore, dealing with 3D vision is an important aspect in the area of machine vision. Depth estimation describes the process of determining the distance of each point in a certain scene to the camera lens. The initial scenario could be a three-dimensional scene which has a certain complexity and that holds e.g. different objects. Looking at this scene or taking a picture offers but one perspective which results in a distorted image. Each point in this scene is thereby transformed onto the projection plane which is nothing more than a two-dimensional image (2D projection). After the transformation, the depth information (one dimension) has been removed. The purpose of depth estimation algorithms is to regain the depth value out of a two-dimensional image. This information can be encoded as a depth map, e.g. a grayscale 2.5D representation that gives feedback of the distance to each point via a different shade of gray as illustrated in figure 1.

This thesis is to present new knowledge and feedback about GPU-accelerated depth estimation and about parallel processing which can be used to improve the development of projects in this area. Therefore, the overall aim in this thesis is to implement a parallelized correspondence-based pairwise depth algorithm and analyze the behavior and the performance of the algorithm in regards of its speed as well as resource utilization of the GPU.

Figure 1: Example image set "teddy"[1] from the Middlebury dataset [10] for stereo vision, as well as the ground truth of the scene. Surfaces that are displayed in a darker shade are further away, while lighter ones are closer to the camera.

1 http://vision.middlebury.edu/stereo/data/scenes2003/, retrieved 2018-08-01

From the depth map, the spatial structure of this given scene can be retrieved. There are several strategies that enable this goal. There is a distinction between active and passive methods, both approaches have their advantages and difficulties. Active methods, e.g. light-based depth estimation, have naturally a high accuracy, but are expensive. They illuminate the scenery and interpret the reflected light [2]. On the other hand, the passive methods solve depth estimation by computations on the captures image(s) in other words they are called algorithms. There is a distinction between monocular and Multi-view approaches. As the name already indicates monocular solutions use one image to retrieve the depth information, e.g. *depth-from-defocus* is based on the focus characteristics of the image [2]. The other approach is called Multi-view which uses two or more images for the depth estimation. These algorithms have gained a lot of popularity, because of their features. It is possible to let certain types of algorithms run in real-time or realize them on a GPU using parallel computing. The correspondence-based pairwise depth estimation algorithm is one of these approaches which uses two images [2]. An image pair is selected, the second picture has a horizontal offset of the scene. It is similar to the human vision where each eye obtains a slightly different image of the scene which allows the person to perceive the depth. The goal is to find the corresponding points and obtain the depth of them. These points can be at any position and don't necessarily need to be equal to the pixel grid. Pixels are a specific unit for digital images. Triangulation Multi-view geometry, which is based on triangulation is used to finally retrieve the three-dimensional coordinate.

1.1 Background and problem motivation

The capabilities of graphics processing units (GPUs) have increased even more in the last couple of years which allows not only real-time execution of video processing algorithms, but also the option of parallelizing certain processes and even balancing the computing load. The Realistic 3D research team at the Mid Sweden University is currently working with a system of NVIDIA Jetson TK1 single-board general purpose computers that supports GPU acceleration. The device are provided with data from a multi-camera system which produces simultaneous video streams.

Depth estimation itself is an important part of this procedure, why it seems inspiring and challenging to realize an algorithm that can perform this task. But it can prove cumbersome, because the algorithm has to be adapted to suit the particular parallel-processing toolchain. Finding e.g. the right mix of the speedup achieved by parallel execution and slowdown from the parallel data management is one of the difficulties in this process as well as including them in a GPU-accelerated system.

1.2 Overall aim

This thesis is to present new knowledge and feedback about GPU-accelerated depth estimation and about parallel processing which can be used to improve the development of projects in this area. Therefore, the overall aim in this thesis is to implement a parallelized correspondence-based pairwise depth algorithm

and analyze the behavior and the performance of the algorithm in regards of its speed as well as resource utilization of the GPU.

1.3 Scope

The study has its focus on investigation, implementation and finally evaluation of an already existing depth estimation algorithm that was chosen prior to the thesis start. The work also includes a literature study of GPU-acceleration and of the selected depth estimation algorithm. Furthermore, it covers the research of general knowledge in this area. Another focus lies on possible bottlenecks and speed limitations of the used technology and the algorithm. The emphasis lies on the implementation of the existing algorithm on a GPU and the analysis of its performance especially in regards of parallelization. Excluded from this thesis is the development of wholly new depth estimation methods. The transformation of two captured images to rectified ones is also not part of this thesis, it will only be worked with images that meet these requirements already.

1.4 Concrete and verifiable goals

The concrete goals of this thesis are:

- Implementation of a parallelized depth estimation algorithm with CUDA on a NVIDIA capable GPU
- Evaluation of the developed algorithm

The evaluation goal of the thesis has the objective to answer the following questions:

- What sections of the algorithm benefit from parallelization?
- How significant are the imposed overheads?
- How can real-time or near-real-time execution be achieved?
- How can the algorithm facilitate constantly updating image-pair input data streams?

1.5 Outline

Chapter 2 describes the theory behind the realization and provides the reader with a found background knowledge about GPU programming, CUDA and the stereo correspondence problem. Further, chapter 3 deals with the methods that are used to conduct this study and introduces the experiments. Chapter 4 shows the implementation of the actual algorithm with its different components. In chapter 5 the results of the measurements and experiments are presented and chapter 6 offers a conclusion with ethical aspects of this thesis and a personal discussion about the study.

1.6 Contributions

The actual algorithm consists of different components, two components – in detail the Census Transform and the calculation of the Hamming Distance were taken from the free GitHub repository[2] of D. Hernandez-Juarez et al. [13]. During the development process these two functions were slightly altered and therefore are not exactly the same anymore. Besides the support and helpful suggestions of the supervisors Elijs Dima and Martin Kjellqvist was the study only carried out by the author herself.

2 https://github.com/dhernandez0/s, retrieved 2018-08-01

2 Theory

This chapter covers the theory and basic concepts of information of GPU programming and explains the background of stereo correspondence with its different components, challenges and possible solutions. It further describes related work and approaches in this area which have been taken into account for this study.

2.1 GPU programming

This chapter provides some information about GPU-accelerated computing which describes the use of a graphics processing unit (GPU) in combination with a central processing unit (CPU) to find solutions for certain programming challenges [15]. Furthermore, a basic introduction of the parallel programming platform CUDA.

2.1.1 GPU and CPU

The performance of CPUs has been tremendously increased over the recent years, yet the development was hindered by the limitations in the fabrication of integrated circuits which ultimately lead the developers to the conclusion to look for different ways to increase the overall performance of a computer program [1]. The solution was to combine the strengths of a CPU with the ones of a GPU. In figure 2 the main difference of the structure of a CPU and a GPU is presented:

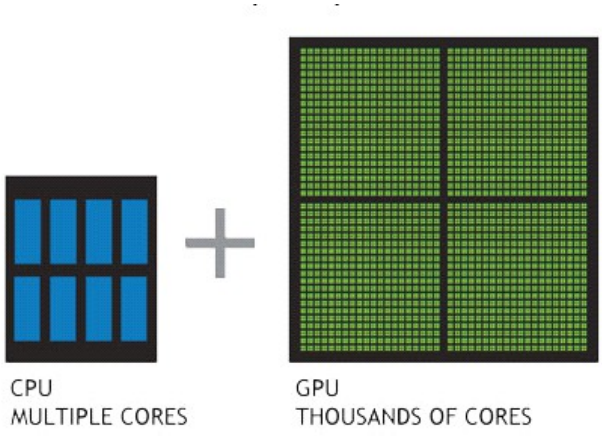

Figure 2: Difference between CPU and GPU[3]

While the CPU holds a few cores optimized for sequential serial processing the GPU consists of an immense parallel architecture of thousands of different cores. These cores are much smaller than the ones in the CPU, but efficient in handling very specific computational instructions, because their main aim lies in working on multiple tasks simultaneously [15]. As an example, it exists the

3 https://graphicscardhub.com/cuda-cores-vs-stream-processors/, retrieved 2018-08-01

so-called SIMD which stands for single instruction, multiple data paradigm. The idea is to apply the same command to multiple places in a dataset. An example can be a simple loop, on the CPU the iterations of the loop run sequentially while on the GPU all iterations could run simultaneously. Originally, a GPU was developed to deal with graphic processing or displaying and video rendering where especially basic mathematical operations are needed which have to be executed over and over again.

There are also some other disfferences between CPU and GPU *cores*, which also describes the limitation of the *graphics* processing unit to the *central* processing unit. A CPU is more flexible, has a larger instruction set, can execute a wider range of tasks and has higher maximum clock rates. Furthermore, the CPU has to take care of the memory management of the GPU which means dealing with allocation, de-allocation and the data transfer from and to the GPU. This results that GPU *cores* are instead often referred to as a mathematical pipeline. Figure 3 shows the relationship between the CPU and the GPU which has to be kept in mind when a program has to be developed:

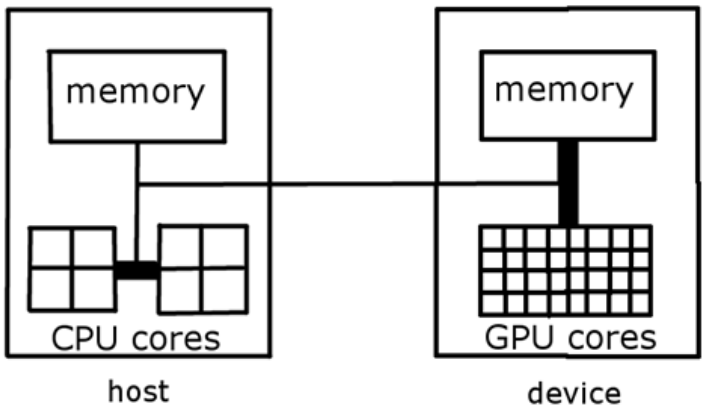

Figure 3: Relationship between CPU and GPU

As already implied, the relationship between these two chips can be described as the one that a human has to a calculator. The CPU which is often referred to as the *host* is in charge, does the higher-level thinking, the structure and the management. Only computationally intensive instructions that would have a slow execution time on a CPU are sent to the GPU which is also called the *device*. The bandwidth between the two units that is shown as a thin line is called a peripheral component interconnect express (PCIe). A PCIe is limited why data should always be sent as late and as rare as possible. The thick line between the GPU cores and its memory indicates that there is a high bandwidth which is the reason that parallel tasks are a lot faster on the GPU, yet there are different kinds of memory types which will be described in better detail in the following section.

2.1.2 GPU parallelism and architecture

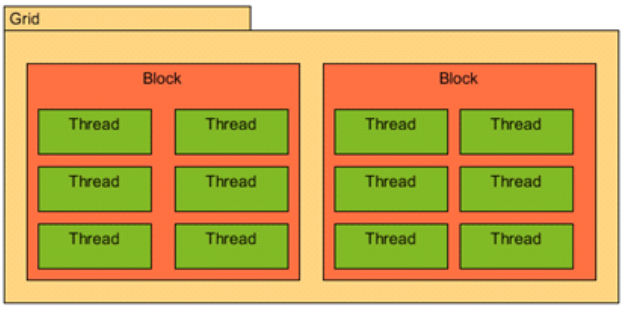

Figure 4: Thread, Block, Grid in a GPU

To be able to understand the workings of a GPU and its capabilities it is important to introduce the most important terminologies. Figure 4 gives a first impression of it and should be explained in greater detail in this subsection. The smallest unit that has to be addressed is a *thread* which handles a sequence of instructions. Threads can be organized within a *block*. These groups of threads are important to the effect, because different options for memory management are available on a GPU. The so-called grid consists of the blocks used to accomplish a task, therefore all threads within a grid execute the same instructions but for example with different values. The instructions or functions that are running on the GPU are referred to as *kernels*. The memory options are various and shown in table 1:

Table 1: Overview of memory types

Memory	**Global**	**Constant**	**Texture**	**Local**	**Shared**	**Register**
Access	Read/Write	Read	Read	Read/Write	Read/Write	Read/Write
Scope	All threads and CPU	All threads and CPU	All threads and CPU	Per thread	Per block	Per thread
Lifetime	All	All	All	Like thread	Like block	Like thread
Speed	Slow, cached	Slow, cached	Slow, cached	Slow, cached	Fast	Fast

Using different kind of memories will have different effects on the program and has therefore also be taken into consideration. The per-thread memory is also often referred to as Registers and is very fast but can only be accessed by the particular thread and is limited in its size. The shared memory is used block-wise and is also accessible quite fast. Besides this, there also exists the global, constant and texture memory. All three of them are slower than the ones already named and are cached, they are organized directly by the host. The global memory is the largest one and simple to work with, the constant memory is a read--only memory and it is optimal if all the running threads in a warp (collection of 32 threads) read exactly the same address. The texture memory offers particular addressing perks which can be in some cases useful for image processing.

A topic related example would be to use shared and global memory in 3D vision. An image with its pixel values is first stored in the global device memory

and has therefore be copied from the host to the device. Then a kernel can be launched, and the needed size of shared memory is allocated within the kernel directly on the device. Every thread of a block can fetch one pixel out of the global memory and store it into the shared memory, this is also possible because each thread gets a unique identifier/index that can be used to address and access the right threads and values. Storing these pixel values in shared memory has the effect that every other thread in the same block can access this value very fast and efficient to do further calculations depending on the purpose of the application. So, a block of a hundred threads each reads and stores the value of a pixel in shared memory which would allow them to access also the other 99 pixels even if they have just read one. This is also one example for the parallelization happening in a GPU, because this command is executed simultaneously by each thread. A possible danger if using shared memory which has to be kept in mind are the so-called *race conditions*.

Race conditions mean that threads have a “race” for particular resources like values stored in shared memory. This can have unexpected effects and can lead to wrong results why they have to be dealt with. If for example calculations are performed and the results are stored in shared memory - to be accessed again in further operations - the case can appear that some threads have not yet finished their calculations and wrong values are taken from other threads that don’t wait on them. Therefore, organizing everything in blocks is also practical for the synchronization of threads. It is possible to let threads within a thread wait at a certain location – it acts like a barrier - in the code until every thread has reached the same position and fulfilled the given task so far.

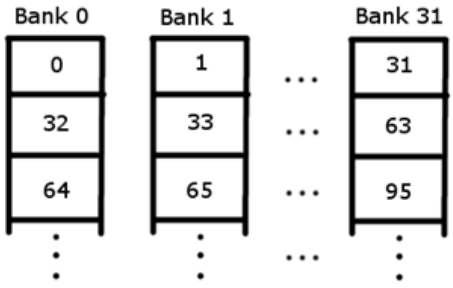

Figure 5: Memory banks. There are 32 different memory banks available, Bank 0 stores words/values with indices divisible by 32, Bank 1 holds words that are divisible by 32 with a reminder of 1 and so forth. The words can be accessed by their indices.

Bank conflicts on the other hand describe a memory access problem that results in a delay and a higher clock rate. (Shared) memory is divided into memory banks – their structure is shown in figure 5 - in which the values or also called words are stored. Each bank is able to access one word over its index at a time. If a warp (collection of 32 threads) requests the values of different indices from one bank at the same time, it has to be serialized which slows down the whole operation. This has to be taken into account if values are stored within the shared memory and the structure should ensure that the words requested at the

same time are in different banks. This can be achieved by adding an extra value/padding that results in an offset of the stored values.

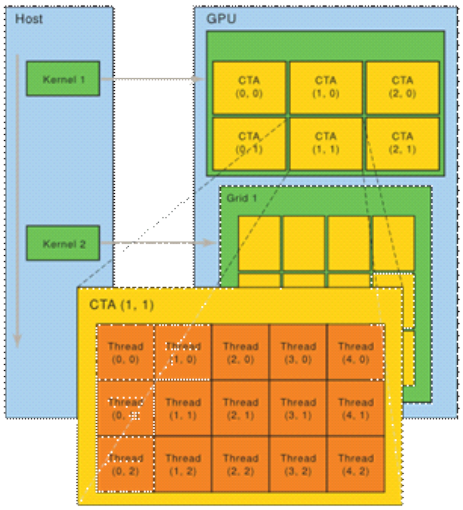

Figure 6: GPU architecture[4]

As already indicated, the main advantage of GPUs is their ability to parallelize instructions which is a result of their special architecture. Figure 6 shows a detail of the GPU architecture. The CPU, or in the figure named the host, sends an instruction-set (kernel) to the GPU. After this, the GPU executes several duplicate realizations of this kernel with its threads. The number of parallel executions depends on the launch configuration for every single kernel, in other words how many blocks and threads have been launched which is also described in 2.1.3. These threads in turn are purposefully grouped into *Cooperative Thread Arrays (CTA)* or simply called blocks. Threads, blocks and even grids can have up to three dimensions, depending on the purpose and the task of the kernel. The figure shows for example two dimensional threads and blocks which is set by the developer before the kernel is launched.

2.1.3 CUDA

CUDA stands for Compute Unified Device Architecture and is a parallel computing platform for programming on GPU and was developed by NVIDIA [16]. Furthermore, it is closely related to C, but also offers C++ capabilities. CUDA uses the key concepts and the architecture of a GPU which were introduced in section 2.1.2.

Figure 7 shows the basic principle of a CUDA program and the steps a developer has to follow. The *__global__* function is a kernel function which is called from the CPU e.g. from the main-function. Inside of the triple chevrons after the function name *some_kernel* is where the actual parallelization is initialized

4 https://docs.nvidia.com/cuda/parallel-thread-execution/graphics/thread-batching.png, retrieved 2018-08-01

by defining the number of blocks and the number of threads within a block that work parallel on the kernel function.

```
kernel<<<1,1>>>()
kernel<<<1,2>>>()
kernel<<<2,3>>>()
```

As an example, the composition 1,1 in a kernel launch executes the kernel one time and therefore acts like a CPU function. The setup of 1,2 means that one block is launched that holds two threads. Each thread is capable of executing the kernel which means that the function is called twice. Another composition example is 2,3. There are two blocks launched and each block has three threads, therefore the kernel is called six (2*3) times. In figure 7, it also follows the important principle of C to allocate the memory on the host and the device that is needed for the program and freeing it in the end.

```
__global__ void some_kernel(...){...}

int main (void){
  // Declare all variables.
  ...
  // Allocate host memory.
  ...
  // Dynamically allocate device memory for GPU results.
  ...
  // Write to host memory.
  ...
  // Copy host memory to device memory.
  ...
  // Execute kernel on the device.
  some_kernel<<< num_blocks, num_theads_per_block >>>(...);

  // Write GPU results in device memory back to host memory.
  ...
  // Free dynamically-allocated host memory
  ...
  // Free dynamically-allocated device memory
  ...
}
```

Figure 7: Skeleton example of a CUDA program

Furthermore, there is the function definition __device__ which indicates that this method is called directly from a kernel function and is not accessible from the host. For the avoidance of the in 2.1.2 named Race Conditions CUDA offers the single-line instruction __syncthreads() which is written at the location in the code where the rest of the threads have to wait.

2.2 Stereo correspondence problem

The stereo correspondence belongs in the image processing of computer vision and describes the challenging problem of determining which pixel of the first image corresponds to which pixel in the second image. Solving it results in a disparity and after some calculation in a depth map which ultimately holds the

depth information for each pixel. One of the applications where depth maps are necessary is the reconstruction of the spatial structure of the three-dimensional scene. A popular approach for this problem is correlation-based and has the aim to determine if the location respectively the feature of a pixel in the first image is similar enough to one of the pixels in the second one.

2.2.1 Epipolar geometry and image rectification

The Epipolar geometry problem has to be dealt with in stereo vision when a pair of differently positioned cameras observe a scene from two different angles. The initial situation is that one picture of each camera is taken at exactly the same time.

