DEPARTMENT OF
COMPUTER SCIENCE

UNIVERSITÄT BAYREUTH

Bachelor's Thesis in Informatics

Efficient Large Scale Stereo Matching On GPU

Felix Herrmann

UNIVERSITÄT
BAYREUTH

DEPARTMENT OF
COMPUTER SCIENCE

UNIVERSITÄT BAYREUTH

Bachelor's Thesis in Informatics

Efficient Large Scale Stereo Matching On GPU

Author:	Felix Herrmann
Student ID Number:	1540860
Supervisor:	Prof. Dr. Michael Guthe
Advisor:	Johannes Jakob
Submission date:	13. January 2020

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Bayreuth, 13. January 2020

Abstract

The dense and accurate reconstruction of depth information from large-size images within real-time constraints is a computationally demanding challenge and crucial for many industrial applications like robotics and autonomous vehicles. This thesis presents a parallelized implementation of the efficient large-scale stereo matching algorithm ELAS proposed by Geiger et al. on the GPU. Based on an analysis of its underlying data dependencies a detailed description of the algorithm's parallelization using CUDA is provided. Experimental results are compared to the CPU implementation provided by the authors, demonstrating that significant speedup has been achieved while maintaining accuracy. For 750×900 pixel images the GPU implementation runs at 27 fps on a NVIDIA Geforce GTX 680M. The results show promising for the usage in many real-time applications that include obstacle avoidance like autonomous vehicles.

Contents

1 Introduction

1.1 Motivation

In recent history the capabilities of autonomous robot systems to sense and act in their environment have been significantly improved. Especially in presence of dynamic and uncertain environments these systems must be able to efficiently reconstruct their environment to enable intelligent decision making. In the field of computer vision, various methods have been evolved that attempt to model a complex visual environment [1].

Stereo vision has become one of the most active research areas in computer vision [2]. The goal of stereo vision is to extract depth information from a pair of images captured by two cameras that are set slightly apart. The main problem is to find the pixel correspondences within the images that belong to the projection of the same 3D point of the captured scene. The resulting shift of the projection of that point within the images is referred to as disparity [3]. The taxonomy mainly divides stereo vision algorithms into two groups. First, there are global methods that produce some of the best qualitative results but require large computational efforts. Second, there are local methods that profit from their simplicity and are typically much faster then global methods. Generally, a compromise between speed and accuracy has to be found [4].

Computing the disparity value for each pixel within the image, i.e. the disparity map, allows to reconstruct the distances of the captured objects using triangulation. Therefore, the disparity map contains valuable information for many applications like grasping of objects in robotics or object avoidance and detection for autonomous vehicles [5]. Many of these applications require the disparity map computation to be close to real-time [6]. Additionally high resolution images are needed to achieve a good accuracy for the estimated depth. Moreover, stereo vision is a computationally demanding procedure and together with these requirements it becomes an even more challenging task to produce a result within real-time constraints. Even though local methods are preferred by real-time applications, implementations solely residing on the central processing unit (CPU) can not satisfy the real-time requirements for large size images [7, 8]. To meet the above mentioned real-time requirements, the computation is usually performed on powerful computers which are not suitable for implementation in autonomous mobile robot systems [9].

With the introduction of the CUDA Architecture – which enables General-Purpose programming on NVIDIA enabled graphics cards – implementing algorithms on the graphics processing unit (GPU) became a much easier

task and allows a more flexible usage of the GPU [4]. Moreover, recent developments in GPU hardware allow for a more affordable and efficient implementation in embedded systems which make it suitable for above mentioned cases. The highly parallel computing capabilities of modern GPUs optimized to efficiently operate on image related data are well suited for many computer vision algorithms [10]. Many of the involved tasks for stereo matching using local methods, e. g., the cost computing or cost minimization phase, can be computed independently. Therefore, a GPU implementation of a local stereo matching algorithm seems promising to benefit from parallel execution.

1.2 Related Work

Local methods typically only consider pixels within a finite window around the referenced pixels to compute the matching costs. For each pixel the disparity is picked that minimizes the resulting costs between the windows of the corresponding pixels under some distance metric [11]. In 2009, a comparison between a straight forward CPU and GPU implementation of a local method using census transform to compute the matching costs was given by Weber et al. [10]. On a 450×375px resolution image and computing 60 disparity levels they reached around 0.01 frames per second (fps) on an Intel Core2Duo 2GHz CPU and 0.1 fps on a NVIDIA GeForce GTX 208 GPU. With the GPU implementation being around ten times faster than the CPU implementation, the results indicate that local methods can profit from parallel execution. More recently, frame rates around 55 fps have been achieved on a NVIDIA Jetson TX1 using the simpler absolute difference method on 900×750px resolution images [9]. Although GPU implementations of traditional local based methods achieve significant run-time improvements, their main disadvantage is their high percentage of mismatched pixels. The low matching ratios especially in low-textured areas of the image result in noisy disparity maps that are not suited for the high accuracy requirements of autonomous vehicles and other applications [12].

A popular approach that aims to overcome the above-mentioned problems is Semi-Global Matching (SGM) that propagates consistency constraints along several paths across the image [7]. Zuh et al. provide a CPU and GPU implementation of SGM using eight path directions in 2012. For an image resolution of 384×228px and 64 disparity levels the GPU implementation reaches 13 fps on a NVIDIA GeForce GTX 295 being around four times faster than their CPU implementation run on a Intel Core2 Q9450. In a recent work, Hernandez-Juarez et al. propose a SGM implementation on the GPU using eight path directions and calculating 128 disparity levels. They state that their algorithm achieves 19 fps for 640×480px images with good qualitative results on a NVIDIA Tegra X1 embedded GPU system [13].

1.3 Purpose and Procedure

This thesis proposes an implementation of a stereo matching algorithm on GPU using CUDA that computes accurate disparity maps for high resolution images achieving high frame rates through parallel execution without the need for global optimization. The algorithm is based on the *'Efficient-LArge-Scale-Stereo'* matching algorithm *ELAS* proposed by Geiger et al. [14]. Their approach builds a prior probability distribution over the pixels using a set of support points that can be robustly matched, that allows for an efficient exploitation of the disparity search space and meanwhile reduces matching ambiguities of the remaining pixels.

The remainder of this thesis is structured as follows. In Section 2 the reader is introduced to basic concepts underlying both stereo matching as well as the CUDA programming model. A description of *ELAS* and its main components is given in section 3. Section 4 investigates the suitability of ELAS for parallelization by analysing data dependencies and the computational load balance of the main components. Section 5 presents the parallel implementation of *ELAS* on GPU. An evaluation of the experimental results comparing the CPU with the GPU implementation in terms of run-time and quality of the produced disparity maps is given in section 6. Finally, section 7 concludes and and anticipates possible future research.

2 Theoretical Background

2.1 Stereo Matching

The stereo matching problem is motivated from the observation that humans, and other two-eyed species, are able to perceive depth through a process named Stereopsis. As human eyes are set slightly apart, the derived visual information results in slightly different images perceived by each eye. Importantly, objects that are close to the viewer result in a bigger shift between the resulting images than objects that are far away [15].

The same observation applies to a pair of images, taken with two cameras that are set slightly horizontally apart. The resulting images of such a camera composition are referred to as stereo images. Stereo matching, also known as stereo disparity describes the process of finding the matching pairs of pixels, i.e. pixel correspondences, that are the projection of the same 3D point in the captured scene [11]. However, without any knowledge of the camera system calibration, a pixel in the left image could possibly correspond to any other pixel in the right image, resulting in a search space over the whole image. Section 2.1.1 describes how the search space for the correspondence of a pixel can be reduced using Epipolar geometry and image rectification. Then section 2.1.2 introduces the basics concepts of stereo matching algorithms. Finally, section 2.1.3 describes a straight forward implementation of a local method.

2.1.1 Epipolar Geometry and Image Rectification

In a camera system where both optical axes are perfectly horizontally aligned the pixel correspondences are restricted to be within the same row in the image. However, in practice such a perfect arrangement is not always the case. Knowing the orientations and positions of both cameras, the *epipolar geometry* can be used to describe the relation between the 3D scene and its 2D projection, allowing to restrict the search space of a pixel and to compute its depth [16]. In the following the basic concepts of epipolar geometry are described.

Given two images I_L and I_R every pixel X_L in the left image defines a line going through X_L and the center of projection of the left image C_L. The projection of that line to the right image plane is referred to as the *epipolar line*. Therefore, only pixels along the epipolar line need to be considered for the matching of X_L [17]. Figure 1 depicts the main elements of the epipolar geometry.

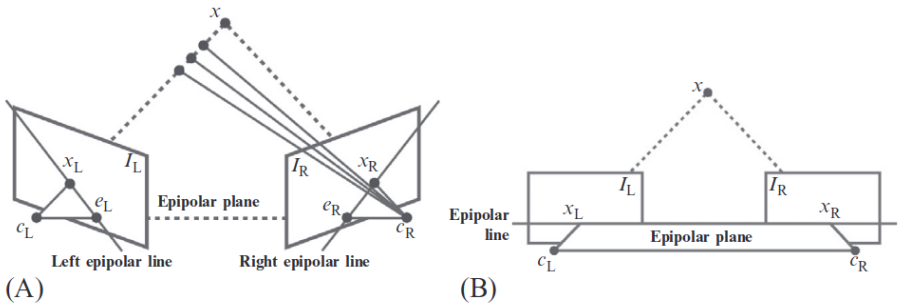

Figure 1: Main elements of the epipolar geometry. (A) Epipolar geometry. (B) Epipolar geometry rectified.

A stereo image pair is considered as *rectified*, if all epipolar lines are horizontal and aligned to the same row as the observed pixel as shown in (B). The process of rectifying images is not trivial and beyond the scope of this thesis. [1] From this point on all stereo image pairs are assumed to be rectified.

Assuming rectified images as the input for the matching process simplifies the correspondence problem for a pixel to search only within the same row in the other image. Moreover, for a pixel $X_L = (u_l, v_l)$ in the left image the corresponding pixel in the right image $X_R = (u_r, v_l)$ will always appear to be shifted to the left. Therefore, the matching problem reduces to the problem of finding a value d such that

$$(u_l, v_l) = (u_r - d, v_l).$$

In other words, the value d suffices to describe the correspondence between two pixels and is referred to as the *disparity* of X_L. Hence, the disparity is a measure for the shift between two corresponding pixels where small disparity values indicate that an object is further away than larger disparity values. The formula that describes the exact relationship between the distance of an object and its disparity is

$$D = \frac{B \cdot f_p}{d},$$

where D describes the distance between the center of projection and the object, B the baseline of the stereo cameras and f_p the focal length in pixels [5]. Figure 11 in the appendix A provides a reference exemplifying the stereo camera setup and the used variables. Stereo matching algorithms typically concentrate only on computing the disparity values for each pixel, as its computation is independent of specific camera calibrations.

