

# Performance Analysis of a Stereo Matching Implementation in OpenCL

DIPLOMARBEIT

zur Erlangung des akademischen Grades

**Diplom-Ingenieur**

im Rahmen des Studiums

**Visual Computing**

eingereicht von

**Stephan Rotheneder, BSc**

Matrikelnummer 0625931

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Mag. Dr. Margrit Gelautz

Mitwirkung: Dr. Govinda Lilley

Dr. Nicolas Thorstensen

Wien, 3. Mai 2018

---

Stephan Rotheneder

---

Margrit Gelautz



# Performance Analysis of a Stereo Matching Implementation in OpenCL

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

**Diplom-Ingenieur**

in

**Visual Computing**

by

**Stephan Rotheneder, BSc**

Registration Number 0625931

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Mag. Dr. Margrit Gelautz

Assistance: Dr. Govinda Lilley

Dr. Nicolas Thorstensen

Vienna, 3<sup>rd</sup> May, 2018

---

Stephan Rotheneder

---

Margrit Gelautz



# Erklärung zur Verfassung der Arbeit

Stephan Rotheneder, BSc  
Schönbrunner Schloßstraße 9/18  
1120 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 3. Mai 2018

---

Stephan Rotheneder



# Acknowledgements

I would like to thank my supervisor Margrit Gelautz and my supervisor at IVISO GMBH Govinda Lilley for their support throughout the whole process of this master thesis.

Further, I would like to thank Nicolas Thorstensen CEO of IVISO GMBH for providing support and infrastructure at his company.

Furthermore, I would like to thank my parents who always believed in me and my friends who helped by proof reading and giving me valueable feedback. Especially, I would like to thank Sandra for supporting me in so many ways and helping me to not lose track of the big picture.

This diploma thesis was carried out in close collaboration with IVISO GMBH.





# Abstract

Stereo matching is one of the first steps in the process of calculating 3D information from two 2D images. To triangulate a 3D point from two corresponding 2D features, the displacement in pixels, or the so-called disparity, must be estimated. From the estimated per-pixel disparity, using a projective camera model, 3D data for large portions of an image may be calculated.

The 3D scene information can be used in applications ranging from obstacle detection and collision avoidance systems in the automotive industry to pick-and-place or human safety systems in the robotics industry. As time is an important factor in most of these applications, the subject of real-time stereo matching has gained importance while quality and accuracy aspects retain their importance. Benchmarks such as the KITTI Benchmark or the Middlebury Benchmark aim at providing stereo test data as well as ground truth to evaluate different matching algorithms against each other with regard to accuracy, coverage and runtime. However, they fall short in measuring the computational efficiency as the reported runtime as well as the real-time capability of the listed stereo matching algorithms are highly hardware dependent.

In this thesis we explore the possibilities of real-time stereo matching and the constraints imposed by the used hardware. Therefore, we implemented a stereo matching algorithm in Open Computation Language (OpenCL) in order to evaluate the runtime of a specific algorithm on multiple devices. Using this runtime data, we discuss the limitations of runtime measurements with respect to varying computational power. Further, we suggest a method to compare the efficiency of various algorithms based on the reported runtime and hardware data, which is provided by the Middlebury Benchmark. This enables us to estimate the real-time capability of a given algorithm with a known problem space size on an arbitrary device with a manufacturer specified or measured performance figure. Finally, we observe that the problem space size, the device performance figure and the algorithm's runtime complexity directly correlate with the matching rate given in Frames per Second (FPS).



# Kurzfassung

Einer der ersten Schritte des Prozesses, welcher 3D-Information aus 2D-Bildern berechnet, ist der des Stereo Matching. Um einen 3D-Punkt aus übereinstimmenden 2D-Features zu triangulieren, muss deren Abstand in Pixel, die sogenannte Disparität, berechnet werden. Mit der berechneten Disparität pro Pixel und einem projektiven Kameramodell, können 3D-Daten für den Großteil eines Bildes berechnet werden.

Die Berechnung von 3D-Szeneninformation findet Anwendung in Bereichen der Hinderniserkennungs- und Kollisionsvermeidungssysteme der Automobilindustrie und pick-and-place- oder Mensch-Roboter-Kollaboration-Systeme (MRK-Systeme) in der Roboterindustrie. Da Zeit ein wichtiger Faktor in den meisten dieser Anwendungen ist, hat das Thema Echtzeit-Stereo-Matching an Wichtigkeit gewonnen, während Qualitäts- und Genauigkeitsaspekte ihren Stellenwert beibehalten haben. Benchmarks wie der KITTI- oder Middlebury Benchmark stellen Test-Stereodaten und Ground Truth für die Evaluierung von Matching-Algorithmen zur Verfügung. Diese werden genutzt, um verschiedene Algorithmen, bezüglich Genauigkeit, Flächendeckung und Laufzeit miteinander zu vergleichen. Allerdings liegt eine Schwäche im Hinblick auf die Messbarkeit der Recheneffizienz vor, da die Laufzeit- und Echtzeit-Fähigkeiten von verglichenen Stereo-Matching Algorithmen stark von der verwendeten Hardware abhängen.

Das Hauptaugenmerk dieser Arbeit liegt auf der Untersuchung der Möglichkeiten von Echtzeit-Stereo-Matching unter den mit der verwendeten Hardware verbundenen Einschränkungen. In Hinblick auf diesen Aspekt wurde ein Stereo-Matching Algorithmus in OpenCL implementiert, um die Laufzeit eines einzelnen Algorithmus auf verschiedenen Geräten evaluieren zu können. Mit diesen Laufzeitdaten besprechen wir die Beschränkungen von Laufzeitmessungen bezüglich variierender Rechenleistung und stellen eine Methode zum Vergleich der Laufzeitkomplexität verschiedener Algorithmen, basierend auf den hardwareabhängigen Laufzeitmessungen des Middlebury Benchmarks, vor. Ausgehend hiervon, stellen wir eine Methode zur Abschätzung der Echtzeit-Möglichkeiten eines Algorithmus bei festgelegter Problemgröße auf gegebenen Geräten, mit vom Hersteller bestimmter oder gemessener Rechenleistung, vor.

Abschließend argumentieren wir, dass die Problemgröße, die Rechenleistung des Gerätes und die Laufzeitkomplexität des Algorithmus direkt mit der Matching-Rate, welche in Frames pro Sekunde (FPS) angegeben ist, zusammenhängen.



# Abbreviations

<b>1D</b>	1-dimensional . . . . .	35
<b>2D</b>	2-dimensional . . . . .	1
<b>3D</b>	3-dimensional . . . . .	4
<b>API</b>	Application Programming Interface . . . . .	1
<b>ARM</b>	Advanced RISC Machine or Acorn RISC Machine . . . . .	1
<b>ASIC</b>	Application-Specific Integrated Circuit . . . . .	42
<b>CNN</b>	Convolutional Neural Network . . . . .	40
<b>CPU</b>	Central Processing Unit . . . . .	1
<b>CRV</b>	Conference on Computer and Robot Vision . . . . .	83
<b>CVPR</b>	Conference on Computer Vision and Pattern Recognition . . . . .	83
<b>DLT</b>	Direct Linear Transformation . . . . .	14
<b>DSI</b>	Disparity Space Images . . . . .	19
<b>DSP</b>	Digital Signal Processor . . . . .	42
<b>ECCV</b>	European Conference on Computer Vision . . . . .	83
<b>FLOPC</b>	Floating point Operations per Pixel Comparison . . . . .	81
<b>FLOPS</b>	Floating point Operations Per Second . . . . .	84
<b>FOI</b>	Feature Of Interest	
<b>FPA</b>	Fronto-Parallel Assumption . . . . .	3
<b>FPGA</b>	Field Programmable Gate Array . . . . .	32
<b>FPS</b>	Frames per Second . . . . .	ix
<b>GFLOPS</b>	Giga FLOPS . . . . .	89
<b>GP</b>	Giga Pixel . . . . .	28
<b>GPU</b>	Graphical Processing Unit . . . . .	1
<b>HCI</b>	Heidelberg Collaboratory for Image Processing . . . . .	25
<b>HSV</b>	Hue Saturation Value	

<b>KITTI</b>	Karlsruhe Institute of Technology and Toyota Technological Institute . . . .	25
<b>MP</b>	Mega Pixel . . . . .	28
<b>MPC</b>	Mega Pixel Comparison . . . . .	91
<b>NCT</b>	Normalized Calculation Time . . . . .	84
<b>OOI</b>	Object Of Interest	
<b>OpenCL</b>	Open Computation Language . . . . .	ix
<b>OpenCV</b>	Open Source Computer Vision Library . . . . .	14
<b>OpenGL</b>	Open Graphics Library . . . . .	1
<b>PC</b>	Pixel Comparison . . . . .	84
<b>PDA</b>	Personal Digital Assistant . . . . .	2
<b>PFM</b>	Portable Float Map . . . . .	96
<b>PGM</b>	Portable Gray Map . . . . .	95
<b>PNG</b>	Portable Network Graphics . . . . .	95
<b>POI</b>	Point Of Interest	
<b>PPM</b>	Portable Pixel Map . . . . .	127
<b>RGB</b>	Read Green Blue	
<b>RISC</b>	Reduced Instruction Set Computing	
<b>RMS</b>	Root-Mean-Square	
<b>ROB</b>	Robust Vision Challenge 2018 . . . . .	83
<b>ROI</b>	Region Of Interest	
<b>SAD</b>	Sum of Absolute Differences . . . . .	19
<b>SDK</b>	Software Development Kit . . . . .	51
<b>SGM</b>	Semi Global Matching . . . . .	90
<b>SIFT</b>	Scale-Invariant Feature Transform . . . . .	39
<b>SSD</b>	Sum of Squared Differences . . . . .	19
<b>VHDL</b>	Very high speed integrated circuit Hardware Description Language . . . . .	42

# Contents

<b>Abstract</b>	<b>ix</b>
<b>Kurzfassung</b>	<b>xi</b>
<b>Abbreviations</b>	<b>xiii</b>
<b>Contents</b>	<b>xv</b>
<b>1 Introduction and Motivation</b>	<b>1</b>
<b>2 Background and Theoretical Foundation</b>	<b>3</b>
2.1 Basic Concepts of Stereo Matching . . . . .	3
2.2 Middlebury Benchmark . . . . .	25
2.3 OpenCL . . . . .	32
<b>3 State of the Art</b>	<b>39</b>
3.1 Sparse Stereo Matching . . . . .	39
3.2 Dense Stereo Matching . . . . .	40
3.3 Local Stereo Methods . . . . .	41
3.4 Global Stereo Methods . . . . .	41
3.5 Real-Time Stereo Matching . . . . .	42
3.6 Real-Time Hardware for Stereo Matching . . . . .	42
<b>4 A Stereo Matching Algorithm</b>	<b>43</b>
4.1 Sparse Census Cost Function . . . . .	43
4.2 Aggregation Strategy . . . . .	46
4.3 Sub Pixel Refinement . . . . .	48
4.4 Left-Right Consistency Check . . . . .	49
<b>5 Implementation</b>	<b>51</b>
5.1 Overview . . . . .	52
5.2 RectificationKernel . . . . .	58
5.3 CensusKernel . . . . .	61
5.4 DiffCubeKernel . . . . .	64
	xv

5.5	CostXCubeKernel . . . . .	67
5.6	CostYCubeKernel . . . . .	70
5.7	MinimumKernels . . . . .	71
5.8	CostCacheKernels . . . . .	74
5.9	ParabolicFittingKernel . . . . .	77
5.10	ConsistencyKernel . . . . .	78
5.11	Conclusion . . . . .	80
<b>6</b>	<b>Evaluation</b>	<b>81</b>
6.1	Timing results . . . . .	82
6.2	Implementation-Result Comparison . . . . .	95
6.3	Result Errors . . . . .	99
6.4	Additional Results . . . . .	107
<b>7</b>	<b>Summary and Outlook</b>	<b>111</b>
	<b>Appendix 1</b>	<b>113</b>
	<b>Appendix 2</b>	<b>125</b>
	<b>Appendix 3</b>	<b>127</b>
	<b>List of Figures</b>	<b>130</b>
	<b>List of Tables</b>	<b>132</b>
	<b>List of Algorithms</b>	<b>133</b>
	<b>Bibliography</b>	<b>135</b>



# Introduction and Motivation

The problem of stereo matching is an often discussed matter in computer vision. Many different methods have been proposed over the years. Stereo matching can be used to estimate depth from two 2-dimensional (2D) images of the same scene from slightly different vantage points. Stereo matching can be used in many areas such as the robotic industry, geodesy, scene reconstruction from video or autonomous systems like self driving cars or drones.

According to [SSZ01] stereo matching methods can be organized by various aspects e.g. density in feature space (dense versus sparse methods), size of considered chunks in image space (global methods versus local methods) or optimization method (winner takes all, dynamic programming, scanline optimization, graph cut).

Many implementations of newly proposed algorithms are designed to run in a single thread on a personal computer Central Processing Unit (CPU). However, this way of implementation binds the implementation to a single platform and is usually neither efficient nor fast. Some implementations are designed for Graphical Processing Units (GPUs). These implementations are usually very good when the runtime is compared. However, different authors use different GPUs, which makes us question the comparability of these runtime results. Further problems exist when low energy hardware is used. Many Advanced RISC Machine or Acorn RISC Machine (ARM) processors and ARM GPUs do not support Application Programming Interfaces (APIs) like OpenCL or Open Graphics Library (OpenGL) that would allow to uniformly control these devices' processing units. Furthermore, ARM processing units are usually low performance devices and therefore have lower limits to how much data they can process. Additionally, these limits impose the question of which algorithm can be used for high performance stereo matching on such devices and what are the parameters for such a task.

In this thesis, we will show an example how OpenCL can be used to implement a stereo matching algorithm. We will show that an OpenCL implementation can be

compiled for multiple different platforms. This can be used to run stereo matching algorithms on ARM devices that support OpenCL such as mobile phones or Personal Digital Assistants (PDAs).

Further, we will show an algorithm specific metric that makes algorithms more comparable throughout different devices by taking the floating point performance values of the used devices into account.

Finally, we will present a way to estimate the upper limit for the image dimensions of the input images in order to achieve a target matching frame rate for a specific device.

# Background and Theoretical Foundation

In this chapter, we will introduce the basic concepts, strategies and technologies that are used in later chapters. We will start in Section 2.1 with the basics of stereo matching and the used terms and ideas. This will be followed by Section 2.2, where we will explain the basic concepts of taxonomy measurements in stereo matching. We will then shortly explain the metrics used by [SSZ01] in their benchmark<sup>1</sup>. Finally, in Section 2.3 we will describe how OpenCL is organized, how memory and buffers are managed and how computations are parallelized.

## 2.1 Basic Concepts of Stereo Matching

In this section, we will explain the basic concepts of stereo matching i.e. epipolar geometry, rectification, stereo matching, the Fronto-Parallel Assumption (FPA), cost functions, disparity space images and cost volumes, support aggregation, disparity optimization and disparity refinement. We will shortly discuss the connection of left and right images due to epipolar geometry and how rectification simplifies the process of tracing these connections. We will explain the concept of disparity, how rectified images are used in stereo matching, and how the Fronto-Parallel Assumption is used in cost functions for the purpose of stereo matching. We will review disparity space images as a representation of the problem space for stereo matching, how the matching costs are calculated, and how cost aggregation reduces ambiguities in the matching process. We will discuss how aggregated cost functions can be efficiently applied for multiple disparities values by reusing intermediate results and further explain how cross correlation checks are done to improve the overall result by reducing erroneous disparity values.

---

<sup>1</sup><http://vision.middlebury.edu/stereo/>

### 2.1.1 Epipolar Geometry

Given are two cameras  $C_0$  and  $C_1$ , with their respective camera positions  $c_0$  and  $c_1$ , looking into a scene and a known rotation matrix  $R$  and translation vector  $t$  to transform  $C_0$  into  $C_1$ . This transformation is given by:

$$c_1 = [R, t] \cdot c_0 \quad (2.1)$$

As shown in Figure 2.1 a point  $p$  that is seen by  $C_0$  is projected onto the image plane ( $I_0$ ) of  $C_0$ , at image coordinate  $x_0$  and projected to infinity at point  $p_\infty$ . The projection of  $p$  onto  $C_1$ 's image plane ( $I_1$ ) lies on an epipolar line segment ( $l_1$ ). The segment  $l_1$  is defined as the line bounded by the projection of  $p_\infty$  onto  $I_1$  and the projection of  $c_0$  onto  $I_1$ . The projection of one camera center point onto the other camera's image plane is called an epipole. The epipole  $e_0$  is the camera center  $c_1$  projected onto  $I_0$  and vice versa. In order to find the projection  $x_1$  of  $p$  onto  $I_1$  we only have to examine the point of  $I_1$  that coincides with  $l_1$ . The point  $p$  and the projected point  $p_\infty$  can be calculated using  $x_0$  and  $c_0$ . The epipolar line  $l_1$  can be calculated using  $e_1$  and the projection of  $p_\infty$  onto  $I_1$ . To find  $x_1$ , only points lying on  $l_1$  have to be examined.

The epipolar line  $l_0$ , is defined by epipole  $e_0$  and point  $x_0$ . It can be shown for any point  $p$  in the 3-dimensional (3D) scene that if  $p$ 's projection onto  $I_0$  lies on  $l_0$ , then its projection onto  $I_1$  will lie on  $l_1$ .

For all points on  $l_0$ , the best match can be found on  $l_0$ s corresponding  $l_1$ .

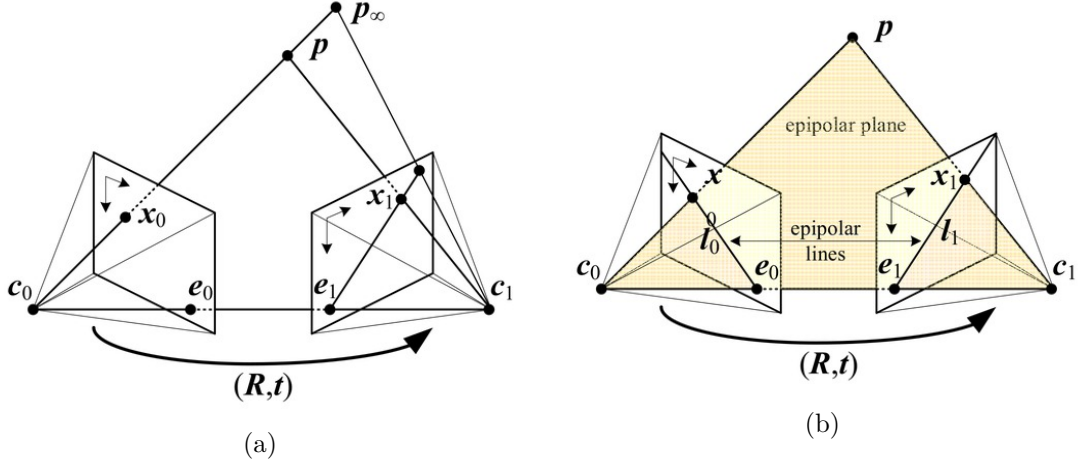


Figure 2.1: Epipolar geometry [Sze10]: The point  $p$  is a point in the 3D scene. The points  $x_0$  and  $x_1$  are the projections of  $p$  onto the image planes  $I_0$  and  $I_1$ . The camera center point  $c_0$  projected onto  $I_1$  is the epipole  $e_1$  and camera center point  $c_1$  projected onto  $I_0$  is the epipole  $e_0$ . The lines  $l_0$  and  $l_1$ , going through  $x_0$  and  $x_1$  and the epipoles  $e_0$  and  $e_1$  are called epipolar lines. It can be shown for any point  $p$  in the 3D scene that, if  $p$ 's projection onto  $I_0$  lies on  $l_0$ , then its projection onto  $I_1$  will lie on  $l_1$ .

### 2.1.2 Rectification and Undistortion

In their chapter on rectification, [Sze10] state that *"a more efficient algorithm can be obtained by first rectifying the input images"*. Rectification in this context means to warp the input images (usually undistorted images) in a way that corresponding epipolar lines coincide with horizontal scanlines. Figure 2.2 shows how rectification warps the input images.