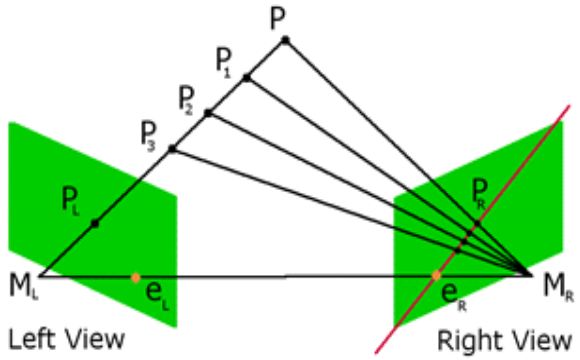

Figure 8: Epipolar Geometry

As can be seen in figure 8, a setup with two different cameras with their focal centers M_L and M_R .The point P lies in the three-dimensional space and is projected on both image planes of the cameras as P_L and P_R . The relationship between P, M_L and M_R is described as coplanar, because they lie in a common plane which is the so-called Epipolar plane. The figure also displays the epipoles e_L and e_R which are the projections of the other camera's centers in the image. They are important because they form together with the projected points the so-called epipolar lines e.g. in figure 8 the red line with e_R and P_R [2], [3]. With this information it is possible to determine which pixels should be considered to find the matching ones. It is just necessary to search along the epipolar line for the corresponding pixel and not the whole image. This is an important basis for a matching algorithm. Furthermore, it is possible to recreate based on this gained knowledge with triangulation the original three-dimensional scene which is also described in 2.2.2. Image rectification itself is the process of transforming both images with the result that these epipolar lines are now horizontal which simplifies this search even more. The only difference remaining between those two pictures is therefore a horizontal offset e.g. between pixel p and corresponding pixel q which is also named parallax or disparity that is shown in the following equation. The height y is for each corresponding pixel the same in both images why only the value x that stands for the width is needed. [2]

$$(p, q) = x_p - x_q \tag{1}$$

2.2.2 Triangulation: Depth calculation

As in formula (1) stated the disparity is gained by finding the corresponding pixels in the second image and it stands for the distance between them. For retrieving the depth of each pixel triangulation is used and therefore, it is further necessary to have some additional information about the cameras and their set-up.

The following formula (2) displays the calculation of the depth:

$$depth = \frac{baseline * focal}{disparity} \tag{2}$$

The focal length of the camera describes the distance of the lenses to the image planes and the baseline stands for the distance between the two cameras to each other. Furthermore, it can be noted that depth is inversely proportional to disparity [17].

2.2.3 Depth representation

The depth maps are the means to represent the depth after it has been reconstructed. In this chapter popular depth map types are presented. The first approach has the advantage that it can be easily compared to other depth maps like in the Middlebury database [10] with the help of the so-called ground truth.

- *2.5D representation:* As in the introduction of this thesis already mentioned the 2.5D gray scale disparity map represents the depth of each pixel with a shade of grey, which means the lighter the area is the closer is it. Because the depth information is directly stored in a two-dimensional picture this map is called 2.5D depth map representation [2]. Similar to the first disparity map the color 2.5D representation also produces a two-dimensional image which has the depth information stored directly, but it uses different, customizable color variations to indicate the distance [2]. Figure 9 visualizes these representation options.

Figure 9: Overview of different depth map types [2], left: original image, middle: 2.5D grayscale representation, right: 2.5D color representation[5]

5 http://cdn.intechopen.com/pdfs/37767/InTech-Depth_estimation_an_introduction.pdf, retrieved 2018-08-01

- *Pseudo-3D representation:* This representation provides the information of the depth differently than the two approaches before. It offers already viewpoints on the recreated scene [2]. It can almost be described as the three-dimensional scene in a two-dimensional image.

2.2.4 Challenges in stereo vision

There are several challenges in stereo vision which have to be taken into account when trying to solve a particular correspondence problem. They are compactly explained in this section. Some challenges are overlapping with each other in their meaning and it is possible that images are affected by more than one.

- *Image ambiguity*: Images can contain a wide range of different characteristics like materials and surfaces. Yet, there is a risk that they result in similar colors and intensities that might result in wrong disparity values.

- *Heterogeneous areas*: The heterogeneous areas describe a different kind of ambiguity. I means that certain regions that lie e.g. on the same surface and therefore have the same depth, but don't have the same characteristics. This is based on different interpretations of certain points by the two cameras, either because of different reflections on the surface of the object or also perspective distortions between the images. An assignment can be therefore quite challenging.

- *Homogeneous areas*: Homogeneous areas describe regions where the colors and the intensities are so similar that an assignment of correspondences is difficult to handle, because the disparities don't really vary.

- *Insufficient or repetitive textures*: Repetitive textures can also be a problem since they lead to repetitive characteristics. Figure 10 shows an image with repetitive textures that has proven to be difficult to analyze for some local solutions of stereo vision.

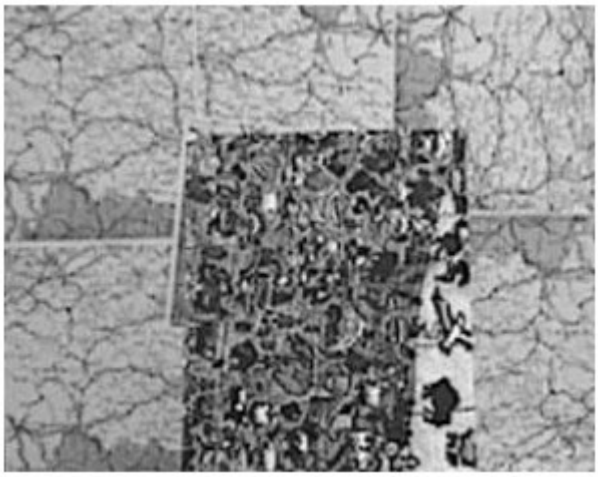

Figure 10: Right image of the picture set "Map" from the Middlebury [10] webpage[6]

6 http://vision.middlebury.edu/stereo/data/scenes2001/, retrieved 2018-08-01

- *Occlusions*: In the stereo vision two pictures of a certain three-dimensional scene are taken by two cameras that provide different angles. Based on these angles it is possible that some objects are partly or even completely covered. As a result, these pixels just appear in one of the images and therefore don't have another correspondent pixel. Affected pixels have to be identified and a disparity value has to be assigned based on neighboring pixels.

2.3 Stereo matching algorithms

One solution to the correspondence problem is a stereo matching algorithm, it uses the input images to find along the epipolar lines the corresponding pixels based on the used methods and approach of the algorithm. As result of the stereo matching algorithm the disparity which is the distance – measured in pixels – the corresponding pixel in the second image is moved in comparison to the original pixel in the first image is determined. This subchapter deals with the typical structure of such an algorithm, the different approaches for them and also a quite popular evaluation tool for these.

2.3.1 Taxonomy of a stereo matching algorithms

Scharstein and Szelsiki [8] developed a taxonomy that provides a step-by-step guide for solutions in the area of stereo correspondence in four basic steps [8]:

1. Matching cost computation
2. Cost (support) aggregation
3. Disparity computation/optimization
4. Disparity refinement

It is recommended to use these steps as an orientation to develop the algorithm and achieve optimized results, yet it is likely that some algorithms have their focus on certain parts of the taxonomy while other steps are trivial or less important [8]. This depends mostly if they are local or global approaches, some algorithms like semi-global algorithms combine both characteristics.

To calculate the matching costs, which describes the cost of matching two given pixels, different solutions according to the method types have been proposed like sum of the absolute differences (SAD) measures, normalized cross-correlation and census transform with a calculation of the hamming distance [8]. Sometimes as a result, for all the matching costs for each pixel and disparities the disparity space image (DSI) is introduced which has as the name already indicates three different coordinates. It can be imagined as a three-dimensional structure which associates each pixel in the reference image with a range of possible correspondences – within the maximum disparity range. [8] In the following thesis it is also referred to as the 3D-coststructue. The cost or support

aggregation is the second step and means in the context of local and window-based approaches that the prior calculated matching costs are either summed up or averaged over a certain support region. [8] The DSI is sometimes used to describe this area. Global methods often skip the second step and lie a strong emphasis on the third one. This third step deals with the computation of the disparity and takes care of the optimization. For local methods it is less important, their focus lies on step one and two. The goal for local ones is to select the disparity with the minimum cost for each pixel. This can be done with the Winner-take-all (WTA) optimization. [8] On the other hand, the global methods have a big emphasis on this optimization step, graph-cuts or dynamic programming are popular approaches to solve this. [2] In the final step the computed disparity map is refined to remove unwanted artifacts and improve the quality of the disparity map. Possible post-processing procedure are sub-pixel computations and the median filter which deals with mismatches and occlusions. [8]

2.3.2 Evaluation and Middlebury

Scharstein and Szelsiki not just developed the taxonomy described in section 2.3.1 that helps structure a stereo matching algorithm [8], but also created a method to evaluate the own algorithm. The evaluation of an algorithm is in stereo vision very useful to obtain feedback about its quality and speed. Middlebury [10] provides a testbed for current and past stereo correspondence algorithms. It allows not only an evaluation, but also rates and ranks the developed algorithms in regards of their calculated depth map which is measured in relation to the ground truth, like shown in figure 11. A ground truth is itself again a depth map which is often created by active methods like structured lightning which are naturally accurate yet quite costly.

Set: test dense test sparse training dense training sparse
Metric: bad 0.5 bad 1.0 bad 2.0 bad 4.0 avgerr rms A50 A90 A95 A99 time time/MP time/GD
Mask: nonocc all
plot selected show invalid Reset sort Reference list

Date	Name	Res	Avg	Austr	AustrP	Bicyc2	Class	ClassE	Compu	Crusa	CrusaP	Djemb	DjembL	Hoops	Livgrm	Nkuba	Plants	Stairs
03/06/18	NOSS	H	5.04 1	3.57 2	2.84 3	3.99 2	1.93 1	5.15 1	3.34 1	3.32 1	3.15 1	2.32 2	8.55 2	7.45 1	7.06 2	12.5 2	5.20 3	10.0 7
06/22/17	LocalExp	H	5.43 2	3.65 3	2.87 4	2.98 1	1.99 2	5.59 2	3.37 2	3.48 3	3.35 2	2.05 1	10.3 3	9.75 3	8.57 5	14.4 10	5.40 6	9.55 5
01/24/17	3DMST	H	5.92 3	3.71 4	2.78 2	4.75 4	2.72 5	7.36 5	4.28 3	3.44 2	3.76 3	2.35 3	12.6 8	11.5 5	8.56 4	14.0 9	5.35 5	8.87 4
03/10/17	MC-CNN+TDSR	F	6.35 4	5.45 13	4.45 17	6.80 18	3.46 14	10.7 12	6.05 10	5.01 9	5.19 9	2.62 7	10.8 4	9.62 2	6.59 1	11.4 1	6.01 10	7.04 1
05/12/16	PMSC	H	6.71 5	3.46 1	2.68 1	6.19 14	2.54 3	6.92 3	4.54 4	3.96 4	4.04 5	2.37 4	13.1 10	12.3 6	12.2 7	16.2 18	5.88 9	10.8 9
10/19/16	LW-CNN	H	7.04 6	4.65 7	3.95 9	5.30 9	2.63 4	11.2 16	5.41 7	4.32 6	4.22 6	2.43 6	12.2 7	13.4 9	13.6 15	14.8 13	4.72 1	12.0 14
04/12/16	MeshStereoExt	H	7.08 7	4.41 6	3.98 11	5.40 10	3.17 10	10.0 7	6.23 12	4.62 7	4.77 8	3.49 19	12.7 9	12.4 7	10.4 6	14.5 11	7.80 17	8.85 3
10/12/17	FEN-D2DRR	H	7.23 8	4.68 8	4.11 13	5.03 6	3.03 9	8.42 6	6.05 10	4.90 8	5.32 10	3.20 16	11.5 6	14.1 11	13.4 14	13.9 7	5.06 2	14.3 19

Figure 11: Overview of depth algorithms, testing methods like bad 4.0 available[7]

2.3.3 Different methods

As discussed in section 2.3.1, there exist several types of methods for addressing the stereo correspondence problem with an algorithm, they are selected according to the needs and requirements of the algorithm like real-time execution or highest accuracy. The methods group into global and local ones, but there

7 http://vision.middlebury.edu/stereo/eval3/, retrieved 2018-06-03

also exists hybrid solutions that use a semi-global approach. In the following subsections an short overview of these methods is given.

Global methods. The global methods see the correspondence problem as an optimization task by using certain smoothness assumptions and by defining and minimizing a global cost and energy function [2], [6]. The following formula (3) shows a global energy function:

$$E(d) = E_{data}(d) + \delta E_{smooth}(d) \tag{3}$$

The first term in this formula determines how well the disparity function matches with the input images by using the DSI respectively the cost structure and therefore the aggregated matching costs. The second term deals with the smoothness assumptions made by the algorithm and can be restricted to just take the neighboring or previous pixel into account like walking along certain scanlines within the image. Graph cuts matching is often used in global methods and delivers good results [6], [8].

Local methods. Local methods are another approach to solve the correspondence problem. Comparing just one pixel in the first picture with another pixel in the second pixel might lead to more noise in the final result, because they are not decisive enough. Hence, local methods use certain areas around a pixel of interest which are then compared. These areas are most of the time referred to as windows, also their shape is important for the final performance of these local methods. The goal is to find the window position that has the greatest synergy with the reference window. Difficulties can arise if the image contains low-textured areas which have barely any useful information to distinguish the pixel. The main advantage of local methods is that they can be parallelized which make them an attractive choice for GPU programming to achieve real-time execution. [9], [8], [2]

One of the approaches to create a local algorithm is the adaptive support--weights (ASW) method. Yoon and Kweon published 2006 their approach of an adaptive support-weight algorithm for correspondence search [5]. They used a local window-based method with varying support weights. Furthermore, the results of the gestalt grouping in color similarity and geometric proximity were used to adjust the support-weights. Their approach was one of the first in this particular area and performed very well in relation to other local methods at the time.

Kowalczuk et al. also proposed in 2013 a method for adaptive-support weights which was realized in CUDA [4]. They developed a method that uses a two-pass approximation of adaptive support-weight aggregation and a low-complexity iterative disparity technique. The focus is also on a real-time execution.

Semi-global methods. Semi-global methods combine both local and global matching approaches to build the disparity maps. A popular implementation of one of these methods is called Semi-Global Matching (SGM) and was published by Hirschmüller [7]. His algorithm is able to use the mutual information

to match pixel pixel wise and it also approximates a global 2D smoothness constraint by combining several 1D constraints. The goal is to aggregate matching costs along individual paths (often four, eight or twelve) in the image. Each direction is there taken into account equally to prevent an effect called streaking which is often suffered by solutions that use pure dynamic programming.

Another approach worth mentioning is realized by Hernandez-Juarez et al. [13] and resulted in an embedded and robust real-time depth algorithm in the demanding area of robotics for driver assistance systems. This algorithm with its features and components is strongly based on Hirschmüller's SGM [7] but focuses even more on efficiency. It was realized in CUDA on a single Tegra X1 at 42 frames per second and offers the selection of either four or eight paths as well as either a disparity level of 64 or 128. This thesis uses two methods (Census Transform and Hamming Distance calculation) which were developed by Hernandez-Juarez et al. [13] with minor changes and that are available on GitHub[8].

8 https://github.com/dhernandez0/sgm, retrieved 2018-08-01

3 Methodology

This chapter aims to explain how the thesis was structured in order to answer the questions in regard of the goals presented in section 1.4. Therefore, it is necessary to understand the undertaken process of this thesis which is visualized in figure 12.

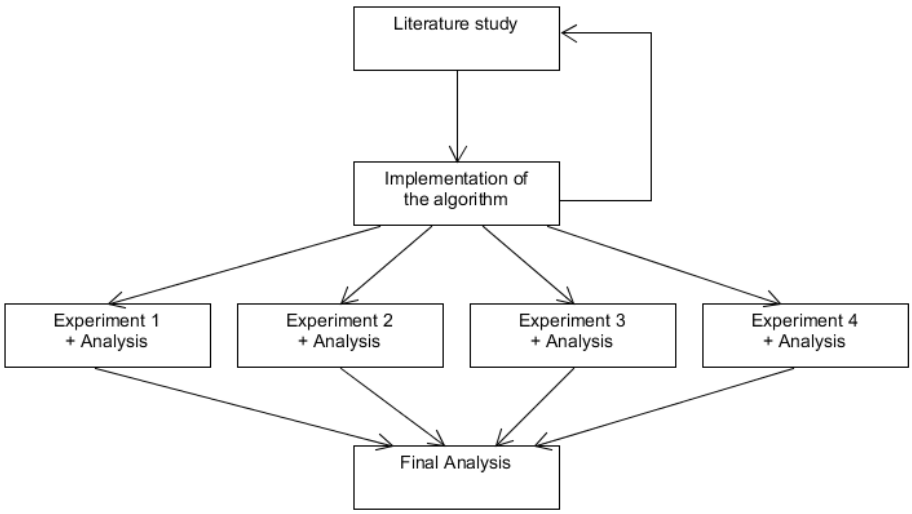

Figure 12: Overview for work process of thesis

To start the work process, a literature study was done by reading related and relevant research papers on the proposed algorithm and about stereo vision in general published by universities and companies. Furthermore, an important part of the literature study, which was also more practical oriented was getting acquainted with CUDA and GPU programming. After collecting the necessary knowledge, the prior selected algorithm was implemented and adapted to fit the particular situation and needs of this thesis. A crucial part was also the evaluation of the algorithm which was divided in four different experiments and their analysis. These are described in greater detail in subchapter 3.2. The final analysis takes all results of the prior experiments into account, summarizes and elevates them and also prepares for the conclusion.

3.1 Selected tools

There are several tools which are of importance in this thesis which will be shortly described in this section.

3.1.1 Evaluation of results and comparison

Middlebury [10] is just one tools that offers a testbed for the evaluation process of stereo correspondence algorithms in form of datasets with left and right images as well as their ground truths, explained in 2.3.2. The figure 13 visualized this setup with some later one used image sets. With these test sets it is possible to check the own results (disparity/depth maps) and get reliable feedback that can be used to measure the accuracy of the proposed algorithm. Yet, also different algorithms can be compared in regards of their execution time. Another similar also quite popular testbed is offered by the KITTI Vision Benchmark Suite [11].

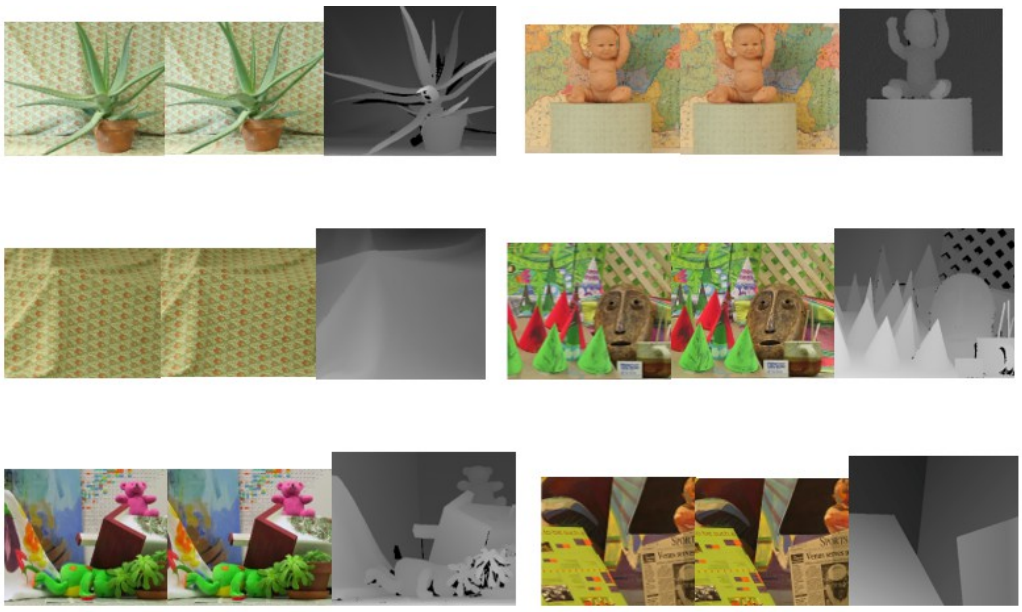

Figure 13: Examples of the available stereo data sets with their ground truth, always sorted like: left image, right image, ground truth, the used images in the first row to the last from left to right: aloe, baby1, cloth1, cones, teddy, venus[9]. The image sets are from different years/different data sets. Further-more, the available images have different characteristics that lead to challenges during the disparity estimation, these challenges are also described in section 2.2.4.

3.1.2 Evaluation of performance

The visual profiler [12] is a well-established tool for CUDA programs and is contained in the CUDA Toolkit 9.1 to review the memory usage and get feedback on the time spent for certain operations and parts of the algorithm while it is also displayed chronological in a timeline and in diagrams. It allows e.g. to see which exact part of the source code is decelerating the whole program or incriminating for the memory utilization. It is recommended by NVIDIA and their developers if the goal is to create a powerful and performance-strong algorithm.