[1] There is extensive literature on image rectification, for more information the interested reader is referred to [16, 18, 19]

2.1.2 Stereo Matching Algorithms

The challenge of a stereo matching algorithm is to find the pixel correspondences within the input images I_L and I_R that relate to the same object in the scene. Without loss of generality, in the following the left image is considered as reference image.

The aim is to produce a *disparity map* that stores a disparity estimate for each pixel position of I_L. Therefore, the disparity map can be considered as an image of the same dimension as I_L holding at each pixel position the disparity of the corresponding pixel in the right image. Figure 2 shows how a disparity map can be visualized by interpreting the disparity value as a pixel color.

Figure 2: Ground truth disparity map for the *Teddy* stereo image from the Middlebury dataset. Red is close, yellow and green is in the middle, blue is far [11].

Every stereo matching algorithm has to make some assumptions about the physical world that the matching process is based on. For example, most algorithms assume that objects look the same when viewing them from two slightly different angles [11]. Typically, the term *matching cost* is used to describe the similarity of two pixels. A lower matching cost indicates a higher probability for the pixels to be a match. Early stereo matching algorithms are based on the idea to find correspondences by searching for the pixel within the same row of the other image that has the closest intensity value compared to the intensity of the reference pixel [5]. A common way to compute the costs of a pixel $X_L = (u, v)$ with disparity d is to take the *absolute difference* of their intensity values

$$C(X_L, d) = |I_l(u, v) - I_r(u, v - d)|$$

where I_l and I_r return the intensities of a pixel in the left and right image respectively. The function C is referred to as the *cost function* or *energy function* and returns the matching costs for a given pixel and disparity. Hence, the disparity computation reduces to minimizing the energy function

for each pixel in the reference image. However, the correspondences are very ambiguous when only considering the absolute difference between two pixels as there is a high chance that there are more pixels with the same intensity values. Therefore, many different cost functions and methods have been proposed and evaluated that attempt to compute more accurate disparity maps [20].

Most of these methods can be assigned to one of the two main classes of stereo matching algorithms. On the one hand, there are *local (window based) methods* that include the pixels of a finite window around each pixel for finding the pixel correspondence. This approach minimizes the probability for ambiguous matches, as more features that describe the uniqueness of a pixel are taken into consideration. However, choosing an adequate size for the window is not trivial. Smaller windows produce low matching ratios within texture-less areas of the image and larger windows produce poor results at the border of objects [14]. On the other hand, there are *global methods* that attempt to minimize a global cost function while explicitly making smoothness assumptions, for example by applying penalties for disparity discontinuities [21]. Global methods typically perform much better on the aforementioned problems. However, minimizing such a global cost function is computationally very expensive [14]. Since ELAS is a local method, global methods are not further investigated here. In the following section 2.1.3 a standard local approach to compute a disparity map is described.

2.1.3 A Local Method: SAD

This section describes a straight forward implementation of a local method using the *Sum-of-Absolute-Differences* (SAD) as matching cost function. The matching cost between two pixels using the SAD is computed by summing the absolute intensity differences of all pixels within a finite window of the size $(2n+1) \times (2m+1)$:

$$C(X_L, d) = \sum_{i=n}^{-n} \sum_{j=m}^{-m} |I_l(u+i, v+j) - I_r(u+i, v-d+j)|.$$

The algorithm performs the disparity computation as follows. For each pixel X_L in the left image the SAD for all valid pixels in the right image is computed. As mentioned earlier, only pixels that are within the same row and left to X_L in the right image have to be considered. Moreover, pixels are skipped if the SAD window around it would overlap the borders of the image. The disparity that minimizes the energy function is assigned to the observed pixel. After all disparities are computed the final disparity map is received.

2.2 CUDA Concepts

This section provides an introduction to the basic concepts of GPU programming using CUDA, a programming model that enables general purpose programming for NVIDIA graphics cards. The CUDA programming model allows to program applications for the GPU using C++ code with a few extensions [22].

In CUDA the computation units are divided into the host, which is normally a CPU and the device, typically a GPU. The code that is run on the device is specified in a CUDA kernel. A kernel is essentially a function that is executed by a large number of threads to exploit the data parallelism. Kernels are launched by the host that specifies the number of threads that execute a kernel. Threads are grouped into blocks which are again organized in a grid. Within a kernel, threads can be accessed using 1D, 2D or 3D indexing. This allows the threads in a kernel to do operations based on their index. CUDA enabled GPUs are build up of tens of multithreaded processing units called streaming multiprocessors (SMs) each consisting out of 32 arithmetic logic unis (ALUs). Within a SM all ALUs are executed in Single-Instruction, Multiple-Data (SIMD) fashion. Due to this architecture, a warp consisting of 32 threads is scheduled to one SM and delivers best performance as all cores are running. Moreover, it is important to avoid different execution paths within the same warp, otherwise the different execution paths have to be executed sequentially. Threads that take much longer than the remaining threads within the same warp lead to a poor utilization of the available parallel resources. Additionally, the allocated resources by a block, e.g. shared memory, are only released if all threads have finished their computations and could delay the scheduling of following blocks. Therefore, a kernels performance highly depends on an equal distribution of the computational work amongst the threads [22, 23].

Furthermore, the management of memory accesses within a kernel plays an important role in CUDA programming for a program's performance. CUDA enabled devices provide several separate memory spaces with different characteristics. All threads have access to the same global memory. It is the biggest memory and is used for data transfer between host and device. Shared memory is a scarce resource that that can only be shared among threads within the same block. Memory fetches from shared memory are about 100 times faster then reading data from global memory. If threads within the same block access overlapping data locations in global memory a common strategy is to load the needed data from global memory to shared memory, perform the calculations and then write the results back to global memory [4]. When developing an application using CUDA, the above mentioned-concepts should be kept in mind as its performance highly depends on them.

3 Description of ELAS

The GPU implementation presented in this thesis is based on the stereo matching algorithm *ELAS (Efficient LArge-scale Stereo)*[2] that was proposed by Geiger et al. in 2010 [14]. An introduction to the key concepts of the algorithm is given in section 3.1. The main building blocks of the algorithm are described in more detail in section 3.2.

3.1 Efficient Large-Scale Stereo

This section provides an overview of the stereo matching algorithm *ELAS*. Geiger et al. describe a local method that operates on two rectified frames and produces a dense disparity map, i.e. a disparity estimate at each pixel. As described in section 2.1, local methods for stereo matching are usually much more efficient then global methods, but suffer from border bleeding artifacts and often show lower matching ratios. These problems arise due to highly ambiguous correspondences for instance in low-textured areas of the image. However an important observation by the authors is that despite these ambiguous correspondences, some pixels can be robustly matched which they refer to as *support points*. Assuming that the world's or scene's geometry is made up of piecewise-smooth surfaces, these support points contain valuable information about the disparity values of their surrounding that can be used to disambiguate the remaining pixels. For example, knowing the disparity value of one particular pixel, one can assume that the surrounding pixels show similar disparity values. This information can then be used to determine which disparity to chose, by favouring disparities that surrounding support points show.

Geiger et al. model this property by forming a Delauny triangulation over the coordinates of a sparse set of support points. This yields a 2D mesh of triangles that is used to compute a prior probability distribution over the remaining pixels. The prior allows to approximate the disparity values for the remaining pixels, hereby significantly reducing the disparity search space and the possibility for ambiguous matches. This is done by interpolating the disparity values associated to the corners of its surrounding triangle at the position of the currently observed pixel. Then only a small search range around the approximated disparity is considered in the matching process. However, this approach is error-prone in areas of the image with higher disparity discontinuities like at the edge of objects. To handle those cases, the search range is extended with a set of support point disparities in a small 20×20 neighborhood around the observed pixel. These properties are incorporated in an energy function that is described in the following section.

[2]The C++ source code of Geiger et al. is available at `http://www.cvlibs.net`

3.1.1 Energy Function

The energy function that is used for the matching process of the remaining pixels is established using a prior probability distribution of the disparity space and an image likelihood, describing the similarity between two observations. The authors use Bayes theorem to compute the posterior probability distribution of the disparity space. Taking the negative logarithm of the posterior yields the energy function. Before further describing how the energy function is derived some formalities are introduced.

The authors define "$\mathbf{S} = \{\mathbf{s}_1, ..., \mathbf{s}_M\}$ *to* be a set of robustly matched support points. Each support point, $s_m = (u_m, v_m, d_m)^T$, is defined as the concatenation of its image coordinates, $(u_m, v_m) \in N^2$, and its disparity, $d_m \in N$. Let $\mathbf{O} = \{\mathbf{o}_1, ..., \mathbf{o}_N\}$ be a set of image observations, with each observation $\mathbf{o}_n = (u_n, v_n, \mathbf{f}_n)^T$ formed as the concatenation of its image coordinates, $(u_n, v_n) \in N^2$, and a feature vector, $\mathbf{f}_n \in \mathbb{R}^{\mathbb{Q}}$, e.g., the pixel's intensity or a low-dimensional descriptor computed from a small neighborhood. We denote $\mathbf{o}_n^{(l)}$ and $\mathbf{o}_n^{(r)}$ as the observations in the left and right image respectively. Without loss of generality, in the following we consider the left image as the reference image" (see Geiger et al. page 5).

Furthermore let $p(d_n|\mathbf{S}, \mathbf{o}_n^{(l)})$ be the prior that describes the probability for an observation $\mathbf{o}_n^{(l)}$ to correspond to an observation with disparity d_n. The authors state that the prior is proportional to a combination of a uniform distribution and a sampled Gaussian

$$p(d_n|\mathbf{S}, \mathbf{o}_n^{(l)}) \propto \begin{cases} \gamma + \exp\left(-\frac{\left(d_n - \mu(\mathbf{S}, \mathbf{o}_n^{(l)})\right)^2}{2\sigma^2}\right) & \text{if } |d_n - \mu| < 3\sigma \vee d_n \in N_{\mathbf{S}} \\ 0 & \text{otherwise} \end{cases}. \tag{1}$$

The function $\mu(\mathbf{S}, \mathbf{o}_n^{(l)})$ returns the mean disparity value for an observation $\mathbf{o}_n^{(l)}$ based the disparity values that surrounding support points show. This is done by interpolating the disparity values of the enclosing triangle of $\mathbf{o}_n^{(l)}$. Hence, the probability decreases with respect to the distance to the mean disparity. To reduce the disparity search space, all disparities that are further than 3σ away from the mean value are excluded. Additionally, $d_n \in N_{\mathbf{S}}$ allows to locally expand the search range to the disparity values exhibited by a 20×20 pixel neighborhood of support points.