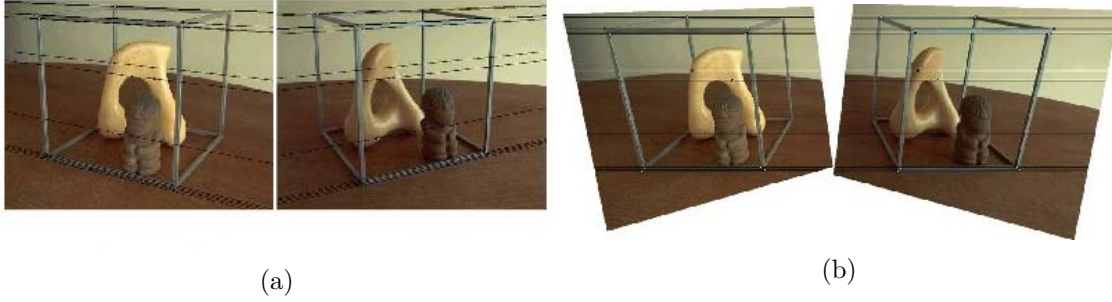


Figure 2.2: Rectification [Sze10]: (a) epipolar lines in left and right image; (b) rectified images with coinciding epipolar lines

Rectification of an image can be achieved by using the camera matrices' focal lengths  $f_x$  and  $f_y$  in pixel units, principal point  $(c_x, c_y)$ , radial distortion coefficients  $k_1, k_2$  and  $k_3$ , tangential distortion coefficients  $p_1$  and  $p_2$ , and the projection matrices' focal lengths and principal points  $f'_1, f'_2, c'_1$  and  $c'_2$ .

In this section, we will describe the concepts of image rectification and image undistortion based on the book of [BK08]. We will show a basic camera model (the pinhole model, see Figure 2.3 and Figure 2.4) and how a point in a 3D scene is projected onto an image plane. We will extend and modify the model to simplify the formulas connected to the model. From that we will show why the parameters  $f_x, f_y, c_x$  and  $c_y$  are needed for depth reconstruction. Further, we will discuss radial and tangential lens distortions and how the distortion coefficients  $k_1, k_2, k_3, p_1$  and  $p_2$  are applied in the undistortion process. Moreover, we will explain the process of camera calibration and how this process obtains the above mentioned distortion parameters, the principle point and the focal lengths.

#### 2.1.2.1 Camera Model

The simplest camera model is the pinhole camera model. It can be used to explain the concepts of intrinsic camera parameters. The combination of intrinsic parameters and extrinsic parameters describe the projective geometry of the camera setup, which will be explained later on. Extrinsic parameters translate and rotate the whole camera system using 6 degrees of freedom: Translation in x, y and z direction and rotation around the corresponding x, y and z axis.

A 3D point  $X$  is projected through the pinhole in the pinhole plane onto the image plane as point  $x$ . The distance between the image plane and the pinhole plane is called the

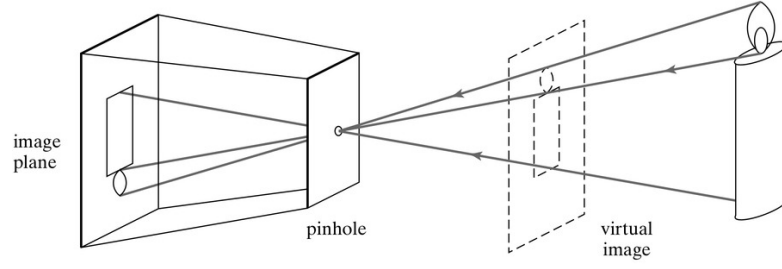


Figure 2.3: Pinhole camera sketch by [FP11]: This image shows how a 3D scene is projected through the pinhole of a pinhole camera onto the image plane.

focal length  $f$ . The orthogonal axis from the image plane through the pinhole is called the optical axis. The distance between  $X$  and the pinhole plane along the optical axis is called the depth  $Z$ . Figure 2.4 illustrates the projection of a point through the pinhole model.

In a real world camera, the image plane might be a photographic plate or some kind of imaging sensor. In the case of an imaging sensor, the sensor usually has some kind of rasterization with a number of pixels per square unit of length, where the pixels can be either rectangular or squared. Therefore, the number of pixels per unit of length in both  $x$  and  $y$  direction are defined as  $s_x$  and  $s_y$ .

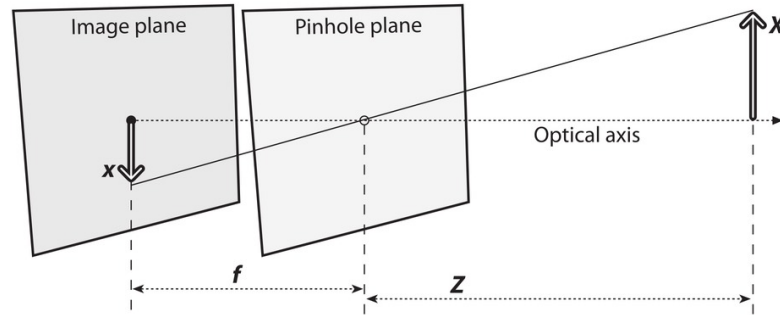


Figure 2.4: Pinhole imaging model [BK08]: A 3D point with  $X$  as its  $x$ -component and the relation between the point  $X$  and its projection  $x$  through similar triangles.

Originating from the pinhole, the relation  $x$  to  $X$  can be modeled via similar triangles  $-x/X = f/Z$  and the relation  $y$  to  $Y$  can be modeled via similar triangles  $-y/Y = f/Z$ . These relations can be rewritten as:

$$-x = f \cdot \frac{X}{Z} \quad -y = f \cdot \frac{Y}{Z} \quad (2.2)$$

By modifying this model in the way that the image plane is virtually moved along the optical axis by two times the focal length  $f$ , a virtual image plane is created. This moves the image plane in front of the camera but preserves the distance between image plane and pinhole plane. Figure 2.5 shows how the translation of the image plane reduces the pinhole to the center of projection and demonstrates the similarity of triangles when looking at a point  $Q$  in the 3D scene and its projection  $q$  onto the virtual image plane. The similarity of triangles is visible in Figure 2.5 where point  $q$  coincides with  $Q$ 's projection line to  $o$ . The relation  $-x/X = f/Z$  is modified to  $x/X = f/Z$  which can be interpreted as tipping of the image plane around the  $y$  axis. This is also true for the  $y$ -coordinate and the  $x$ -axis. This rewrites Formula 2.2 as:

$$x = f \cdot \frac{X}{Z} \quad y = f \cdot \frac{Y}{Z}, \quad (2.3)$$

Due to slight manufacturing errors the center of the image plane i.e. the center of the imaging sensor is not always perfectly aligned with the center of projection. To account for these errors, [BK08] introduced a translational vector  $\mathbf{c}$  containing the correction along the  $x$  and  $y$ -axes  $c_x$  and  $c_y$ .

In order to calculate the pixel position  $q = (x_{\text{px}}, y_{\text{px}})$  of point  $Q = (X, Y, Z)$  in the sampled image, we will use the pixels per unit of length parameters  $s_x$  and  $s_y$  and the point to projection relation in Formula 2.3. The pixel position  $q$  is calculated using the following formula:

$$x_{\text{px}} = f \cdot s_x \cdot \frac{X}{Z} + c_x \quad y_{\text{px}} = f \cdot s_y \cdot \frac{Y}{Z} + c_y \quad (2.4)$$

### 2.1.2.2 Projective Geometry

The projective geometry of a setup consists of intrinsic and extrinsic camera parameters. The extrinsic parameters express the rotation and translation of the camera. We define the parameters  $t_x$ ,  $t_y$  and  $t_z$  as the translation parameters and  $\alpha_x$ ,  $\alpha_y$  and  $\alpha_z$  as the rotation parameters corresponding to the Euclidean axis of their respective index. This means that the extrinsic parameters can be expressed as a single transformation matrix:

$$\mathbf{T_E} = \begin{vmatrix} \mathbf{R}_{\alpha_x} \cdot \mathbf{R}_{\alpha_y} \cdot \mathbf{R}_{\alpha_z} & t_x \\ & t_y \\ & t_z \end{vmatrix} \quad (2.5)$$

The variables  $\mathbf{R}_{\alpha_x}$ ,  $\mathbf{R}_{\alpha_y}$  and  $\mathbf{R}_{\alpha_z}$  are three-by-three rotation matrices of angle  $\alpha_i$  around the axis corresponding to the index of  $\alpha_i$ .

Using Formula 2.4 rewritten to  $X/Z$  and  $Y/Z$ . Further, we can express the 3D position  $(X, Y)$  of a point  $P$  from a given image as a function of the 3D depth of the point  $P$ .

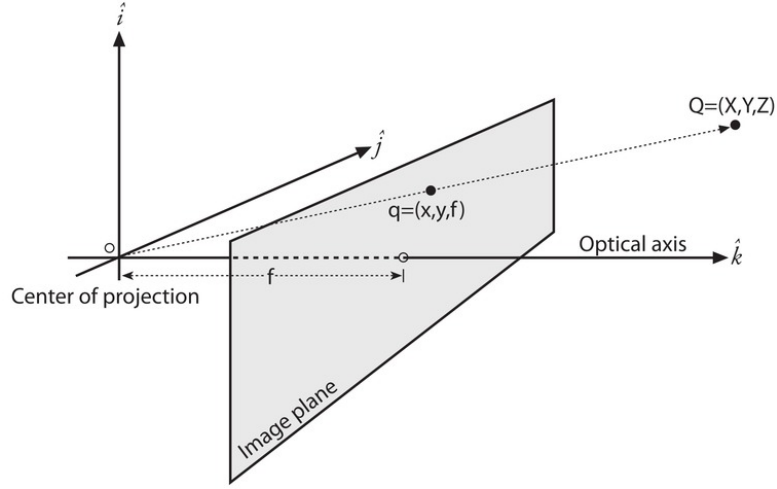


Figure 2.5: Extended pinhole imaging model [BK08]: The image plane in the regular pinhole model is virtually moved along the optical axis by two times the focal length  $f$ . Therefore, the pinhole is reduced to the center of projection  $o$ . The point  $q$  is the projection of a 3D point  $Q$  onto this virtual image plane. The similarity of triangles is visible in this figure at point  $q$  which coincides with  $Q$ 's projection line to  $o$ . The relation  $-x/X = f/Z$  is modified to  $x/X = f/Z$ .

Usually the parameters  $f$ ,  $s_x$ ,  $s_y$ ,  $c_x$  and  $c_y$  are not precisely known. In order to calculate the parameters needed the process of camera calibration is used. We will discuss camera calibration in Section 2.1.2.5. In the book, [BK08] focal lengths in  $x$  and  $y$  direction  $f_x$  and  $f_y$  are introduced as:

$$f_x = s_x \cdot f \quad f_y = s_y \cdot f \quad (2.6)$$

The reason to combine  $f$  and  $s_x$ , and  $f$  and  $s_y$  into  $f_x$  and  $f_y$  is that, with camera calibration only the products  $f_x$  and  $f_y$  can be determined but not their factors. The parameters  $f_x$  and  $f_y$  are unit free pixel counting factors. This can be seen when we look at the meaning of  $s_x|_y$  and  $f$ . The focal length is measured in units of length (e.g. m, mm, inch, etc.) and  $s_x$  and  $s_y$  are a number of pixels per unit of length. This results in "number of pixels" as pseudo unit for  $f_x$  and  $f_y$ . The parameters  $f_x$ ,  $f_y$ ,  $c_x$  and  $c_y$  are used to define the intrinsic camera matrix  $A$  as described by [BK08] and [Sze10]:

$$\mathbf{A} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \quad (2.7)$$

### 2.1.2.3 Radial Distortion

A major problem with pinhole cameras is that very little light reaches the image plane through the pinhole. This results in very high exposure times for every image. In order



to reduce exposure time and still gain enough light, one could suggest to widen the hole of the pinhole camera. This however, leads to a blurred image. The reason for this can be explained with a simple thought experiment.

Let us assume instead of widening the pinhole  $P_0$ , a second pinhole  $P_1$  is added to the pinhole plane. This results in two different sets of vector  $\bar{c}$ , i.e.  $\bar{c}_0$  and  $\bar{c}_1$ . The focal length does not change, because it is dependent on the pinhole plane, and the imaging sensor parameters  $s_x$  and  $s_y$  also do not change, because  $P_0$  and  $P_1$  project onto the same imaging sensor. We can use Formula 2.4 to calculate  $q_0 = (x_0, y_0)$  and  $q_1 = (x_1, y_1)$  for all 3D points  $Q$  in the scene. This results in two unaligned images overlaying one another. Every additional pinhole  $P_i$  would add another unaligned version of the projected scene to the image on the image plane. An argument can be made that a widening of the pinhole  $P_0$  by diameter  $r$  can be approximated by adding multiple additional pinholes within the radius  $r$  surrounding  $P_0$  and therefore, would be an aggregation of infinitely many unaligned projections of the scene onto the image plane.

To solve the problem of wider holes in cameras, lenses can be used to focus the multiple projected images into one projection point. The usage of lenses introduces the problem of lens distortion. Most lenses in cameras are not perfect, and therefore distort the projected image. A common distortion introduced by lenses is radial distortion. This occurs when the used lens is not perfect and the focus point of outer regions of the lens is different to the focus point of inner regions. If the outer part of the lens refracts light more than the inner region (see Figure 2.6), this leads to an effect called "barrel distortion" which can be seen in Figure 2.7. If the outer part of the lens refracts light less than the inner region this leads to "pincushion distortion".

The distortion introduced by the lens' imperfection can usually be approximated by a function in one parameter  $D_R(r)$ . This function represents the refraction property of the lens at radial distance  $r = \sqrt{x^2 + y^2}$  to the center of the lens. This means that, due to the circular form of lenses, the refraction properties of a lens can be modeled by circular level lines of refraction values around the center of the lens. The undistortion function can be approximated by a Taylor series as shown by Brown in [Bro64] and [Bro71]:

$$\begin{aligned} x_{\text{rad,corr}} &= x \cdot (1 + k_1 \cdot r^2 + k_2 \cdot r^4 + k_3 \cdot r^6 + \dots) \\ y_{\text{rad,corr}} &= y \cdot (1 + k_1 \cdot r^2 + k_2 \cdot r^4 + k_3 \cdot r^6 + \dots) \end{aligned} \quad (2.8)$$

For most cases of radial distortion it is sufficient to only use the first few terms of the Taylor series. This results in:

$$\begin{aligned} x_{\text{rad,corr}} &= x \cdot (1 + k_1 \cdot r^2 + k_2 \cdot r^4) \\ y_{\text{rad,corr}} &= y \cdot (1 + k_1 \cdot r^2 + k_2 \cdot r^4) \end{aligned} \quad (2.9)$$

for regular cameras and in

$$\begin{aligned} x_{\text{rad,corr}} &= x \cdot (1 + k_1 \cdot r^2 + k_2 \cdot r^4 + k_3 \cdot r^6) \\ y_{\text{rad,corr}} &= y \cdot (1 + k_1 \cdot r^2 + k_2 \cdot r^4 + k_3 \cdot r^6) \end{aligned} \quad (2.10)$$

for fish-eye cameras.

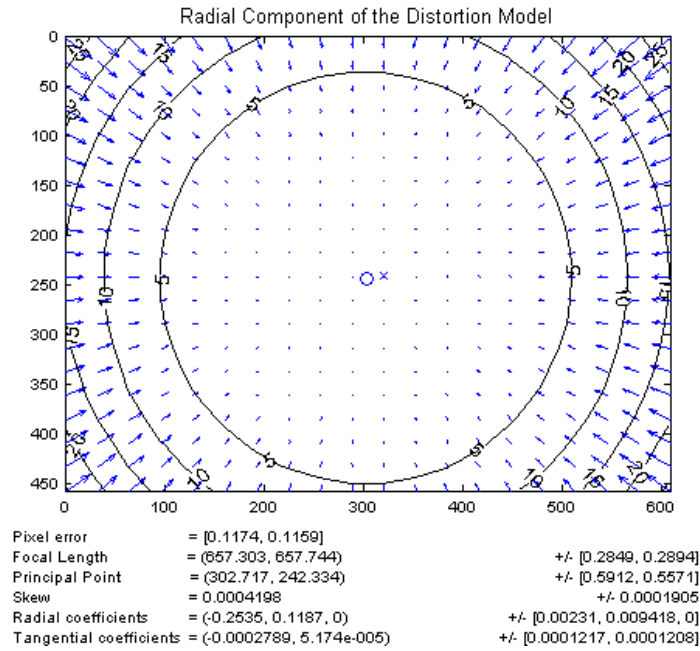


Figure 2.6: Radial Distortion plot (Jean-Yves Bouguet in [Bou]): This plot shows the displacement of pixels sampled in an uniformly spaced grid. The arrows show the displacement direction and their lengths show the strength of the displacement. This is an example visualization of radial distortion.

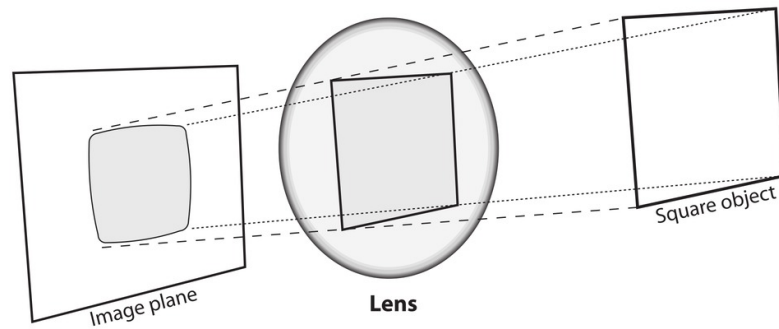


Figure 2.7: Barrel distortion [BK08]: A square object is projected through an imperfect lens onto an image plane. Light rays from the object hit the lens and are bent to focus the image on the image plane. Due to imperfections of the lens, light rays are refracted more strongly, the farther away from the lens center they arrive. Corner rays of the square object are more refracted towards the center of the image than the center edge rays. This lets projections of straight lines seemingly bulge away from the center of the image, creating the barrel distortion.

### 2.1.2.4 Tangential Distortion

Another distortion effect in digital imaging stems from the process how cameras are produced. Inside a digital camera behind a shutter and lens system lies an imaging sensor. This sensor represents the image plane. Due to different production techniques it is not guaranteed that the optical axis aligns with the sensor's normal vector i.e. the sensor not being parallel with the lens. If the sensor and lens are not parallel, then the sampled image will have a trapezoid form, which can be seen in Figure 2.8. This causes the focus points of the lens system to no longer be colinear. This effect is also called 'decentering distortion' [Bro66] and adds a tangential distortion and radial distortion to the image.

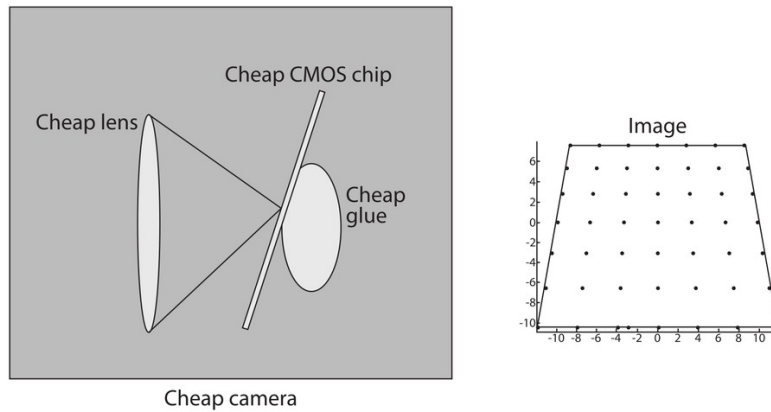


Figure 2.8: Tangential distortion (Sebastian Thrun in [BK08]): An example of how cheap production can lead to tangential distortion. On the left, a cheap camera, where the imaging sensor is not parallel to the lens is shown. On the right an example of how tangential distortion may look is shown.