9 http://vision.middlebury.edu/stereo/data/, retrieved 2018-08-01

3.2 Experiments

As can be seen in figure 12 four different experiments are part of the research process that are to evaluate the proposed algorithm in different aspects and answer the questions introduced in chapter 1.4.

3.2.1 Benefits from parallelization

This experiment belongs to the first question about the evaluation in section 1.4 and is to determine whether and where parallelization is beneficial in the algorithm. Therefore, a comparison between sequential and parallel execution of the code is drawn. The sequential processing is handled by the CPU where executing certain computationally intensive calculations are very costly and it isn't constructed to provide multiple threads yet is also known to have a higher clock rate. The parallel processing is done by different for testing available GPUs where the execution can be due to its architecture of threads and blocks extremely fast, yet time loss like of memory copies over the PCIe or access of certain memory types will occur. In particular, the number and ratio of threads and blocks created to run a kernel function have a certain influence on the execution time. The experiments determine the milliseconds respectively the seconds needed to execute single code snippets especially calculations in these two manners.

3.2.2 Imposed overheads

Imposed overheads are named in the second question of section 1.4 where the aim is to determine their relevance and significance in the proposed algorithm. They are a result of initializing devices, launching kernel functions and memory management or transfer and have an impact on the performance of the program. Another important factor is whether a kernel function is called asynchronous and synchronous, meaning if the CPU can hide the latency of a kernel function in executing other code or forcing it to wait until the kernel function has been completed. So, overheads describe the costs often in regard of time and memory usage or even in compiled code length. The significance of these overheads is evaluated with the time needed to execute the program with the focus on the processing time of the GPU. Also, launching empty kernel functions suffices to simply show the overhead of these. The cost of memory allocation on CPU is also something that is investigated using different allocation functions. Allocation on the CPU is correlated to the transfer of memory onto the GPU which is measured based on the different types of storage and memory options for data on host and device. The aim is to use different data sizes in MB that are allocated in various ways and to measure the time in milliseconds that it takes on average to do so. Furthermore, the actual transfer time of the stored data is measured in milliseconds and averaged. Also, the throughput of the data in MB/s is investigated and averaged. Accordingly, these results are presented in diagrams or via tables. The x-axis of the diagrams shows the allocated MB and the y-axis the spent time in milliseconds respectively the data throughput in MB/s.

3.2.3 Real-time execution

The third question in section 1.4 deals with (near) real-time execution and wants to find answers how this can be obtained and where the challenges are. Furthermore, it is analyzed what parts of the algorithm benefit in particular from the parallelization and result in an increase in performance and speed and if there are parts that should not be parallelized at all but should or have to run linear instead. In image processing in general, a real time program fulfills the task of producing output (almost) simultaneously with the input e.g. from a camera system. The time available for calculating one frame is extremely limited. As an upside, the images of the camera system are supposed to arrive in a continuous and time-consistent manner. Also, there is a distinction between cameras which provide frames at a lower rate like 20 fps or higher rate like 60 fps. The number of images that have to be processed in a certain time to achieve the same results varies in these cases. E.g. the camera has a capture rate of 30 fps then 30 frames are to be processed in a second. This results in an output of ca. 33ms/frame (1000ms/30frames). The single algorithm steps and components are evaluated to determine which instructions are taking too much time, and which instructions can be optimized further.

3.2.4 Handling image-pair input data streams

The last question in section 1.4 that ought to be answered asks about facilitating constantly updating image pair input data streams. It is investigated, if it is possible and what it would take to let the whole algorithm run multiple times as copies concurrently on the same GPU but while handling different sets of input images which would also benefit the aspect of (near) real-time execution. The solution with collected results about the performance, partitioning and other unexpected bottlenecks or challenges, restrictions and limitations are described.

4 Implementation

In this chapter details and information about the actual implementation of the disparity estimation algorithm are described. The focus is on the different components/methods, their mathematical aspect and the overall structure is stated. In Appendix A there is also a more detailed description about the code, especially the functions. It is important to first note that the algorithm works with rectified, greyscale image pairs. Figure 14 describes as an activity diagram the pipeline of the algorithm:

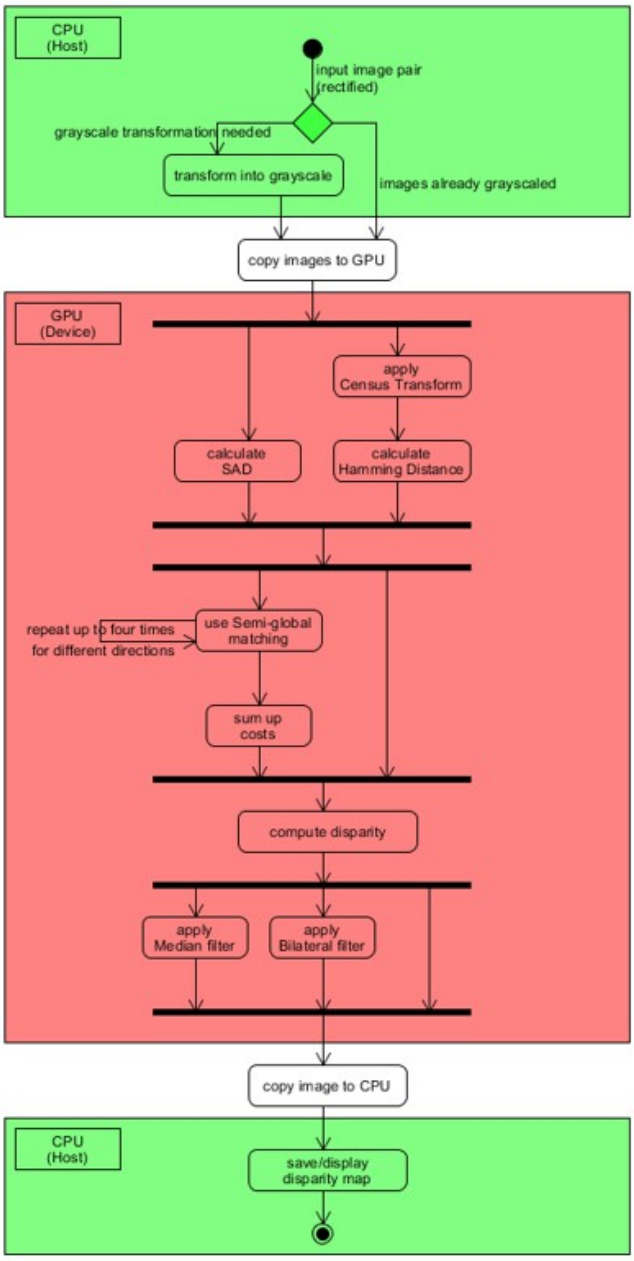

Figure 14: Pipeline of the algorithm, activity diagram of the structure

As can be seen in figure 14, the host fetches the rectified images and transforms them into grayscale images if they are not already transformed. In the next step, the images are copied to the device memory to make the following operations on them possible. After this, the matching cost is computed, this can either be achieved with a Census Transform of the images and subsequently the calculation of the hamming distance or simply with the calculation of the sum of the absolute differences (SAD). Both approaches result in the 3D-matching cost structure $MC_{width,height,max.disparity}$ (Matching cost or also referred to as DSI [7]) that is later used in the semi-global matching that has the main goal to smooth and improve the final result. $Max.disparity$ which is the third dimension of the cost structure is the amount of pixels in the second image that are compared to the pixel in the first image to find the corresponding pixel-pair, important to note: $Max.disparity = 125$. For each pixel in range is a disparity value calculated and stored in the cost structure. After that, the minimal disparity itself needs to be computed which is achieved with a Winner-takes-all approach. Before sending back the disparity image a filter can be applied. This can either be a Median filter or a Bilateral filter. The components are explained in greater detail in the following subchapters.

4.1 Census Transform and Hamming Distance

After retrieving the rectified, grayscale image pair the Census Transform (CT) is applied. In figure 15 is exemplary visualized how CT works. Within the set window 9x7 pixel the center pixel is compared with every other pixel. The intensity difference between these two defines whether the value of the pixels is set to 0 or to 1. If the center pixel is brighter the other pixel is set to 1, if it is darker it is set to 0. Formula (4) displays the assignment of the values mathematically.

$$CensusTransform(I(x,y)) = \begin{cases} 1, if\ I(x,y) \geq central\ pixel\ value \\ 0, if\ I(x,y) < central\ pixel\ value \end{cases} \quad (4)$$

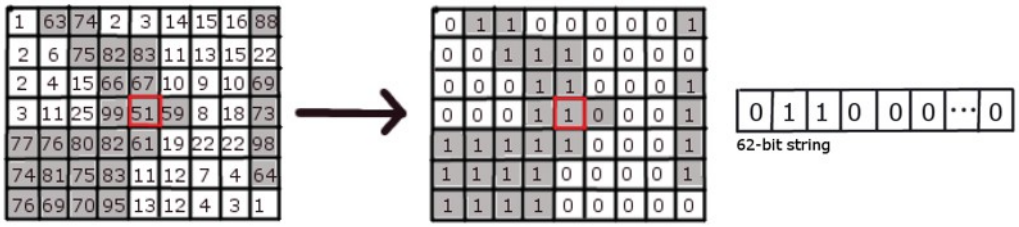

Figure 15: Census Transform example for a grayscale image, window-size 9x7 pixel, center pixel is marked red, result-ed in 62-bit strings

In the end 62-bit strings are created which can then be used to calculate the Hamming Distance. Figure 16 shows an example of how an image looks like when a census transform is applied:

Figure 16: Left: Original image "cones"[10] from the Middlebury testbed [10], right: result of Census Transform

The next step calculates the Matching Cost (MC) or in this case more specifically the Hamming Distance. Two 62-bit strings in both first and second image are therefore compared and for each difference 1 is added. The more similar the intensities (I) of the left and the right image are the smaller the value. The comparison is carried out along horizontal lines in the image for a certain number of pixels which is referred to as disparity ($d = [0, Max.\ disparity_{range} - 1]$). If the result of the comparison is very small the higher the chance that these pixels are the corresponding pixels. The formula (5) to calculate the Matching Cost can be expressed as follows:

$$MC_{width, height, max.disparity} = HammingDist\left(CT\left(I_L(x, y)\right), CT\left(I_R(x+d, y)\right)\right) \quad (5)$$

Figure 17 shows an exemplary comparison between two 62-bit strings. The difference of the two strings is two which is an indication that these areas have a high similarity.

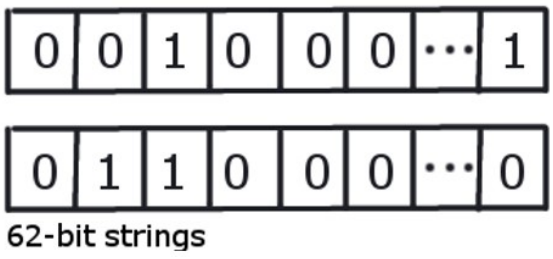

Figure 17: Exemplary comparison between two 62-bit strings, is can be seen that the second and the last pixel are not the same which makes a difference of 2

10 http://vision.middlebury.edu/stereo/data/scenes2003/, retrieved 2018-08-01

4.2 SAD

The sum of the absolute differences (SAD) is also used to calculate the disparity between an image pair respectively the Matching Cost (MC). This function uses the absolute values therefore the result is never negative. In this scenario the grayscale images are used and the pixel intensity (I) of one is subtracted from the pixel intensity of the other. The result describes the similarity and the smaller the result the higher is the probability that the pixels are corresponding.

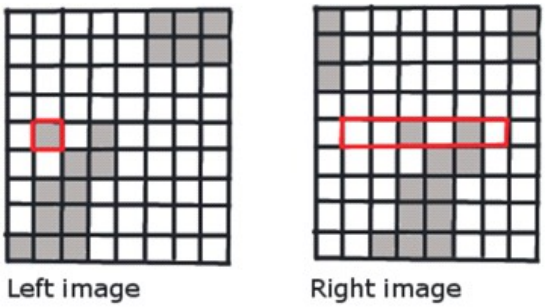

Figure 18: Example for SAD

Figure 18 shows how a pixel in one image is compared to a range of pixels in the seconds image. These pixels are limited to horizontal pixels with the same height along a scanline. The number of pixels that have to be compared are based on the preset disparity value (d).

The following formula (6) describes how the value is calculated:

$$MC_{width,height,max.disparity} = ABS.(I_L(x,y) - I_R(x+d,y)) \quad (6)$$

4.3 Semi-global matching

Local methods like the Hamming Distance or SAD are often not distinctive enough to give results of a certain accuracy and contain still a lot of noise, therefore it is enhanced with a semi-global function. This describes the calculation of disparities along certain scanlines in the 3D-cost structure for each pixel. In this algorithm four scanlines are used, two for x- and two for the y-axis. The main difference between those scanlines or also called paths is the direction (dir) in which the new disparities are calculated, can abstractly be visualized like in figure 19.

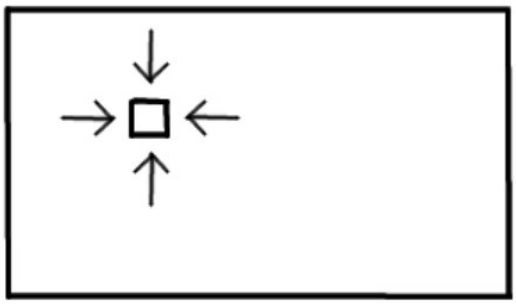

Figure 19: four different paths/scanlines are implemented, x-pos (arrow pointing from left to right), y-pos (arrow pointing from top to bottom), x-neg (arrow pointing form right to left) and y-neg (arrow pointing from bottom to top)

The following equation (7) shows how the semi-global matching is conducted, it contains three different terms. The first term describes the previous calculated matching cost (MC) either with the calculated Hamming Distance or SAD. The second term adds the minimum disparity cost of the previous pixel, the previous pixel in either x-direction or y-direction. If the previous pixel in x-direction is calculated, $dir.y=0$, and $dir.x=1$ for x-pos or $dir.x=-1$ for x-neg. If the scanline is calculated in y-direction, then $dir.x=0$, and $dir.y=1$ for y-pos or $dir.y=-1$ for y-neg. In this more extensive operation the disparity (d) has an important role, because each disparity of the current pixel is compared with each disparity of the previous pixel. If the disparity is for both pixels the same no penalty term has to be added, if there is a difference of 1 a small penalty ($P_1=5$) is added. The large penalty term ($P_2=25$) is added if the disparities diverse more than 1. The variable i accepts values between $[0, Max.disparity_{range}-1]$, but excluding the disparity values which are covered by the first two parts of the second term, e.g. the disparity for the current pixel is 15. If now the minimum cost for the previous pixel is calculated and the cost is checked for the disparity 15 as well, no penalty is added. If the cost is calculated for the disparity 14 or 16 the small penalty is added to the cost which is supposed to make is more unlikely to get chosen in the end as the correct disparity, because it increases the value. Lastly, the other disparities are checked, excluding 14, 15 and 16, and the large penalty is added.

The penalty terms P_1 and P_2 have different effects on the result. The smaller penalty takes care of slanted and curved surfaces. P_2 , on the other hand, smooths results and makes abrupt changes difficult. [13] In the end, the last term in subtracted to prevent the overall term from getting out of bounds respectively too large. The variable k can be expressed as , consequently the minimums of the second and the third term don't necessarily need to be the same and have to be distinguished why both the variables i and k are needed.

$$\begin{aligned} &3\text{DCostpath}(x,y,d)= \\ MC(x,y,d)+min&\begin{cases} 3\text{DCostpath}(x-dir.x,\ y-dir.y,\ d) \\ 3\text{DCostpath}(x-dir.x,\ y-dir.y,\ d\pm 1)+P_1 \\ \min\limits_i\left(3\text{DCostpath}(x-dir.x,\ y-dir.y,\ i)+P_2\right) \end{cases} \\ &-\min_k\left(3\text{DCostpath}(x-dir.x,\ y-dir.y,\ k)\right) \end{aligned} \tag{7}$$

The following figure shows how the algorithm is selecting the minimum disparity for each value pair, in this example it uses the scanline in x-pos direction and examines always the previous pixel. Figure 20 visualizes how the cost is calculated.

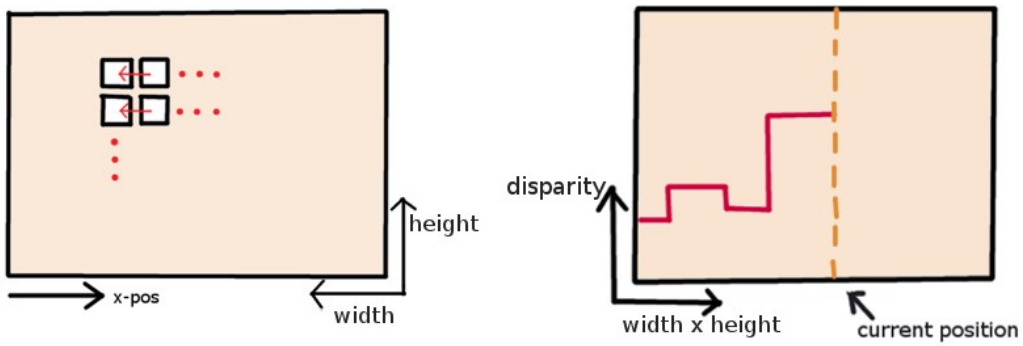

Figure 20: two dimensional illustrations of the 3D-cost structure, smallest disparity for each pixel is determined (red line) in the left image, on the right image the path through the image is shown, x-pos direction for scanline, cost of previous pixel is used

The semi-global matching has a very distinct limitation regarding parallelization possibilities. For the calculation of the current pixel cost the cost of the previous pixel is required. Therefore, parallelization of the current scanline is not possible. Yet, there is the option to parallelize the scanlines alongside each other and assign threads and blocks only to certain disparity levels within a path, e.g. one thread is responsible for one whole scanline exactly for on disparity of the current pixel responsible but has to calculate the minimum for each disparity cost of the previous pixel. It is important to note that this makes a thread synchronization alongside the paths necessary, because all disparities for the current pixel which will be the previous pixel in the next step of the semi-global matching have to be already calculated. Before the disparity can be computed it is necessary to sum up the newly calculated 3DCostpath and the originally calculated, like in formula (8).

$$3\text{DCost}(x,y,d)=3\text{DCostpath}(x,y,d)+MC(x,y,d) \tag{8}$$

4.4 Disparity computation

The result of the previous methods is a 3D-cost structure and can therefore not be used for the 2D disparity image, also it still has to be determined which disparity is the *right* one to choose. In this structure image height and width are one plane of the cuboid while the disparity range is the depth. For each pixel

and each disparity, the calculated value is found and based on this simply the smallest one is selected for the disparity map. This is called Winner-takes-all (WTA) and is shown in formula (9).

$$Disparity(x, y) = \min_{d} \sum 3\mathrm{DCost}_{x,y,d}(x, y, d) \tag{9}$$

4.5 Filter

To smooth respectively refine the disparity maps two filters with different characteristics are implemented. The filters are applied after the disparity computation which means that they have to be executed after the disparity computation from section 4.4.

4.5.1 Median filter

The Median filter is mainly used for noise reduction and a clear final disparity map without blur. A window around each pixel with a certain size (e.g. 5x5 px) is selected and moved through the image, the goal is to select a (new) center pixel that comes close to the average value of the area. It is especially powerful when it comes to single respectively outlier noise points, these will be filtered out quite easily, just larger areas of noise, especially on the edges of objects, might pose some problems. Figure 21 shows exemplary how the values within the windows are reordered.