The likelihood function is expressed as a constrained Laplace distribution

$$p(\mathbf{o}_n^{(r)}|\mathbf{o}_n^{(l)}, d_n) \propto \begin{cases} \exp\left(-\beta||\mathbf{f}_n^{(l)} - \mathbf{f}_n^{(r)}||_1\right) & \text{if } \begin{pmatrix} u_n^{(l)} \\ v_n^{(l)} \end{pmatrix} = \begin{pmatrix} u_n^{(r)} + d_n \\ v_n^{(r)} \end{pmatrix} \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

with $\mathbf{f}_n^{(l)}$, $\mathbf{f}_n^{(r)}$ the feature vectors of the left and right observation respectively. Due to the shape of the Laplace distribution correspondences with similar feature vectors are strongly favoured. As the algorithm expects rectified images, only observations that are located on the same horizontal line are considered. This restriction is expressed through the if-condition.

Geiger et al. construct an energy function for an observation $\mathbf{o}_n^{(l)}$ from equation (1) and (2) by taking the negative logarithm of their product

$$E(d) = \beta||\mathbf{f}^{(l)} - \mathbf{f}^{(r)}(d)||_1 - \log\left[\gamma + \exp\left(-\frac{\left(d_n - \mu(\mathbf{S}, \mathbf{o}_n^{(l)})\right)^2}{2\sigma^2}\right)\right] \tag{3}$$

where $\mathbf{f}^{(r)}(d)$ denotes the feature vector that is located at pixel $(u^{(l)} - d, v^{(l)})$. Again, note that due to the characteristics of Eq. (1) and Eq. (2) only disparities d need to be considered with $|d - \mu| < 3\sigma$ or if $d \in N_{\mathbf{S}}$.

3.2 Algorithmic Building Blocks

In the previous section the key concepts of the algorithm were introduced. The algorithm is based on the main observation that some pixels in an image can be robustly matched. Ambiguities for the remaining pixels can be reduced by limiting the disparity search range based on the disparity values of neighboring support points. These properties are expressed in a compact form through the energy function. The disparity maps are obtained by minimizing the energy function for every pixel. In practice, the authors additionally apply some refinement steps on the disparity maps including a left/right consistency check, removing small segments and interpolating small disparity gaps. Those are commonly used refinement steps that are used in many other stereo matching algorithms [11]. Since the focus of this thesis is to implement the core features described in their paper, only the left/right consistency is presented. Therefore, the remaining refinement steps are not further investigated in this thesis. In the following the algorithm is described in more detail. The algorithm can be decomposed into six building blocks. The description of the building blocks proceeds as follows:

First, section 3.2.1 describes the computation of a descriptor. The descriptor aggregates image features for every pixel in a descriptor vector that describes a pixels characteristics through its texture. The descriptor vector is used to evaluate the similarity between two pixels. Section 3.2.2 shows how

the computation of robust support points is realized. The authors establish different criteria that support points have to satisfy. Those will be described throughout the section. To evaluate the energy function for a pixel, the disparity values of support points of a 20×20 pixel neighborhood have to be accessed. Section 3.2.3 describes the computation of the disparity grid, a data structure that allows to access these disparities efficiently. Additionally, a triangulation has to be formed on the set of support points that is briefly described in section 3.2.4. Section 3.2.5 shows how the previously computed data structures are efficiently used to compute the disparities of the remaining pixels. Finally, section 3.2.6 describes the left/right consistency check that produces the final disparity map.

3.2.1 Descriptor

Similar to the straight forward implementation of a stereo matching algorithm presented in section 2.1 ELAS used the SAD to compute the matching costs between two pixels. However, the image features that are used to compute the similarity between two pixels are not the raw intensity values of the pixels within a finite window. Instead, the authors apply an edge detection on the images using the horizontal and vertical Sobel filter and use the resulting filtered images for the cost computation. More precisely, a descriptor vector is computed for every pixel that gathers features from the filtered images. The descriptor vector for a pixel is the concatenation of 16 features taken from a 5×5 pixel window around the corresponding pixels in the filtered images. The matching cost between two pixels is computed by taking the SAD of their respective descriptor vectors.

3.2.2 Support Points

This section focuses on the computation of a sparse set of support points that can be robustly matched. The authors establish different criteria that candidates, i.e. pixels that could possibly serve as support points, have to meet in order to be accepted as support points. Choosing the right set of candidates is important for the efficiency and accuracy of the algorithm. If the initial set of candidates is too large it could become a limiting factor of the algorithms performance since the verification of the support point criteria is computationally expensive as shown later on in section 4. On the other hand, if the set is too small the prior will lose accuracy which reduces the quality of the produced disparity map. The authors find that choosing the pixels from a 5×5 pixel grid as the initial set of candidates is an adequate choice. Hereby, the size of candidates is strongly reduced while covering the whole image. The following steps are executed for each candidate to impose robustness:

First it is checked whether each candidate shows enough texture. Pixels

with low texture are prone to be mismatched and are therefore discarded. Then the matching costs for the whole valid disparity range are computed. To obtain more precise results the matching cost between two pixels is computed by summing the SAD between the descriptor vectors of four neighboring pixels of the observed pixels. During the matching cost computation the disparity and matching cost of the best and second best match are stored. Candidates whose ratio between the matching costs of the best and the second best match exceed a fixed threshold are eliminated as a high ratio indicates that the correspondence of the observed candidate is ambiguous. Moreover, only correspondences that can be matched from left-to-right and right-to-left are retained. That is, if the previously obtained correspondence of a candidate in the right image matches to a pixel in the left image that is more then two pixels away from the candidates position it is removed. Additionally, inconsistent matches are removed by deleting candidates that show dissimilar disparity values from surrounding candidates. To reduce the number of support points and hence speed up the triangulation process, redundant candidates that have similar disparity values along their five pixel horizontal and vertical surrounding are removed as well. After that last step is finished the final set of support points is retrieved.

3.2.3 Disparity Grid

To evaluate the energy function for the remaining pixels, the disparity values exhibited by support points within a 20×20 neighborhood have to be accessed for each pixel. They are used to locally extend a pixels disparity search space to handle areas of the image with disparity inconsistencies. Since it would be quite computationally expensive to do that for each pixel, an approximation is calculated using the so called *disparity grid*. The disparity grid is a data structure that allows to efficiently read an approximation of the disparity values exhibited by support points within a 20×20 neighborhood of a given pixel. Thereby, the image is divided into 20×20 pixel tiles and for each tile the disparity values of its enclosing support points are stored. Furthermore, for each tile the disparity values of its surrounding tiles are added to the own set of disparity values. The disparities of a pixels neighborhood can be retrieved by accessing its corresponding tile within the disparity grid based on the pixels position.

3.2.4 Triangulation

For a given set of points P a triangulation divides the plane into a set of triangles $T(P)$, where each triangle side is entirely shared by two adjacent triangles. Moreover, a Delauny triangulation $DT(P)$ guarantees that no point in P is inside the circumcircle of any triangle in $DT(P)$. As a result of that, narrow triangles are avoided [24]. These are useful properties to

interpolate values that are associated with the corners of a triangle. By forming a Delauny triangulation over the set of support points the disparity of an observation $\mathbf{o}_n$ can be approximated by interpolating the disparities of the support points associated with the surrounding triangle of $\mathbf{o}_n$.

3.2.5 Disparity Computation

This section describes the computation of the disparity values of the remaining pixels. This is done by iterating over every triangle and evaluating the energy function for all valid disparities of the inner pixels and picking the disparity that minimizes the energy function. In section 3.1 it was shown that for an observation $o_n^{(l)}$ only disparities d need to be considered that are within a range of 3σ around the mean value $\mu(S, o_n^{(l)})$. Additionally, the disparity search range is expanded for disparities exhibited by support points within a 20×20 pixel neighborhood. To obtain the mean value the authors express μ as a piecewise-linear function that is defined for every triangle t_i as its describing plane in Hesse normal form:

$$\mu_i(o_n^{(l)}) = a_i u_n + b_i v_n + c_i.$$

Plugging in the coordinates (u_n, v_n) of an observation $o_n^{(l)}$ inside of triangle t_i yields the mean disparity of the observation.

Based on the mean value the disparity search range can be easily computed. For every disparity d within the search range $|d_n - \mu| < 3\sigma$ or that is shown by support points of the observations neighborhood $d \in N_S$ the energy function is evaluated. The disparity that minimizes the energy function is assigned in winner-takes-it-all fashion to the current observation. Figure 3 depicts the disparity interpolation of an observation $o_n^{(l)}$ and its 3σ search range.

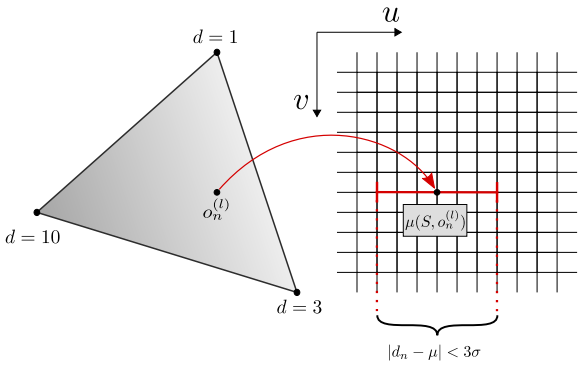

Figure 3: Interpolation of the disparity of an observation $o_n^{(l)}$ and its 3σ search range.

3.2.6 Left/Right Consistency Check

In the final step a left/right consistency check is applied to remove spurious mismatches and disparities in occluded areas of the image. For the disparities of all pixels in the left image it is checked if the difference to the corresponding disparity in the right image is higher then a fixed threshold. In that case the disparity in the left image is removed. Analogously, the same check is applied for all pixels in the right image. After the left/right consistency check is applied the final disparity maps are received.

4 Analysis of ELAS

In the previous section a detailed description of ELAS was given. This section provides an analysis of the CPU implementation provided by the authors in terms of its computational load distribution and its underlying data dependencies to serve as a reference and guidance for the parallel implementation described in section 5. First section 4.1 gives an overview of the run-time of the different components to identify the components that would benefit from parallel execution the most. Then section 4.2 analyses the data dependencies of the algorithm and works out the computations that can be parallelized. Furthermore, it is emphasized where previous parallel computations have to be synchronized due to dependencies between operations.