In the works of [Str12] and [Was57] the thin prism model was used to calculate the distortion introduced by the inherent camera faults. The thin prism model was first proposed by [Con19] and uses the mathematical model of the Seidel aberration defined by [Sei57].

In the article of [Bro66] the thin prism model is extended to handle the additional radial distortion of the lens decentering. The model introduced by [Bro66] is called the "plumb bob model" or the "Brown-Conrady model".

A 2D visualization of tangential distortion has an axis of maximal distortion. Orthogonal to the axis of maximal tangential distortion lies the zero line of tangential distortion. This can be seen in Figure 2.9 where the axis of maximal tangential distortion lies near the y axis and the zero line of distortion is approximately parallel to the x axis, lying at a y value of nearly 220.

To correct the distortion two additional parameters  $p_1$  and  $p_2$  are needed. [BK08] define

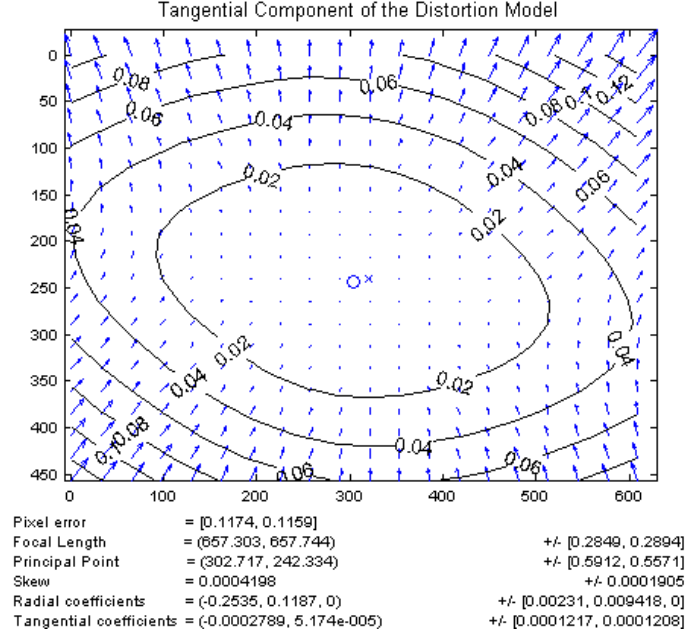


Figure 2.9: Tangential Distortion plot (Jean-Yves Bouguet in [Bou]): This plot shows the displacement of pixels sampled in an uniformly spaced grid. The arrows show the displacement direction and their lengths show the strength of the displacement. This is an example visualization of tangential distortion. The arrows at  $y$  positions with a high value i.e. the lower part of the plot, are pointing towards the center. Arrows on the upper end of the plot i.e. at low  $y$  positions have an outwards direction. This distortion model distorts a rectangle to a trapezoid as shown in Figure 2.8 right.

the correction for tangential distortion as<sup>2</sup>:

$$\begin{aligned} x_{\text{tan,corr}} &= x + [2 \cdot p_1 \cdot x \cdot y + p_2 \cdot (r^2 + 2 \cdot x^2)] \\ y_{\text{tan,corr}} &= y + [p_1 \cdot (r^2 + 2 \cdot y^2) + 2 \cdot p_2 \cdot x \cdot y] \end{aligned} \quad (2.11)$$

Formula 2.11 is based on the work of [Con19], where a horizontal and a vertical component of aberrational displacement are defined by use of Seidel aberrations by [Sei57]. The

---

<sup>2</sup>The book [BK08] in version 1 is erroneous at this point, having  $x_{\text{tan,corr}} = x + [2 \cdot p_1 \cdot y + p_2 \cdot (r^2 + 2 \cdot x^2)]$  and  $y_{\text{tan,corr}} = y + [p_1 \cdot (r^2 + 2 \cdot y^2) + 2 \cdot p_2 \cdot x]$ . Please refer to the official errata of the document: <http://www.oreilly.com/catalog/errata.csp?isbn=9780596516130>

original form of the displacement formula in [Con19] is defined as:

$$\begin{aligned}
 \text{Horiz.comp.} &= g_1 \cdot S^2(2 \cdot \cos(\Phi) + \cos(2 \cdot E - \Phi)) \\
 &\quad + g_2 \cdot S \cdot V(3 \cdot \cos(\Psi) \cdot \cos(E) + \sin(\Psi) \cdot \sin(E)) \\
 &\quad + 3 \cdot g_3 \cdot V^2 \cdot \cos(\chi) \\
 \text{Vert.comp.} &= g_1 \cdot S^2(2 \cdot \sin(\Phi) + \sin(2 \cdot E - \Phi)) \\
 &\quad + g_2 \cdot S \cdot V(\cos(\Psi) \cdot \sin(E) + \sin(\Psi) \cdot \cos(E)) \\
 &\quad + g_3 \cdot V^2 \cdot \sin(\chi)
 \end{aligned} \tag{2.12}$$

The parameters  $g_1$ ,  $g_2$  and  $g_3$  in Formula 2.12 represent the magnitude of three different decentering defects and the corresponding angles  $\Phi$ ,  $\Psi$  and  $\chi$  represent the orientation of the defects. The variable  $V$  is the angle of the field of view and variable  $S$  represents the semi-aperture. The angle  $E$  was defined in earlier work of [Con18], and represents the angle  $PQR$ . The point  $Q$  is the center of a principal pencil ray where it cuts the lens.  $P$  is the axis defined by the lens center and  $Q$ . Point  $R$  is a point lying on the circumference of the pencil ray in  $Q$ .

The first term in Formula 2.12 represents the comatic<sup>3</sup> aberration of the lens and the second term represents the astigmatism of the lens. Conrady argues that the first term of the equation is constant within the field of view and does therefore not affect the relative distances of projected points and can thus safely be ignored. Conrady also argues that the astigmatism is *bound to be very small in any respectable instrument* and can therefore be neglected in the displacement calculation [Con18].

The remaining term is defined as a function of the angle of image orientation relative to the direction of decentration  $\chi$  and the angle of view  $V$ . The angle  $\chi$  can also be understood as the deviation of the angle of the polar form of a given point  $p$  relative to the direction of maximal tangential distortion. The remaining horizontal component of displacement  $\rho$  and the remaining vertical component of displacement  $\tau$  are therefore defined as:

$$\begin{aligned}
 \rho &= 3 \cdot g_3 \cdot V^2 \cdot \cos(\chi) \\
 \tau &= g_3 \cdot V^2 \cdot \sin(\chi)
 \end{aligned} \tag{2.13}$$

Using a base transformation and setting  $\chi$  to the angle of maximum tangential distortion as  $\Phi + \beta$ , the tangential distortion can be expressed as:

$$\begin{vmatrix} \Delta_x \\ \Delta_y \end{vmatrix} = P_c \cdot \begin{vmatrix} (2 \cdot (\frac{d_x}{r})^2 + 1) \cdot \cos(\phi) + 2 \cdot \frac{d_x \cdot d_y}{r^2} \cdot \sin(\phi) \\ (2 \cdot (\frac{d_y}{r})^2 + 1) \cdot \sin(\phi) + 2 \cdot \frac{d_x \cdot d_y}{r^2} \cdot \cos(\phi) \end{vmatrix} \tag{2.14}$$

With  $\cos(\beta) = \frac{d_y}{r}$  and  $\sin(\beta) = \frac{d_x}{r}$ .

---

<sup>3</sup>see [Gat55]

[STH80] express the distortion profile  $p_3 \cdot V^2$  in  $\rho$  and  $\tau$  as a polynomial<sup>4</sup>  $P_c$  in distance  $r$ :

$$p_3 \cdot V^2 = P_c = K_4 \cdot r^2 + K_5 \cdot r^4 \quad (2.15)$$

Using Formula 2.15 [STH80] rewrites Formula 2.14 to<sup>5,6</sup>:

$$\begin{vmatrix} \Delta_x \\ \Delta_y \end{vmatrix} = \begin{vmatrix} 2 \cdot d_x \cdot d_y \cdot p_1 + (2 \cdot d_x^2 + r^2) \cdot p_2 \\ (2 \cdot d_y^2 + r^2) \cdot p_1 + 2 \cdot d_x \cdot d_y \cdot p_2 \end{vmatrix} \quad (2.16)$$

### 2.1.2.5 Camera Calibration

The process of measuring the cameras intrinsic parameters for radial and tangential distortion ( $k_1, k_2, k_3, p_1$ , and  $p_2$  described in Formula 2.10 and Formula 2.16), the focal lengths ( $f_x$  and  $f_y$ ) and translation parameters ( $c_x$  and  $c_y$ ) as described in Formula 2.4 is called camera calibration. For this purpose, one or multiple images of a calibration object are taken. In the multi image case, the images are taken from multiple different orientations and positions. A calibration object is an object with well known properties such that the object's position and orientation can easily be determined in every image. From the discrepancies of how the object should be projected onto the image plane, from the known position and orientation, and the actual projection of the image onto the image plane, the intrinsic parameters and the distortion parameters can be calculated.

A method for camera calibration from one image was shown by [Fau93] and is known as Direct Linear Transformation (DLT). This method is dependent on the calibration object being a 3D calibration rig. In the works of Zhang [Zha00] and [Zha99], the author proposed a method of camera calibration by solving a homogeneous linear system derived from multiple images of a planar calibration object.

The method of [Zha00] is widely used in multiple applications. This is due to its simplicity in preparation of the calibration object and to the availability of a free and well tested implementation of a variation of [Zha00] using the model described by [Bro71] to solve the linear system in the Open Source Computer Vision Library (OpenCV). This is why we will shortly review the method implemented in the OpenCV library as it is described by [BK08].

The method starts by defining the homography matrix for every one of the used images. The homography matrix defines the transformation between the image plane and the model plane, i.e. the checkerboard plane. This matrix can be defined as

---

<sup>4</sup>Correction note for polynomial  $P_c$  in the work of [STH80] page 483:  $K_4 \cdot r^2 + K_5 \cdot r$  should be  $K_4 \cdot r^2 + K_5 \cdot r^4$ .

<sup>5</sup>Correction note for formula (9.30) in the work of [STH80]:  $P_c$  on the right part of the equation should be a scalar factor for the whole vector not only for  $2 \cdot (\sin(\beta)^2 + 1) \cdot \cos(\phi)$ .

<sup>6</sup>Note for formula (9.30) and formula(9.29) in the work of [STH80]: The terms in formula(9.29) should be negative ( $\sin(\alpha - \frac{\pi}{2}) = -\cos(\alpha)$  and  $\cos(\alpha - \frac{\pi}{2}) = \sin(\alpha)$ ), but this is implicitly corrected by formula(9.30) as it uses a rotation matrix equal to our  $-\mathbf{R}$  instead of  $\mathbf{R}$ .

$$\mathbf{H} = \mathbf{A} \cdot \mathbf{E}_T \quad (2.17)$$

Matrix  $\mathbf{A}$  is the intrinsic camera parameter matrix as described in Formula 2.7 and  $\mathbf{E}_T$  is the extrinsic camera parameter matrix from Formula 2.5. The extrinsic matrix can be written as the combination of its column vectors. A projection to the image plane can be written as

$$s \cdot \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \mathbf{A} \cdot \begin{bmatrix} \bar{r}_1 & \bar{r}_2 & \bar{r}_3 & \bar{t} \end{bmatrix} \cdot \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \quad (2.18)$$

Without loss of generality [Zha99] assumes that in the model plane  $Z = 0$ . This reduces Formula 2.18 by column vector  $\bar{r}_3$  and results in

$$s \cdot \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \mathbf{A} \cdot \begin{bmatrix} \bar{r}_1 & \bar{r}_2 & \bar{t} \end{bmatrix} \cdot \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix} \quad (2.19)$$

The homography is rewritten as

$$\mathbf{H} = \begin{bmatrix} \bar{h}_1 & \bar{h}_2 & \bar{h}_3 \end{bmatrix} = s \cdot \mathbf{A} \cdot \begin{bmatrix} \bar{r}_1 & \bar{r}_2 & \bar{t} \end{bmatrix} \quad (2.20)$$

Due to the orthogonality of  $\bar{r}_1$  and  $\bar{r}_2$ , [Zha99] formulate two constraints:

$$\bar{h}_1 \cdot \mathbf{A}^{-T} \cdot \mathbf{A}^{-1} \cdot \bar{h}_2 = 0 \quad (2.21)$$

and

$$\bar{h}_1 \cdot \mathbf{A}^{-T} \cdot \mathbf{A}^{-1} \cdot \bar{h}_1 = \bar{h}_2 \cdot \mathbf{A}^{-T} \cdot \mathbf{A}^{-1} \cdot \bar{h}_2 \quad (2.22)$$

For the closed form solution of the linear system [Zha99] proposed an additional parameter  $\gamma$  for the intrinsic matrix  $\mathbf{A}$  such that  $\mathbf{A}'$  is given as:

$$\mathbf{A}' = \begin{bmatrix} f_x & \gamma & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \quad (2.23)$$

This results in the following matrix  $\mathbf{B}$  used for the closed form solution.

$$\mathbf{B} = \mathbf{A}'^{-T} \cdot \mathbf{A}'^{-1} = \begin{vmatrix} \frac{1}{f_x^2} & -\frac{\gamma}{f_x^2 \cdot f_y} & \frac{c_y \cdot \gamma - c_x \cdot f_y}{f_x^2 \cdot f_y} \\ -\frac{\gamma}{f_x^2 \cdot f_y} & -\frac{\gamma^2}{f_x^2 \cdot f_y^2} + \frac{1}{f_y^2} & -\frac{\gamma \cdot (c_y \cdot \gamma - c_x \cdot f_y)}{f_x^2 \cdot f_y^2} - \frac{c_y}{f_y^2} \\ \frac{c_y \cdot \gamma - c_x \cdot f_y}{f_x^2 \cdot f_y} & -\frac{\gamma \cdot (c_y \cdot \gamma - c_x \cdot f_y)}{f_x^2 \cdot f_y^2} - \frac{c_y}{f_y^2} & \frac{(c_y \cdot \gamma - c_x \cdot f_y)^2}{f_x^2 \cdot f_y^2} + \frac{c_y^2}{f_y^2} + 1 \end{vmatrix} \quad (2.24)$$

In [BK08] the original matrix  $\mathbf{A}$  is used, which works under the assumption that  $\gamma = 0$ .

$$\mathbf{B} = \mathbf{A}^{-T} \cdot \mathbf{A}^{-1} = \begin{vmatrix} \frac{1}{f_x^2} & 0 & -\frac{c_x}{f_x^2} \\ 0 & \frac{1}{f_y^2} & -\frac{c_y}{f_y^2} \\ -\frac{c_x}{f_x^2} & -\frac{c_y}{f_y^2} & \frac{c_x^2}{f_x^2} + \frac{c_y^2}{f_y^2} + 1 \end{vmatrix} \quad (2.25)$$

It can be seen in Formula 2.24 that matrix  $\mathbf{B}$  is symmetric. [Zha99] define a vector  $\mathbf{b}$ , which contains  $\mathbf{B}$ s defining values for later rewriting of Formula 2.21 and Formula 2.22:

$$\bar{\mathbf{b}} = \begin{vmatrix} B_{11} \\ B_{12} \\ B_{22} \\ B_{13} \\ B_{23} \\ B_{33} \end{vmatrix} = \begin{vmatrix} \frac{1}{f_x^2} \\ -\frac{\gamma}{f_x^2 \cdot f_y} \\ -\frac{\gamma^2}{f_x^2 \cdot f_y^2} + \frac{1}{f_y^2} \\ \frac{c_y \cdot \gamma - c_x \cdot f_y}{f_x^2 \cdot f_y} \\ -\frac{\gamma \cdot (c_y \cdot \gamma - c_x \cdot f_y)}{f_x^2 \cdot f_y^2} - \frac{c_y}{f_y^2} \\ \frac{(c_y \cdot \gamma - c_x \cdot f_y)^2}{f_x^2 \cdot f_y^2} + \frac{c_y^2}{f_y^2} + 1 \end{vmatrix} \quad (2.26)$$

Based on the aforementioned assumption of  $\gamma = 0$ , vector  $\bar{\mathbf{b}}$  can be written as

$$\bar{\mathbf{b}} = \begin{vmatrix} \frac{1}{f_x^2} \\ 0 \\ \frac{1}{f_y^2} \\ -\frac{c_x}{f_x^2} \\ -\frac{c_y}{f_y^2} \\ -\frac{c_x^2}{f_x^2} + \frac{c_y^2}{f_y^2} + 1 \end{vmatrix} \quad (2.27)$$

For the mentioned rewriting of Formula 2.21 and Formula 2.22 a vector is needed to map the components of the column vectors of the homography matrix into the factors of the product of Formula 2.22:

$$\overline{v_{i,j}} = \begin{vmatrix} h_{i,1} \cdot h_{j,1} \\ h_{i,1} \cdot h_{j,2} + h_{i,2} \cdot h_{j,1} \\ h_{i,2} \cdot h_{j,2} \\ h_{i,3} \cdot h_{j,1} + h_{i,1} \cdot h_{j,3} \\ h_{i,2} \cdot h_{j,3} \\ h_{i,3} \cdot h_{j,3} \end{vmatrix} \quad (2.28)$$



This allows us to write  $\overline{h_i}^T \cdot \mathbf{B} \cdot \overline{h_j}$  as

$$\overline{h_i}^T \cdot \mathbf{B} \cdot \overline{h_j} = \overline{v_{i,j}}^T \cdot \overline{b} \quad (2.29)$$

With this the constraints in Formula 2.21 and Formula 2.22 can be rewritten as

$$\left| \frac{\overline{v_{1,2}}^T}{(\overline{v_{1,1}} - \overline{v_{2,2}})^T} \right| \cdot \overline{b} = \mathbf{V} \cdot \overline{b} = \overline{0} \quad (2.30)$$

$\mathbf{V} \cdot \overline{b} = \overline{0}$  is solved by calculating the eigenvector of  $\mathbf{V}^T \cdot \mathbf{V}$ . From vector  $\overline{b}$  matrix  $\mathbf{A}$  can be calculated. This in turn allows the calculation of the rotation and translation column vectors in Formula 2.18.

After the calculation of the camera parameters the distortion parameters are calculated. This is done by a combination of Formula 2.10 and Formula 2.16 given by [BK08]

$$\begin{vmatrix} x_p \\ y_p \end{vmatrix} = (1 + k_1 \cdot r^2 + k_2 \cdot r^4 + k_3 \cdot r^6) \cdot \begin{vmatrix} x \\ y \end{vmatrix} + \begin{vmatrix} 2 \cdot p_1 \cdot x \cdot y + p_2(r^2 + 2 \cdot x^2) \\ p_1(r^2 + 2 \cdot y^2) + 2 \cdot p_2 \cdot x \cdot y \end{vmatrix} \quad (2.31)$$

Using multiple points in multiple images a list of equations is collected and solved for the distortion parameters. This is followed by a reestimation of the intrinsic and extrinsic parameters using the newly calculated distortion parameters for correction of the point positions.

### 2.1.2.6 Fronto-Parallel Assumption

A central assumption in stereo matching is the FPA as explained by [EE14]. The basic idea is that a point's appearance will change when it is being looked at from different vantage points. Based on this it is assumed, that all pixels in one image that belong to the same object have the same disparity when matched to the second image. In other words, the cost value of all pixels belonging to the same object should be minimal at the same disparity value  $d$ . The FPA also results in pixels of an object to align the same way in both images, which means that a patch of pixels in one image will perfectly correlate to a patch of pixels in the second image. Patch wise matching of pixels under the FPA gives the matching algorithm robustness against noise.

There are multiple problems with this assumption. One is shown by the sketch of [EE14] in Figure 2.10. The problem of the sketch comes from the fact that in reality, not all objects are aligned perfectly parallel to the camera planes. The necessity arises that there have to be multiple disparity levels within one patch. This in turn faults the FPA. Another problem is that, the mentioned perfect correlation of a patch of pixels, needs all of the patches pixels to belong to the same object. This leads to problems for any approach that does not consider object boundaries and relies on the FPA.