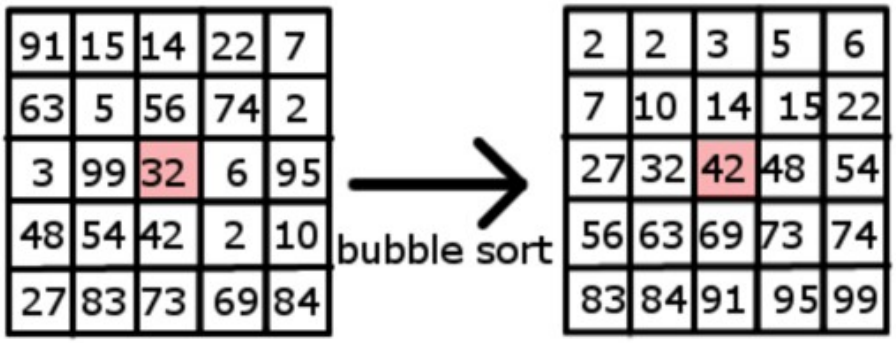

Figure 21: Median filter: Rearranging of pixel values within the window in ascending order, window size: 5x5 px

4.5.2 Bilateral filter

The Bilateral filter is similar to the Median filter in section 4.5.1 also used for noise reduction, but it is also capable of the preservation of edges and results in a blurred disparity map, similar with a Gaussian blur. This filter replaces the value of the center pixel with the weighted average value of neighbouring pixels (window size = 5x5 px). Depending on the set smooth value e.g. between 20 and 100 the final disparity map will change the blur factor.

4.6 Additional information

This subchapter provides some additional information about the implementation of the algorithm but doesn't cover details about the actual components and methods of the algorithm. There is a momentary need for the use of image libraries, so that the algorithm can be tested, because the final integration in the pipeline of the Multiview camera system has not taken place yet. Furthermore, to facilitate the integration process different versions besides the simple standalone executable for the algorithm are created like a static library. These facts are described for the sake of completeness in the rest of the subchapter.

4.6.1 Image libraries

Furthermore, it is necessary for the implementation to use image libraries to fetch the input images and also to save or to display the disparity map. The first library used is OpenCV-3.4.1 [26] that also provides a simple solution of transforming the input images into grayscale images with a built-in functionality. Yet, this library is too large in its scope to use efficiently for test purposes on the Jetson TK1 devices that only have a limited memory space available. That's why another, smaller image library is used, too. LodePNG [27] is selected, but the transformation from colored images into grayscale ones has to be implemented manually. This library makes it also necessary to use PNG-images as input images.

4.6.2 Multiple implementations

The algorithm was implemented in several ways to make it easier to use in different situations. The first one is a standalone executable application that either uses LodePNG or OpenCV as an image library. It fetches one image pair per execution and saves the final disparity map. The second version is a static library with no image library included and only holds the methods and components of the GPU pipeline. The algorithm requires *uint8_t pointer* to the image data as initial parameters and it also returns the disparity map as such, this makes it possible to use it as part of the Multiview camera system, because there the image data is transferred as *vector<uchar>* directly from the cameras these can be transformed into a *uint8_t pointer*. A exemplary Frontend for the library is also delivered, it uses LodePNG respectively OpenCV as image library, fetches one image-pair, uses the library functions to calculate the disparity and saves the final disparity map that is returned. It also shows the possible changes of the variables for the algorithm like how many paths are used or which filter and which window size for it. The last solution is realized with OpenCV and also uses the static library, the goal is to simulate a continuous image input stream and execute the algorithm in multiple repetitions, the disparity maps are not saved permanently, but displayed in a window and as soon as the window is closed the next image pair is fetched.

5 Results

This chapter states, visualizes and explains the objective results of the experiments introduced in subchapter 1.4 and described in 3.2. If necessary for the understanding it also provides some background information e.g. to used functionalities. Important to mention are also the test environments and hard-ware properties that have an influence on the outcome of the result. Used operating systems are Windows 10 in combination with the GeForce 840M and GeForce GTX 1070 and Ubuntu 17.10 with the Jetson TK1 device. Farther, the algorithm is developed with the latest CUDA version at the time: 9.1. Table 2 shows some essential device properties that have been queried.

Table 2: Overview of important device properties

NVIDIA GPU / **Components**	**Jetson TK1**	**GeForce 840M**	**GeForce GTX 1070**
Architecture	Kepler	Maxwell	Pascal
Compute capability	3.2	5.0	6.1
Base clock	1058 MHz	1020 MHz	1506 MHz
Memory clock	924 MHz	1001 MHz	2002 MHz
Memory available	8 GB	2,048 GB	8 GB
Number of CUDA cores	192	384	1920
Multiprocessor (MP) count	1	3	16
Concurrency	Able	Able	Able
Device copy overlap	Able	Able	Able
Integrated GPU	True	False	False
Max. threads per block	1024	1024	1024
Threads in warp	32	32	32
Tested CUDA version	9.1	9.1	9.1
OS System	Ubuntu 17.10	Windows 10	Windows 10

Each GPU is designed with a different architecture which also affects the results of the experiments. The *Kepler micro-architecture* [18] was released in April 2012 as a successor to the *Fermi* architecture. The focus of *Kepler* is energy efficiency, programmability and performance [19]. In February 2014, the first

graphic cards that use a *Maxwell micro-architecture* is released, it further introduced an improved design of the streaming multiprocessors (SM) that was supposed to increase the power efficiency [20]. The *Pascal* architecture is the successor of the Maxwell architecture, released 2016 and offers several improvements and new features. E.g. it uses NVLink which is a high-bandwidth bus between the CPU and GPU that allows higher transfer performance, it also provides more dedicated space for shared memory and a dynamic load balancing scheduling system that manages multiple tasks and guarantees that the resources of the GPU are utilized as much as possible [21]. Regarding shared memory, using a Kepler architecture the shared memory and the L1 cache share the same on-chip storage while Maxwell and the Pascal architecture provide a space that is dedicated to shared memory and therefore increase their capacity in that area [22].

For some experiments the duration of certain CUDA instructions are measured and therefore the functionality *cudaEvent* is used, while usually the CPU will not wait until the kernel is finished and execute the next instruction (asynchronous execution) it is now forced to wait till the kernel is completely finished and only after this measure the time (*cudaEventSynchronize*). To measure the duration of tasks running on the CPU the library *ctime* is used.

5.1 Benefits of parallelization

The first experiment deals with the question of the benefits of parallel execution of the algorithm, therefore a CPU version of the algorithm is implemented and tested against the three available GPUs. This also allows a first comparison between the performance of the GPUs themselves. Important to note is that not simply the performance of the whole algorithm is measured, but the single components of it. This makes it possible to identify the methods that benefit the most from parallelization and which might not show the desired effect. The test images have the dimensions of 450x375 px (width x height) and the maximal disparity distance that is scanned in the second image is set to 125. The results of this investigation are shown in table 3.

Table 3: Comparison between CPU and GPU execution time of different components of the algorithm

Hardware / Components	CPU[11]	Jetson TK1	GeForce 840M	GeForce GTX 1070
Census Transform	0.18 secs	0.0006 secs	0.02 secs	0.00215 secs
Hamming Distance	0.44 secs	0.0001 secs	0.03 secs	0.0035 secs
SAD	0.85 secs	0.78 secs	0.09 secs	0.026 secs
Semi-global matching (scanlines)	~ 113.15 secs/path	~ 0.78 secs/path	~ 6.75 secs\path	~ 0.7533 secs/path
Sum-up calculated costs of semi-global matching	0.07 secs	0.0009 secs	0.02 secs	0.0014 secs
Disparity computation, WTA	0.34 secs	0.15 secs	0.03 secs	0.0036 secs
Median filter	0.3 secs	0.01 secs	0.08 secs	0.0054 secs
Bilateral filter	1.95 secs	0.16 secs	0.08 secs	0.0062 secs

This table shows that all methods of the algorithm benefit to some extend from a parallelization and executing the calculation directly on the GPU. Especially in regards of the aspired near real-time execution even the smaller speed-ups are of importance and will have a final influence on the results. The performance boost that stands out the most is for the semi-global matching itself, the speed-up even for the slowest GPU version is 20 times the execution time of the CPU per path. The table also allows the comparison between the single GPUs and their performance. The Jetson TK1 has an overall better performance than the other GPUs, the only component that runs slower is SAD.

5.2 Overhead investigation

The second experiment displays the different overheads that can influence the performance of the algorithm. Investigated are overheads created by memory allocation, memory transfer and kernel launches. There also is a notable overhead from one up to several seconds of the image library functions that are responsible for fetching the images. This overhead can be overlooked for now, because the actual aim is to integrate this algorithm and to get the input images directly from the camera system over a pipeline and don't fetch them out of a directory. The focus is on the overheads that are likely to occur in this type of scenario.

Before the data (e.g. the input images) can be accessed on the GPU is has to be allocated on the CPU and then transferred to the GPU memory. After some operations the result is often copied back to the host. First the memory allocation and realizing is investigated, three different options storing increasing amounts

11 Intel(R) Core(TM) i5-4210U CPU @ 1.70GHz, 2401 MHz, 2 core(s), 4 logical processor(s)

of data on the CPU were examined. Figure 22 shows the results for these allocation types.

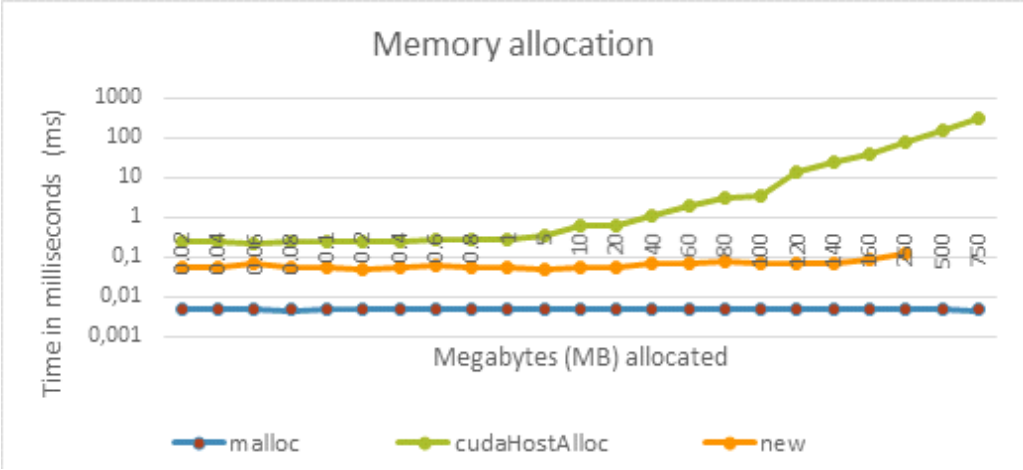

Figure 22: Memory allocation on CPU, note the logarithmic scales

The C standard library function *malloc* is often used, it allocates memory from the *heap* and returns a *void pointer*. To allocate memory it is necessary to specify the required size in bytes. In the context of using CUDA it is important to state that the so-called *pageable host memory* respectively *non-pinned memory* is allocated with this method. The experiment shows that *malloc* is a very stable function regardless of the data size from 0.02 up to 750 MB to allocate the time remains rather stable at an absolute low of approximately 0.0048 ms.

The second memory type examined is allocated with *cudaHostAlloc* which allocates a buffer of *page-locked* respectively *pinned memory*. As shown in the figure above, it is significantly more expensive to use than *malloc*. But the time needed for allocation remains relatively stable until about 1 MB, after this it starts to increase. Allocating the largest size of 750 MB takes over 300 ms. The last allocation approach is using *new*, which is the equivalent to *malloc* in C++. It allocates memory from the *free store* and returns a fully typed pointer. The tests show that the time needed remains relatively stable over all data sizes, but it is more expensive than *malloc*, it costs about 0.055 ms. Also, it has to be stated that on the test machine problems occurred when allocating data sizes over 500 MB.

For every allocation method exists an own de-allocation function: *malloc/free*, *cudaHostAlloc/cudaFreeHost*, *new/delete* that have to be called at the end of the program. Figure 23 visualizes in what correlation the amount of released MB and the needed time stands. Even after executing the transfer multiple times and choosing the average value as a result some spikes remain during the de-allocation process, e.g. at 80 MB for *cudaFreeHost* or at 0.04 MB for *free*. Like for the allocation process it can be seen that *cudaFreeHost* is more expensive than free or delete, yet besides some outliers the times for free and delete remain relatively stable. The function *cudaFreeHost* has a time increasement for data larger than 60 MBs.

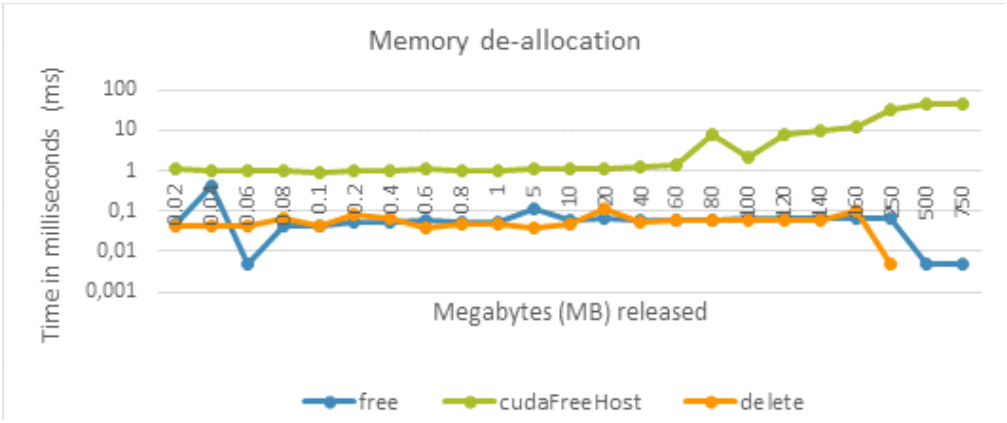

Figure 23: Memory de-allocation on CPU, note the logarithmic scales

The second possible overhead examined is based on the time needed for the transfer or copying of the allocated data from the CPU (host) to the GPU (device) and vice versa. This part of the experiment is run on the Jetson TK1 and the GeForce 840M which allows to compare the results afterwards.

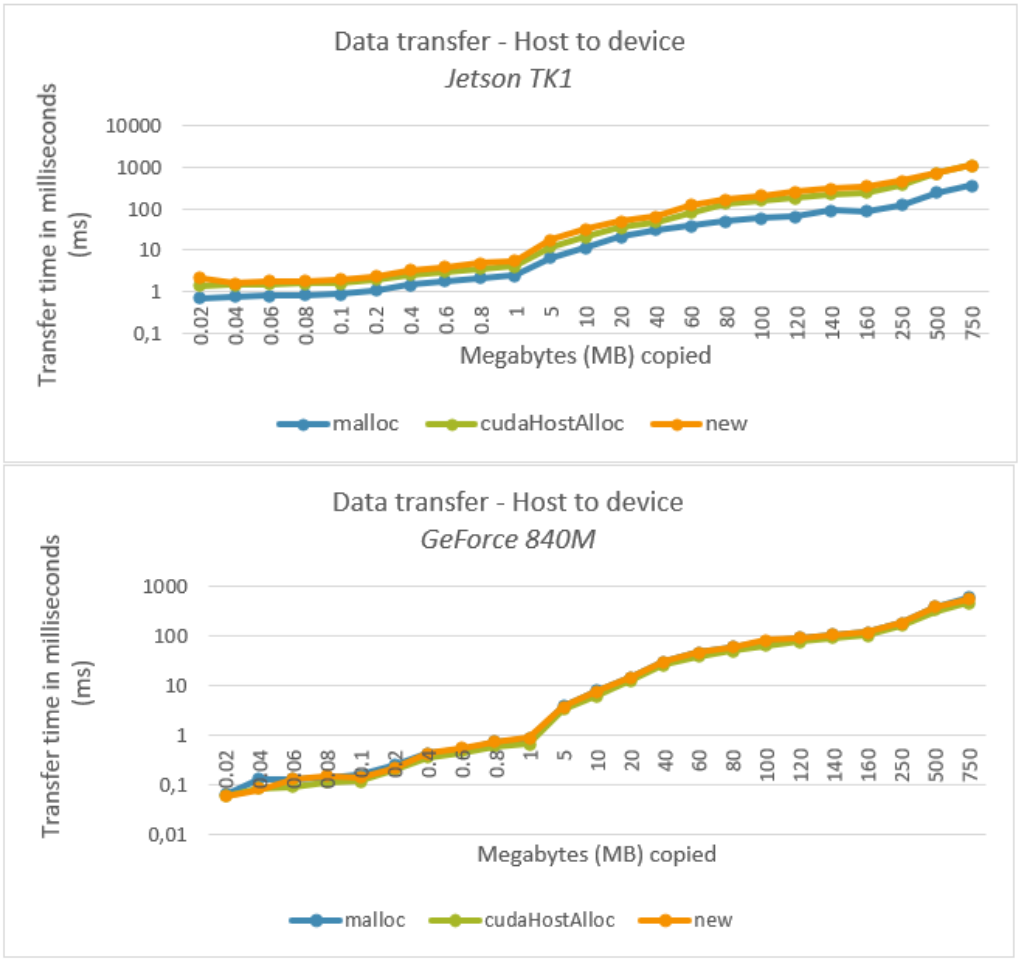

Figure 24: Transfer time: Jetson TK1and GeForce 840M comparison, host to device, note logarithmic scale

Figure 24 shows how the different allocated memories behave when they are

copied from the host to the device. For both GPUs there is a slow increase in time needed for the transfer if the data size expands as well, the values for the GeForce are closer together. There is a faster increase for the GeForce 840M after the 1 MB mark is passed. The method *new* requires the most amount of time, *malloc* is the fastest function, but the differences between the three options when using this are not very significant. By comparing single data sizes of both diagrams and the needed time it is noted that the GeForce is slightly more efficient, this is e.g. visible for 0.02 where the GeForce is about 0.64 ms or 1 MB ca. 1.07 ms faster, because the values of these three methods are close together the average value from those is chosen for the result. In addition, the measured time for the data transfer on the GeForce 840M never crosses the 1000 ms line for the largest data size.

On the other hand, the differences of the data transfer back from the device to the host are more evidently which is shown in the figure 25. For the Jetson TK1, there is a drop for memory that is transferred with *cudaHostAlloc* at a data size of 60 MB and from that point it starts to increase again. Different than for the transfer from the host to the device, in this direction the fastest method is *cudaHostAlloc*. *New* and *malloc* are almost equally efficient. For the GeForce 840M, it looks remarkably similar to the data transfer from the host to the device in figure 25 (bottom diagram). The main differences are the missing decrease are 60MB on the GeForce and some slight variations for the smaller data sizes, even the spike after 1 MB is visible.

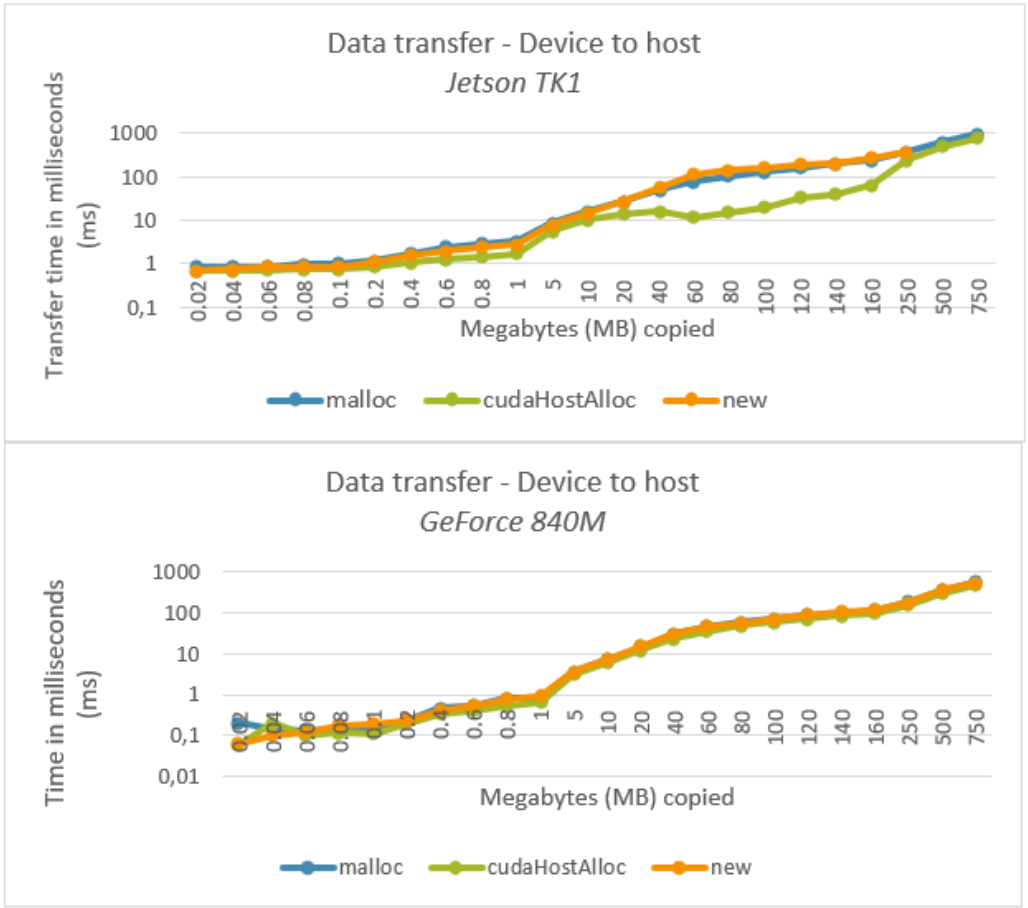

Figure 25: Transfer time: Jetson TK1 and GeForce 840M comparison, device to host, note logarithmic scale

As already mentioned at the beginning of this section there is a difference between pinned/page-locked (*cudaHostAlloc*) and non-pinned/pageable (*malloc*) memory. If the memory is pinned, the operating system guarantees that it will never page this memory out to disk, further it ensures that it will remain in physical memory with a known address [1]. This has the advantage that the GPU has direct memory access – it is safe to be directly accessed. On the other hand, the copy performance for pageable memory allocated by *malloc* depends on the PCIe transfer speed and the system front-side bus speeds. Usually it is stated that page-locked memory has about two times the performance advantage over standard pageable memory when it is used to copy data between GPU and CPU [1]. In addition to the transfer time also the data throughput from the CPU (host) to the GPU (device) and vice versa is measured and visualized in figure 26.