4.1 Computational Load Distribution

To get an oversight on the performance of the different components, the implementation provided by the authors was run several times on an Intel i7-3720QM 2.60GHz CPU measuring each components runtime. As input the *Cones* stereo image pair with a resolution of 900×750 pixel from the Middlebury dataset was used [11]. Figure 4 shows the mean runtime of each components averaged over ten executions.

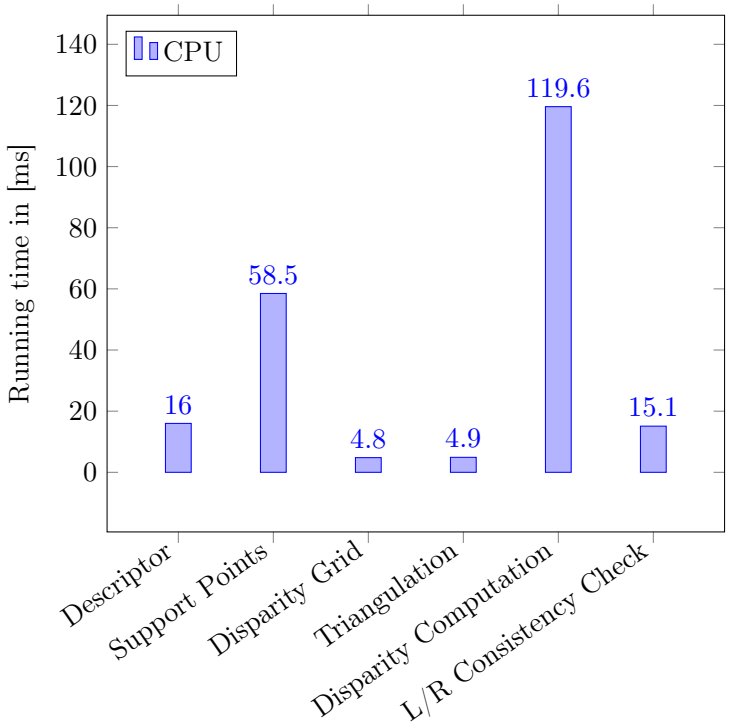

Figure 4: Run-time of the different components of ELAS averaged over ten executions

The total mean runtime for the *Cones* image is 218.9ms. It is noticeable that the disparity computation takes with 119.6ms over half of the total execution time. Moreover, computing the support points makes up more then 25% of it with an average of 58.5ms. Parallelizing these components is therefore particularly advisable as it could result in considerable run-time improvements. Although the other components make up less then 20% of the total run-time, improving each is still desirable as the goal is to minimize the overall run-time.

4.2 Data Dependencies

This section provides an analysis of the data dependencies that underlie this algorithm. An algorithm can not run faster as its longest chain of dependent computations, since computations that depend on prior computations have to be executed sequentially. However, the algorithm includes many computations that can be executed independently. It is the aim of this section to find the parts of the algorithm where no data dependencies apply and computations can be executed in parallel. Furthermore, this sections works out the locations in the algorithm where previous parallel computations are forced to be synchronized due to dependencies between operations. This section proceeds as follows: First the dependencies between the main components of the algorithm are discussed. Then the dependencies of the calculations within each main component are examined.

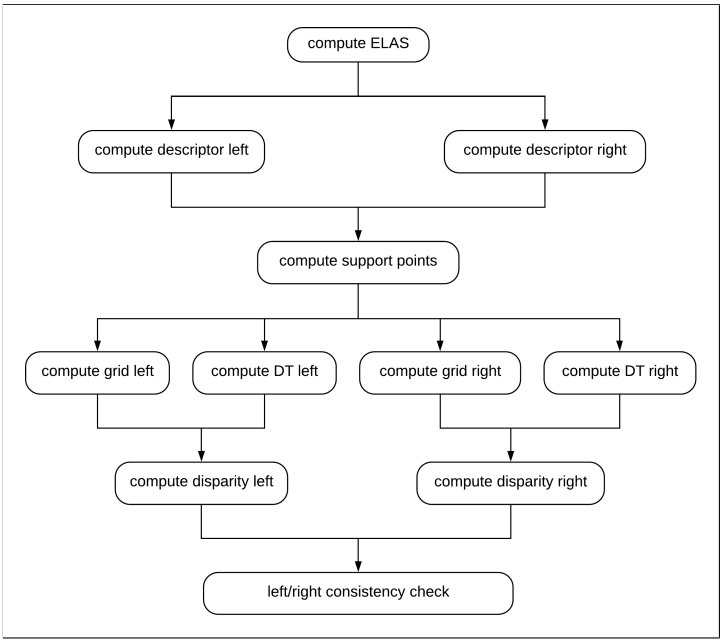

Figure 5: Dependencies between the main computations of ELAS.

Figure 5 illustrates the dependencies between the main computations of ELAS. Computations that are on the same level can be executed simultaneously. The computation of the descriptor for the left and right image can be executed concurrently, since their calculation only depends on the left and right image respectively. The computation of the support points depends on the data produced during the descriptor computation and therefore has to wait for it to finish. The disparity grid and the Delauny triangulation are computed afterwards, as both use the support points for their computations. However, both computations can be executed in parallel, since they do not have any data dependencies during their computations. Moreover, both computations can be executed independently with respect to the left and right image. The disparity computation of the remaining pixels uses the disparity grid and the triangles produced by the Delauny triangulation. Therefore, it has to wait for them to finish. After both disparity maps are computed, the left-right consistency check can be executed in parallel for the left and right disparity map.

In the following the dependencies of the computations within the main components are investigated. Large computations can often be subdivided into smaller computations, that can be solved at the same time. For image related computations an operation that is applied on the whole image often breaks down into applying the same operation on all pixels or the pixel-related data independently [8]. Operations like these are good candidates to profit from parallel execution. However, in other cases there might be dependencies between the operations that forbid independent execution. In the following the data dependencies between the computation within each building block are described and computations that can be parallelized will be worked out. Note that the Delauny triangulation will not be investigated in this section as this would exceed the scope of this thesis.

Descriptor. The descriptor computation breaks down into two main operations. First the horizontal and vertical Sobel filter is applied, then features from the resulting filtered images are gathered to form the descriptor. Applying the Sobel filter to an image is one of the above mentioned cases, where the computation can be divided into applying the Sobel mask on each pixel. The results of the operation are written to the corresponding location in the filtered image, therefore the operation can be executed on each pixel independently. During the gathering process the values of the filtered images are used. Therefore, all parallel computations of the Sobel filter have to be synchronized before computing the descriptor.

Support Points. The candidates from a 5×5 grid that satisfy the robustness criteria described in section 3.2.2 make up the set of support points.

There are several computations executed on each candidate to validate if they satisfy these criteria. All of these computations built up on each other as they might change the validity of a candidate that changes the behavior of preceding computations. Therefore, these computations have to be executed in order. The following will further investigate these computations:

First it is checked if a candidate has enough texture which can be done for each candidate independently. Then the disparities from left-to-right have to be computed for each pixel. Hence, the matching costs for all valid disparities have to be computed. The disparity computation can be done for each candidate and each disparity simultaneously. After that the consistency check is applied. Similarly, all matching costs for all valid disparities from right-to-left have to be computed. Again this can be done simultaneously for each candidate and each disparity. Then inconsistent candidates are removed that exhibit dissimilar disparity values from their surrounding candidates. The authors implement this by iterating over every candidate and flag a candidate as deleted if the number of candidates with similar disparities in their neighborhood is under a certain threshold. Since they write the results back to the same memory location this changes the neighborhood for other candidates and therefore following computations depend on previous ones. Thus, the nature of their implementation requires this computation to be done sequentially. Alternative implementations of this operation should be considered that produce similar results but can be computed concurrently to make use of the parallel resources. After that, redundant candidates are removed. Candidates are considered as redundant if they show similar disparities compared to its horizontal and vertical neighborhood. Geiger et al. apply this operation line by line iterating over every candidate from left-to-right. Again, if a candidate is removed it will change the neighborhood for following candidates. However computations of different rows are independent which allows to compute them concurrently. Analogously, the vertical redundancy check can be executed independently for every column.

Disparity Grid. To compute the disparity grid two main computations have to be carried out. First, the disparities for each support point have to be loaded to the corresponding position in the disparity grid. This can be done for each support point independently. Then for each grid position the disparity values of surrounding grid positions are aggregated. The aggregated set of disparities is added to the corresponding position in a copy of the grid. Therefore, no data dependencies apply allowing to parallelize this operation for each grid position.

Disparity Computation. The disparity computation of the remaining pixels is done by minimizing the energy function of Eq. 3. To evaluate the

energy function the mean value at that pixel position has to be computed. The disparities of the support points of a pixels neighborhood are provided in the disparity grid. To compute the mean disparity the plane parameters of the triangle that corresponds to the observed pixel have to be computed. The plane parameters can be derived independently for every triangle by solving a linear system. Once that is achieved the energy function can be evaluated for every pixel and every disparity independently [14].

L/R Consistency Check. The left/right consistency check removes spurious mismatches and disparities in occluded areas of the image. The check can be applied for every pixel in the left and right image independently.

5 Implementation

The previous section has provided an analysis of the data dependencies that underlie the algorithm. This section presents details to the parallel implementation of ELAS using CUDA. First, section 5.1 describes the used variables and data structures. Throughout the implementation common implementation strategies were used that are covered in section 5.2. Finally, section 5.3 outlines the highlights of the parallel implementation of ELAS.

5.1 Variables and Data Structures

This section lists and describes important variables and data structures that are used throughout the implementation. The image width and height is referred to with the variables *width* and *height* respectively. There are a variety of parameters that can be set that change the behaviour of the algorithm. Those parameters include the maximum disparity value, the step size of the candidate grid, thresholds for the support point criteria and many others. A listing of all parameters with a description is provided in the appendix in table 4. Table 1 provides an overview of the main arrays that are used during the computation. Note that the input image and other pixel-related data is stored row by row from left to right in linear fashion. In the following these arrays are described.

Name	Datatype	Dimension	Description
d_img0	uint8_t	$width \times height$	input image
d_desc0	uint8_t	$(16 \times width) \times height$	image descriptor
d_can	int32_t	$\lfloor \frac{width}{5} \rfloor \times \left\lfloor \frac{height}{5} \right\rfloor$	candidate grid
d_sp0	Point2	-	support point coordinates
d_grid0	int32_t	$(256 \times \lceil \frac{width}{20} \rceil) \times \left\lceil \frac{height}{20} \right\rceil$	disparity grid
d_tri0	Tri	-	triangles
d_disp0	float	$width \times height$	disparity values

Table 1: Overview of the main arrays used in the implementation. The index 0 and 1 at the end of a variable refers to the left and right image respectively. Only the arrays referencing the left image are listed.