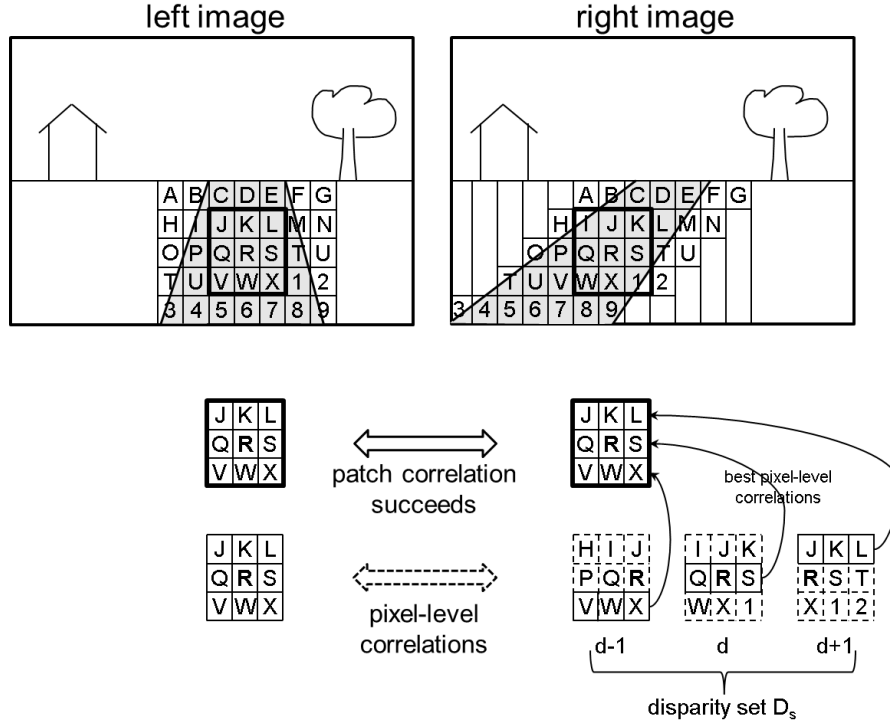


Figure 2.10: Fronto-Parallel Assumption (FPA) Problem [EE14]: This sketch shows the problem of FPAs, that the "street" object is not parallel to the image planes of the two cameras. This results in multiple disparity levels within one image patch.

### 2.1.3 Disparity Estimation and Disparity Maps

The operation of stereo matching can be described as the process of matching two images  $I_0$  and  $I_1$  to one another.  $I_0$  and  $I_1$  are projections of a 3D scene onto two image planes (see Figure 2.1). For any point  $x_0$  a matching point  $x_1$  is searched for. To match two images means to match as many pixels of  $I_0$  as possible, as well as possible to the pixels of  $I_1$ . A match is considered to be a perfect match, if the pixel in  $I_0$  and the pixel in  $I_1$  are projections of the same point in the 3D scene. The absolute distance of the matching points  $d = |x_0 - x_1|$  is called disparity. An image with a disparity value for point  $(x, y)$  at pixel  $(x, y)$  is called a disparity map. The 3D depth component  $Z$  can be calculated from  $d$  with the length of the baseline  $B$  between the cameras which is the length of the translation vector  $t$ , and  $f$  as the focal length:

$$Z = f \cdot \frac{|t|}{d} = f \cdot \frac{B}{d} \quad (2.32)$$

In order to formalize the comparison of dense stereo matching methods [SSZ01] introduced four building blocks to map stereo matching algorithms. These building blocks or a

subset of thereof, generally can be mapped to the steps of most dense stereo matching methods. These steps are listed below:

- Matching cost computation
- Disparity computation / optimization
- Cost (support) aggregation
- Disparity refinement

Some of the ideas in stereo matching are best explained by using the concept of Disparity Space Images which will be discussed in this section as well.

### 2.1.3.1 Cost Functions

A cost function in stereo matching is a function that expresses the dissimilarity between two image patches  $p_0$  and  $p_1$ . The lower the cost, the higher the chance that the examined patches are projections of the same part of the scene. The Sum of Absolute Differences (SAD) and the Sum of Squared Differences (SSD) are among the most used dissimilarity measures. Usually the cost of matching a patch  $p_0$  with itself, should be the lowest possible value for the used cost function. In general, intensity values of the pixels of the compared patches are used to determine the costs. Depending on the cost function the input patches may be represented in various color spaces e.g. gray scale, RGB, HSV.

For most patch based cost functions, the patch size  $s$  has to be the same for  $p_0$  and  $p_1$ . In general,  $s$  has a constant value for all compared patches. Every patch has a center point  $(x, y)$ . Pixel based cost functions can be interpreted as patch based cost functions with a window size of 1. For a patch at  $(x, y)$  in  $I_0$  the matching costs for multiple patches in  $I_1$  have to be calculated. In a rectified setup, only the patches in  $I_1$  at position  $(x', y') = (x + d, y)$  with  $d$  running from 0 to a maximal disparity ( $d_{\max}$ ) have to be considered for matching.

Therefore, the cost function is a function of point  $(x, y)$  and disparity  $d$ :  $C(x, y, d)$

### 2.1.3.2 Disparity Space Images

A Disparity Space Images (DSI) is the collection of values that correspond to the dimensions, width and height ( $w \times h$ ) of the two images, over multiple discrete disparity steps. A DSI of the cost function is also called a cost volume. Figure 2.11 shows how the left and right image are combined into a DSI by using a function  $G(x, y, d)$ .

The left image (blue) and the right image (green) in Figure 2.12 are combined into a DSI using a function  $G(x, y, d)$ . The left lower part of the graphics 2.12(a) and 2.12(b) shows that  $G(x, y, d)$  overlays the left and right image with a shift of  $d$  pixels. The overlapping region (marked by the light blue border) may produce results for  $G(x, y, d)$  (aggregation

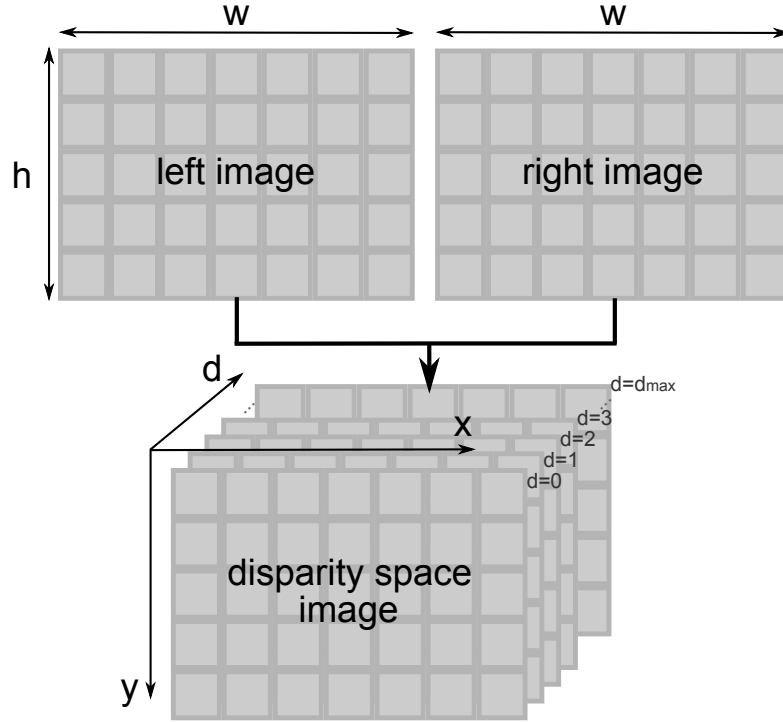


Figure 2.11: Disparity Space Images: In stereo matching the left and right image are usually of the same width ( $w$ ) and height ( $h$ ). A Disparity Space Images (DSI) is a memory volume of size  $w \times h \times D$ . The value of  $D$  is the number of considered disparity steps and is usually  $D = \lceil [d_{\min}; d_{\max}] \rceil$ . In most cases  $d_{\min}$  is set to 0, which results in  $D = d_{\max} + 1$ . This memory volume is filled with a function  $G(x, y, d)$ .

steps discussed in Section 2.1.3.3 not taken into account). The results of  $G(x, y, d)$  for the overlapping area are written to the DSI's corresponding disparity layer, also highlighted by light blue borders. If  $x$  is traversed from 0 to  $w$ , then  $x$  reaches the point  $x = w - d + 1$  where due to the disparity shift no more data of the left image is available to be compared to the right image. A common strategy to deal with this problem is to replicate the last column, marked in yellow in the upper left part of graphics 2.12(a) and 2.12(b), of the left image  $d$  times and using this data as if  $(x, y, d)$  was still in the overlapping region. This results in a part of the DSI (marked in yellow) that might not be as meaningful as the rest of the DSI. Figure 2.12(a) shows how a DSI layer for  $d = 1$  with one replication of the last column is filled. Figure 2.12(b) shows how a DSI layer for  $d = 4$  with four replications of the last column is filled.

### 2.1.3.3 Support Aggregation

As stated by [HIG02],[HZW<sup>+</sup>10], and [Hir05], single pixel matching costs are often ambiguous due to noise or repetitive structures in the image. A known problem is that,

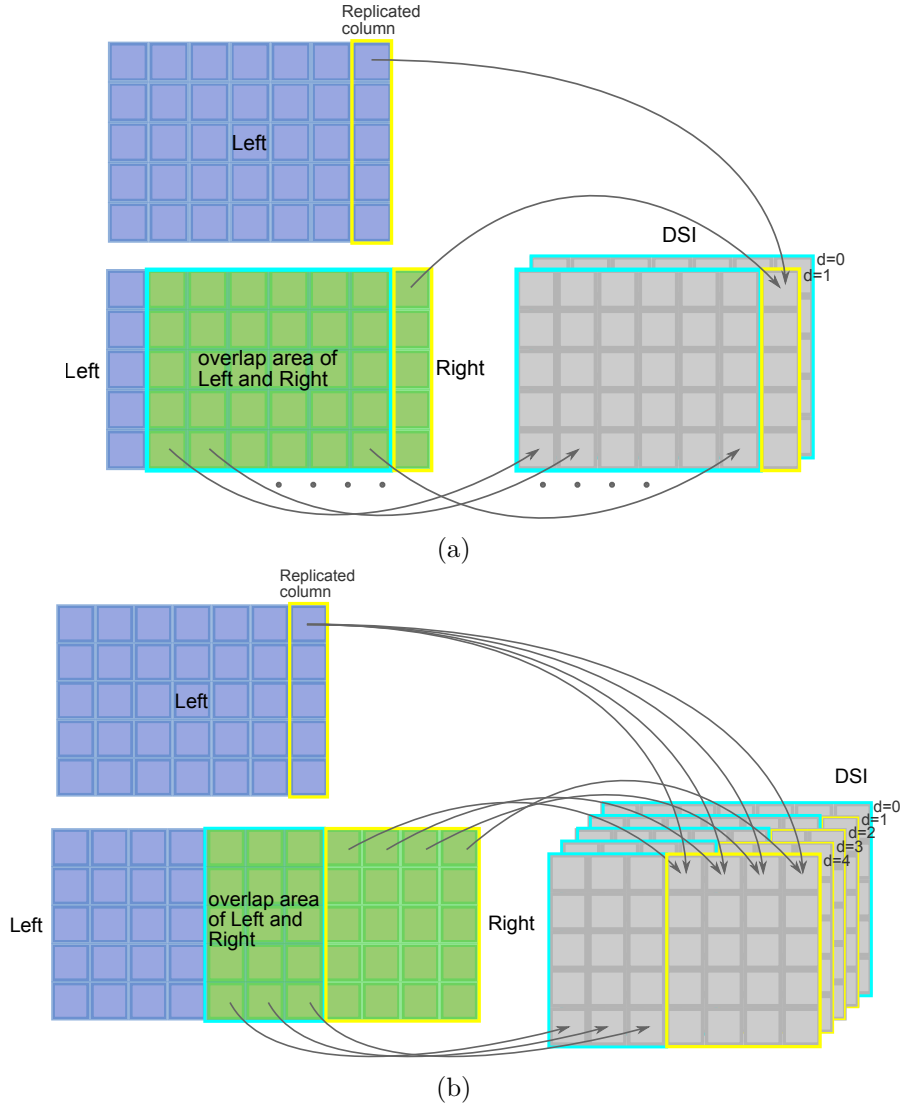


Figure 2.12: The left image (blue) and the right image (green) are combined into a DSI using a function  $G(x, y, d)$ . The results of  $G(x, y, d)$  for the overlapping area are written to the DSI's corresponding disparity layer, highlighted by light blue borders. At point  $x = w - d + 1$  no more data of the left image is available to be compared to the right image due to the disparity shift. (a): This part of the figure shows how DSI layer  $d = 1$  is filled with one replication of the last column and the costs of the pixels in the overlapping area. (b): This part of the figure shows how DSI layer  $d = 4$  is filled with four replications of the last column and the costs of the pixels in the overlapping area.

due to noise, a non-matching pixel can have lower matching costs than the correct match. To lower the chance of mismatched pixels, a smoothness term is added to the cost function. This smoothness term works under the assumption that, within a region around a pixel, neighboring pixels have a similar disparity value (i.e. the FPA) and therefore penalizes high disparity steps. Because of the assumption, that the disparity values of the pixels within a support window should be the same, it is possible to calculate multiple support aggregations for different disparities in prior.

A DSI of a support aggregation of a cost function is also called an aggregation cost volume.

The work of [TMDSA08] gives a short overview on different cost aggregation methods and evaluate these methods based on accuracy and computational requirements. We will shortly review some of the ideas for support aggregation i.e. rectangular windows, varying window sizes, multiple window selection, unconstrained shapes, weight variation and adaptive weights, as they were reviewed by [TMDSA08].

**Rectangular Windows** In a rectified setup with two rectified images  $I_0$  and  $I_1$ , a reference point  $p = (x, y)$  in  $I_0$ , a target point  $q = (x + d, y)$  in  $I_1$ , with a cost function  $C(x, y, d)$  as described in Chapter 2.1.3.1, the matching costs for multiple points surrounding  $p$  can be calculated and aggregated. The authors of [TMDSA08] denote such an aggregation window of size  $n$  as  $w_n^{I_0}(i, j)$  and  $w_n^{I_1}(i, j)$  respectively for  $I_0$  and  $I_1$  centered on point  $(i, j)$ . They also introduce  $W_n(i, j, d)$  as the pair  $w_n^{I_0}(i, j), w_n^{I_1}(i + d, j)$  and they denote  $S_V(p, q) = S_V(x, y, d)$  as the aggregation of the cost function results over one of these windows.

Aggregation in this context is usually the summation of the cost function of the corresponding pixels in the window pair. For readability we denote  $x_n$  and  $y_n$  as the  $n^{\text{th}}$   $x$  and  $y$  position in an arbitrary enumeration of the pixel positions in a given window.

$$S_V(x, y, d) = \sum_{n=0}^N C(x_n, y_n, d) \quad (2.33)$$

**Varying Window Size and Offset** There exist multiple strategies to improve accuracy of stereo correspondence by using a set of windows and selecting the best support value  $S_V(x, y, d)$  of this set. The set of windows is denoted by [TMDSA08] as  $S(p, q) = S(x, y, d)$ .

Two commonly used ideas for the creation of images is the displacement of the windows

$$S(x, y, d) = \{W_n(i, j, d) : i \in [x - n, x + n], j \in [y - n, y + n]\} \quad (2.34)$$

and the variation in the windows size.

$$S(x, y, d) = \{W_n(x, y, d) : n \in [N_{\min}, N_{\max}]\} \quad (2.35)$$

A more general combination of these two ideas is:

$$\begin{aligned} S(x, y, d) = \{ & W_n(i, j, d) : i \in [x - n, x + n], \\ & j \in [y - n, y + n], \\ & n \in [N_{\min}, N_{\max}] \} \end{aligned} \quad (2.36)$$

**Multiple Window Selection** After calculating multiple support values for a set  $S(p, q)$  more accuracy can be gained by using more than one of the calculated support values. Different strategies for this include selection, based on the distance to a depth edge in the examined window and on which side of this depth edge  $p$  and  $q$  lie. Another proposed selection scheme is defined as

$$S(x, y, d) = W_n(x, y, d) \cup \{W_n(x \pm n, y \pm n, d)\} \quad (2.37)$$

A method to unify multiple selected window sets, is to calculate the average of the cost function.

**Unconstrained Shapes** In order to better adapt to the characteristics of the data, [BVZ98] proposed a variable window approach using the idea of not constraining the window shape to be rectangular. There are different techniques to determine the best window shape for a pixel.

[BVZ98] used a plausibility measure based on photometric properties to determine whether a pixel belongs to the surrounding area of pixel  $p$ . For best results the largest set of plausible connected pixels was chosen to constitute the shape.

In the work of [Vek01] the shape of the support window is represented as a polygonal line around  $p$  and in the work of [GB06] a segmentation method is used to split the images into segments. The support window's shape for a point  $p$  is expressed as the intersection of the segment belonging to  $p$  and a squared window around  $p$ .

**Weight Variation and Adaptation** Another idea for better support values is the use of weights. According to [TMDSA08], weights can be used for example to penalize high matching costs more or less depending on the corresponding pixels distance to  $p$ . The variation of weight depending on the distance to the center point of the window or, more general, depending on the position in the window, allows for more complex cost and error measures to be devised.

Other strategies of determining weights for support aggregation may be dependent on the local data around point  $p$  e.g. distance to color discontinuities or color gradients.

#### 2.1.3.4 Disparity Estimation

As stated by [Sze10], disparity computation and optimization is one of the four basic steps in most stereo matching algorithms. The usual approach is a minimization of the

cost function, to find the best disparity for any pixel. In order to find the minimum of the cost function, cost volumes or aggregation cost volumes can be used. To find the minimal cost for every pixel, the used cost volume  $V$ , is traversed in disparity direction to find the smallest value per pixel position  $(x, y)$ . Figure 2.13 shows a cost volume and how it can be traversed in disparity direction  $d$  to find the minimal i.e. best cost value. This can be represented as

$$d_{x,y} = \operatorname{argmin} V_{x,y}(d) \quad (2.38)$$

where  $V_{x,y}$  represents all cost values in cost volume  $V$  for position  $(x, y)$ .

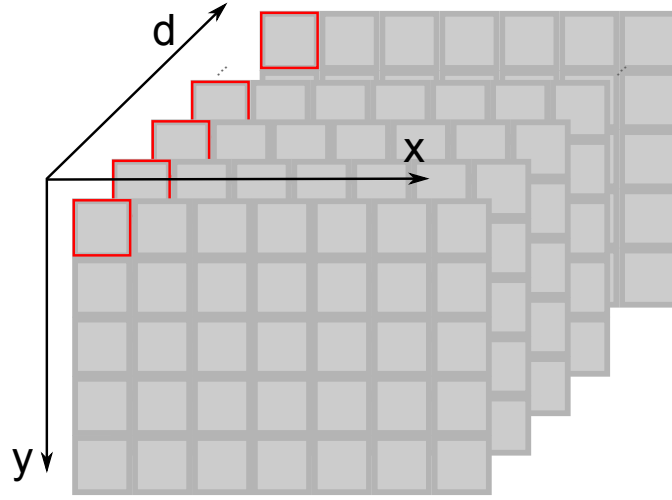


Figure 2.13: Visualization of a cost volume with cost values for  $x$  and  $y$  positions over the disparity  $d$ . To find the best disparity for one position  $(x, y)$ , the cost volume can be traversed in direction  $d$  at position  $(x, y)$  to find the disparity level with the best cost value. This is illustrated by the red highlighted pixels along the axis  $d$ . This can also be applied to aggregation cost volumes.

### 2.1.3.5 Disparity Refinement

A measure to improve the quality of a calculated disparity map is called disparity refinement. This includes cross correlation checks and sub pixel refinements.