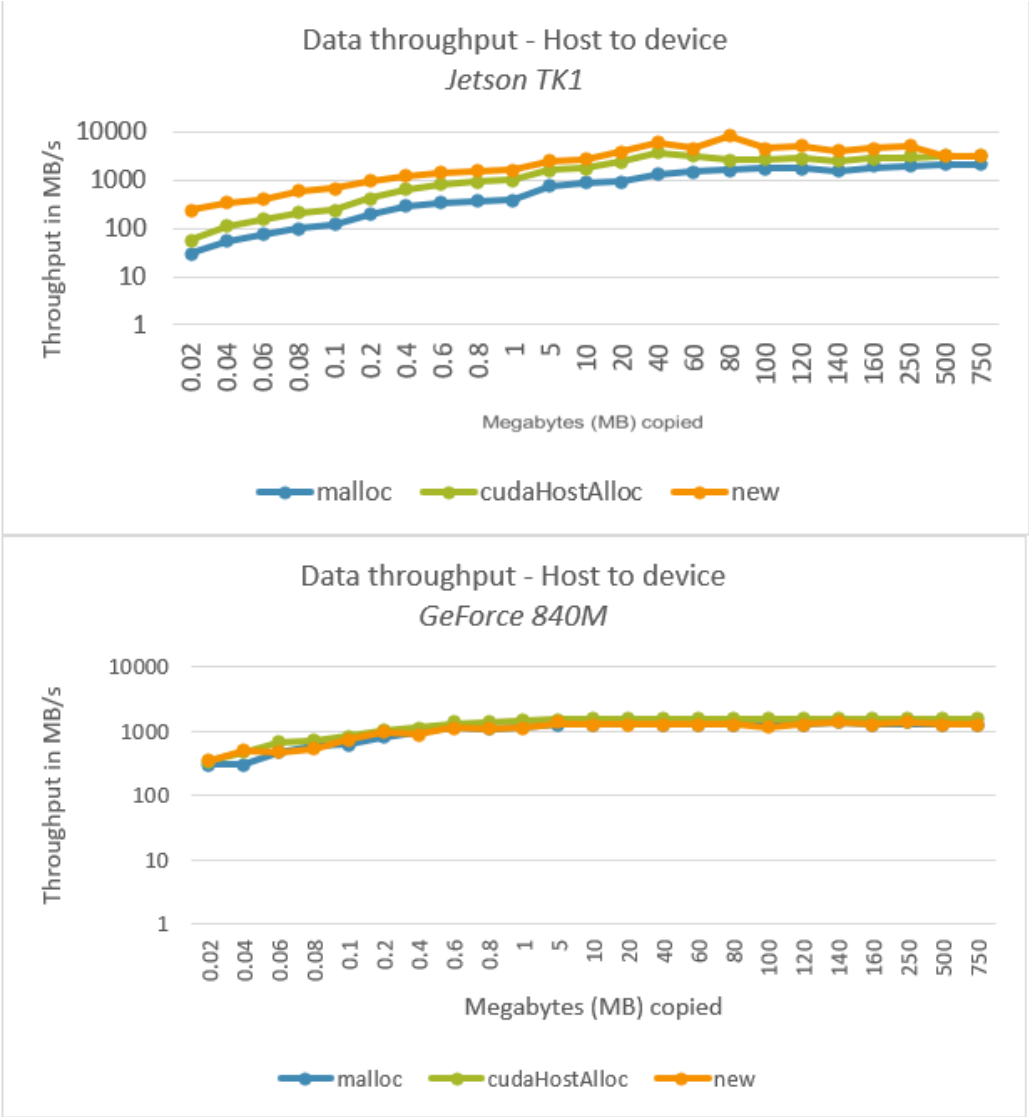

Figure 26: Data throughput: Jetson TK1 and GeForce 840M comparison, host to device, note logarithmic scale

Regarding the data throughput from the host to the device which is visualized in figure 26, it can be seen that *new* is the method with the highest data rate per

second for each data point on the Jetson TK1, followed by *cudaHostAlloc*. But all three methods show similar behavior and increase in the throughput in MB/s till about 60 MB, after this the three methods start to even themselves out and don't climb anymore. The data throughput of the GeForce 840M looks similar but it reached its peak between 0.6 and 0.8 MBs and remains steady for all three functions. In figure 27 the data throughput from the device to the host is shown. For the Jetson TK1, the three methods remain relatively steady until a data size of 10 MB. After this, there is a larger spike of throughput for *cudaHostAlloc* for over 5000 MB/s for data sizes between 60 and 100 MBs, for sizes above this mark the throughput falls again and is almost even with the other two methods at 250 MBs. The results for the GeForce 840M look quite different, it starts at with a steady increase of throughput up to a data size of about 1 MB, then the throughput levels out, important to note that memory allocated with *cuda-HostAlloc* has the largest throughput/s in general on both devices.

Figure 27: Data throughput: Jetson TK1 and GeForce 840M comparison, device to host, note logarithmic scale

For the throughput on the GeForce it is furthermore noted that it starts for the function *malloc* at about 100 MB/s and for the other two functions at ca. 300 MB/s, while the throughput on the Jetson starts at 0.7 MB/s which is a quite remarkable difference.

On top of the memory overhead, also overheads created by kernel launches are examined. Therefore, empty kernels are launched with different configurations and the times are measured. Table 4 shows an overview of the results of this measurement.

Table 4: Times for empty kernel launches with different launch configurations

Device / Launch configuration	**Jetson TK1**	**GeForce 840M**	**GeForce GTX 1070**
Kernel<<<1,1>>>(...)	0.0324 ms	0.0075 ms	0.0051 ms
Kernel<<<100,100>>>(...)	0.05 ms	0.0087 ms	0.0054 ms
Kernel<<<1000,1000>>>(...)	0.87 ms	0.0454 ms	0.0124 ms

Kernel launches are quite expensive if compared with the execution of instructions on a CPU[12] which have in general much higher clock rates. There is a decrease of performance if a larger set of blocks and threads is launched. The GeForce GPUs are also faster for every kernel launch.

5.3 Real-time execution

This extensive experiment examines the possibility for (near) real-time execution of this algorithm and measures the single steps of the algorithm. Different image sizes, certain methods, kernel configurations and the usage of shared memory are investigated to achieve a better performance. Also included in the results is the accuracy of the disparity maps. Further tested are concurrency and streams of algorithm components.

5.3.1 Kernel launch configuration

During the tests, it is measured that a change in some kernel launch configurations achieve a notable increase in the performance of certain functions. After the semi-global matching cost is calculated it has to be added to the already calculated matching cost. Because it is simply the addition of two equal-sized arrays with the size of $image_{width} * image_{height} * max.\,disparity$, it makes sense to use a kernel function and assign each thread with the task of adding up one value pair. The experiments show that a kernel launch that has for either a block or a thread configuration a 1 assigned would lead to a performance, as can be seen in table 5.

12 Intel(R) Core(TM) i5-4210U CPU @ 1.70GHz, 2401 MHz, 2 core(s), 4 logical processor(s)

Table 5: Overview of performance improvement for function "sum_upCosts" that is responsible to add up the matching costs and the calculated semi-global matching cost, image size is 450x375, width: 450, height: 375, max. disparity: 125

Kernel launch configuration for the function: *sum_upCosts*	<<< 1, width * height * max. disparity >>>	<<< Max. disparity, width * height >>>
Jetson TK1	0.696 secs	0.0009 secs
GeForce 840M	0.4856 secs	0.018 secs
GeForce GTX 1070	0.2432 secs	0.0014 secs

After reconfiguration of the kernel launch and exchanging the 1 with the value 125 (Max. disparity) a tremendous increase over all GPUs is noted. And in terms of aiming to reach (near) real-time execution this increase, as shown in the table 5, is very important.

5.3.2 Shared memory

Furthermore, the addition of shared memory during the calculation of the Hamming Distance also has a huge impact on the execution speed of the algorithm on the Jetson TK1 which is displayed in the table 6.

Table 6: Overview of performance improvement for function "calculateHammingCost" that is responsible to deter-mine the hamming distance after the census transform and results in a 3D-cost structure, image size is 450x375, width: 450, height: 375, max. disparity: 125

Kernel: *calculateHammingCost*	<<< width, height >>> No shared memory	<<< height, max. disparity >>> + shared memory
Jetson TK1	0.5877 secs	0.0137 secs
GeForce 840M	0.0434 secs	0.031 secs
GeForce GTX 1070	0.0131 secs	0.0035 secs

The results of this experiment show that the GPUs have a different reaction towards the usage of shared memory. While the GeForce 840M doesn't experience very significant speed-up when using shared memory with this method, the Jetson TK1 shows a large performance boost of over 570 ms. This becomes even clearer if different sizes of images are tested, the calculation of the hamming distance - without shared memory used - takes up over five seconds when images with a size of 975x813 px are used, while the performance of the GeForce 840M is barely decreased.

5.3.3 Examination of algorithm components and image sizes

Figure 28 displays in a diagram how much time the components of the algorithm tested on the Jetson TK1 need to compute a disparity map and it is evident that the semi-global matching slows the disparity estimation down immensely, e.g. it takes for images of the size 450x375 approximately 0,8 seconds per scanline and for images of the size 975x813 already ca. 3,5 seconds. The algorithm pipeline for each image is visualized vertically starting at the bottom, the legend on the right is also in the same order as the components that can be seen in the diagram. It further has to be stated that the total time calculated is not the actual time of the algorithm, because more components than needed are tested here, usually just one filter (either Median or Bilateral is chosen, or neither). The same applies for the local methods (Census Transform + Hamming Distance and SAD), also some parts of the algorithm are so quickly finished that their partition in the diagrams is rather small, in the later section of 5.3.3 there are more diagrams focusing on other smaller and different parts of the algorithm. The following diagram emphasizes on the parts that take approximately the longest like the calculations along the scanlines (y-pos, x-pos, y-neg, x-neg) or the local method SAD. The disparity range used for the calculation is always 125.

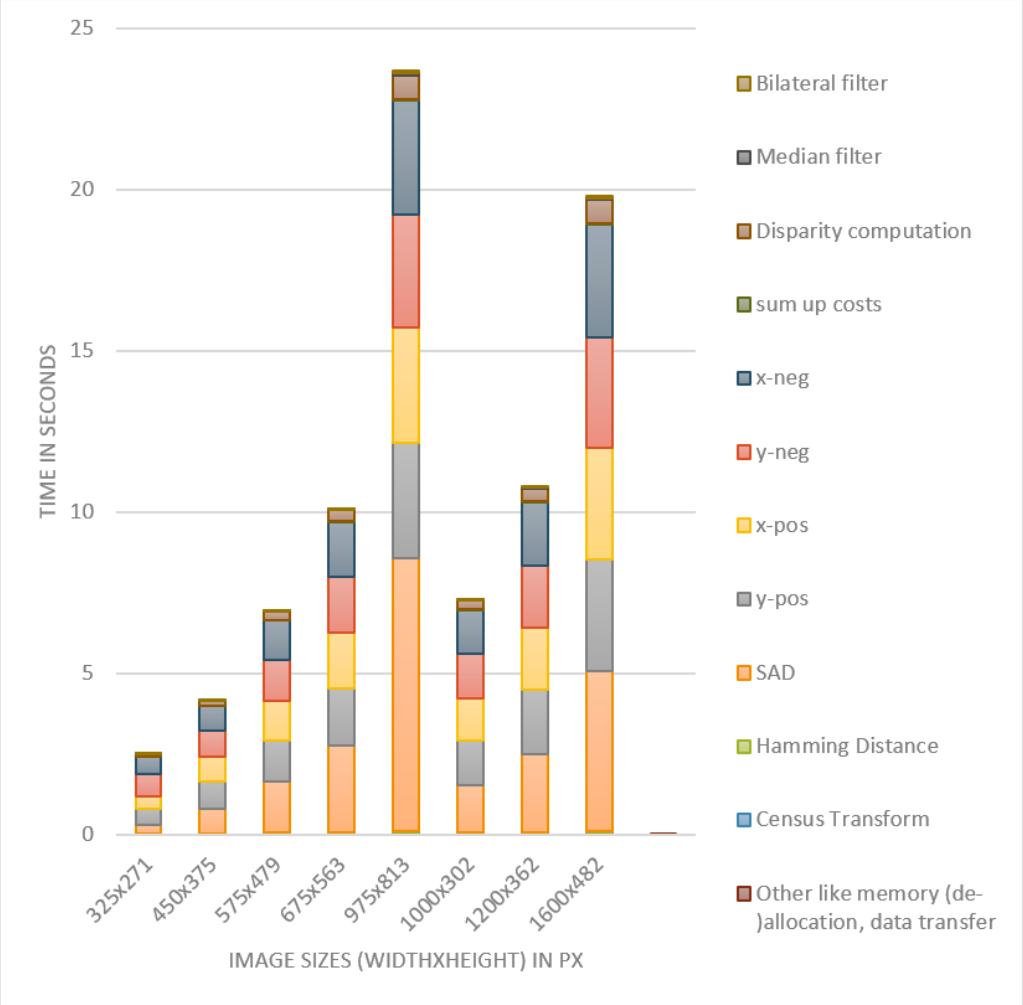

Figure 28: Visualization of the different components of the algorithm and their performance, tested on the Jetson TK1

The dimensions of the image don't change the duration of the semi-global matching and its calculations along certain scanlines in horizontal and vertical direction. Even if the image dimensions are something like 1000x302 px calculating in x- and y-direction takes nearly the same amount of time.

Table 7: Comparison between the two local methods (Census Transform+ Hamming Cost and SAD), image size: width x height

	Jetson TK1			**GeForce 480M**		
	Census Transform	**Hamming Distance**	**SAD**	**Census Transform**	**Hamming Distance**	**SAD**
450x375	0.00667 secs	0.0137 secs	0.783 secs	0.0198 secs	0.031 secs	0.0919 secs
675x563	0.0362 secs	0.014 secs	2.704 secs	0.0432 secs	0.0686 secs	0.194 secs
975x813	0.0738 secs	0.029 secs	8.469 secs	0.0879 secs	0.141 secs	0.398 secs
1200x362	0.04 secs	0.016 secs	2.443 secs	0.048 secs	0.787 secs	0.2336 secs

Table 7 displays the performance of the two local methods of the algorithm (Hamming Distance and SAD), because the calculation of the Hamming Distance relies on images that are census transformed, this function is also a part of the comparison. The performance difference of the two GPUs, especially in regard of SAD is remarkable. It can be seen that the duration for this method is much worse on the Jetson TK1, e.g. for images of the size 975 x 813 px is takes over eight seconds longer to calculate the result, while the GeForce 840M is still finished under one second.

In general, SAD is slower in its execution than the Hamming Distance and further delivers less accurate results, as can be seen in figure 29. A combination of the two local functions where the average value of both calculation results is taken to find corresponding pixels also doesn't increase the final result. Therefore, SAD is not considered in the following tests and consequently diagrams when investigating (near) real-time execution. Instead the disparity maps are calculated only with Census Transform and Hamming Distance calculations.

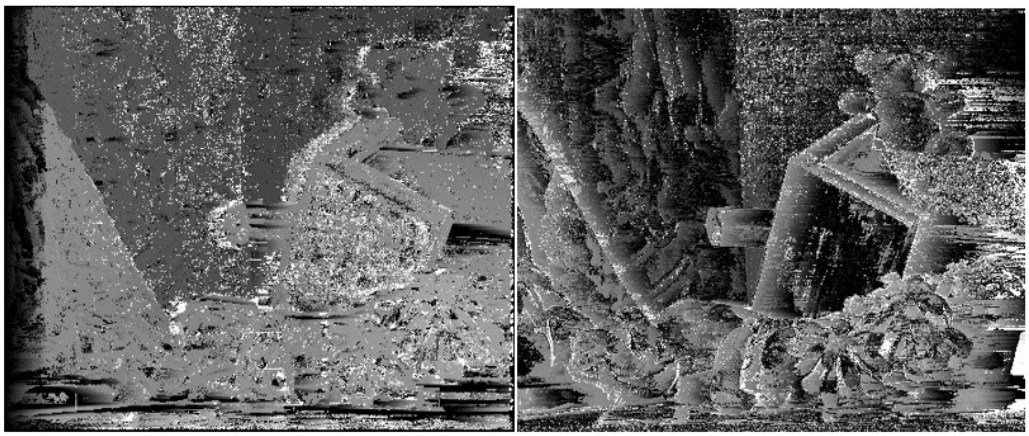

Figure 29: Comparison between Hamming Distance (left) and SAD (right), the original image pair and its ground truth is found in 3.1.1

Farther, the time needed for the calculation of the disparity map can be decreased if the number of scanlines is reduced. The resulting disparity map is still quite accurate but suffers in certain circumstances at some places from the

so-called *streaking*-effect along the used scanline. This is known in dynamic programming too and originates from minimizing the disparity values alongside single paths without considering neighboring image rows respectively columns. One solution is to use two or more scanlines that are perpendicular to each other (e.g. x-pos and y-pos) or a filter that smooths the result and takes these neighboring pixels into account which will be shown in a later part of this section.