The image and the descriptor are stored in the array *d_ img0* and *d_ desc0* respectively. The disparities of the candidates from a 5×5 grid are stored in the array *d_ can*. Each entry in the grid at position (x, y) corresponds to the disparity of the pixel at position $(5x, 5y)$ in the left image. Setting a value in the grid to -1 is interpreted as removing the candidate. The coordinates of the support points are stored in *d_ sp0*. The entries are of type *Point2* that is a structure of two integers holding the x and y coordinate for a given support point. The array is used as the input for the Delauny triangulation. The resulting triangles are stored in the data structure *Tri* that is a structure of three integers each referring to the index of a support point in the input array *d_ sp0*. The disparity grid is stored in *d_ grid0*. For each position in the grid 256 integers are allocated. The first entry at a grid position refers to the number of disparities that are available followed by the individual disparities that are exhibited by the support points in the corresponding area. In *d_ disp0* the disparity values of all pixels are stored.

To simplify the specification of the number of bits for an integer, the data types defined in the 'stdint.h' header were used[3]. For the *Cones* image pair from the Middleburry dataset with a resolution of 900×750 pixels the peak memory load on the GPU is around 28,5MB.

5.2 Common Implementation Strategies

Throughout the implementations some common implementation strategies were used that will be presented in this section. When working with pixel-related data, it is often more convenient to use a pixels position to access its corresponding data. However, the data of an array can only be accessed using 1D indexing. As mentioned in the previous section, the arrays used during this implementation store the image data row by row from left to right. Therefore, for an array *arr* of dimension $w \times h$ the value of row y and column x can be accessed using $arr[y \cdot w + x]$.

Moreover, CUDA allows 2D indexing of threads when executing a kernel that simplifies the mapping of threads to pixels. The number of threads and blocks in x- and y-dimension has to be specified during the kernel call. Within a kernel threads can access their x- and y-index using:

Listing 1: Computing the 2D index within a kernel

```
1 int x = blockIdx.x * blockDim.x + threadIdx.x;
2 int y = blockIdx.y * blockDim.y + threadIdx.y;
```

In the following code excerpts of a kernel it is assumed that the x- and y-index is stored in the variables x and y as shown above whenever 2D indexing of threads is used.

[3] more information on the 'stdint.h' header can be found here [25]

The matching cost computation for the support points and the remaining pixels requires many accesses to the descriptor. Reading the values from a descriptor vector individually results in 16 memory fetches. Doing so would lead to considerable increases in run-time. CUDA provides vector loads that allow coalesced memory reads for up to 128 bytes [23]. Hence, one feature vector consisting of 16 bytes can be coalesced to one memory read. The following code piece exemplifies how a coalesced read to register memory is implemented:

```
uint8_t desc_block[16];
(int4*) desc_block = *((int4*) (&d_desc0[...]));
```

For simplification the kernel function *loadDescriptorBlock* was implemented which efficiently loads a descriptor vector from global memory.

5.3 Parallel Implementation

In section 4.2 an analysis of the data dependencies of the algorithm are provided. Figure 5 shows the dependencies between the main components. The execution order and general structure of the GPU implementation is adopted as depicted in the graph. Moreover, it was shown which computation can be executed independently and where computations have to be synchronised. Based on those findings the following sections describe the parallel implementation of each building block. To achieve best performance it is important to comply with the CUDA performance guidelines presented in section 2.2. Therefore, it is emphasized how the computational work is distributed evenly across the GPU's parallel resources and how the mapping between tasks and threads is realized. Moreover, global memory reads can quickly reduce the algorithm's performance due to bandwidth limitations and high latency. Thus, it is stressed how the memory accesses are organized for the different kernels. Note that for simplicity, the following sections always refer to the computation of the disparity map for the left image. In practice, all computations have to be executed for the left and right image.

5.3.1 Descriptor

To compute the descriptor two kernels are implemented. The kernel *kerComputeSobel* computes the horizontal and vertical filtered images from an input image. Each pixel is mapped to one thread using 2D indexing. Within each thread the convolution of the image and the Sobel mask is computed for the corresponding pixel. The result is written to the respective location in the filtered images. After that the kernel *kerComputeDescriptor* is launched that computes the descriptor from the filtered images. For every pixel in the image a descriptor vector is computed that gathers 16 features

from the filtered images within a 5×5 area around the currently observed pixel. Every thread is mapped to one pixel using 2D indexing and stores the concatenation of the features from the filtered images at the corresponding location in the descriptor array *d_ desc0*.

5.3.2 Support Points

This section describes the implementation of the support point computation. First, the disparities for all candidates have to be computed. Thereby, the disparity that minimizes the matching costs has to be found. Computing the matching costs includes taking the SAD of four surrounding descriptor vectors of the observed pixels. For the whole disparity range that results in many matching cost computations and descriptor accesses. Different strategies have been considered to parallelize the matching cost computation. The following will present two possible implementations and compares them using a disparity search range of 256 pixels and a 900×750 pixel image as reference.

One approach is to load the descriptor into shared memory. As the descriptor vector for one pixel is accessed multiple times during the disparity computation, loading the descriptor to shared memory reduces memory latency for following accesses. However, shared memory is a sparse resource with about 48KB storage space [22]. The descriptor of the reference image occupies more then 10MB. Therefore, only portions of the descriptor can be loaded to shared memory and a good data segmentation has to be found. A common strategy is to process the candidates of the same row in one block and load the descriptor row into shared memory that is needed for the matching process [4]. The disadvantage of this approach is that it does not scale well with the image resolution as the number of threads per block is limited. For the reference image that results in 45 matching cost computations per thread.

Another approach is to omit loading the descriptor to shared memory. Instead, the descriptor accesses are managed in a way to enforce cache locality to make up for the slower memory latency of global memory. Cache locality can be enforced by grouping threads in blocks that process connected areas of the image. Neighboring candidates have overlapping disparity search ranges, therefore the descriptor of the corresponding pixel is accessed by multiple candidates. This approach allows to choose the number of threads per disparity computation more freely. For example, by mapping one block of 32 threads to compute the disparity of one candidate, one thread has to calculate only 8 disparities. Importantly, the number of disparities per thread does not depend on the size of the image. Since the aim of this thesis is to provide an implementation that allows to compute the disparity map for high resolution images the latter approach was chosen as it scales well

with the size of the image. The following will present more details on its implementation.

Mapping one block of 32 threads to one candidate as in the example above proved to work well. First, all disparities for matching the candidates from left-to-right are computed. The computation of the matching cost for all valid disparities of a candidate is shared evenly across all threads in a block. Each thread calculates the matching costs for the best and second best match of its share. Shared memory is allocated for every block to communicate their results. Once every thread of a block has calculated the matching costs they are written together with their associated disparity to the shared memory. A simplified excerpt of the main CUDA kernel *kerComputeSupportMatches* is shown below:

Listing 2: excerpt of *kerComputeSupportMatches*

```
1  int d = disp_min_valid+loc_tidx;
2  while(d<=disp_max_valid){
3      desc1_block_addr = &d_desc1[(y*width+(v-d))*16];
4      loadDescriptorBlocks(desc1_block_addr,
5                           desc1_block);
6      for(int i=0; i<4; i++)
7          for(int j=0; j<16; j++)
8              sum += abs(desc0_block[i][j]
9                         -desc1_block[i][j]);
10     // best + second best match
11     if (sum<min_1_E) {
12         min_2_E = min_1_E;
13         min_2_d = min_1_d;
14         min_1_E = sum;
15         min_1_d = d;
16     } else if (sum<min_2_E) {
17         min_2_E = sum;
18         min_2_d = d;
19     }
20     d+=blockDim.x;
21 }
22 writeToSharedMem(min_1_E, min_2_d, min_1_E, min_1_d);
23 __syncthreads();
```

The threads are synchronised afterwards to guarantee that all results have been written. In the next step, the first thread of each block searches through the results from all threads for the best and second-best match. If a disparity is available and the ratio between the best and second-best match

is below the support threshold it is written to corresponding location in the candidate grid.

The same kernel can be launched to check whether the found correspondences can be matched from right-to-left as well by setting the boolean *right_ image* to true. The mapping of the threads to candidates remains the same. For every candidate the disparity for the corresponding pixel in the right image is calculated in the same fashion. If the difference between the disparities matching from left-to-right and right-to-left is greater than the left-right threshold, the respective candidate is removed.

The kernel *removeInconsistentSupportPoints* is implemented to remove spurious outliers. Hereby, every candidate is mapped to one thread that counts the number of similar disparities in its 5×5 neighborhood. If the support is lower then the minimum inconsistency support threshold, the candidate is removed.

Lastly, the kernel *removeRedundantSupportPoints* removes redundant candidates along the horizontal and vertical line. As pointed out in the analysis section, this check can not be done for every candidate independently. Thus, for the horizontal check every row is mapped to one thread that executes the redundancy check for every candidate from left-to-right. If a candidate has a similar disparity compared to other candidates in both directions along its five pixel neighborhood, meaning that their difference is below or equal to the redundancy threshold, the candidate is removed.

5.3.3 Disparity Grid

The disparity grid allows to efficiently lookup the disparities exhibited by support points in a 20×20 pixel neighborhood for each pixel. The computation of the disparity grid splits up into two kernels.

The first kernel *kerLoadGridValues* operates on a temporary grid *d_ temp* and stores the disparities of all support points to their corresponding location in *d_ temp*. For each grid position in *d_ temp* an array of *max_ disp*+1 integers is allocated. Each thread is mapped to one candidate in the candidate grid *d_ can*. If a candidate has a valid disparity d then the value at index d of its corresponding array in the grid is set to one. After all computations of the kernel finished each array holds the disparities exhibited by the support points within its associated 20×20 pixel neighborhood.

The kernel *kerCreateGrid* is implemented to diffuse the grid and aggregate the disparity values in a compact representation. To diffuse the grid, all grid positions include the values from the surrounding grid positions. This is achieved by applying a bit-wise *OR* on the associated arrays and writing the results to the corresponding position in the final grid *d_ grid0*. Finally, the values at each grid position are aggregated as illustrated in figure 6.

At each grid position the first value of the corresponding array indicates the number of available disparities followed by all occurring disparity values

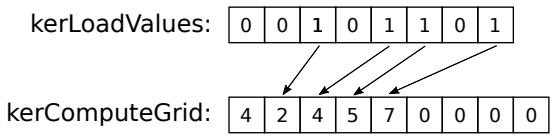

Figure 6: disparity grid gather step

consecutively. The work is distributed by mapping each grid position to one thread. The diffused array is temporarily stored in registers to reduce read an write operations to global memory. Reading the disparity values from surrounding grid positions is implemented using vectorized memory accesses to reduce the amount of read operations.