A cross correlation check in stereo matching is a check of validity for a disparity value  $d$  at a position  $(x, y)$ . This is based on the assumption that the given disparity map  $D_0$ , was calculated by taking patches from  $I_0$  and matching them to  $I_1$ . A second disparity map  $D_1$ , is calculated, using patches in  $I_1$  and matching them to  $I_0$ . The disparity value at position  $(x, y)$  in  $D_0$  is denoted as  $D_0[x, y]$ . A pixel  $(x, y)$  passes the cross correlation



check if the absolute difference of its disparity value  $D_0[x, y]$  and the disparity value of the allegedly corresponding pixel in  $D_1$  fall below a threshold  $T$ :

$$|D_0[x, y] - D_1[x + D_0[x, y], y]| < T \quad (2.39)$$

In the work of [HZW<sup>+</sup>10] a seemingly arbitrary threshold of 5 is chosen. We argue that the threshold for the disparity cross correlation check should be a function of the maximal disparity  $d_{\max}$ . The parameter  $d_{\max}$  is in turn the result of a function of the image width  $w$ . For the Middlebury Data Sets (see Section 2.2) which provide a  $d_{\max}$  for every data set, the maximal disparity is roughly:

$$d_{\max} \approx \frac{w}{10} \quad (2.40)$$

Based on the given  $d_{\max}$  parameters of the Middlebury data sets, ranging from approximately  $450 \times 700$  pixel images to approximately  $1800 \times 2800$  pixel images, a threshold of  $T = 5$  is equivalent to  $(2 \text{ to } 6)\% \cdot d_{\max}$ .

Sub pixel refinement is an attempt to improve the result of a function that is working on discrete data (e.g. gray values or pixel indexes) and is delivering discrete output (e.g. pixel indexes or pixel exact disparity values) by trying to estimate the result of the corresponding continuous function, using multiple results of this discrete function. This can be achieved through aggregation functions like weighted means or polynomial interpolation.

## 2.2 Middlebury Benchmark

There are many stereo benchmarks available e.g. [GLU12] provided the Karlsruhe Institute of Technology and Toyota Technological Institute (KITTI) benchmark, [KNM<sup>+</sup>14] provided the Heidelberg Collaboratory for Image Processing (HCI) benchmark suite and [SSZ01] provided the *Middlebury Stereo Benchmark*<sup>7</sup>.

Due to the fact that the *Middlebury Stereo Benchmark* provides more than one runtime metric (many benchmarks provide only one metric), is freely accessible, and provides a relatively easy to use framework for executing the benchmark, our main evaluation is done with the *Middlebury Stereo Benchmark*.

In this section we will explain the taxonomy for measurements in stereo matching. We will shortly present the metrics used by [SSZ01]. This will include a comparison of sparse versus dense stereo matching, error metrics and evaluation masks "nonocc" versus "all". The metrics presented by [SSZ01] were implemented in an online system, providing data sets and evaluation tools for online and local use.

<sup>7</sup>see <http://vision.middlebury.edu/stereo/>

After this we will present the data sets that are provided at the website of the university of Middlebury <sup>8</sup> and the available resolutions and qualities of these data sets.

### 2.2.1 Sparse and Dense Stereo Matching

As mentioned in Section 2.1.3, stereo matching algorithms are categorized according to their results into sparse and dense algorithms. A dense algorithm is a method that tries to solve the correspondence problem for every pixel in the image. A sparse algorithm only solves the correspondence problem for specific points or objects in the scene.

Under this premise it is not clear how a sparse algorithm can be compared to a dense algorithm or even how a sparse algorithm can be compared to a different sparse algorithm. This ambiguity results from the fact that for one set of images  $S_I = \{I_0, I_1\}$  the number of calculated pixels, as well as the quality of the result may vary greatly. The remaining question is, whether and how a result with only few but highly accurate result pixels is comparable to a result with many result pixels with a lower accuracy. The original work proposed by [SSZ01] states that this problem is outside the scope of their then current work and focused therefore on dense stereo matching. This is why the *Middlebury Stereo Benchmark* only supported submissions for dense algorithms.

The latest version of the *Middlebury Stereo Benchmark* (at this point version 3), introduced in May 2015)<sup>9</sup> now supports the submission of sparse algorithm results. Submissions are now allowed to have both dense and sparse results, or only either one of them. The strategy to allow this, implemented in the *Middlebury Stereo Benchmark*, is a scanline based hole filling algorithm to create dense results from sparse results in case only sparse results are provided and the usage of the dense results in the sparse table if only dense results are provided. In order to make sparse and dense results distinguishable in the table of sparse results, the percentage of invalid pixels is displayed alongside the results.

### 2.2.2 Error Metric

In this section, we will shortly discuss the error metrics for quality and runtime evaluation, which are used in the *Middlebury Stereo Benchmark*. These metrics are: *percentage of bad pixels*, *average absolute error*, *root-mean-square disparity error*, *X-percentiles*, *absolute time*, *time per number of pixels* and *time per disparity hypotheses*.

#### 2.2.2.1 Percentage of Bad Pixels

The percentage of bad pixels metric  $P$  represents the number of pixels where the calculated disparity ( $d_C$ ) and the ground truth disparity ( $d_T$ ) at this pixel's position differ more than a chosen  $\delta_d$ . This is normalized to the total number of pixels  $N$ . In [SSZ01] this was proposed with a fixed  $\delta_d = 1$ . The first version of the benchmark was built according to these specifications.

---

<sup>8</sup>see <http://vision.middlebury.edu/stereo/data/>

<sup>9</sup>see <http://vision.middlebury.edu/stereo/eval3/MiddEval3-newFeatures.html>

$$P = \frac{1}{N} \sum_{(x,y)} (|d_C(x,y) - d_T(x,y)| > \delta_d) \quad (2.41)$$

Since version 2 of the *Middlebury Stereo Benchmark*<sup>10</sup> multiple thresholds  $\delta_{0.5}$ ,  $\delta_{0.75}$ ,  $\delta_{1.0}$ ,  $\delta_{1.5}$  and  $\delta_{2.0}$  are introduced. In version 3 the provided thresholds are changed to  $\delta_{0.5}$ ,  $\delta_{1.0}$ ,  $\delta_{2.0}$  and  $\delta_{4.0}$ .

### 2.2.2.2 Root-Mean-Square Disparity Error and Average Absolute Error

The Root-Mean-Square Disparity Error (Root-Mean-Square) is available in the *Middlebury Stereo Benchmark* since version 1 and was proposed as metric in [SSZ01] as

$$\text{RMS} = \left( \frac{1}{N} \sum_{(x,y)} |d_C(x,y) - d_T(x,y)|^2 \right)^{\frac{1}{2}} \quad (2.42)$$

with  $N$  as the total number of pixels and  $d_C$  and  $d_T$  as the calculated disparity and the ground truth of the disparity.

An additional statistical error metric, the average absolute error  $A$ , was defined as

$$A = \frac{1}{N} \sum_{(x,y)} |d_C(x,y) - d_T(x,y)| \quad (2.43)$$

and is available in the *Middlebury Stereo Benchmark* since version 3.

### 2.2.2.3 X-Percentiles

Another metric added to the *Middlebury Stereo Benchmark* since version 3 is the percentile of the absolute disparity error. The available percentiles are 50%, 90%, 95% and 99%. An x-percentile of a series of numbers is the value  $k$  at which x percent of the values of the series are below the value  $k$ .

### 2.2.2.4 Absolute Time

The current version (version 3) of the *Middlebury Stereo Benchmark*, provides three different metrics for runtime comparison. The first runtime metric, is the absolute runtime for an image set. This is the runtime measurement taken by the submitter and is the absolute time in seconds that their algorithm needed to complete.

### 2.2.2.5 Time per Number of Pixels

The second and third runtime metrics are a function of the first runtime metric. The second metric, the time per number of pixels  $T_P$ , is the runtime value normalized to the

<sup>10</sup>see <http://vision.middlebury.edu/stereo/eval/newFeatures.html>

number of pixels for the current image set. With the runtime  $t$  and the number of pixels  $N$  this is

$$T_P = \frac{t}{N} \quad (2.44)$$

The visualization of the *Middlebury Stereo Benchmark* has a normalization to seconds per million pixels or Mega Pixel (MP) instead of seconds per pixel.

#### 2.2.2.6 Time per Disparity Hypotheses

The third runtime dependent metric is the time per disparity hypotheses  $T_D$ . The total runtime of the method is normalized to the number of disparity hypotheses, which is the number of pixels  $N$  times the disparity depth  $d_{\max}$ .

$$T_D = \frac{t}{N \cdot d_{\max}} \quad (2.45)$$

The visualization of the *Middlebury Stereo Benchmark* has a normalization to seconds per billion pixels or Giga Pixel (GP) instead of seconds per pixel.

The normalization of the runtime  $t$  to dimensions of the problem space i.e. image dimensions  $x$  and  $y$ , and disparity depth  $d_{\max}$ , has the advantage over the not normalized runtime  $t$  in expressiveness as it allows for runtime estimation of future problems when image dimensions and disparity depth are known.

#### 2.2.3 Evaluation Masks

In the *Middlebury Stereo Benchmark*, evaluation masks are used to prevent or allow the evaluation of the disparity value at a specific position. They consist of a gray scale map of the image where an intensity value of 255 means that the pixel's disparity value should be evaluated and any other value means that the disparity value at this pixel position should not be evaluated. Evaluation in this context means, that the error metrics are calculated and counted towards the result and the value of  $N$  in Formula 2.41 to Formula 2.45.

The *Middlebury Stereo Benchmark* provides two evaluation masks, "nonocc" and "all". The label "all" designates a map with all pixels having a value of 255, which leads to evaluation for every pixel. The label "nonocc" stands for "*non-occluded pixels visible in both views*". This mask also provides different mask values for pixels that are only visible in one image but not the other, and for pixels where the disparity value in the ground truth could not be determined. This allows for later versions of the benchmark to distinguish between values that are not available in the ground truth and values that are not determined through correspondence of two pixels.

#### 2.2.4 Data Sets

The *Middlebury Stereo Benchmark* provides training data for different versions of the benchmark. The training sets are provided with a left and right image and ground truth disparity maps. The different data sets are labeled by the year of publication and can

be found on the website<sup>11</sup> of the *Middlebury Stereo Benchmark*. The available data sets are labeled as "2001 datasets", "2003 datasets", "2005 datasets", "2006 datasets" and "2014 datasets". In this section we will shortly talk about these data sets, the provided resolutions and quality traits.

#### 2.2.4.1 2001 datasets

The data set "2001 datasets" in Figure 2.14 was created by D. Scharstein, P. Ugbabe, and R. Szeliski for the work of [SSZ01]. It contains eight image sets of which the set "map" is provided by courtesy of *Microsoft Research* and the set "tsukuba" is provided by courtesy of the *University of Tsukuba*. The set "tsukuba" has a resolution of  $384 \times 288$  pixels and the set "map" has a resolution of  $284 \times 216$  pixels. The remaining sets have a resolution of  $434 \times 380$  pixels. The ground truth disparities are scaled with factor eight i.e. a disparity value of 80 corresponds to a disparity of 10 pixels.

#### 2.2.4.2 2003 dataset

The data set "2003 datasets" in Figure 2.15 was created by D. Scharstein, A. Vandenberg-Rodes, and R. Szeliski using the method described by [SS03]. The two sets are available in the resolutions  $1800 \times 1500$  pixels (F - full size),  $900 \times 750$  pixels (H - half size) and  $450 \times 375$  pixels (Q - quarter size). The disparity ground truth, available for the quarter size resolution sets, is scaled by factor four i.e. disparity value 40 corresponds to a disparity of 10 pixels. A disparity value of 0 encodes "unknown" disparities.

#### 2.2.4.3 2005 dataset

The data set "2005 datasets" in Figure 2.16 was created by A. Blasiak, J. Wehrwein, and D. Scharstein using the method described by [SP07] and [HS07]. The nine sets are available in the resolutions  $1330$  to  $1390 \times 1110$  pixels (full size),  $665$  to  $695 \times 555$  pixels (half size) and  $443$  to  $463 \times 370$  pixels (third size) (resolutions  $r = 3$ ). Each set contains seven views ( $v = 7$ ), three different illumination situations ( $i = 3$ ) and three different exposure times ( $e = 3$ ), resulting in  $v \cdot r \cdot e \cdot i = 189$  different images available per set. The disparity ground truth, is available for view 1 and 5. The ground truth scale factor is 1 for full resolution, 2 for half resolution and 3 for third resolution. For the image sets "Computer", "Drumsticks" and "Dwarves" the disparity ground truth was withheld.

#### 2.2.4.4 2006 dataset

The data set "2006 datasets" in Figure 2.17 was created by B. Hiebert-Treuer, S. Al Nashashibi, and D. Scharstein using the method described by [SP07] and [HS07]. The 21 sets are available in the resolutions  $1240$  to  $1396 \times 1110$  pixels (full size),  $620$  to  $698 \times 555$  pixels (half size) and  $413$  to  $465 \times 370$  pixels (third size) (resolutions  $r = 3$ ). Corresponding to "2005 dataset", each set of "2006 dataset" contains seven views ( $v = 7$ ),

<sup>11</sup>see <http://vision.middlebury.edu/stereo/data/>

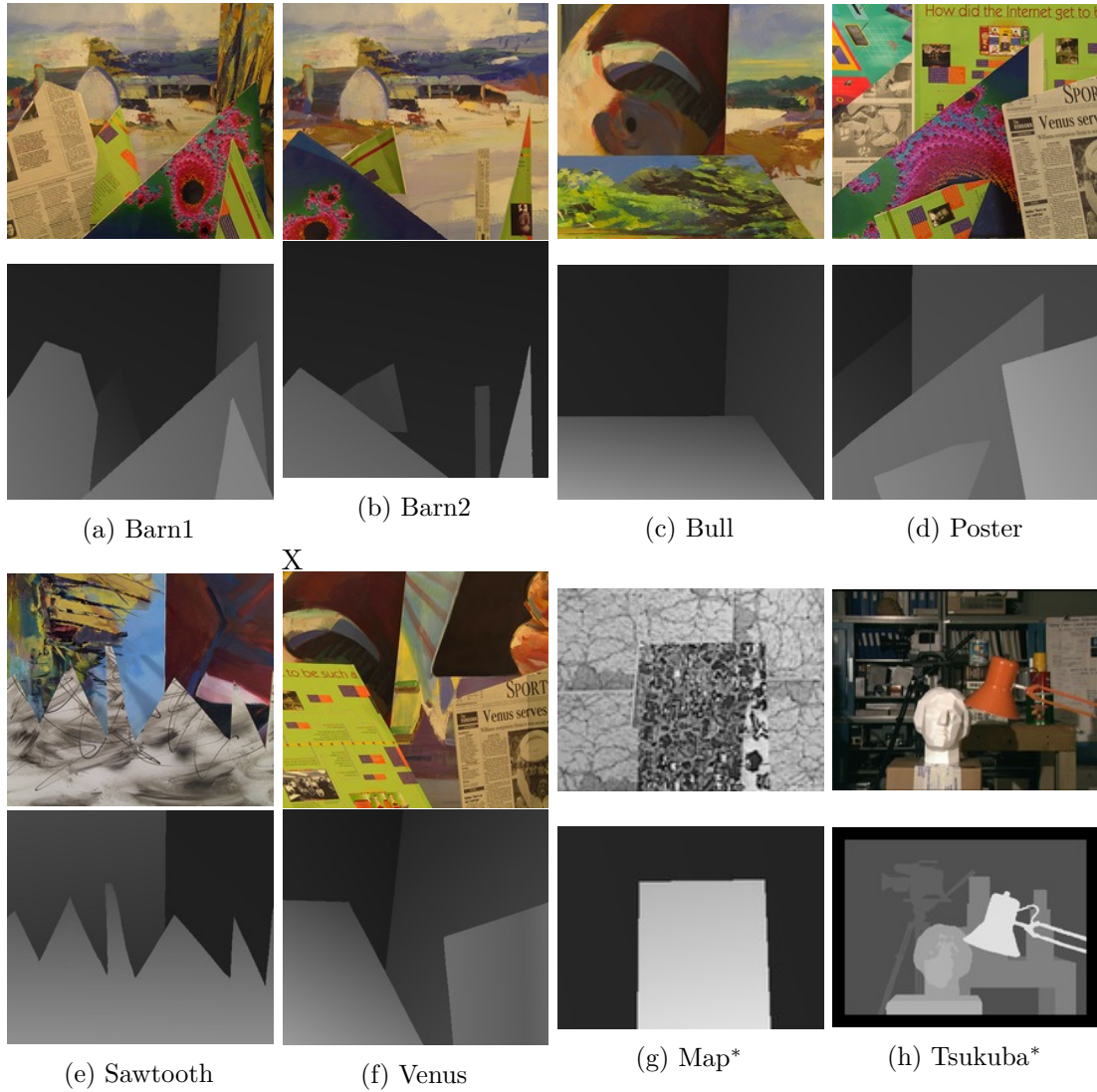


Figure 2.14: *Middlebury Stereo Benchmark* data set: "2001 dataset". The images "tsukuba" and "map", marked with an asterisk (\*) were created by different authors. Both image sets were used in the work of [SZ99]. The image set "tsukuba" is by courtesy of the University of Tsukuba and the image set "map" is by courtesy of Microsoft Research.



(a) Cones

(b) Teddy

Figure 2.15: *Middlebury Stereo Benchmark* data set: "2003 dataset"


(a) Art

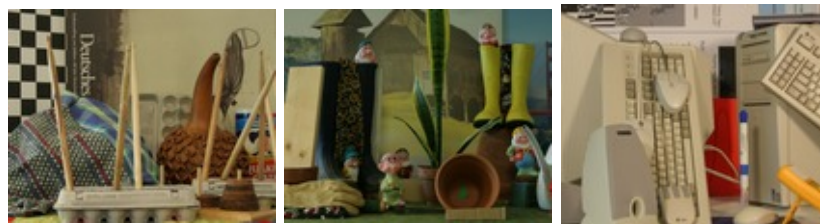
(b) Books

(c) Dolls

(d) Laundry

(e) Moebius

(f) Reindeer



(g) Drumsticks

(h) Dwarves

(i) Computer

Figure 2.16: *Middlebury Stereo Benchmark* data set: "2005 dataset"

three different illumination situations ( $i = 3$ ) and three different exposure times ( $e = 3$ ), again resulting in  $v \cdot r \cdot e \cdot i = 189$  different images available per set. The disparity ground truth is available for view 1 and 5. The ground truth scale factor is 1 for full resolution, 2 for half resolution and 3 for third resolution.

#### 2.2.4.5 2014 dataset

The data set "2014 datasets" was created by N. Nesić, P. Westling, X. Wang, Y. Kitajima, G. Krathwohl, and D. Scharstein using the method described by [SHK<sup>+</sup>14]. The 33 sets are available in resolutions ranging roughly from  $2630 \times 1840$  to  $2000 \times 1000$  pixels (F - full size),  $1315 \times 920$  to  $1000 \times 1000$  pixels (H - half size) and  $657 \times 460$  to  $500 \times 500$  pixels (Q - quarter size) (resolutions  $r = 3$ ). Each set contains a left and a right view ( $v = 2$ ), four different illumination situations ( $i = 4$ ) and eight different exposure times ( $e = 8$ ), resulting in  $v \cdot r \cdot e \cdot i = 192$  different images available per set.

The data set "2014 datasets" is parted in 10 training sets (Figure 2.18), 10 test sets (Figure 2.19) and 13 additional sets. Parts of the training sets and the test sets are part of the training and test sets for version 3 of the *Middlebury Stereo Benchmark*.