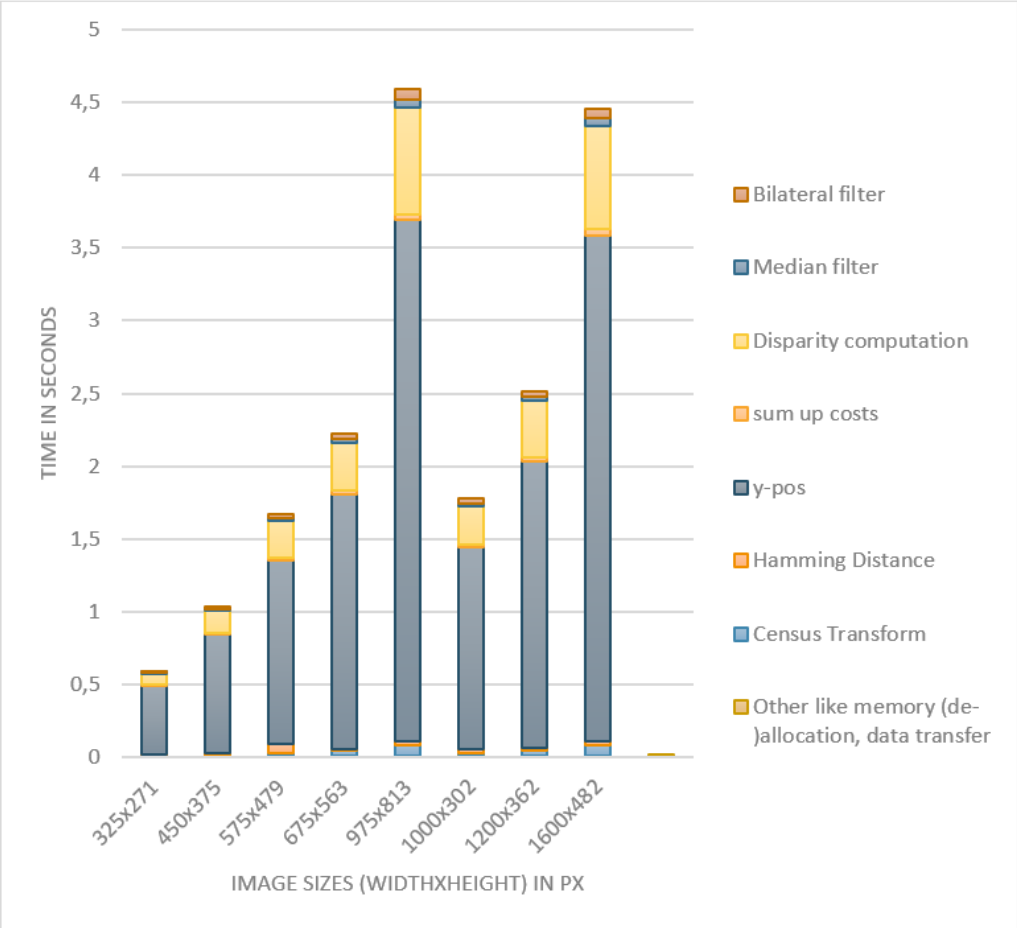

Figure 30: Visualization of the different components of the algorithm and their performance, but just calculating semi-global matching along one scanline (y-pos), also SAD calculation is removed, tested on the Jetson TK1

The figure 30 - which results of tests on the Jetson TK1 - shows the execution speed of the algorithm with just one scanline (y-pos) and one local method used, it increases the performance of the algorithm. For the image with the dimension 975x813 px which approximately took about 23 seconds before needs the algorithm now about 4.7 seconds which is an improvement of over 18 seconds, note that also the SAD calculation which took eight seconds was removed. Also, for the other image sizes is the performance boost notable.

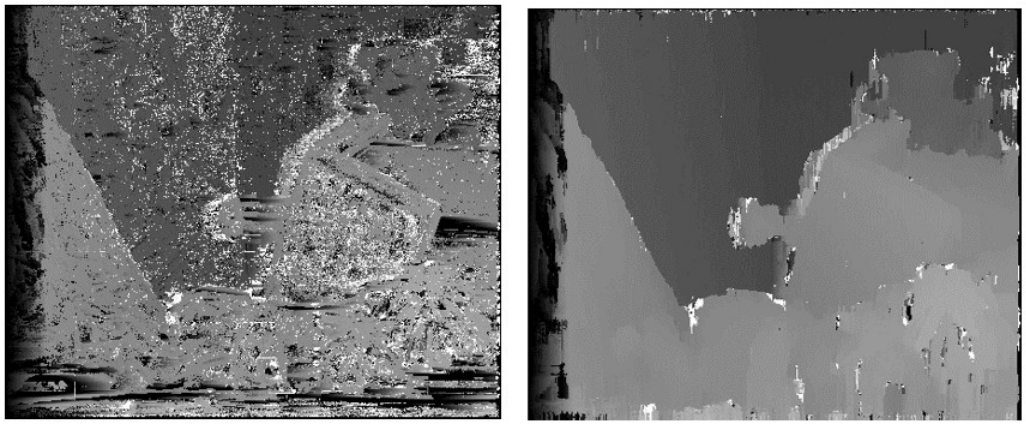

Figure 31: Comparison disparity map, left side: Hamming Distance, right side: Hamming Distance plus scanline along y-pos, filters are not applied to either image, because they are supposed to show the raw result of using just one scanline after calculating the Hamming Distance and no smooth-factor of the filter.

The right side of figure 31 displays the disparity map if there is besides the Hamming Distance just one other scanline used like in figure 30 – but without any filter, in this case along the y-axis in positive direction. It contains some noise areas alongside the edges of objects and further suffers from the streaking effect.

In figure 32 the performance of the filters is assessed, and it is noted that the Median filter (window size 5x5 px) is on average for the measured image sizes about 10 ms faster than the Bilateral filter (window size 5x5 px). An increase in window size increases accuracy, but always decreases the performance, in combination with one scanline that already smooths the result it is not necessary to increase the window size further.

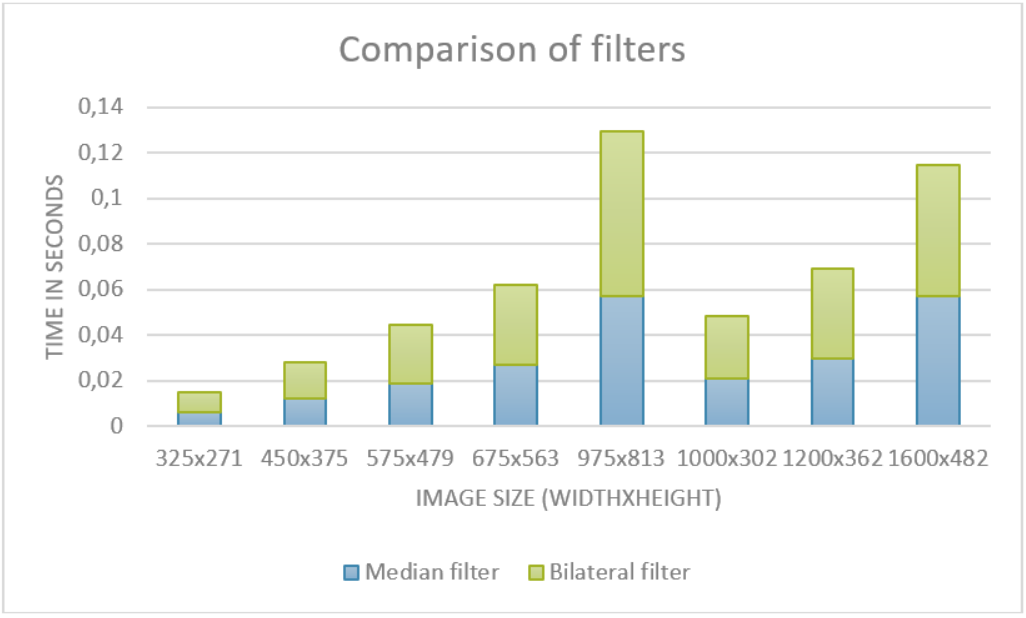

Figure 32: Time comparison between Median and Bilateral filter

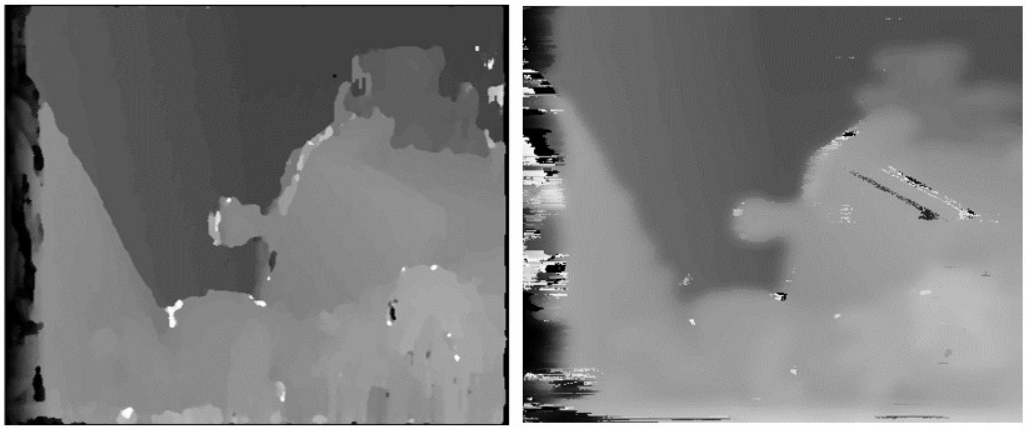

Figure 33: To the right disparity map in figure 32 is a filter applied, left side: Median filter, right side: Bilateral filter

Figure 33 displays the teddy disparity map and an applied filter – after using the Hamming Distance calculation and one additional scanline (right side of figure 31), on the left side the result of the Median filter and on the right side of the Bilateral filter is displayed. In comparison to the right side of figure 31 the accuracy is increased, even though there are still some noise areas left, but the so-called streaking effect which is a result of just using one scanline has vanished.

Farther, other example input images which can be found in 3.1.1 are used for testing and evaluating the algorithm. Whether one scanline suffices to provide results accurate enough, the results show that there are some problematic areas, especially around edges or small, but distinctive surfaces. The results of theses tests with different image-pairs can be seen in figure 34.

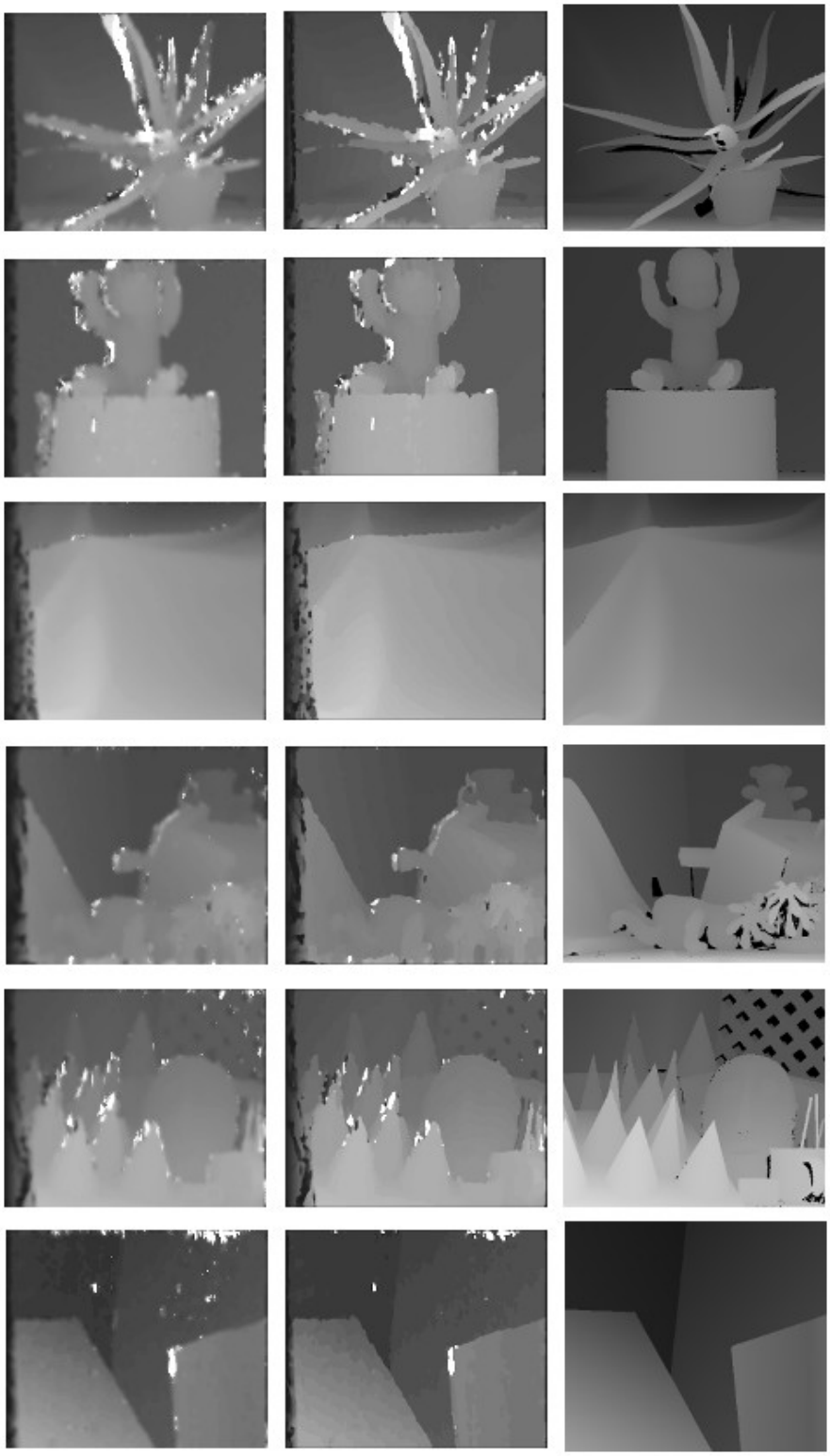

Figure 34: Example disparity maps comparison, left side of each pair: results of using one scanline with Median filter, middle: using one scanline with Bilateral filter, right side: ground truth, the ground truths are from the Middlebury datasets[13] and the images are called from left to right, first row: aloe, baby1, cloth1, teddy, cones and venus.

13 http://vision.middlebury.edu/stereo/data/, retrieved 2018-08-01

In the next step of the experiment, the semi-global matching calculation is removed completely – because it makes (near) real-time execution impossible – and the disparity map is only calculated with the Hamming Distance and WTA, before the Median filter is used to refine the result.

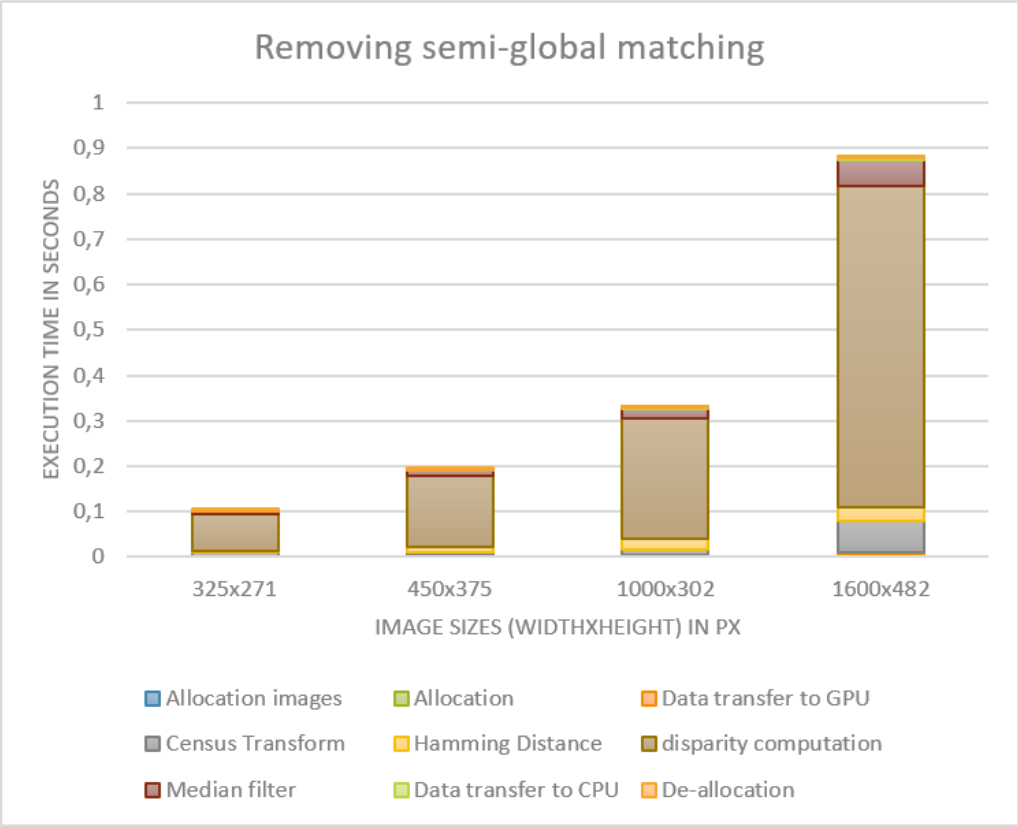

Figure 35: Visualization of the different components of the algorithm and their performance, after removing all semi-global matching calculations, tested on the Jetson TK1

While the accuracy is decreased, and the noise values increased the performance of the algorithm improves significantly. Even the largest tested image finishes within a second on the Jetson TK1, more specifically after almost 0.9 seconds. Furthermore noteable, figure 35 also allows a better overview of parts of the algorithm that finish very quickly like the calculation of the hamming distance which could not be displayed as clear in the figure 28 and 30.

To refine the result more, different window sizes for the Median filter are tested, using window sizes larger than 5x5 increases the accuracy, removes noise, but are a bit more costly – yet still have a better performance than using an extra scanline.

Figure 36 displays these results of the algorithm if the semi-global matching is completely removed and just the Hamming Distance is calculated before the Median filter is used to smooth the result. As can be seen, there still is noise left, but depending on the used window size of the Median it can be reduced as well.

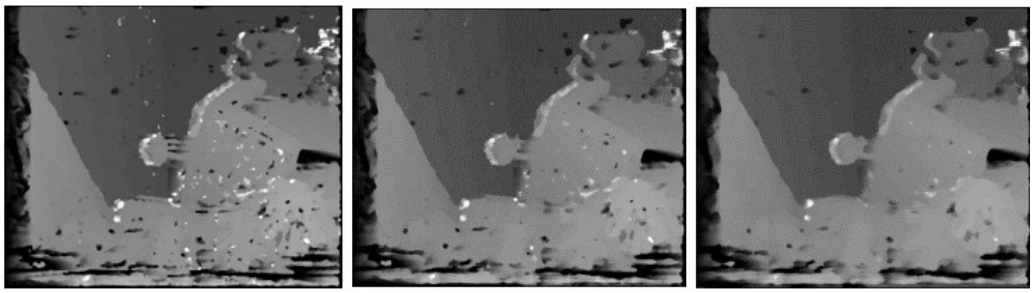

Figure 36: Disparity map only generated with Hamming Distance and the Median filter, left image: window size 5x5 px, middle image: window size 7x7 px, right image: window size 9x9 px

During the experiments, it is also examined how changes in the semi-global matching calculation along the scanlines will affect the performance and the final result of the algorithm. It is tested what happens to the result and more important to the execution speed if the scanline itself is bisected and parallelized or even quartered and parallelized. The results of this investigation are shown in the table 8. It is tested on the Jetson TK1.

Table 8: Overview of different types of scanlines/path calculation, full path: normal version, calculating semi-global matching along whole direction of the image; bisected path: scanline is divided by two and parallelized; quartered: scanline is divided by four and also parallelized, as a result the approximate duration of the scan-line calculation has been taken for each image pair, tested on the Jetson TK1

Path-type / Image size	Approx. secs/full path	Approx. secs/bisected path	Approx. secs/quartered path
450x375	0.78 secs	0.69 secs	0.5 secs
775x646	2.4 secs	1.9 secs	1.9 secs
1000x302	1.3 secs	1.1 secs	0.9 secs
1600x482	3.4 secs	3 secs	2.3 secs

Even though there is no significant difference of the single durations of scanline calculations along different directions (x-direction and y-direction) when using the full path, there are some changes between the times of the single path calculations if they are divided by two or respectively by four. As an example, the algorithm needs for the image with 1000x302 px - which has a significant longer width/x-dimension - for the scanlines in x-direction when divided by four ca. 1.13 seconds, but in y-direction approximately only 0.7 seconds. It is also notable that e.g. the bisection of the scanline doesn't cut the spent time for this instruction simply in half, but only reduces it to some extent.