5.3.4 Triangulation

The GPU implementation of the Delauny triangulation named *gDel2D* by Cao Thanh Tung et al. was used [26]. Additionally, the algorithm can be executed using a CPU implementation *Triangle* by Jonathan Shewchuk [27] due to significant performance differences between GPU and CPU Delauny implementations [28]. It was found that even though using the CPU version requires to copy data from the device to host and back, using a hybrid version is still faster then the GPU alternative. The GPU version can be enabled or disabled by setting the macro *GPU_ONLY* to one or zero respectively.

Both implementations take an array of 2D coordinates as input and return a set of triangles as output. Currently the coordinates of the support points are only indirectly available through their position within the candidate grid. Therefore, the array *d_sp0* is computed holding the coordinates of the support points. To distribute the work evenly across multiple threads some preprocessing steps have to be applied. First, an array of the same size as the candidate grid is allocated. A kernel was implemented that flags all positions in that array with one if the value at the same position in the candidate grid is greater or equal to zero. As introduced in section 5.3.2 these are the support points. Thereby, every position in the candidate grid is mapped to one thread. Then an exclusive prefix sum is calculated on the array using the *ExclusiveSum* function from the CUB library.[4] The exclusive prefix sum returns an array of the same length as the input array with one additional entry. The value at position i in the resulting array is equal to the sum of all values from zero to $i-1$ in the input array. Figure 7 exemplifies computing the prefix sum for an array of size eight.

Therefore, the last value in the returned array is equal to the number of positions in the input array flagged with one which represents the number

[4]More information to the CUB library at `https://nvlabs.github.io/cub/`

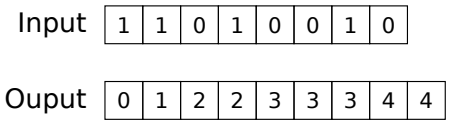

Figure 7: prefix sum

of support points. Lastly a kernel is started that computes the final array holding the coordinates of the support points. Every position in the prefix sum array is mapped to one thread excluding the last entry. The threads are indexed using 2D indexing associating each thread with a position in the candidate grid. The following listing exemplifies the computation of the support point array.

Listing 3: pseudo for computing the support point array

```
1 int offset = y*d_can_width+x;
2 if(d_prefix_sum[offset] != d_prefix_sum[offset+1]){
3     d = d_can[offset];
4     point.x = x*5;
5     point.y = y*5;
6     d_sp[d_prefix_sum[offset]] = point;
7 }
```

The result is stored in array *d_sp0* that holds the coordinates of the support points. The Delauny triangulation is formed on the set of coordinates. The triangulation yields an array of triangles *d_tri0* where each triangle stores the indices of the coordinates in the input array that make up the triangle. Hence, the coordinates of a triangle can be easily retrieved by reading *d_sp0* at the indices provided in each triangle. When using the CPU version *d_sp0* has to be copied from device to host before the triangulation. Afterwards, the result has to be copied back to *d_tri0* on the device.

5.3.5 Disparity Computation

This section describes how the disparity computation of the remaining pixels is implemented. The disparities of the remaining pixels are computed by minimizing the energy function. Hence, the energy function has to be evaluated for every valid disparity. A pixel's disparity is valid if it is within the range of 3σ around the mean value or exhibited by support points within its 20×20 neighborhood. The disparity grid allows for an efficient access to the neighboring disparities based on the pixels position. To calculate the mean value the enclosing triangle of the observed pixel has to be known. However,

accessing the enclosing triangle based on a pixel's position can not be done efficiently.

To do so, the whole triangle array *d_tri0* would have to be searched trough. Therefore, a simple kernel that calculates the disparities for the remaining pixels using a thread to pixel mapping is not enough. Different approaches have been considered to solve the problem.

One possible option would be to map every triangle in the array *d_tri0* to a thread that calculates the disparities for every pixel within the corresponding triangle. Using this approach the enclosing triangle of each processed pixel is trivially known. The mean value can be computed by interpolating the disparity values of the support points corresponding to the corner of the triangle. However, the disadvantage of this approach is that the triangles produced by the Delauny triangulation are rarely of the same size. Hence, the amount of disparity computations within the threads could vary significantly. Threads that process smaller triangles would finish their computation much earlier than threads processing larger ones which would lead to a poor utilization of the parallel resources.

The approach that is implemented is similar to the one explained above but uses a pre-processing step to reduce the amount of work per thread. The disparity computation is split up into two kernels. The first kernel *kerComputeMean* computes only the mean value for the remaining pixels and stores it in the array *d_mean*. As in the previous approach, each triangle in *d_tri0* is mapped to one thread. In the kernel every thread computes the mean value of all pixels within the corresponding triangle and writes the results to *d_mean*. Although the problem of different sized triangles is still given, the computational work per thread is strongly reduced as the disparity computation is transferred to the kernel *kerComputeDisparities*. The pre-processing step enables direct access to a pixel's mean disparity in the array *d_mean* without the need to search for its enclosing triangle. That allows to equally distribute the disparity computation amongst the threads.

Similar to the support point computation, the disparity computation of the remaining pixels requires many accesses to the descriptor. However, in this step the disparity space is significantly reduced and the matching cost computation requires to access only one descriptor vector which results in less matching cost computations per pixel. Therefore, it was decided to load the descriptor to shared memory to enable fast descriptor reads. In practice, each row of the image is mapped to one block of 1024 threads which is the maximum amount of threads per block. The corresponding descriptor row is loaded to shared memory. Then the computation of the disparities of the row is shared equally amongst the threads. The following listing depicts the main functionality of the kernel:

Listing 4: kernel for computing Delauny input

```
1  // load descriptor row to shared memory
2  for(int pos=x; pos<width; pos+=blockDim.x)
3   loadDescBlock(d_desc1+(y*width+x)*16,s_desc+16*x);
4  __syncthreads();
5  for(; x<width; x+=blockDim.x){
6   //load descriptor of current pixel
7   loadDescBlock(d_desc0+(y*width+x)*16,desc_block);
8
9   // disparities within 3*sigma around mean
10  int mean = d_mean[y*width+x];
11  for (d=mean-3*sigma; d<=mean+3*sigma; d++) {
12   updatePosteriorMinimum(s_desc+16*(x-d),desc_block,
13                          d,min_val,min_d);
14  }
15
16  // disparities of neighboring support points
17  int* grid_block = d_grid+getGridOffset(x,y);
18  int num_disps = grid_block[0];
19  for (int i=1;i<=num_disps; i++) {
20   d = grid_block[i];
21   updatePosteriorMinimum(s_desc+16*(x-d),desc_block,
22                          d,min_val,min_d);
23  }
24  d_disp0[y*width+x] = min_d;
25 }
```

In lines 2-3 the descriptor row is loaded to the array *s_desc* that is allocated in the shared memory. The threads are synchronized in line 3 to verify that all threads have finished loading the descriptor. Lines 5-24 shows the computation of the disparities for the pixels that are associated to the thread. First the descriptor of the current reference pixel is loaded to a register array *desc_block*. Then the energy function is evaluated for all valid disparities. The function *updatePosteriorMinimum* evaluates the energy function using the descriptors of the observed pixels and updates the new matching cost minimum in *min_val*. The corresponding disparity is stored in *min_d*. In lines 10-14 the evaluation of the energy function for disparities within the sigma range is shown. Lines 17-25 depicts the evaluation of the energy function for the disparities stored at the corresponding position in the disparity grid. Finally, in line 24 *min_d* is written to the disparity map *d_disp0* in global memory.

5.3.6 L/R Consistency Check

The left/right consistency check is implemented in the kernel *kerLeftRight-ConsistencyCheck*. Before applying the left/right consistency check a copy of the current disparity maps *d_ disp0* and *d_ disp1* is made. If a disparity is removed the previous value can be read from the copy in case it has to be accessed for the consistency check of other disparities. Every pixel is mapped to one thread. Within the kernel each thread computes the check for the corresponding pixel in the right and left image respectively. After the kernel returns the final disparity maps are received.

6 Evaluation

In this section a comparison between the CPU implementation by Geiger et al. and the proposed GPU implementation will be presented. The experimental results for the CPU are computed on a Intel Core i7-3720QM at 2.6 GHz and for the GPU on a NVIDIA GeForce GTX680M graphics card. Both implementations are compared in terms of runtime and quality. During the experiments all parameters are set to be the same. The maximum disparity range was set to 256. A list for all parameter settings can be found in table 5. The images used for the comparison all originate from the Middlebury 2005 and 2006 datasets [11].

6.1 Runtime Comparison

This section compares the runtime of the proposed GPU implementation with the CPU implementation given by the authors. Throughout all experiments the algorithms were run with the same compute steps activated. Both implementations compute the disparity map for the left and right image. Afterwards, the left/right consistency check is applied. All other refinement steps are omitted as their implementation is not part of this thesis. The time measurement for the GPU implementation includes copying the images from the host to the device. Figure 8 depicts the execution times of both implementations averaged over ten executions on the *Cones* image pair scaled to different input sizes.

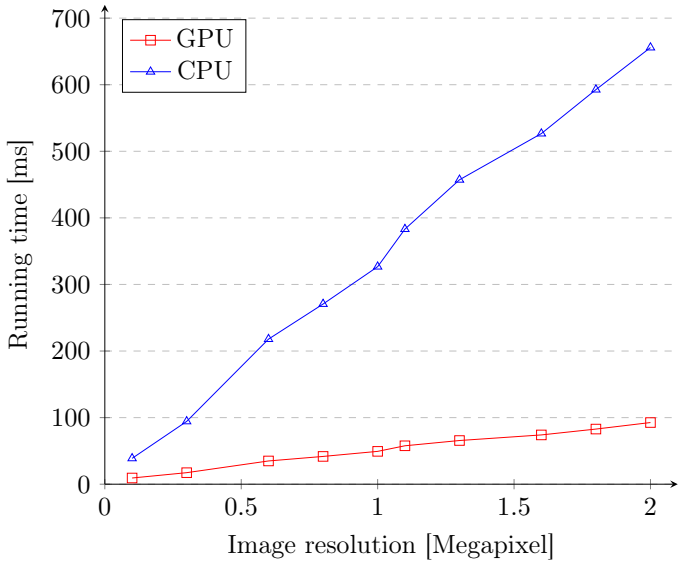

Figure 8: Run-time comparison between GPU and CPU implementation on different image resolutions

The results show that significant run-time improvements were achieved. The proposed implementation computes the disparity maps for the left and right image for all image resolutions in less then 100ms. Moreover, the graph shows that the GPU implementation scales approximately linear with the size of the input image.