## 2.3 OpenCL

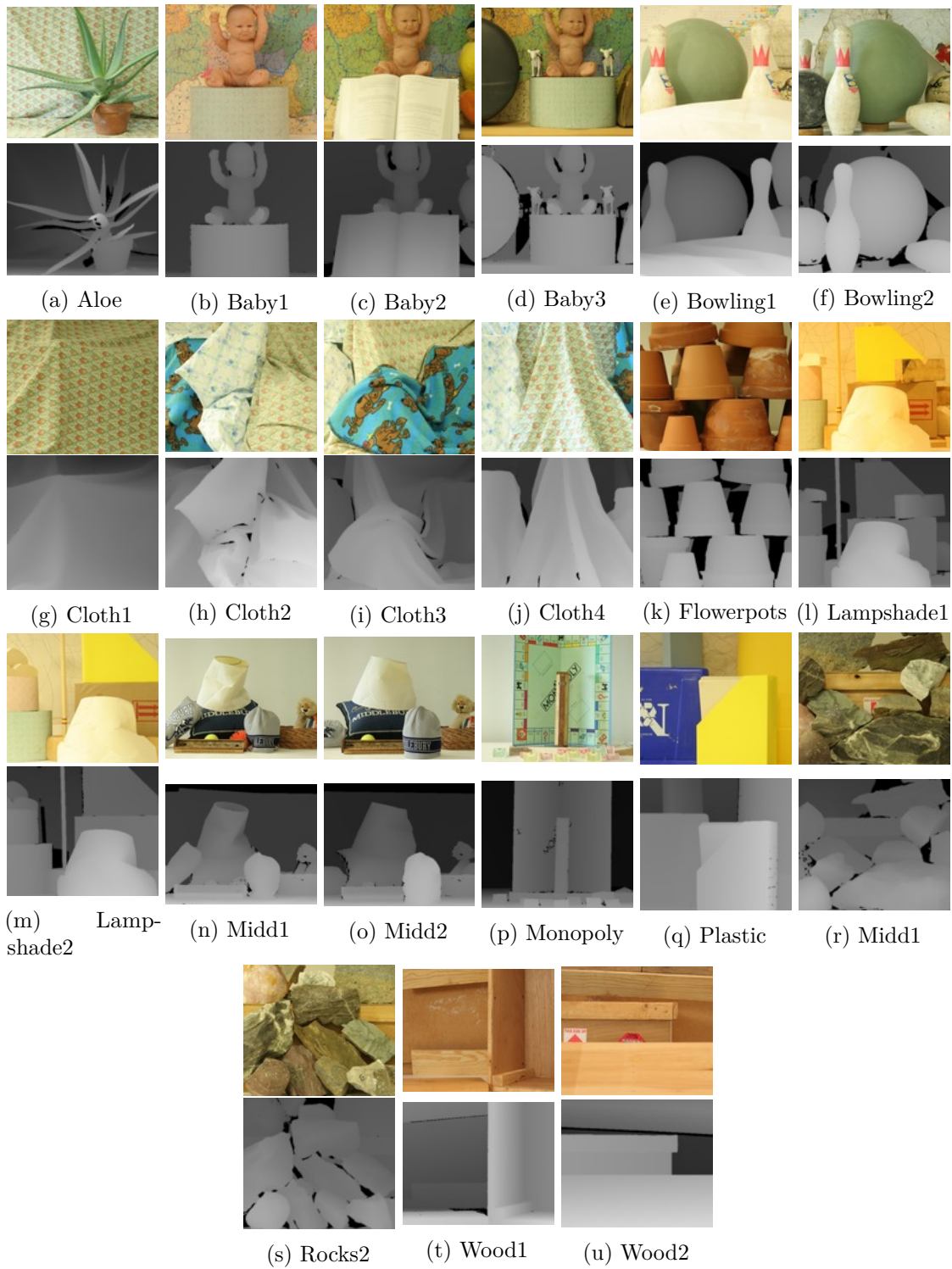
In this section, we will show the basic architectural principles of the *Open Computation Language* (OpenCL) and OpenCL devices. In this section, the concepts of devices, device parameters, local and global memory, work units, work groups, work dimensions, and kernels will be explained. For this chapter we will greatly rely on the work of [MGMG11].

OpenCL is an interface for heterogeneous hardware, that allows the use of said hardware for calculations in a parallel setup. For this reason, hardware like graphic cards, CPUs or Field Programmable Gate Arrays (FPGAs) are wrapped as devices. These OpenCL devices represent handles with which the hardware can be addressed for memory read and write operations as well as for running programs on the hardware. For this to be possible, a corresponding OpenCL driver has to be installed on the host system to make the connected devices available via OpenCL. The OpenCL drivers for most graphics hardware as well as CPUs are provided by their corresponding vendors.

The system hosting OpenCL and the calculation devices is called host. One host may have multiple devices available to it. Each physical device has a software wrapper (from its OpenCL driver) making it available as OpenCL device. The OpenCL device handle (in short OpenCL device) can be used to run programs on the physical devices. These programs are called *kernels*.

Let us assume that we have a data vector  $\bar{v} = \{v_1, v_2, \dots, v_N\}$  of length  $N$  and let us assume that we have to perform a calculation  $C(x)$  for every element in  $\bar{v}$  and that every one of these calculations is independent of the results of all the other calculations such that we have a result vector  $\bar{r} = \{C(v_1), C(v_2), \dots, C(v_N)\}$ . In conventional programming, this would be solved with some kind of looping structure iterating over the elements of  $\bar{v}$



Figure 2.17: *Middlebury Stereo Benchmark* data set: "2006 dataset"

## 2. BACKGROUND AND THEORETICAL FOUNDATION

---

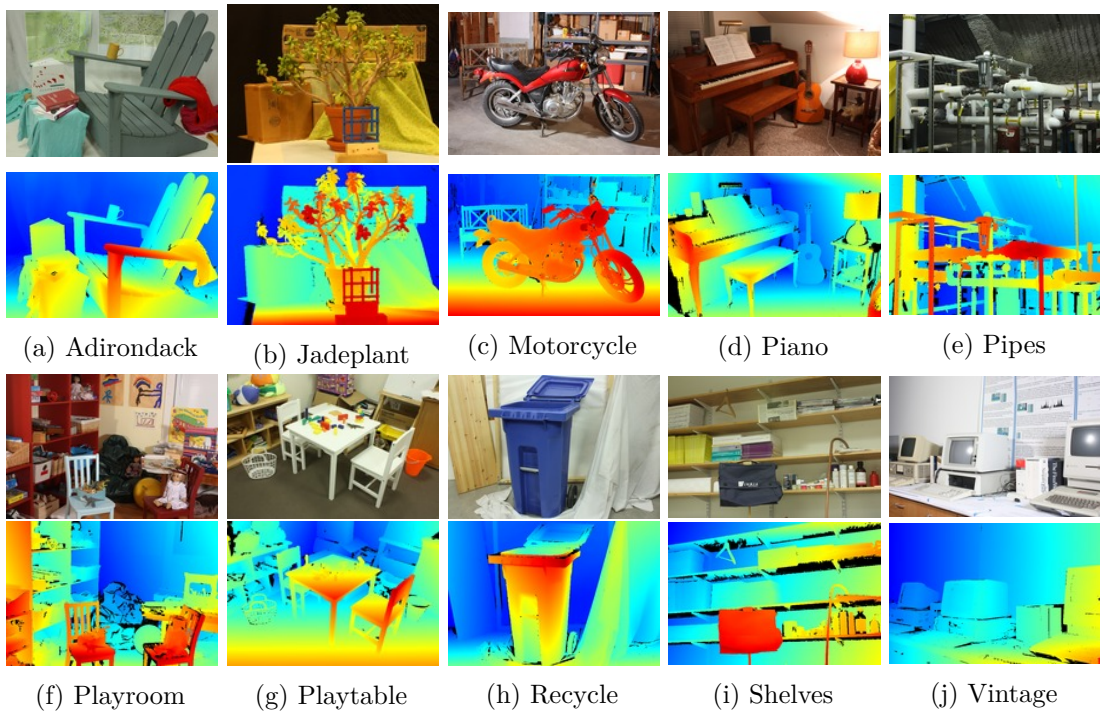


Figure 2.18: *Middlebury Stereo Benchmark* training data set: "2014 dataset"

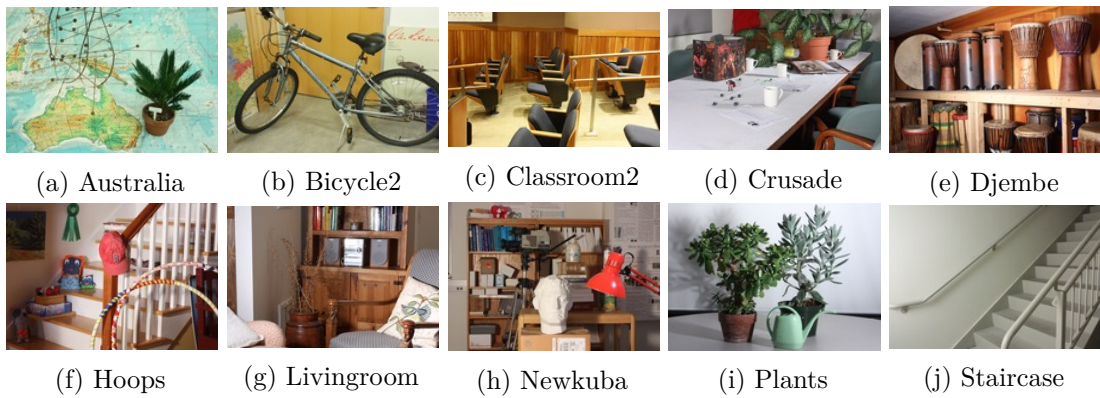


Figure 2.19: *Middlebury Stereo Benchmark* test data set: "2014 dataset"

and calling  $C(v_n)$  sequentially. A more adept approach might use some kind of multi threading to improve the runtime.

A kernel can be interpreted as the calculation function  $C(x)$ . It represents a function that has to be evaluated multiple times and is independent of the results of its previous evaluations. The task of parametrization and evaluation of one instance of the kernel for one specific parameter set, that is done by one processing element, is called *work item*.

OpenCL devices are organized in a number of *compute units*, which in turn contain multiple *processing elements*. Processing elements are the singular units used to execute kernels. Within one compute unit and within the context of one command, all active processing elements execute the same kernel. The maximum number of processing elements that can be used by one compute unit is fixed for every device. Every compute unit of one device has the same maximum number of processing elements as every other compute unit on that specific device. The organization of host, OpenCL devices, compute units and processing elements can be seen in Figure 2.20.

For one compute unit the processing elements are considered to execute the kernel simultaneously. The execution of kernels is usually dependent on the size and dimension

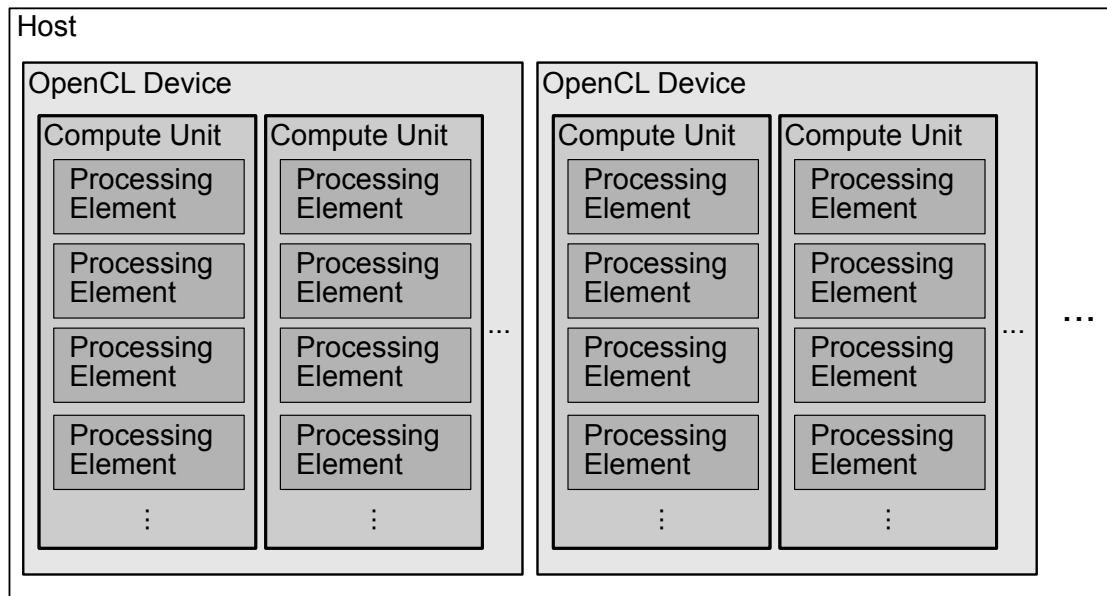


Figure 2.20: Figure reinterpreted from [MGMG11] Figure (1.6). Multiple OpenCL devices are available to one host. Every OpenCL device has a number of compute units. All compute units on one OpenCL device have the same number of processing elements.

of the problem space. In our previously defined example of vector  $\bar{v}$  the problem space has size  $N$  and is 1-dimensional (1D). Different OpenCL devices provide a different number of possible dimensions, but in most cases three dimensions are available. Also

the maximum size for a dimension may be different for every available dimension on an OpenCL device e.g. three dimensions:  $1024 \times 512 \times 128$ . These parameters can be retrieved by predefined OpenCL commands.

The total size of a problem (*work size* or *global size*) might exceed the number of processing elements available to a compute unit. It might even exceed the total number of processing elements available to the whole device. This leads to the conclusion that at some point, a process of serialization has to be introduced in order to process problems with a high number of *work items*. This is why any problem to be handled by OpenCL is partitioned into *work-groups*. *Work groups* are executed on the level of compute units.

Problems consisting of fewer *work items* (i.e. problems with smaller *work group size*) than the used OpenCL device's number of processing elements per compute unit, are naturally fitted into one single *work-group*. The partition of larger problem spaces is parametrized by the programmer and can be crucial for runtime performance.

Due to the fact that OpenCL devices are mostly independent devices in respect to the host system, it is important to keep in mind that OpenCL devices can not access the host's memory but have their own memory. This means that data that is required for a kernel to run properly, has to be transmitted towards the OpenCL device before the kernel execution is started. In the context of an OpenCL program there are four types of memory. These types are host memory, global memory complemented by global memory cache, local memory and private memory.

Host memory is the memory that is directly accessible by the host, without the help of an OpenCL command. It is usually very big compared to the other types of memory. The architecture of host, host memory, global memory, global memory cache, local memory, work-groups, private memory and work items is outlined in Figure 2.21.

Global memory is the OpenCL device's biggest memory that is accessible to the host through OpenCL commands. OpenCL provides commands for the host to efficiently read and write big chunks of this memory. Every processing unit may access allocated global memory areas, as long as a handle for the corresponding memory region is provided by the host. Global memory usually is the slowest memory of an OpenCL device, this however, is partially compensated by the global memory cache. Random access to global memory by processing elements, may lead to heavy usage of paging, and this is why the way the problem space is partitioned may be crucial. Global memory is also the memory where space for constants can be allocated.

Local memory is smaller but faster than global memory. Every compute unit has its own local memory, which is accessible for every processing element within this compute unit. Kernels can trigger a synchronized bulk loading of data from global memory into local memory. This allows for fast and efficient access for the processing elements to that bulk of data. This can under certain conditions prove beneficial for the runtime. Let us assume that the time for bulk loading a chunk of data of size  $s$  into local memory, takes  $t_s$  time units. Every work unit will have  $k$  random access read or write actions within this chunk of data. The time needed for one read / write action to local memory

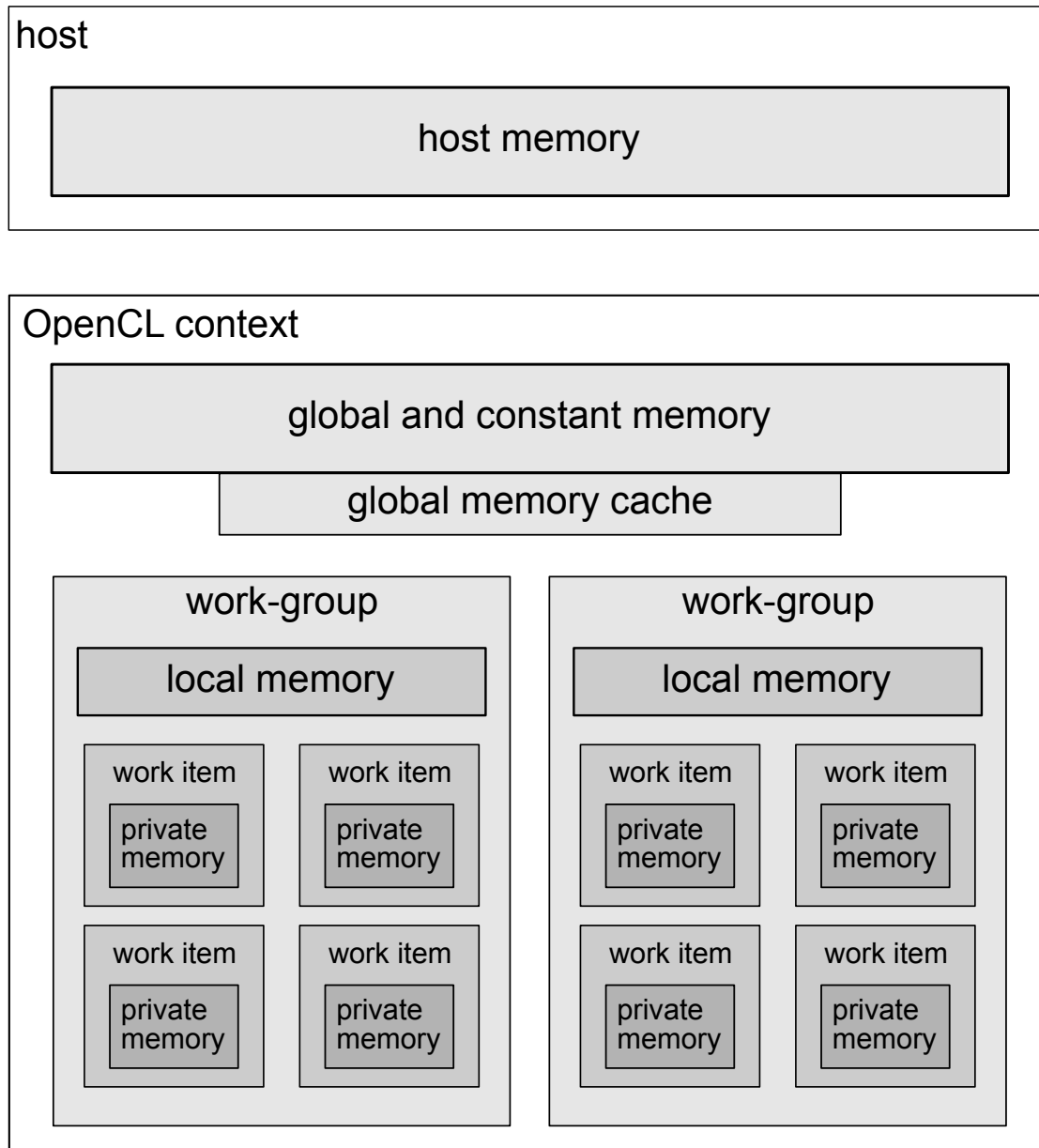


Figure 2.21: Figure reinterpreted from [MGMG11] Figure (1.8). Visualization of host and OpenCL device in terms of memory. The host can access global device memory exclusively via commands provided by the OpenCL API. Local memory access is coordinated by the device and initiated by the work items. Private memory is exclusively available to the corresponding work item.

is denoted as  $t_l$  and the time needed for one read / write action from global memory cache is denoted as  $t_g$ . We assume that the relation  $t_l < t_g$  holds. Loading a chunk of global memory into local memory is beneficial to the runtime if the number of read / write actions  $k$  is large enough to satisfy:

$$k \cdot t_g > t_s + k \cdot t_l \quad (2.46)$$

Memory that is only available to one processing element, is called this processing element's *private memory*. Private memory is the smallest and fastest memory available to an OpenCL device. This is where a work item's variables are located.

## State of the Art

In the field of computer vision, stereo vision is a topic which is often discussed. The problem of stereo correspondence is a key element of many stereo vision methods. Stereo matching is a way to solve the stereo correspondence problem. [Sze10] define stereo matching as *the process of taking two or more images and estimating a 3D model of the scene by finding matching pixels in the images and converting their 2D positions into 3D depths*<sup>1</sup>. The result of the stereo matching process is usually a mapping from image space to disparity space. This mapping is often represented as a 2D map of disparity values for all pixels.