5.3.4 Concurrency and CUDA streams

Moreover, about (near) real-time execution and how it can be achieved: streams, concurrency and its challenges are examined and tested. *Concurrency*

describes the ability to perform simultaneously a multitude of CUDA instructions that goes beyond the already multi-threaded parallelism, so it means e.g. kernel launches, memory transfer or simply operations directly on the CPU [23]. The introduction to chapter 5, table 2 shows important device properties, concurrency and the overlapping of memory copies are crucial for realization of streams in CUDA, in general most GPUs with a compute capability 2.x or higher are able to execute different instructions concurrently [24]. Figure 37 displays an abstract example of concurrency:

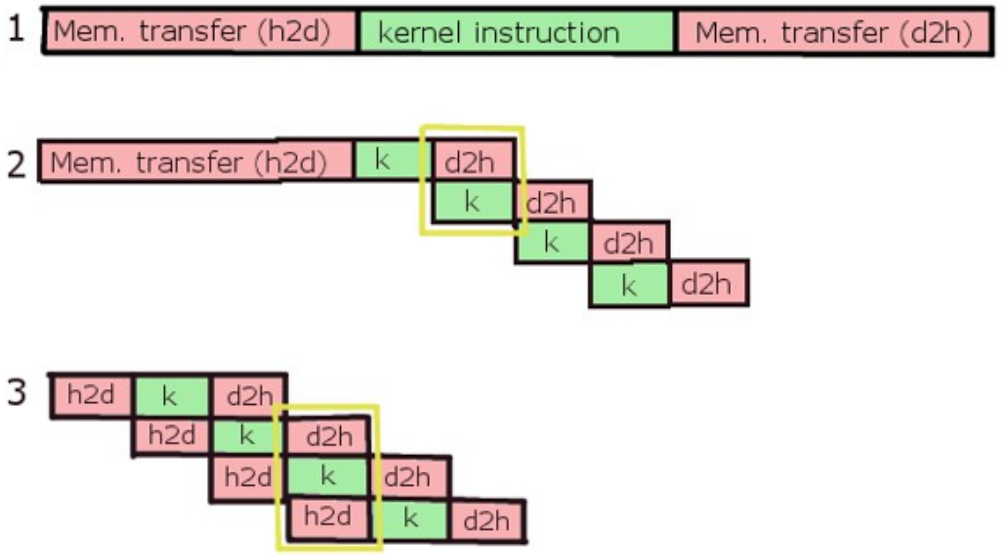

Figure 37: Abstract overview of possibilities of concurrency, h2d: memory transfer from host to device, d2h: memory trans-fer from device to host, (1) shows iterative/serial execution of instruction set, (2) shows two-way concurrency: one kernel and memory transfer d2h is executed concurrently. (3) displays three-way concurrency: kernel execution and memory transfer from CPU to GPU and vice versa are concurrent, in the first row one default stream is used, the second and third row uses 4 different streams

As can be seen for (3) in figure 37, memory copy overlapping is used, data is concurrently copied from the CPU to the GPU and vice versa. Usually, kernels run asynchronously which means that the CPU doesn't have to wait till the kernel is finished but can proceed with the instructions and tasks which makes concurrent execution possible. Instructions like e.g. *cudaEventSynchronize* prevent concurrency.

Even though the algorithm in this study consists of various different components and methods within the pipeline, it is often unfavorable to run concurrently, because it has several dependencies and results from prior functions that are needed in the next calculation, e.g. Census Transform needed for calculation of Hamming Distance or result of Hamming Distance is needed for the semi-global matching. Also, there is besides the memory transfer of the input images at the beginning of the algorithm and the transfer of the result back to the CPU in the end no further data transfer which could run concurrently and improve the performance, further the CPU itself doesn't need to finish any other tasks or do calculations on its own.

Still, one part where concurrency is reasonable within the algorithm is investigated on the GeForce 840M. During the semi-global matching, the path or scanline calculations itself do not depend on each other and the results for each path at the end are simply added together, there are no further dependencies. For the concurrency experiments four streams are created, each responsible for one path. The serial execution uses the default stream that is always automatically created. Figure 38 displays the results of serial and concurrent execution visible in the NVIDIA profiler. The slight overlap of kernel execution in the bottom part of the figure shows the concurrency in a timeline, the reason that there is not more overlap is that there are no more resources on the GPU available.

Figure 38: Overview of comparison between serial and concurrent execution of the scanlines on the GeForce 480M, top part of the figure shows the serial execution where just one default stream exists and every kernel is executed after another. The bottom part displays the execution of the concurrent four kernels in four different streams, it can be seen that there are slight overlaps during the execution (next kernel starts before the previous one is finished).

It is notable that the actual performance of the algorithm – when using concurrency – is not improved, because it takes longer to finish the single scanlines which can be seen in table 9. Still, the algorithm ends approximately after the same amount of time, because of the overlap in the kernel execution. If streams are created they run as soon as possible that they have not started earlier is an indication that there are not enough GPU resources available at an earlier time.

Table 9: Comparison between serial and concurrent execution of scanlines during semi-global matching, concurrent execution uses 4 streams for the paths and on default stream, used image size 450x375 px.

Direction / Execution type	y-pos	x-pos	x-neg	y-neg
serialized	6.76 secs	6.819 secs	6.827 secs	6.76 secs
concurrent	7.857 secs	8.51 secs	7.83 secs	7.09 secs

Furthermore, important to note, the values which are used for calculating are stored within one place of the memory (in one 3D-coststructue), the results are also saved in one place, consequently the single streams don't necessarily need their own resources. There is also no difference of the result or of the duration/performance of the semi-global matching if each concurrent kernel gets its own set of resources assigned, respectively its own variable to store the result of the calculation.

5.4 Investigation of concurrent CUDA applications and processes

In the fourth experiment it is examined how the algorithm could facilitate constantly updating image-pair input data streams, therefore it is necessary to examine solutions for the possibility of running multiple CUDA applications/processes on the same GPU concurrently, e.g. letting the GPU calculate two or more different disparity maps with different input images simultaneously. This functionality is not simply possible to implement and usually not intended, on the contrary if there is a need of running more than one CUDA applications it is recommended to use multiple GPUs [1].

If this still is to be attempted or if just one GPU is available, it is crucial to ensure that the accessible device memory is big enough for multiple processes running at the same time, therefore there are limitations of parallelization within the single processes (kernel launch configurations) but also the usage of shared memory. In general, it needs to be very carefully designed, especially in regards of the resource utilization of the kernels. Image processing is one of the more expensive operations, especially if working with single pixels and assigning single threads directly to them. CUDA itself does not provide any built-in functionality that would allow to partition GPU resources directly and assign e.g. cores to a particular kernel or stream. If e.g. two CUDA applications are launched – maybe even on multiple CPU cores - the host code runs always concurrently, but the kernels - respectively the device code - are serialized and there is a high risk that it will lead to a low GPU resource and capability utilization [25].

But there is an option to run whole processes like constantly updating image-pair input data streams concurrently. It is called Multi-process service (MPS) and is an alternative, binary-compatible client-server run-time implementation of the CUDA API [25].

5.4.1 MPS architecture types

Two different architectures based on the available GPU are introduced *Pre-Volta* and *Volta* architecture. Volta comes with more features like direct connection to the GPU without having to pass the instructions through the MPS server or a dedicated GPU address space for each client/process, figure 39 shows a comparison between the two architectures.

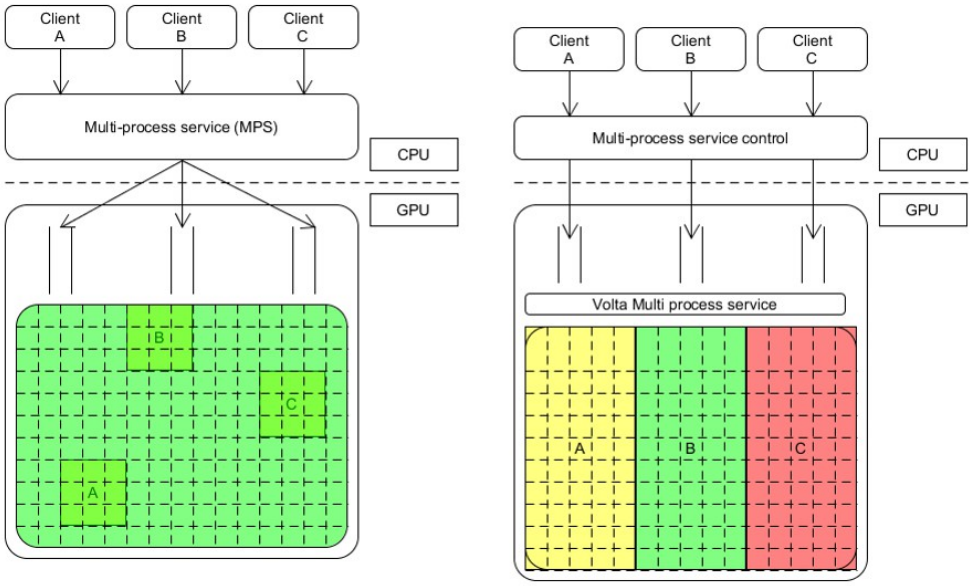

Figure 39: Comparison Pre-Volta architecture left and Volta architecture right

MPS consists – based on its architecture – of multiple components: A *server* is the shared connection for each *client* to the GPU and ensures the concurrent execution on it. The *control deamon process* administrates the server – startup and shutdown of the server – it also manages the connections between all the clients and the server. If the connection from a client to the server fails, the application will run normally – as it would without MPS. For communication pipes and UNIX domain sockets are used.

Moreover, Pre-Volta architecture can handle up to 16 CUDA applications which means e.g. 16 clients with one CUDA process or eight clients with two running CUDA applications each. On the other hand, Volta MPS servers support up to 48. [25]

5.4.2 Limitations and challenges of this technology

Yet, there are constrictions to this functionality that have to be considered. The two main ones are: it is only supported by a Linux OS and it requires a GPU with a compute capability of at least 3.5. Furthermore, Unified Virtual Addressing has to be available, only 64-bit applications are supported and the usage of dynamic parallelism which means e.g. dynamic allocation of shared memory is not possible. For Pre-Volta usage: out-of-range errors don't give a direct indication or throw an exception, but just lead to unexpected behavior. This results out of the fact that as can be seen in figure 39 the single partitions don't have a dedicated space on the GPU. Regardless which architecture is used, if an exception is thrown it will be reported to all clients but won't give an indication which one is the reason for the error, a fatal exception will terminate all clients.

5.4.3 Setup

The available amount of threads for each running CUDA application can be adapted, a recommended strategy is to divide the threads by half of the number of expected clients. This allows the load balancer more freedom in overlapping

kernel execution between different clients, because of the remaining resources in standby. [25]

The following input shows an exemplary startup of the server which allows if successfully immediate execution of concurrent CUDA processes and doesn't need further instructions:

```
export CUDA_VISIBLE_DEVICES=0
nvidia-smi -i 0 -c EXCLUSIVE_PROCESS
nvidia-cuda-mps-control -d
```

It has to be performed having admin privilege, the first instruction selects the GPU 0 which always is the first or in single-GPU systems the only device available. The second instruction line sets the GPU to the exclusive mode which guarantees that only one MPS server is using the GPU. The last line simply starts the *deamon*.

After launching the MPS server successfully it is e.g. possible to write and start a bash script that starts multiple applications or for test purposes two copies of one application.

5.5 Final analysis

This final section serves the purpose to shortly summarize the extensive previous results and prepare for the conclusion in the next chapter.

The first experiment shows that there are huge benefits throughout the algorithm if it is parallelized, especially the calculations of the scanlines are enhanced immensely. The comparison between the GPUs themselves returns different results in performance that also surprised to some extent. The execution of the scanlines is much faster on the Jetson TK1 than on the GeForce 840M, the same is true for the calculation of the Hamming Distance. On the contrary to this, the calculation of SAD is faster on the GeForce 840M than on the Jetson TK1. But the semi-global matching is also the most costly components for the GPUs.

The second experiment demonstrates that there are indeed differences of the available allocation functions that also affect the performance for different data sizes if the data is transferred from the CPU to the GPU and vice versa. It also shows, that they have an impact on the throughput itself and that transfers from CPU to GPU take not necessarily always the same amount of time as the transfers back from the GPU to the CPU. Moreover, launching kernel functions are expensive in comparison to execute a program only on the CPU, especially if blocks and threads are launched in a bigger dimension.

The third experiment gives a good impression which parts of the algorithm can be improved further and where there are some challenges regarding real-time execution. Kernel launch configurations and the usage of shared memory have an important influence on the speed. Simple changes in the *sum_upCosts* and

calculateHammingCost in this matter result in a notable improvement in overall performance. Also investigated are different image sizes and their impact on the speed, to determine which dimensions are still optimal of a fast execution like 450x375 px or 1000x302 px.

Different components like the local methods (Hamming Distance and SAD) are compared and the first one shows a better overall performance. Reducing the algorithms of some components like SAD or even some of the scanlines lead to an increase in speed, but in the case of removal of the scanline the accuracy of the disparity map is likely to suffer. Using just local methods the Jetson TK1 is capable to produce with a Median filter disparity maps within one second for most of the image dimensions. Changing the window size of the Median filter is also a way to increase the quality of the disparity again without it being as expensive as a scanline.

In an attempt to improve the semi-global matching more efficient the scanlines were parallelized (two respectively four times) and an improvement of speed is measured but which was overall smaller than expected. Also, executing the scanlines concurrently doesn't lead to a faster execution. While there is overlapping in the kernel functions, the overall duration for the scanlines increases as well.

The last experiment discussed the possibility to not just parallelize single parts of the algorithm like memory transfers and kernel functions but let the whole application run concurrently with a copy of itself. This is not so simple to achieve, but a service called MPS allows the developer with certain limitations (e.g. Linux, compute capability of 3.5 or higher) to run CUDA processes simultaneously on one GPU and ensures a GPU utilization as efficient as possible.

6 Conclusions

In this chapter a conclusion for the study is provided and some personal input from the author. First, the ethical aspects are described, then the goal fulfillment and after this the challenges and difficulties are discussed. Another important section of this chapter is possible future work.

6.1 Ethical aspects

This particular study falls short of many usual ethical aspects, because the implementation and evaluation of the actual depth estimation algorithm doesn't really pose any direct threats or imminent risks to the general meaning of safety, security or health of people, especially if the goal is to use the results only within the lab environment. Depth estimation in 3D-vision generally is often inter alia used for the improvement in the car industry to develop help systems for people to keep track of their surroundings within the car [13] or it even supports the development of self-driving vehicles. Another field of interest is the possibilities for the usage of depth estimation systems in robotics and their navigation [14]. But for some applications like for the usage of a depth estimation algorithm in a car it is important to note that there is a huge responsibility on the developer to make this process absolutely reliable and safe for the people using it on the streets.

One ethical aspect of this thesis itself is definitely worth mentioning. The algorithm is developed with the homogeneous CUDA which is endorsed by NVIDIA, it also only works with NVIDIA graphics cards, so it is consequently impossible to use cards from other companies with this implementation. It forces the developer to use a certain hard-ware for the application. This has to be kept in mind when ordering a new card, e.g. either because the old one malfunctioned or when cards need to be replaced with newer and better ones or a more affordable one. It also limits the developer to a certain IDE depending on the chosen operating system (e.g. Linux or Windows). This can be troublesome for some developer who like their keep their options and freedom in that regard. If a Non-NVIDIA graphics card should be used for this algorithm, it makes it necessary to rework the whole code and use an alternative like OpenCL which is an API for heterogeneous computing. This raises the question whether CUDA really is the optimal choice for such an algorithm: In the case of this thesis project it is the *right* choice, because the algorithm should only expand already implemented applications by the realistic-3D research team at the Mid Sweden University who is using CUDA in their whole setup. The usage of an alternative would therefore not have even been possible to start with.

6.2 Goal fulfillment

In 1.4 the goals for this thesis are listed, they are as follows:

- Implementation of a parallelized depth estimation algorithm with CUDA on a NVIDIA capable GPU
- Evaluation of the developed algorithm
  - What sections of the algorithm benefit from parallelization?
  - How significant are the imposed overheads?
  - How can real-time or near-real-time execution be achieved?
  - How can the algorithm facilitate constantly updating image-pair input data streams?

This subchapter aims to determine how well the goals have been met and the questions have been answered and where there have been challenges and difficulties. The first goal was met by investigating a stereo matching algorithm and fully implementing it with CUDA. Further, it was examined how the resulting disparity map can be transformed with a simple calculation to a depth map. It was pointed out what information about the cameras and the scene stetting are needed, so that the depth information can be extracted. This will be of importance as soon as the algorithm is actually integrated into the pipeline and fetching the input images from the actual cameras of the scene.

Originally, it was planned to test and evaluate the algorithm directly with the input images of the research lab at the Mid Sweden University, yet this was not possible, because the image has not been rectified yet – which was not part of this study -, instead mainly images from the Middlebury testbed [10] were used. They fully sufficed as training data and even provided a wide variety of different images with different stereo matching challenges. Furthermore, it was possible to evaluate the algorithm not only on the Jetson TK1, but also on a GeForce 840M and the GeForce GTX 1070 that use among other things another architecture and show different performance results which is quite interesting to compare.

The first question during evaluation that has to be addressed is what sections of the algorithm would benefit from parallelization. The answer is found by implementing a CPU version of the algorithm, because this would show which sections and components are accelerated the most by parallelization speaking what increases the performance the most. The second question demands to investigate the overheads of a GPU implementation, to provide feedback on a broader spectrum not just overheads imposed by kernel launches and configurations are investigated, but also overheads regarding memory management and transfer in relations to the data size. In the end it was possible to provide an overview of different aspects of overheads which are visualized

for a clearer understanding. Next, the question should be answered how (near) real-time execution can be achieved, therefore the duration of single components is measured to determine which instructions take the longest and if it would be even possible with this algorithm to achieve it. Depending on the image dimensions and if the whole pipeline of the algorithm (with the semi-global matching) is used (near) real-time execution is not possible, because calculations along the scanlines or paths through the image take too much time. If only a local method like the calculation of the hamming distance and a filter like the Median filter are applied to an image pair the dimensions of 1600x482 px the algorithm – running on the Jetson TK1 - is capable of calculating a disparity map within one second (as displayed in figure 35). The results are still affected by noise, because local methods are often less distinctive, using a Median filter might improve the result. The Hamming Distance provides still more accurate results than SAD. Further examinations of the single parts of the algorithm in regards of (near) real-time execution showed that changes in the kernel launch configurations or if shared memory is used lead to a better performance, the later especially when used on the Jetson TK1. Concurrency tests are done with the scanlines to improve the speed which seemed advisable, because the semi-global matching is the slowest part of the algorithm and also the only instruction set that don't have dependencies amongst them. But it does not have the desired effect and doesn't improve the performance. Even though it is not possible to achieve (near) real-time execution with this implementation of the algorithm in its entirety, the challenges and difficulties have been identified and some solutions have been examined as well, e.g. the bisected/quartered calculation of the scanlines or reducing the components of the necessary algorithm. The last question is about how the algorithm is able to facilitate constantly updating image-pair input data streams, this was harder to answer, especially because it is not an intended option to run whole CUDA processes simultaneously on one GPU. But the technology MPS offers theoretically the capabilities to achieve just that. As part of this experiment MPS was investigated, the architecture, the benefits, the set-up, but also the limitations and challenges.

6.3 Challenges and difficulties

In this section challenges and difficulties which occurred during the study are discussed, but not purely from the side of objectiveness, because some personal input and experience is added as well. The purpose of this section is to give further insights in the development process that can be useful for other developers.

- One of the difficulties during the work process was based on the used operating system (Win 10) and the settings of the GPU (GeForce 840M) itself. The kernel execution timeout was active and the so-called TDRdelay was set to two seconds. TDRdelay sets the frame after how many seconds the kernel is terminated when running on the GPU. This had to be changed and adapted manually, so that also kernels with a longer duration could run properly.

- This first difficulty noted, is also an indication that GPU applications are just more challenging to debug than many CPU programs. Error messages are even after querying the exact expression and choosing a verbose option not always easy to understand and assignable to the actual problem. Furthermore, unexpected issues can also very easily occur and cause confusion, e.g. using the input images in the wrong order results in a very distorted disparity map and when only starting with image processing and not having any prior experiences with this topic made it quite challenging to identify this issue in particular.

- While working on the GPU and the CPU version of the algorithm it could be experienced firsthand that this are two very different approaches to the same problem. The CPU version seemed to be more distant and overlooking the process as a whole, while the GPU programming style made it necessary to take a closer look at the actual scenario and unravel the problem at a whole new level. One example would be setting the kernel launches and assigning blocks and threads to the single pixels and calculating the wanted results.

- Another challenge was to get acquainted to the high mathematical part of the program, because after all it is an algorithm. Every method and component are based on equations and during the literature study it was important to really dedicate some time in understanding and transferring these formulas to the actual code.

6.4 Shared memory on different GPUs

As can be seen in table 6, the calculation of the Hamming Distance has been changed in early stages during the evaluation process, so that it now uses shared memory. This was due to the outcome of the first tests on the Jetson TK1, while the Hamming Distance was calculated quite fast on the GeForce 840M and stayed relatively efficient even for larger image sizes, the calculation of it on the Jetson TK1 decreased immensely to a point where it made this change absolutely necessary. Still, the results of this examination were very interesting and simply surprising. While other weaknesses in the algorithm like the performance of the scanlines were kind of clear from the beginning this development was most definitely unexpected and makes it even more necessary to investigate the use of shared memory on the Jetson TK1 device.