Table 2 provides an overview on the achieved speed-up between both implementations averaged over ten executions. Here, the *Teddy, Cones and Aloe* images were used as input. On average the GPU implementation performs the disparity map computation 5.74 times faster as the CPU implementation. Moreover, the speed-up increases with higher image resolutions which is due to a better occupancy of the GPU's resources.

	GPU run-time (10×)	CPU run-time (10×)	Speed-Up
Teddy	11.18 ms	48.69 ms	4.35
Cones	36.06 ms	219.40 ms	6.08
Aloe	68.43 ms	465.69 ms	6.80
Average			5.74

Table 2: Average speedup for different the input images: Teddy $450 \times 375px$, Cones $900 \times 750px$ and Aloe $1282 \times 1110px$ from the Middlebury Datasets.

Next the performance of the different building blocks is compared using the *Cones* image. Since both implementations use the same library 'Triangle' for the Delauny triangulation its runtimes are not included here. Figure 9 shows the run-times of the different algorithm components. The most significant speed-up is achieved during the disparity computation which runs with 10.5 ms more than one order of magnitude faster on the GPU. This is especially relevant as it makes up the most computation time of the CPU implementation. Great speed-up is achieved during the descriptor computation as well. With a speed-up of around four a good speed-up is accomplished for the support point and disparity grid computation. A possible reason for the lower speed-up is that the other computations use a better strategy for memory accesses. In contrast to the disparity computation of the remaining pixels where the descriptor is loaded to shared memory, the descriptor is read from global memory during the support point calculation. Moreover, the runtime of the support point computation is increased as removing redundant candidates requires to process a whole row of the candidate grid by only one thread. The other computations allow for a better distribution of the computational load amongst the threads.

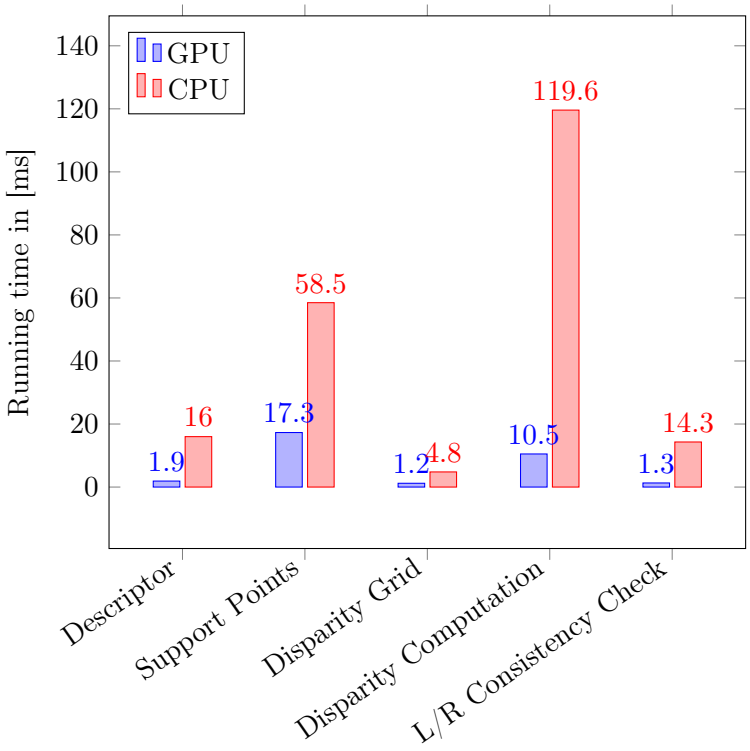

Figure 9: Run-time comparison between the GPU and CPU implementation of the main building components using the *Cones* image.

6.2 Quality Comparison

This section provides a quality comparison of the disparity maps produced by both implementations. As mentioned earlier this thesis focuses on the implementation of the core features of ELAS, thus varies from the CPU implementation. Therefore dissimilarities between the resulting disparity maps are expected. Scharstein et al. provide the Middlebury Evaluation SDK that is commonly used to evaluate the results of dense two-frame stereo correspondence algorithms [11]. They additionally provide ground truth disparity maps and different metrics to evaluate an algorithms performance.

During this evaluation the disparity maps of both implementations are compared to the ground truth disparity maps using their provided SDK. Thereby only the left disparity maps are used for the evaluation. The disparity maps are evaluated using the *bad 2.0* and *bad 4.0* metric as they are commonly used on the Middelbury online evaluation platform [29]. The error metric *bad 2.0* refers to the percentage of pixels whose assigned disparity value differs more than two pixels from the true disparity value. Analogously, the percentage of disparities that differ more then four from the true disparity value are expressed using the *bad 4.0* metric. Additionally, occluded

pixel, i.e. pixels that are only visible in one of the input images, are ignored during the error calculation. The images used for the comparison are from the Middlebury Evaluation Dataset providing a wide range of baselines and image resolutions up to 1500×1000 pixel.[5]

During the experiments both implementations use the same refinement steps. First, a left-right consistency check is applied to remove spurious mismatches and disparities in occluded regions. Then, small pixel segments not larger then 200 pixels are removed. Missing disparities are interpolated in the final refinement step. For all experiments the maximum interpolation gap width was set to three. Table 3 shows the results of the *bad 2.0* and *bad 4.0* metrics on a variate of images for both implementations.

	bad 2.0		*bad 4.0*	
	GPU	CPU	GPU	CPU
Adirondack	8.55 %	7.23 %	5.44 %	4.57 %
ArtL	6.98 %	5.8 %	5.41 %	4.5 %
Jadeplant	6.68 %	6.17 %	4.63 %	4.28 %
Motorcycle	6.2 %	5.76 %	3.49 %	3.36 %
Piano	17.62 %	16 %	12.91 %	11.81 %
PianoL	24.51 %	24.15 %	18.91 %	19.29 %
Pipes	8.56 %	7.45 %	6.4 %	5.74 %
Playroom	14.33 %	12.94 %	8.9 %	8.14 %
PlaytableP	11.47 %	10.33 %	8.01 %	7.2 %
Teddy	5.06 %	4.36 %	3.41 %	2.86 %

Table 3: Results on Middlebury data set.

Even though the GPU implementation uses a slightly different method to compute the descriptor vectors similar results of the *bad 2.0* and *bad 4.0* metrics are achieved. On average the difference between the results of the *bad 2.0* are less then 0.5% and for the *bad 4.0* even less then 0.1 %. Hence, the GPU implementation performs competitive to the authors implementation quality wise. Figure 10 depicts the disparity maps produced by the GPU and CPU implementation for the *Teddy*, *Playroom* and *Motorcycle* stereo image pair, which further emphasises the GPU implementation's competitiveness.

[5]Middlebury Evaluation Datasets at `http://vision.middlebury.edu/stereo/submit3/`

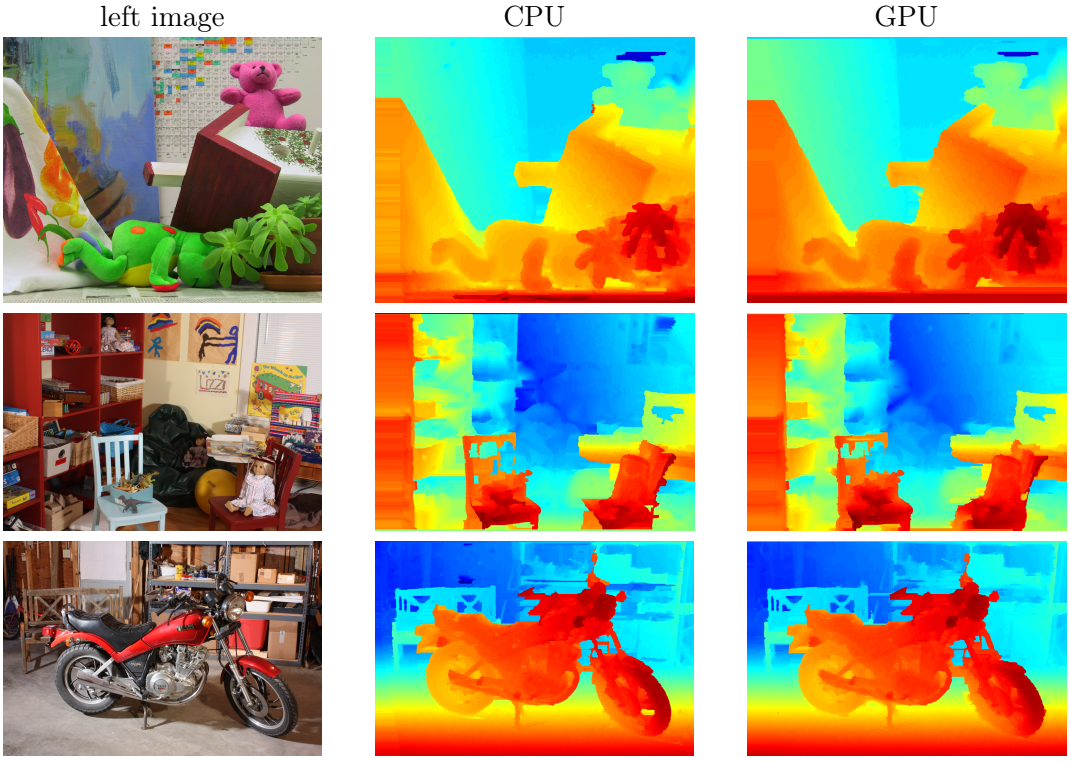

Figure 10: Disparity maps of CPU and GPU implementation. Images from top to bottom: *Teddy*, *Playroom* and *Motorcycle*.

7 Conclusion

In this thesis a parallel implementation of the stereo matching algorithm *ELAS* using the CUDA programming model was proposed, developed and tested. In contrast to standard block matching algorithms that exhibit low matching ratios within texture-less areas, ELAS builds a prior probability distribution over the pixels based on a triangulation over robustly matched support points which decreases the ambiguities of the remaining pixels. The algorithm allows to compute accurate disparity maps of high resolution images without the need for global optimization.

An analysis of the data dependencies that underlie the algorithm are presented in 4. Based on these findings a detailed description of the implementation of the algorithm using CUDA is provided in section 5. Experimental results are presented in section 6 using the training image sets from the Middlebury dataset. The results show that the accuracy of the presented parallel implementation is comparable to the reference algorithm, while being on average more then five times faster. For 750×900 pixel images the implementation runs at 27 fps on a NVIDIA Geforce GTX 680M. Following the guidelines for CUDA programming presented in section 2.2 has turned out to be crucial for the performance of the algorithm. Most importantly, coalescing global memory reads and realizing an evenly distribution of the computational load amongst the threads leads to the most significant runtime improvements.