Some methods that are used in stereo vision have been around since the 1970s and 1980s. The works of [LT98] from 1998, cover some of these methods and give an overview on the techniques and basic ideas that in part are used until today. In 2010 [Sze10] published an updated overview of these methods. They explain the concepts of epipolar geometry, sparse and dense correspondence, DSI, cost functions and the mechanisms used for stereo matching in general.

The search for stereo correspondence can be reduced to single scanline problems when the epipolar geometry of the camera setup is utilized. Using epipolar geometry, the images can be transformed such that for every point  $(x, y)$  in the first image, all matching candidates  $(x_i, y_i)$  in the second image have the same  $y$ -coordinate  $(x_i, y)$  i.e. lie on the same scanline. A sketch explaining epipolar geometry is shown in Figure 2.1.

### 3.1 Sparse Stereo Matching

Sparse correspondence methods are generally feature based. Feature based methods usually start with identifying Points of Interest (POIs) e.g. corner points or Scale-Invariant Feature Transform (SIFT) features, Objects of Interest (OOIs) e.g. edges

---

<sup>1</sup>[Sze10] Chapter 11 - Stereo correspondence

or geometric forms or Regions of Interest (ROIs) e.g. regions containing a specific texture in the stereo image pairs. Then the POIs, OOs or ROIs (further on we will combine these terms into Features of Interest (FOIs)) of one image are matched to their counter parts in the another image. From these matches the disparity is calculated. The resulting disparities are then refined and/or interpolated by post processing steps. Early sparse stereo matching approaches like those proposed by [Han74],[MKS89], [HMJP92] or [Col96] use some patch-based metrics to match the found FOIs of one image into the second image. Later works such as the work of [ZS00] and [LQ02] proposed to "grow" matches from a small set of highly reliable matches. These approaches were extended by, among others [LQ05] and [GSC<sup>+</sup>07], who proposed multi-view stereo methods.

In recent works of [BGM14], focus is set on the process of whether a found match is good and unambiguous, by usage of Gibbs sampling and minimization of an expected loss function. Cooperative matching techniques for sparse stereo matching were discussed in the context of event cameras, as opposed to conventional RGB cameras, by [PBG13] and [PKBG17].

## 3.2 Dense Stereo Matching

Dense correspondence methods in general work on most or all pixels in image space. This results in higher computational demands for these methods in comparison to sparse methods. Window based approaches that depend on intensity values for correspondence, usually have an implicit smoothness assumption included. This smoothness assumption is attributed to support aggregation as it is used by different authors, such as [Han74], [ZW94], [BN98] or [HZW<sup>+</sup>10].

Other dense algorithms minimize a global cost function which usually has an explicit smoothness term. [Sze10] states that the main distinction between these methods is the used procedure of minimization. In his work [Sze10] lists simulated annealing as used by [MMP87] and [Bar89], probabilistic diffusion as used by [SS98], expectation maximization as used by [BNT07], graph cuts as used by [BVZ01] and loopy belief propagation as used by [SZS03] as minimization methods.

Recent works in the field of stereo matching like the works of [JGM14] or [ZL16] use Convolutional Neural Networks (CNNs) for matching. The use of neural networks in stereo matching is observed by the authors of [ZK15], as they ask the question whether or not datasets can be used to learn similarity functions. They show that CNNs can be used for different stages in the stereo matching pipeline.

An example for dense stereo correspondence not using CNNs, is the work of [ZFM<sup>+</sup>14] who use a *bio-inspired* process of stereoscopic correspondence across multiple scales, as observed in human vision.



### 3.3 Local Stereo Methods

In local methods, usually all matching costs provided by a support region in the DSI are aggregated. Support regions may be 2-dimensional (2D) or 3-dimensional (3D). The dimensions of these support regions are not necessarily spatial dimensions. In some applications, the support region is accumulated over time e.g. from a video. Three dimensional spatial support regions allow for better results in scenes with slanted surfaces but are more demanding in calculation due to the additional dimension. Two dimensional support regions are supported by the Fronto-Parallel Assumption. Different implementations of 2D support regions have been provided by authors such as [Arn83], [FRT97], [BI99], [BG05] and [HZW<sup>+</sup>10]

In his work [Sze10] points out various methods using 2D support regions, with different approaches as to connectedness of the components ([BVZ98]), varying window sizes ([OK92], [KO94], [KSC01], [Vek01], [Vek03], [PKBG17]) or the usage of color ([YK06], [TMDSA08]).

In the work of [BRR11] a 3D support plane is created to overcome problems related to slanted surfaces. For more 3D support regions [Sze10] refers to [Gri85], [Pra85] and [ZK00].

Further, in the works of [HRBG11], [DRR03] and [ZCS03] time is used as a third dimension. These methods can be used to reconstruct 3D information from video footage.

Different cost aggregation methods have been reviewed by [GYWG07] and [HS09]. Some aggregation methods use adaptive window sizes as shown by [KO94], adaptive support weights as used by [YK05] or geodesic support weights as implemented by [HBGR09].

### 3.4 Global Stereo Methods

In comparison to local methods, global methods are usually more computationally demanding. Global methods are characterized by minimizing an energy function that considers the whole image in terms of differences and smoothness. Some methods take additional factors into account e.g. [KZ01] use an additional term to handle occlusions. These cost constraints can be resolved for example by dynamic programming as shown by [Bel96], [CHRM96] or [BT99] or with graph cuts as shown by [BVZ01]. Graph cut methods build graphs out of the stereo images and model the problem with energy functions such that flow of energy models the matching costs. Then the authors of [BK04] compare methods that use graph optimization methods to find the best disparity mapping.

Another global approach to solving the stereo correspondence problem are segmentation based techniques. [Sze10] characterize these techniques as methods which segment the image into regions and then try to assign disparity values to the resulting regions. Methods using this approach are described by authors such as [KSK06], [ZK07] or [TWZ08].

Some constraints for global methods are smoothness constraints as described by [BG05], [Hir08] and [TS00], color based constraints like the works of [TS00] and [BG05], edge based constraints like the work of [CZYS14] or visibility or order based constraints like described by [SLKS05]. An experimental evaluation of the incorporation of color in global stereo matching was performed by [BCPG08].

### 3.5 Real-Time Stereo Matching

The work of [TLLA16] is focused on the subject of real-time stereo matching. An overview is given on the performance of different algorithms and the used width, height and maximum disparity parameters used by the various authors of different algorithms are provided along with the runtime comparison. Further, [TLLA16] argue that accuracy and speed are usually competing factors in stereo matching methods.

The subject of real-time stereo matching is discussed by many authors e.g. [KPP13], [HIG02], [ZNPC13], [HZW<sup>+</sup>10], [HBG10], [YWY<sup>+</sup>06], [BM17] and [GYWG07].

### 3.6 Real-Time Hardware for Stereo Matching

In the work of [TLLA16] stereo matching algorithm-combinations are classified into three categories. The first category contains real-time or near real-time results that were achieved on standard processors. The second category contains algorithms with real-time performance on specialized hardware. The hardware types listed in this context are GPUs, FPGAs, Digital Signal Processors (DSPs) and Application-Specific Integrated Circuits (ASICs). The third type are algorithms that did not show real-time or near real-time performance. The hardware evaluations of [TLLA16] give an overview of different algorithm - hardware combinations and comparisons are listed by device type i.e. FPGA, GPU and DSP. However, the compared algorithm results were rarely generated by the same device and comparability of algorithms across devices is therefore not easily achieved.

Stereo matching algorithms designed for embedded systems were presented by [HZW<sup>+</sup>10] and [BM17]. A DSP stereo matching solution is presented by [CLT<sup>+</sup>07]. The most frequently used specialized hardware for stereo matching implementations are GPUs. GPU based implementations were presented by [KPP13], [ZNPC13], [GFGC17]

Further, the works of [CAD<sup>+</sup>12] and [OBDA11] focus on the topic of using OpenCL to generate Very high speed integrated circuit Hardware Description Language (VHDL) for FPGAs. This may be used for GPU based implementations which use OpenCL.

Generally, strategies for evaluating different hardware configurations for stereo techniques can also draw inspiration from related computer vision fields such as video coding (e.g., [SBSG08], [SBGB09]), where real-time implementations have been in the focus of interest for many years.

# A Stereo Matching Algorithm

In this chapter, we will explain the ideas proposed by [HZW<sup>+</sup>10]. This method was chosen as model for our implementation because of three major reasons. The first reason being that on the website of the Middlebury benchmark, algorithms with a good runtime performance are usually window-based like this method and many of these algorithms use a census-based cost metric as well. Secondly, the method proposed by [HZW<sup>+</sup>10] was designed to be "suitable for embedded real-time systems", which makes an implementation based on this method a good candidate for a transfer to FPGA later on. Finally, this thesis was created in cooperation with IVISO Ges.m.b.H. which expressed an interest in an implementation of the method similar to the method of [HZW<sup>+</sup>10].

The method of [HZW<sup>+</sup>10] is a dense stereo matching method, which creates a DSI and provides disparity refinement. We will discuss the proposed techniques for cost volume calculation, support aggregation and disparity refinement as well as cover the topics of census transform, aggregation strategy, subpixel refinement and the left-right consistency check in detail.

## 4.1 Sparse Census Cost Function

In the work of [ZW94], the census transform was used as a non-parametric local transformation in the context of stereo correspondence. In the work of [HZW<sup>+</sup>10] a sparse census cost function was proposed. In this section we will in short explain non-parametric local transformations in general and the census transform in detail.

A local transformation performs a transformation operation  $\mathcal{O}$  on localized data regions  $\mathcal{R}_i$  written as  $\mathcal{O}(\mathcal{R}_i)$ . Every data point in the source data (usually pixels) corresponds to such a data region. A local transformation is therefore a mapping of  $d_i = \mathcal{O}(\mathcal{R}_i)$ , where  $i$  denotes a position in the source data array. The set of regions  $\mathcal{R}_i$  represents

the data regions corresponding to the data positions  $i$ . The set of all  $d_i$  represents the transformation result and is called census word of length  $i$ .

A non-parametric local transformation as described by [ZW94], uses the relative ordering of the data in the examined region, rather than the data values in that region. An example for a non-parametric local transformation is the census transformation.

The census transform in its basic form describes a window of  $3 \times 3$  pixels. One pixel in this window is the reference pixel for this 3-by-3 region. The values of all other pixels  $p_i$  in this region are compared to the value of the reference pixel  $c$ . For every comparison one bit of information is saved, resulting in one byte per 3-by-3 region. The saved information represents whether or not the examined value was lower or equal to the reference value. The following formula shows how the values of the bits  $b_i$  of the census word are determined:

$$b_i = \begin{cases} 1, & \text{if } c > p_i \\ 0, & \text{otherwise} \end{cases} \quad (4.1)$$

The resulting bytes encode the structure of the data values  $p_i$  with respect to the central value. Two pixels in census transformed images are compared for similarity by using the Hamming distance of these bytes. Improvements that have since been proposed mainly entail larger window sizes, more finegrained difference measures from the reference pixel, different reference pixel positions or masking of the examined pixels.

Larger window sizes i.e. larger regions  $\mathcal{R}_i$ , for census transforms result in a higher number of data points that can be compared to the reference pixel  $c$ . If more data points are used, the resulting census word will be longer. Longer census words in turn, result in a higher maximum value for Hamming distances.

Masking of data points describes a strategy where for every region not the whole region, but a subset of data points of the region is used for the creation of the census word. This effects the result in a way that only the selected subset is used for comparison and included in the result. This reduces the result's expressiveness, because data is dismissed. However, this allows to use larger image patches while the calculation costs stay the same and therefore, increases the distinctiveness of the census word.

The algorithm proposed by Humenberger et al. ([HZW<sup>+</sup>10]) uses a census transform and masks every second row and column of the regions in order to reduce calculation density, as shown in Figure 4.1. The shown mask generates a 16 bit census word for a region of 64 pixels. Though the loss of information is 75%, Humenberger et al. claim that the *expressiveness* of the census word does not loose that much of its value and still produces good results when used as similarity measure in stereo correspondence. Humenberger et al. call this method *sparse census mask*.

In order to argue about the qualities of sparse transformations a metric is defined to measure the *expressiveness* of the results of these transformations. This metric is based

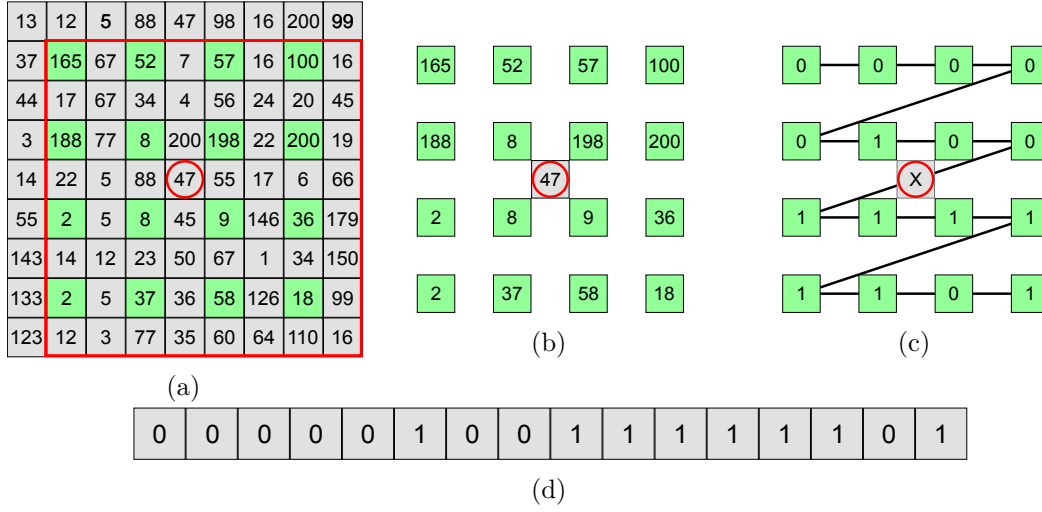


Figure 4.1: Figure in part reinterpreted from [HZW<sup>+</sup>10] Figure (3). Figure 4.1(a) shows region  $\mathcal{R}$  highlighted by a red frame, with a region size of  $8 \times 8$ . The reference pixel  $c$  at filter position (3/3) highlighted by a red circle and the filter mask, leaving out every second row and column in light green, can be seen. Figure 4.1(b) shows only the part of the input data, relevant for the transformation. Figure 4.1(c) shows the result for every pixel based on Formula 4.1. Figure 4.1(d) visualizes how the census word is composed of the pixel wise results.

on the relation of the result of the sparse transformation and the result of the non-sparse (base) transformation of the examined transformation. In our case, this is the regular census transformation of an image patch. For an image patch  $P$  consisting of  $k + 1$  pixels denoted as  $|P| = k + 1$  a base result-word  $v$  consists of  $k$  bits denoted as  $|v| = k$ . Every bit in the base result word corresponds to exactly one pixel in the image patch. In a sparse context the length of the result word  $s$  is smaller than the length of the base word:  $|s| < |v|$

From any sparse result word a reconstruction of the base word can be attempted. The resulting reconstruction  $r(s)$  is of the same length as the base word:  $|r(s)| = |v|$

The reconstructed result word  $r(s)$  is compared to the base result word. A bit is correctly reconstructed if its value is equal to the corresponding bit in the base result word. The expressiveness of a result word is defined as the percentage of correctly reconstructable bits from that result word. This implies that the expressiveness of the base result word is 1.

The lower bound of the expressiveness of a sparse census result word  $s$  is the length  $|s|$  of this word. In a setup where one out of four pixels is transformed, this results in a lower bound for expressiveness of roughly 0.25. The actual expressiveness in such a case is usually higher than 25%. In an image of a natural scene pixels often have color

values similar to their neighbors. This similarity among neighboring pixels is known as homogeneity, as described by [ZHHL06]. Due to homogeneity, the value of a census bit in a full census word can be estimated from the sparse census word by a nearest neighbor estimation. Figure 4.2 gives an example of a homogenous image patch and gives an example where a sparse census word has an expressiveness value of 0.8, by using nearest neighbor estimation for reconstruction. The lower bound for expressiveness of the sparse census word for the example in Figure 4.2 in this example is  $\frac{4}{15} = 0.2666$ .

The argument can be made that the error of the estimation of the full census word from its sparse census word, should roughly be the same as if the image was down-sampled and up-sampled again, which is a known problem that corresponds to the Nyquist–Shannon sampling theorem.

This argument further supports the claim of [HZW<sup>+</sup>10] that sparse census cost functions with a higher window size perform better than dense census cost functions of lower window size. In [HZW<sup>+</sup>10] the authors compare the sparse census transform to the dense census transform with different window sizes. The window size  $16 \times 16$  was chosen by [HZW<sup>+</sup>10] based on their experimental evaluation.

In order to compare two census words in terms of expressiveness, the filter size has to be taken into account. We used the area of the biggest filter as reference size and fitted smaller filter elements into that area. The current example of a  $16 \times 16$  sparse census transform and an  $8 \times 8$  dense census transform was therefore normalized to the area of the larger filter. The expressiveness of the  $8 \times 8$  dense census word is exactly one quarter of the expressiveness of a  $16 \times 16$  dense census word. We have already shown (in Figure 4.2) that the expressiveness of a  $16 \times 16$  sparse census word is at least one quarter of the corresponding dense census word, but can be much better as well. This shows that the worst case scenario of a sparse census word with window size  $16 \times 16$ , is exactly as expressive as the dense census word with window size  $8 \times 8$ . Thus a  $16 \times 16$  sparse census word is at least as expressive but usually better than an  $8 \times 8$  census word.

The estimation of the base census word from the sparse census word shows that more information is contained in sparse census words of length  $n$ , than in their dense counterparts.

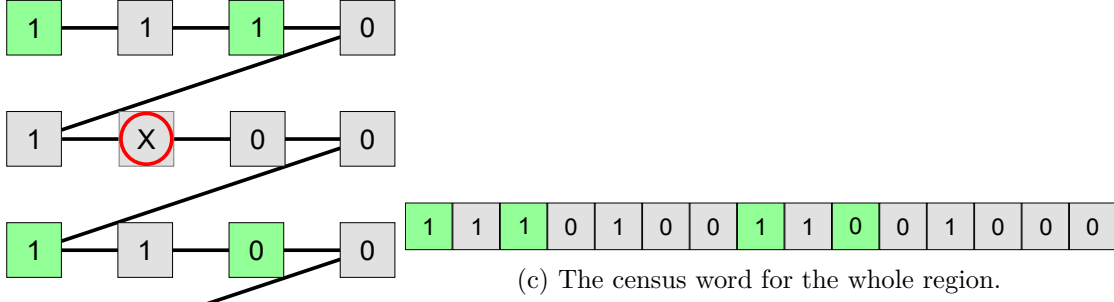
## 4.2 Aggregation Strategy

As stated in Section 2.1.3.3 the aggregation of support costs can influence the quality of stereo matching results. This is why [HZW<sup>+</sup>10] use a variation of rectangular windows (see Section 2.1.3.3) for cost aggregation. The variation by [HZW<sup>+</sup>10] is that the point of origin  $(i, j)$  of  $w_n^{I_{0|1}}(i, j)$  is not centered in the window but is instead positioned in the upper left corner of the window.

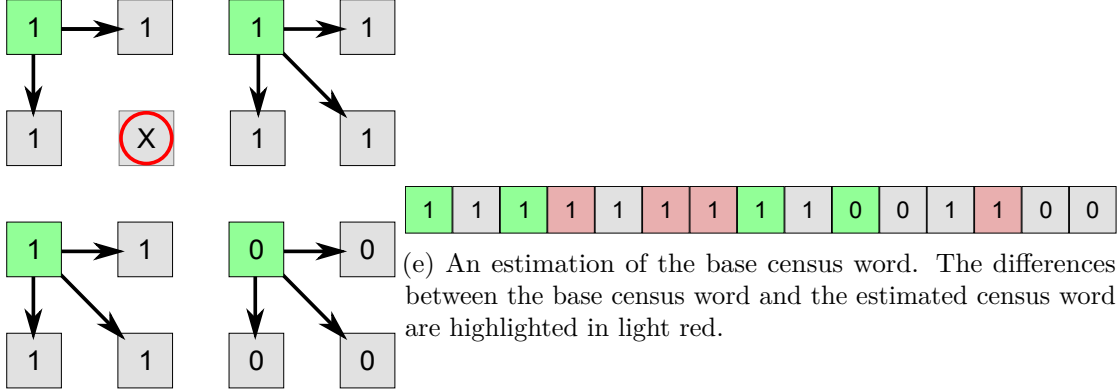
The argument can be made that the support aggregation is done to reduce ambiguities, and it is therefore not important which neighborhood of a pixel is aggregated, as long as the aggregation is done the same way for the left and the right image.