6.5 Final setup recommendations

Even though the algorithm contains several different components which all can technically be used, it doesn't necessary lead to the desired result – regarding accuracy and performance. During the evaluation it was tested which parts of the algorithm are essential in providing these goals, two different solution could be filtered out so far.

If the aim is a high performance while accuracy can be neglected to some extend the focus should lie on the local methods like the Hamming Distance (+ Census Transform), this function can be combined with a filter like the Median

and provide a disparity map rather quickly. Selecting larger window sizes like 7x7 px or even 9x9 px improves the results a lot and it is performance wise still cheaper than using the additional scanlines.

If the goal is to obtain more accuracy and performance is less important, it is recommended to use one or at maximum two different scanlines like y-pos and x-pos. After one scanline results are already very noise reduced and using a filter like the Median (window size 5x5) in the end smooths the disparity map even more.

6.6 Future work

To improve the algorithm in its performance and accuracy of the results some actions are recommended.

- While the algorithm could be evaluated on the Jetson TK1 device it was not possible to integrate the algorithm in the pipeline yet and retrieve the input images directly from the camera system – and not testbed images out of a directory. Which is why it is recommended to include it when working on improvements for the algorithm in general and for further evaluation. Integrating the application would also open up the possibility to know about the camera/scenario settings which would allow to produce not just the disparity map, but also extract the true depth out of those.

- Another in-depth examination possibility is shared memory and the attempt to use this more extensively. While the GeForce 840M which was mainly used for the implementation and testing doesn't show a huge improvement with shared memory, the Jetson TK1 has – as can be seen when calculating the Hamming Distance – a remarkable performance boost. Investigating it further looks therefore promising, integration of shared memory into the filter calculations seems quite reasonable. The most possible gain would definitely be to integrate it into the scanline calculations themselves.

- Furthermore, as could be experienced during the testing the local method, SAD is not efficient right now, because it has a worse performance speed and accuracy than the Hamming Distance. To be able to exchange the local methods and calculate disparity maps with either function it is necessary to improve SAD first. This could be achieved by expanding the method into the adaptive support weight function. It uses a window around the center pixel and determines based on the intensities of its neighbors the support-weights that are supposed to assist in the search for the corresponding pixel. This window around the center pixel might increase the accuracy of the simple SAD function, because more pixels are taken into account.

- Also, it might also be of interest if concurrency of single streams within the algorithm is further investigated. The delay in concurrent execution indicates that the GPU does not have enough resources at the moment to execute those kernels faster simultaneously. Therefore, it seems advisable examining if the scanlines can be more carefully designed, e.g. exactly adapted to the image sizes when integrated in the lab pipeline, scale images down so that less blocks and threads needed and test it with Jetson TK1.

- Even though MPS explicitly states that it requires a compute capability of 3.5 or higher it might be worth a try on the Jetson TK1 (with a compute capability of 3.2). The other solution would be to find a machine that fulfills the requirements of this service (e.g. also uses a Linux OS) and investigate the performance of two or more applications run in parallel on the same GPU to the performance of two or more executions of the process without MPS.

- If continuing using the algorithm also separately from the pipeline it seems desirable to develop a small and simple GUI to allow the developer and user an easier interaction with the algorithm and its variables. Besides the input images other values can be changed as well like window size of the Median filter, the blur radius of the Bilateral filter or even the penalty terms of the single semi-global matching scanlines. It could also be used to set if semi-global matching in general should be used and if so how many paths/scanlines should be calculated.

References

[1] J. Sanders, and E. Kandrot. “CUDA by example: an introduction to general-purpose GPU programming.” Addison-Wesley Professional, 2010.

[2] P. Revuelta Sanz, B. Ruiz Mezcua, and J. M. Sánchez Pena. “Depth estimation – an introduction.” Current Advancements in Stereo Vision. In-TechOpen, 2012.

[3] R. Hartley, and A. Zisserman. “Multiple View Geometry in computer vision”. Cambridge University Press, 2003.

[4] J. Kowalczuk, E. T. Psota, L. C. Pérez. “Real-time stereo matching on CUDA using an iterative refinement method for adaptive support-weight correspondences”, IEEE Transactions on circuits and systems for video technology, vol. 23, no. 1, 2013, pp. 94-104.

[5] K. -J. Yoon, and L. -S. Kweon, “Adaptive support-weight approach for correspondence search”, IEEE Transactions on pattern analysis and machine intelligence, vol. 28, no. 4, 2006, pp. 650-656.

[6] H. Hirschmüller, and D. Scharstein, “Evaluation of stereo matching costs on images with radiometric differences”, IEEE transactions on pattern analysis and machine intelligence, vol. 31, no. 9, 2009, pp. 1582 – 1599.

[7] H. Hirschmüller, “Stereo processing by semiglobal matching and mutual information”, IEEE Transactions on pattern analysis and machine intelligence, vol. 30, no. 2, 2008, pp. 328-341.

[8] D. Scharstein, and R. Szeliski, “A taxonomy and evaluation of dense two-frame stereo correspondence algorithms”, International Journal of Computer Vision, vol. 47, no. 1 , 2002, pp. 7-42.

[9] R. Yang, M. Pollefeys, and S. Li, “Improved real-time stereo on commodity graphics hardware”, IEEE Proceedings of 2004 Conference on Computer Vision and Pattern Recognition Workshop, vol. 3. Washington DC, 2004, pp. 36.

[10] Middlebury Testbed: http://vision.middlebury.edu/stereo/
Retrieved 05-04-18.

[11] KITTI Vision Benchmark Suite:
http://www.cvlibs.net/datasets/kitti/eval_stereo_flow.php?

benchmark=stereo
Retrieved 05-04-18.

[12] NVIDIA Documentation, Profiler User's Guide: https://docs.nvidia.-com/cuda/profiler-users-guide
Retrieved 05-04-18.

[13] D. Hernandez-Juarez, A. Chacón, A. Espinosa, D. Vázquez, J. C. Moure, and A. M. López, "Embedded real-time stereo estimation via Semi-Global Matching on the GPU", ICCS: the international conference on computational science, vol. 80, 2016, pp. 143-153.

[14] S.C. Diamantas, A. Oikonomidis, R.M. Crowder: Depth estimation for autonomous robot navigation: A comparative approach. IEEE International Conference on Imaging Systems and Techniques. Thessaloniki, 2010, pp. 426-430.

[15] NVIDIA, What is GPU computing, http://www.nvidia.com/object/what-is-gpu-computing.html, retrieved 2018-08-01.

[16] NVIDIA Developer, CUDA-zone, https://developer.nvidia.com/cuda-zone
Retrieved 2018-08-01.

[17] UW CSE vision faculty, Lecture 16: Stereo and 3D Vision, https://courses.cs.washington.edu/courses/cse455/09wi/Lects/lect16.pdf
Retrieved 2018-08-01.

[18] NVIDIA: Kepler architecture, https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf
Retrieved 2018-08-01.

[19] GPU-tech-conf, Inside Kepler, http://on-demand.gputechconf.com/gtc/2012/presentations/S0642-GTC2012-Inside-Kepler.pdf
Retrieved 2018-08-01.

[20] NVIDIA Developer, 5 Things you should know about the new Maxwell GPU architecture, https://devblogs.nvidia.com/5-things-you-should-know-about-new-maxwell-gpu-architecture/
Retrieved 2018-08-01.

[21] NVIDIA Developer, Inside Pascal: NVIDIA's newest computing platform, https://devblogs.nvidia.com/inside-pascal/
Retrieved 2018-08-01.

[22] NVIDIA Documentation, Tuning CUDA Applications for Pascal, https://docs.nvidia.com/cuda/pascal-tuning-guide/index.html Retrieved 2018-08-01.

[23] GPU-tech-conf, CUDA C/C++: Streams and Concurrency, http://on-demand.gputechconf.com/gtc-express/2011/presentations/StreamsAndConcurrencyWebinar.pdf Retrieved 2018-08-01.

[24] NVIDIA Documentation, Programming-guide, Asynchronous-concurrent-execution, https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#asynchronous-concurrent-execution Retrived 2018-08-01.

[25] NVIDA Documentation, Multi-process-service, https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf Retrived 2018-08-01.

[26] OpenCV documentation, https://docs.opencv.org/2.4/modules/core/doc/intro.html Retrived 2018-08-01.

[27] LodePNG, https://lodev.org/lodepng/ Retrieved 2018-08-01.

Appendix A: Documentation of own developed program code

Setup configuration

Setup_Configuration.cuh: Setup configuration for the algorithm, set variables for different components.
Important to note: standalone implementation requires more attributes to set in this header while the static library implementation allows the developer to change them in the Front-end and therefore expects instead certain values - like input images, image dimensions or which filter is to be calculated as well as how many paths for the semi-global matching should be used - as parameters during the function calls itself and not already in the configuration file, consequently the *Setup_Configuration.cuh* holds less changeable attributes in the static library, but its functions contain more parameters.

Attributes	
left_image	Path to left input image (rectified)
right_image	Path to the right input image (rectified)
im_height	Height of the input images
im_width	Width of the input images
MAX_DISPARITY	Pixel range for the search for corresponding pixels in the second image • default value: 125 • the larger the disparity the more accurate can pixels close to the camera lenses be determined
window_height	Height of the window in Census Transform • default value: 7
window_width	Width of the window in Census Transform • default value: 9
LEFT	Required for the navigation through the window in Census Transform for the comparison between neighboring pixels' intensities with center pixel's intensity • (window_width-1)/2
TOP	Required for the navigation through the window in Census Transform for the comparison between neighboring pixels' intensities with center pixel's intensity • (window_height-1)/2
SMALL_PENALTY	Small penalty added in Semi-global matching, disparities differ just by one • default value: 5 • right now no distinction between different

	paths
LARGE_PENALTY	Large penalty added in Semi-global matching, disparities differ by more then one • default value: 25 • right now no distinction between different paths
INTMAX	High initial value for finding actual minimum in semi-global matching through comparison between costs
MEDIAN_WINDOW_SIZE	Window size in one dimension, the full window size is the squared value • default value: 5 (25) • select values like: 3 (9),5 (25),7 (49),9 (81) • center pixel in window necessary
BILATERAL_WINDOW_SIZE	Window size in one dimension, the full window size is the squared value • default value: 5
BLUR_VALUE	Determines Gaussian blur value, the higher the value the greater the blur in the final image • default value: 50
calculate_paths	Determine how many paths/scanlines in semi-global matching are used: • possible values: 0,1,2,3,4 • 0: no semi-global matching is used • 1 (y-pos), 2 (y-pos, x-pos), 3 (y-pos, x-pos, x-neg), 4 (y-pos, x-pos, x-neg, y-neg)
calculateHammingDistance	Determine which local method is used to calculate disparity • possible values: 0,1 • 1: Census Transform + Hamming Distance • 0: SAD
calculate_Median	Calculate Median filter • 0: false, 1: true
calculate_Bilateral	Calculate Bilateral filter • 0: false, 1:true
maxThreads	Determine how many threads per block possible for the particular GPU, influences kernel launches for disparity computation • usually: 1024 (or 512)

Disparity estimation

Main file of the algorithm, contains in the executable the main-function and uses either LodePNG or OpenCV as the image library. In the static library, the file holds the function that calls the other components of the algorithm in order and with the parameters selected by the developer. But the two functions that are contained in both variations are described as follows:

Method Summary	
void	allocMemory() Allocate the necessary memory space on the host and the device.
void	freeMemory() Free the allocated memory space on the host and on the device.

Census Transform

Calculates the Census Transform image-pair needed for the calculation of the Hamming Distance.

Method Summary	
__global__ void	censusTransform(const uint8_t *imL, const uint8_t *imR, uint64_t *transformL, uint64_t *transformR) The two input images (imL and imR) are used to calculate the bit-strings. Shared memory is used and first filled with the values of the grayscale images as long as it stays within image bounds. Navigation through the image with LEFT and TOP and comparing neighboring intensities with center pixel's intensity. Based on this, *bit* is set either 0 or 1 and stored in corresponding variable *censusL* or *censusR* before they are stored in *transformL* and *transformR*.

Hamming Distance

Calculates the Hamming Distance with the results of the Census Transform and results in a 3D-cost structure (image width, image height, max. disparity range).

Method Summary	
__global__ void	calculateHammingCost(const uint64_t *d_transformLeft, const uint64_t *d_transformRight, uint8_t *cost) The two results (*transformLeft*, *transformRight*) from the Census Transform are used to calculate the Hamming Distance. This version uses shared memory and stores there values from the Census Transform bit-strings. The result is stored in *cost*, a 3D-cost structure.
__global__ void	calculateHammingCost_NoSharedMem(const uint64_t *d_transformLeft, const uint64_t *d_transformRight, uint8_t *cost) The two results (transformLeft, transformRight) from the Census Transform are used to calculate the Hamming Distance. This version doesn't use shared memory. The result is stored in *cost*, a 3D-cost structure.

SAD

Calculates sum of absolute differences (SAD) with the two input images (gray-scale, rectified) and results in a 3D-cost structure (image width, image height, max. disparity range).

Method Summary	
__global__ void	calculateSAD(const uint8_t *leftIm, const uint8_t *rightIm, uint8_t *costs) The two input images (*leftIm, rightIm*) are used to calculate SAD of every pixel in the first image and every pixel in the second image within the given disparity range and the results of this calculation are stored in *costs*, a 3D-structure. No shared memory is used.

Semi-global matching

Calculates the scanline(s) in one of the for directions (x-pos, x-neg, y-pos, y-neg) through the 3D-cost structure which is either calculated with the Hamming Distance or SAD. There is parallelization along the scanlines, in x-direction every row and for y-direction every column. Furthermore, each disparity value is also parallelized executed.

Method Summary	
__device__ uint8_t	findMin(uint8_t input1, uint8_t input2) Determines the minimum input parameter of the two and returns it.
__global__ void	dir_xpos(const int dirx, const uint8_t *d_cost, uint8_t *d_dir_cost) Calculates the scanline in x-pos direction through the 3D-cost structure. The parameter *dirx* is always 1, *d_cost* is the 3D-cost structure and *d_dir_cost* is the new cost calculated along the scanline.
__global__ void	dir_xneg(const int dirx, const uint8_t *d_cost, uint8_t *d_dir_cost) Calculates the scanline in x-neg direction through the 3D-cost structure. The parameter *dirx* is always -1, *d_cost* is the 3D-cost structure and *d_dir_cost* is the new cost calculated along the scanline.
__global__ void	dir_ypos(const int diry, const uint8_t *d_cost, uint8_t *d_dir_cost)

	Calculates the scanline in y-pos direction through the 3D-cost structure. The parameter *diry* is always 1, *d_cost* is the 3D-cost structure and *d_dir_cost* is the new cost calculated along the scanline.
__global__ void	dir_yneg(const int diry, const uint8_t *d_cost, uint8_t *d_dir_cost) Calculates the scanline in y-neg direction through the 3D-cost structure. The parameter *diry* is always -1, *d_cost* is the 3D-cost structure and *d_dir_cost* is the new cost calculated along the scanline.
__global__ void	dir_xpos_half(const int dirx, const uint8_t *d_cost, uint8_t *d_dir_cost) Only for test purposes. Calculates the scanline in x-pos direction through the 3D-cost structure and is parallelized for each image row as well as each disparity. The scanline itself is also bisected and both halves are calculated in parallel. Also *dirx* is always 1, *d_cost* is the 3D-cost structure and *d_dir_cost* is the new cost calculated along the scanline.
__global__ void	dir_xneg_half(const int dirx, const uint8_t *d_cost, uint8_t *d_dir_cost) Only for test purposes. Calculates the scanline in x-neg direction through the 3D-cost structure and is parallelized for each image row as well as each disparity. The scanline itself is bisected and both halves are calculated in parallel, also *dirx* is always -1, *d_cost* is the 3D-cost structure and *d_dir_cost* is the new cost calculated along the scanline.
__global__ void	dir_ypos_half(const int diry, const uint8_t *d_cost, uint8_t *d_dir_cost) Only for test purposes. Calculates the scanline in y-pos direction through the 3D-cost structure and is parallelized for each image column as well as each disparity. The scanline itself is bisected and both halves are calculated in parallel, also *diry* is always 1, *d_cost* is the 3D-cost structure and *d_dir_cost* is the new cost calculated along the scanline.
__global__ void	dir_yneg_half(const int diry, const uint8_t *d_cost, uint8_t *d_dir_cost) Only for test purposes. Calculates the scanline in y-neg direction through the 3D-cost structure and is parallelized for each image column as well as each disparity. The scanline itself is bisected and both halves are calculated in parallel, also *diry* is always -1, *d_cost* is the 3D-cost structure and *d_dir_cost* is the new cost calculated along the scanline.
__global__ void	dir_xpos_quart(const int dirx, const uint8_t *d_cost, uint8_t *d_dir_cost)

	Only for test purposes. Calculates the scanline in x-pos direction through the 3D-cost structure and is parallelized for each image row as well as each disparity. The scanline itself is quartered and each quarter is calculated in parallel, also *dirx* is always 1, *d_cost* is the 3D-cost structure and *d_dir_cost* is the new cost calculated along the scanline.
__global__ void	dir_xneg_quart(const int dirx, const uint8_t *d_cost, uint8_t *d_dir_cost) Only for test purposes. Calculates the scanline in x-neg direction through the 3D-cost structure and is parallelized for each image row as well as each disparity. The scanline itself is quartered and each quarter is calculated in parallel, also *dirx* is always -1, *d_cost* is the 3D-cost structure and *d_dir_cost* is the new cost calculated along the scanline.
__global__ void	dir_ypos_quart(const int diry, const uint8_t *d_cost, uint8_t *d_dir_cost) Only for test purposes. Calculates the scanline in y-pos direction through the 3D-cost structure and is parallelized for each image column as well as each disparity. The scanline itself is quartered and each quarter is calculated in parallel, also *diry* is always 1, d_cost is the 3D-cost structure and *d_dir_cost* is the new cost calculated along the scanline.
__global__ void	dir_yneg_quart(const int diry, const uint8_t *d_cost, uint8_t *d_dir_cost) Only for test purposes. Calculates the scanline in y-neg direction through the 3D-cost structure and is parallelized for each image column as well as each disparity. The scanline itself is quartered and each quarter is calculated in parallel, also *diry* is always -1, *d_cost* is the 3D-cost structure and *d_dir_cost* is the new cost calculated along the scanline.

Sum up costs

The 3D-cost structure calculated with either Hamming Distance or SAD and the result of the semi-global matching which also is a 3D-cost structure with the same dimensions have to be summed up.

Method Summary	
__global__ void	sum_upCosts(uint8_t *cost, uint8_t *d_dir_cost) The two cost structures are summed up and the result is stored in *cost*. Each value in *d_dir_cost* is added to the corresponding value in *cost*.

Compute disparity

Within the 3D-cost structure it is necessary to find the disparity with the highest likelihood, to determine the two corresponding pixels. Therefore, the find the minimum value is found with the winner-takes-all strategy.

Method Summary	
__global__ void	createDispImage(uint8_t *d_cost, uint8_t *disp_image, int min) The parameter *d_cost* is the 3D-cost structure with the dimensions (image width, image height, max. disparity range). For each pixel the minimum disparity value is found and stored in the *disp*_image that has the same dimension as the input images. The *min* stores a extremely high value in the beginning to ensure that the first disparity value chosen is the smaller one.

Median filter

Apply Median filter to the disparity image that is computed with the winner-takes-all strategy. The goal is to remove noise from the final disparity image.

Method Summary	
__global__ void	medianFilter(uint8_t *inputIm, uint8_t *outputIm) A window with a certain size is moved across the image (*inputIm*) and the values within the window are sorted in ascending order. Different sort algorithms possible. (New) center pixel is stored in the final disparity image (*outputIm*).

Bilateral filter

Apply Bilateral filter to the disparity image that is computed with the winner-takes-all strategy. The goal is to remove noise from the final disparity image.

Method Summary	
__global__ void	bilateralFilter(uint8_t *inputIm, uint8_t *outputIm) A window with a certain size is moved across the image (*inputIm*) and the value of the center pixel is calculated based on the intensitiy values of its neighbors and an additional Gaussian blur value is incorporated into the formula. The result is stored in the final disparity image (*outputIm).*