Future work includes investigating whether the Delauny triangulation can be simplified for the specific use case. The jump flooding approach by Guodong et al. would be appropriate as it allows to compute a discrete Delauny triangulation very fast on the GPU. Guodong et al. state that their approach achieves significant speedup compared to *Triangle*, the currently used Delauny triangulation implementation, which is known to be the fastest implementation of its class [30, 31]. Moreover, different data tiling and memory access strategies will be investigated. This includes considering a different usage of shared memory during the support point computation as it is one of the most time consuming parts of the algorithm. Currently, shared memory is used to communicate the results of the best matches between the threads that are mapped to one candidate. Adopting the strategy of the final disparity computation by loading descriptor rows to shared memory could reduce global memory accesses and improve overall performance.

The findings show that the GPU implementation is suitable to meet the requirements for many real-time applications that include obstacle detection and avoidance like autonomous vehicles. Weber et al. state that for such use cases a frame rate around 15 fps would be sufficient [10]. As the presented

implementation exceeds this frame rate, the additional processing power of the GPU could be used for further optimization. For example, the accuracy of algorithm could be improved by increasing the number of support points, thus resulting in a more precise prior. Likewise it paves the way for further processing steps allowing for more complex algorithms and applications.

8 References

[1] Emilio Frazzoli, Munther A. Dahleh, and Eric Feron. Real-time motion planning for agile autonomous vehicles. *AIAA Guidance, Navigation, and Control Conference and Exhibit*, 25(1), 2000.

[2] Rostam Affendi Hamzah and Haidi Ibrahim. Literature survey on stereo vision disparity map algorithms. *Journal of Sensors*, 2016, 2016.

[3] Tangfei Tao, Ja Choon Koo, and Hyouk Ryeol Choi. A fast block matching algorthim for stereo correspondence. *2008 IEEE International Conference on Cybernetics and Intelligent Systems, CIS 2008*, pages 38–41, 2008.

[4] Ke Zhu, Matthias Butenuth, and Pablo D'Angelo. Comparison of dense stereo using CUDA. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6554 LNCS(PART 2):398–410, 2012.

[5] Adrian Leu, Dan Bacara, and Ioan Jivet. Disparity map computation speed comparison for CPU, GPU and FPGA implementations. *TRANSACTIONS on ELECTRONICS and COMMUNICATIONS*, 55(69):7–12, 2010.

[6] Takeo Kanade, Hiroshi Kano, Shigeru Kimura, Atsushi Yoshida, and Kazuo Oda. Development of a video-rate stereo machine. *IEEE International Conference on Intelligent Robots and Systems*, 3:95–100, 1995.

[7] Matthias Michael, Jan Salmen, Johannes Stallkamp, and Marc Schlipsing. Real-time stereo vision: Optimizing Semi-Global Matching. *IEEE Intelligent Vehicles Symposium, Proceedings*, pages 1197–1202, 2013.

[8] Nan Zhang, Jian Li Wang, and Yun Shan Chen. Image parallel processing based on GPU. *Proceedings - 2nd IEEE International Conference on Advanced Computer Control, ICACC 2010*, 3:367–370, 2010.

[9] Arnas Ivanavičius, Henrikas Simonavičius, Julius Gelšvartas, Andrius Lauraitis, Rytis Maskeliūnas, Piotras Cimmperman, and Paulius Serafinavičius. Real-time CUDA-based stereo matching using Cyclops2 algorithm. *Eurasip Journal on Image and Video Processing*, 2018(1), 2018.

[10] Michael Weber, Martin Humenberger, and Wilfried Kubinger. A very fast census-based stereo matching implementation on a graphics processing unit. *2009 IEEE 12th International Conference on Computer Vision Workshops, ICCV Workshops 2009*, 3:786–793, 2009.

[11] Daniel Scharstein and Richard Szeliski. [Scharstein02 IJCV] A Taxonomy and Evaluation of Dense Two-Frame Stereo Correspondence Algorithm.pdf. (1).

[12] R. Lagisetty, N. K. Philip, R. Padhi, and M. S. Bhat. Object detection and obstacle avoidance for mobile robot using stereo camera. *Proceedings of the IEEE International Conference on Control Applications*, pages 605–610, 2013.

[13] D. Hernandez-Juarez, A. Chacon, A. Espinosa, D. Vazquez, J. C. Moure, and A. M. Lopez. Embedded real-time stereo estimation via Semi-Global Matching on the GPU. *Procedia Computer Science*, 80:143–153, 2016.

[14] Andreas Geiger, Martin Roser, and Raquel Urtasun. Efficient large-scale stereo matching. 2010.

[15] Andreas Koschan. What is new in computational stereo since 1989: A survey on current stereo papers. (August), 1993.

[16] D. V. Papadimitriou and T. J. Dennis. Epipolar line estimation and rectification for stereo image pairs. *IEEE Transactions on Image Processing*, 5(4):672–676, 1996.

[17] R. R. Orozco, C. Loscos, I. Martin, and A. Artusi. *HDR multiview image sequence generation: Toward 3D HDR video.* Elsevier Inc., 2017.

[18] Yun Suk Kang and Yo Sung Ho. An efficient image rectification method for parallel multi-camera arrangement. *IEEE Transactions on Consumer Electronics*, 57(3):1041–1048, 2011.

[19] Andrea Fusiello, Emanuele Trucco, and Alessandro Verri. Compact algorithm for rectification of stereo pairs. *Machine Vision and Applications*, 12(1):16–22, 2000.

[20] Heiko Hirschmüller and Daniel Scharstein. Evaluation of cost functions for stereo matching. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2007.

[21] Heiko Hirschmüller. Semi-Global Matching – Motivation, Developments and Applications. (Figure 1):173–184, 2002.

[22] NVIDIA. *CUDA C++ Programming Guide.* 2019.

[23] NVIDIA. *CUDA C++ BEST PRACTICES GUIDE.* 2019.

[24] Oleg R. Musin. Properties of the Delaunay triangulation. *Proceedings of the Annual Symposium on Computational Geometry*, 2:424–426, 1997.

[25] Reference 'stdint.h' header. https://pubs.opengroup.org/onlinepubs/009695399/basedefs/stdint.h.html. Accessed: 2019-11-27.

[26] Thanh Tung Cao, Ashwin Nanjappa, Mingcen Gao, and Tiow Seng Tan. A GPU accelerated algorithm for 3D Delaunay triangulation. *Proceedings of the Symposium on Interactive 3D Graphics*, pages 47–54, 2014.

[27] Jonathan Richard Shewchuk. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In Ming C. Lin and Dinesh Manocha, editors, *Applied Computational Geometry: Towards Geometric Engineering*, volume 1148 of *Lecture Notes in Computer Science*, pages 203–222. Springer-Verlag, May 1996. From the First ACM Workshop on Applied Computational Geometry.

[28] Ge Li, Xuehe Zhang, Changle Li, Hongzhe Jin, and Jie Zhao. Design and application of parallel stereo matching algorithm based on CUDA. *Microprocessors and Microsystems*, 47:142–150, 2016.

[29] Middlebury stereo website. http://vision.middlebury.edu/stereo/. Accessed: 2019-12-15.

[30] Guodong Rong and Tiow Seng Tan. Jump flooding in GPU with applications to voronoi diagram and distance transform. *Proceedings of the Symposium on Interactive 3D Graphics*, 2006:109–116, 2006.

[31] Guodong Rong, Tiow Seng Tan, Thanh Tung Cao, and Stephanus. Computing two-dimensional Delaunay triangulation using graphics hardware. *Proceedings of the Symposium on Interactive 3D Graphics and Games, I3D 2008*, 1(212):89–97, 2008.

9 Appendix A

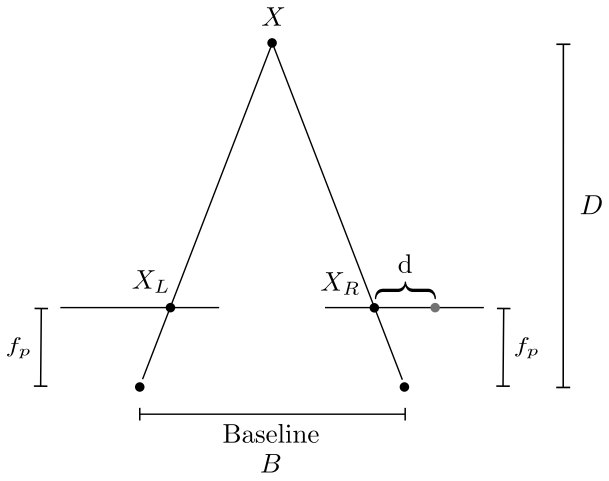

Figure 11: Relationship between disparity and distance.

Parameter	Description
DISP_MIN	min. dispairty
DISP_MAX	max. disparity
CANDIDATE_STEPSIZE	step size of the candidate grid
GRID_SIZE	disparity grid size
SUPPORT_TEXTURE	min. required texture for support points
SUPPORT_THRESHOLD	max. uniqueness ratio
LR_THRESHOLD	disparity threshold for left/right consistency check
ICON_WINDOW_SIZE	window size of inconsistent support point check
INCON_THRESHOLD	disparity similarity threshold for support point to be considered consistent
INCON_MIN_SUPPORT	min. number of consistent support points
REDUN_MAX_DIST	max. distance for redundancy check
REDUN_THRESHOLD	max. number of redundant support points
SPECKLE_SIM_THRESHOLD	similarity threshold for speckle segmentation
SPECKLE_SIZE	maximal size of a speckle
INTERPOL_GAP_WIDTH	interpolate small gaps
SIGMA	prior sigma
SRADIUS	prior sigma radius
GAMMA	prior gamma
BETA	prior beta
MATCH_TEXTURE	min texture for dense matching

Table 4: Description of the parameters used for the implementation.

Parameter	Value	Parameter	Value
CANDIDATE_STEPSIZE	5	LR_THRESHOLD	2
DISP_MIN	0	SPECKLE_SIM_THRESHOLD	1
DISP_MAX	255	SPECKLE_SIZE	200
SUPPORT_TEXTURE	10	INTERPOL_GAP_WIDTH	3
SUPPORT_THRESHOLD	0.85	GRID_SIZE	20
ICON_WINDOW_SIZE	5	SIGMA	1
INCON_THRESHOLD	5	GAMMA	3
INCON_MIN_SUPPORT	5	BETA	0.02
REDUN_MAX_DIST	5	SRADIUS	2
REDUN_THRESHOLD	1	MATCH_TEXTURE	0

Table 5: Parameter values used during the experiments.