3	4	5	10
2	6	5	10
4	3	9	9
3	8	9	10

(a) This shows an example region  $\mathcal{R}_i$  to which a sparse census transform is applied. The reference pixel  $c$  is highlighted with a red circle and the values included by the sparse mask are highlighted in light green.



(b) This shows the result for the census transform for the region shown in Figure 4.1(a).



(d) The nearest neighbor interpolation is used to estimate the base census word from the sparse census word.

Figure 4.2: Example for sparse census transform.

Further, one can argue that, as long as the result position relative to the window position is constant, the resulting aggregation map is roughly the same as when the result position were to be in the center of the aggregation window.

### 4.3 Sub Pixel Refinement

As explained in Section 2.1.3.5, sub pixel refinement estimates a continuous function through discrete data. In the work of [HZW<sup>+</sup>10] the aggregated cost volume of the sparse census cost function is searched for the minimum of the costs along the disparity axis. Therefore the cost volume is traversed for all pixel positions along the disparity axis in discrete disparity steps.

When the minimum cost is found, a parabolic polynomial is fitted into the search space at the minimum point and its two neighbors. The minimum position of this polynomial is then assumed to be more accurate than the discrete disparity value.

The formula in [HZW<sup>+</sup>10] is based on Newton's method (which is also known as the Newton-Raphson method) of iteratively approaching a real valued function's roots:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \quad (4.2)$$

The real valued function in this context would be the first derivation of the polynomial describing the paraboloid fitted into the discrete values. Neither the parameters of the paraboloid polynomial nor the parameters of its first derivation are known. This is why the first derivation is approximated by the method of finite differences,

$$f_C(d) \approx \frac{y(d + \frac{s_1}{2}) - y(d - \frac{s_1}{2})}{s_1} \quad (4.3)$$

where  $d$  represents the optimum disparity found by searching the discrete DSI for a minimum value,  $s_1$  is the step size for this finite difference, the function  $y(k)$  represents the values in the DSI at disparity depth  $k$ .

The first derivation of our function, and therefore the second derivation of the paraboloid polynomial, is approximated in much the same way:

$$f'_C(d) \approx \frac{y(d + s_2) - 2y(d) + y(d - s_2)}{s_2^2} \quad (4.4)$$

where  $d$  and  $y(k)$  are the same as in Formula 4.3 and  $s_2$  is the step size for this finite difference.

The combination of Formula 4.2, Formula 4.3 and Formula 4.4 with  $s_1 = 2$  and  $s_2 = 1$  results in

$$d_{sub} = d_{min} + \frac{y(x_n) - y(x_p)}{2(2y(x_c) - y(x_n) - y(x_p))} \quad (4.5)$$



with  $x_c$  (i.e.  $x$  central) as the optimum disparity found by searching the discrete DSI for a minimum value previously known as  $d$ ,  $x_p$  (i.e.  $x$  previous) as  $x_p = x_c - 1$  and  $x_n$  (i.e.  $x$  next) as  $x_n = x_c + 1$ .

Formula 4.5 represents one step of Newton's method in order to close in on the assumed paraboloid's minimum point. Assuming that the assumption of a paraboloid is correct, the total disparity error should get smaller with the convergence rate of Newton's method, which is quadratic.

## 4.4 Left-Right Consistency Check

As mentioned in Section 2.1.3.5 cross correlation checks can be used to verify if a disparity value for a pixel is valid. In their work, [HZW<sup>+</sup>10] calculate sub pixel disparity maps for both images of every image pair. For this the search direction along the rectified scanlines is from left to right for the right image and from right to left for the left image.

The sub pixel disparity value  $d_r$ , for any position  $(x, y)$  in the right disparity map, is cross checked with the corresponding disparity value  $d_l$ , at position  $(x + d_r, y)$  in the left disparity map. To check whether  $d_r$  and  $d_l$  are correlated, the absolute difference of  $d_r$  and  $d_l$  is verified to be below an arbitrary value of 5. The size of this threshold can be argued to be a function of the maximum disparity depth  $d_{\max}$  and was discussed in Section 2.1.3.5 to be around  $(2 \text{ to } 6)\% \cdot d_{\max}$ .

If the disparities are found to pass the threshold check, the mean of  $d_r$  and  $d_l$  is chosen to be the correct disparity value for this pixel position.





# Implementation

The target of this thesis is to explore a way to make a stereo matching algorithm available on multiple platforms. To this end, we implemented an algorithm and used this implementation on a variety of different platforms. In principle, this is possible with OpenCL. OpenCL kernels can run on platforms like CPUs, GPUs, ARM processors and FPGAs. There are Software Development Kits (SDKs), provided by the vendors Altera and Xilinx, which provide a way to transform kernels into modules for the FPGAs of these manufacturers.

In order to show the cross platform portability of such a method, we chose to create an implementation in OpenCL, similar to the stereo matching algorithm of [HZW<sup>+</sup>10]. We will prove that our implementation can run on CPUs, GPUs and on ARM boards. The results of the *Middlebury Benchmark* of this implementation on these devices will be presented in Chapter 6.

In this chapter we will go into detail concerning our implementation. We will give an overview of our implementation which contains steps for rectification and undistortion, census transformation, cost calculation and cost aggregation, minimum search, parabolic fitting and finally consistency checking.

Finally, we will discuss every one of these steps separately. We will provide code listings, estimated memory consumption and descriptions of method details.

Our calculation of consumed memory in this chapter is based on the assumption that the stereo matching process for one image pair is concluded before the next image pair's processing is started. Also buffers are only used for their designated purpose i.e. the purpose they were allocated for, and are not reused for different purposes along the matching process. An example for this is the buffer *MinimumR2LBuffer*, which is *not* reused as buffer *DisparityMapBuffer* in Figure 5.2 though it can hold the same amount of data and when *DisparityMapBuffer* is needed, *MinimumR2LBuffer*'s data is no longer used in the matching process of the current image pair. Reusing buffers could potentially

reduce the total memory consumption. However, this is not done because of a potential future implementation for matching to cameras' data streams. In such an implementation, the buffers at the beginning of the pipeline could be prepared for the matching of two images, while the matching process for the previous frame set is still in progress.

For this chapter, we introduce the following notation in order to make formulas for memory consumption easier to read. We will use the image dimensions width ( $w$ ) and height ( $h$ ), as well as the maximum disparity ( $d_{\max}$ ). The notation  $|B|$  represents the size of buffer  $B$  in bytes. Further, we introduce the variables  $i_k$  and  $f_k$ . These variables represent integer variables ( $i_k$ ) and floating point variables ( $f_k$ ). The value of  $k$  specifies the size of this variable in bytes. The following examples explain this notation.

A buffer  $B$  containing one-byte gray scale values in the range of 0 to 255 for every pixel, has size  $|B|$  in bytes:

$$|B| = i_1 \cdot w \cdot h = 1 \cdot w \cdot h \quad (5.1)$$

A buffer  $C$  containing three double sized floating point numbers per pixel has size:

$$|C| = (f_8 + f_8 + f_8) \cdot w \cdot h = 24 \cdot w \cdot h \quad (5.2)$$

Finally, a buffer  $D$ , containing one 32-bit integer and a 32-bit floating point number per pixel per disparity depth has size:

$$|D| = (i_4 + f_4) \cdot w \cdot h \cdot d_{\max} = 8 \cdot w \cdot h \cdot d_{\max} \quad (5.3)$$

## 5.1 Overview

In this section, we will give an overview of how the OpenCL kernels of our implementation interact with buffers in global memory. The following figures, Figure 5.1, Figure 5.2 and Figure 5.3 provide a visual overview on the flow of data in our implementation.

Figure 5.1 shows the optional rectification and undistortion step. In this step, the input images from two cameras (left and right camera) are rectified and undistorted. A kernel was implemented which uses bilinear interpolation to generate a rectified image from the distorted input data. For this purpose, two rectification maps that provide the sub-pixel positions of the intensity value in the unrectified image are needed. One map contains the rectified x-coordinates ( $x'$ ) and the other map contains the rectified y-coordinates ( $y'$ ) for every position  $(x, y)$  in the input image. These rectification maps are specific for each of the used cameras, but usually do not change between two images of the same camera. This allows for the rectification maps to be constant for each camera. The coordinates  $x'$  and  $y'$  are floating point values and the intensity value at point  $(x, y)$  is calculated as the bilinear interpolation of the point  $(x', y')$  to the points  $(\lfloor x' \rfloor, \lfloor y' \rfloor)$ ,  $(\lfloor x' \rfloor, \lfloor y' \rfloor + 1)$ ,  $(\lfloor x' \rfloor + 1, \lfloor y' \rfloor)$  and  $(\lfloor x' \rfloor + 1, \lfloor y' \rfloor + 1)$  in the input image.

An overview of the structure of our implementation is shown in Figure 5.2. The algorithm starts with rectified and undistorted images as input, a kernel transforms the input

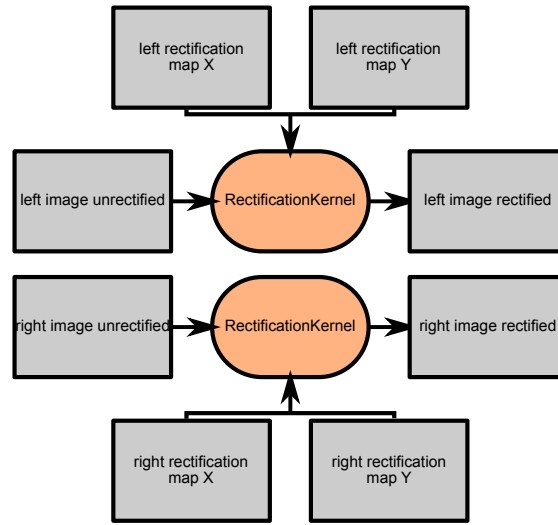


Figure 5.1: Visualization of the optional rectification and undistortion process in our implementation. A rectification map has the original image dimensions and contains displacement correction values.

images into census maps and these census maps are then used to calculate the matching costs. It is possible that the calculation of the matching costs for the whole problem space is not feasible within the available global memory of the OpenCL device. Due to this restriction we provide a strategy for cost calculation in an iterative way. This cost processing loop is shown in Figure 5.3. The result of the cost processing loop are triples of the minimal cost for every pixel position and the two neighboring cost values along the disparity axis of the DSI containing the costs. These triples are generated for both search directions (left-to-right and right-to-left). A parabola is fitted in every one of these triples, in order to find a sub-pixel disparity value. Finally, a consistency check is performed as described in Section 2.1.3.5 and the final result is written to an output buffer.

The iterative calculation of the matching costs is shown in Figure 5.3. The maximum number of lines ( $k_{\max}$ ) that the used OpenCL device is able to process at once is calculated using the image dimensions, the maximum disparity, the size of the cost aggregation window and the maximum memory allocation size of the used device. The problem space is then divided along the image's y-dimension into sections of at most  $k_{\max}$  lines.

For every section, the Hamming distances for all disparity steps and all pixel positions in this section are calculated as described in Section 2.1.3.1 and Chapter 4. The calculated costs are then aggregated as shown in Figure 5.4 and described in Section 2.1.3.3. The implementation of this aggregation is split into two steps. The first step is the aggregation along the x-axis of the DSI containing the costs. The resulting DSI is in a second step processed in y-direction. The aggregation is done this way to avoid unnecessary paging

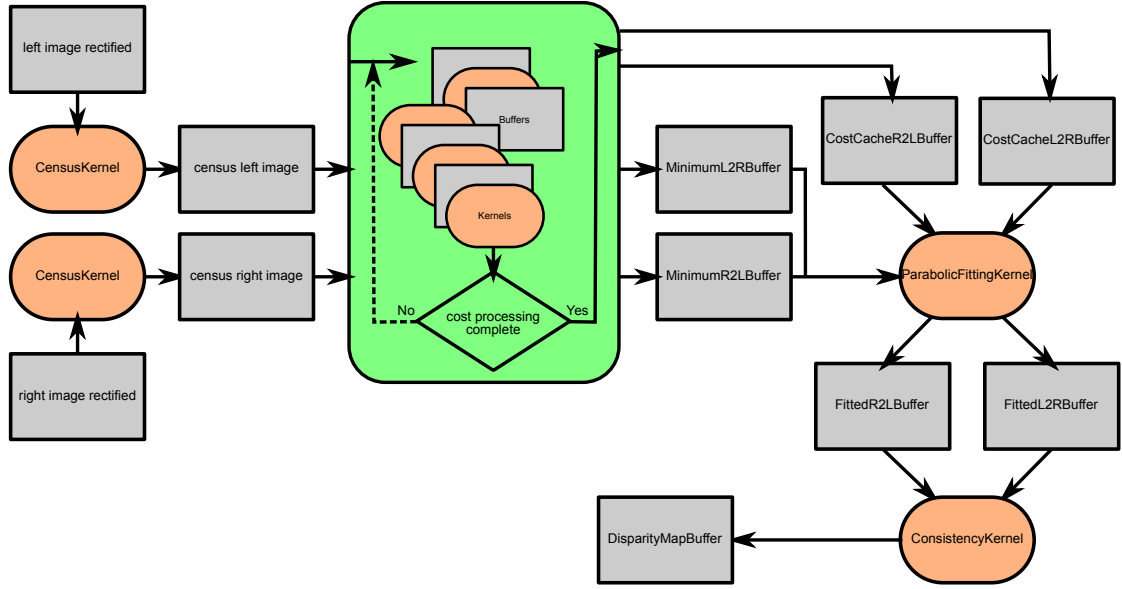


Figure 5.2: Overall flow of data through our OpenCL implementation of the algorithm. The gray rectangles in this figure represent buffers in the global memory of the used OpenCL device. The orange shapes represent OpenCL kernels in our implementation. The arrows represent the flow of data between kernels and buffers. The big, green block represents the cost calculation and minimum cost search which is shown in detail in Figure 5.3.

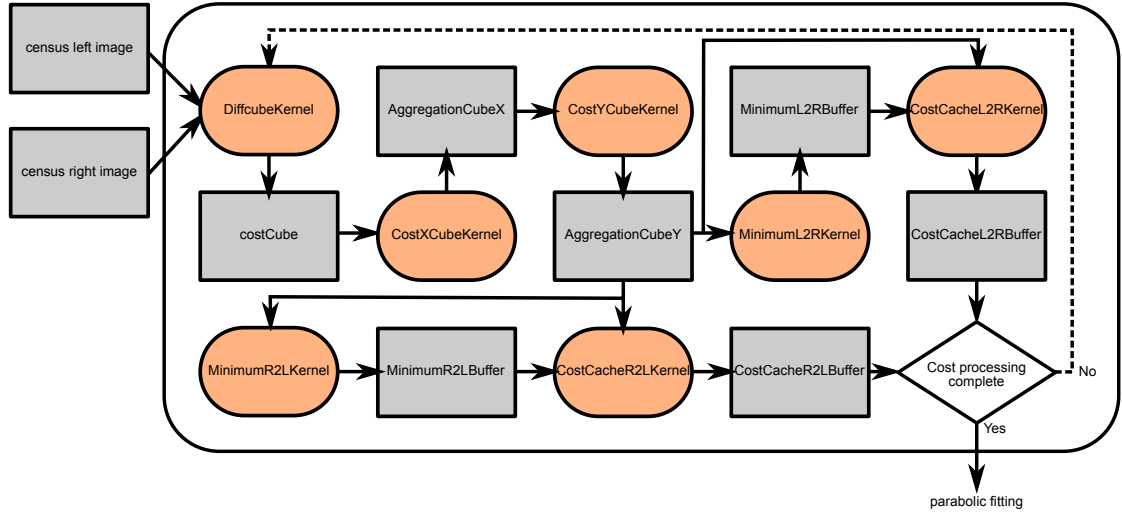


Figure 5.3: Data flow between buffers (gray) and kernels (orange) to iteratively calculate the aggregated costs, find the minimum of the aggregated costs at each pixel position and cache the minimum costs in addition to their neighbors along the disparity axis into a buffer for both left-to-right and right-to-left search direction. The problem space is subdivided along the y-axis of the image dimensions.

due to random access to the global memory.

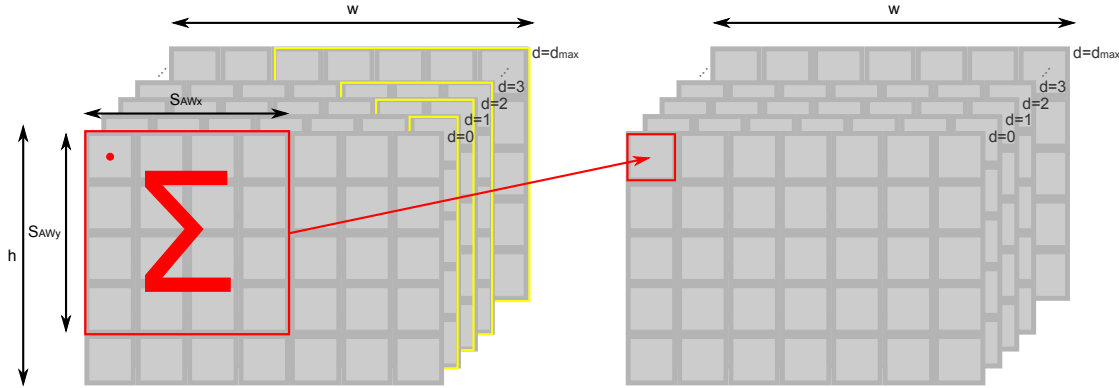


Figure 5.4: Calculation of the DSI for the aggregated cost function. Starting with a non-aggregated DSI (on the left) a window function (highlighted in red) is applied to this DSI, as discussed in Section 2.1.3.3. The calculated sum is then written to a second DSI (on the right). The areas highlighted by yellow borders in the cost volume mark positions containing cost values of non-overlapping areas, as described in Section 2.1.3.2.

After the aggregation steps, the DSI of aggregated matching costs is searched for minimum matching costs along the disparity axis for every pixel position. The aggregated cost value DSI can be used for both right-to-left minimum search and left-to-right minimum search, this is shown in Figure 5.5. The disparities with minimal cost's disparity values are written to the buffers *MinimumL2RBuffer* and *MinimumR2LBuffer*. In an additional step, the aggregated cost values at the minimum cost position and the aggregated costs at the neighboring positions along the disparity axis are written to the buffers *CostCacheR2LBuffer* and *CostCacheL2RBuffer*.

Using the disparity values in the buffers *MinimumL2RBuffer* and *MinimumR2LBuffer* and the disparity cost tripple in *CostCacheR2LBuffer* and *CostCacheL2RBuffer*, a parabolic fitting of the disparity costs is used to determine a sub-pixel disparity. Using Formula 4.5, sub-pixel disparities can be calculated. This is done once for the search direction left image to right image and once for the search direction right image to left image. Formula 4.5 is based on the assumption that the continuous cost function has a minimal turning point that lies near the minimal turning point of its discrete approximation. To locate the minimal turning point near the minimum of the discrete cost function for one pixel, one determines the solution to  $F'_C(d) = 0$ <sup>1</sup> where  $F_C(d)$ <sup>2</sup> represents the continuous cost function, approximated by  $C_{M,N,x,y}(d)$  with  $M$  and  $N$  as the aggregation window's size in x and y direction. Later on we will use  $y(d)$  as a shorthand for cost functions  $C_{M,N,x,y}(d)$  with arbitrary values for  $M$ ,  $N$ ,  $x$  and  $y$ .

We also define  $x_p, x_c, x_n \in \mathbb{R}$  as  $x_n - 1 = x_c = x_p + 1 = y(d_{\min})$ . These variables are

<sup>1</sup> $F'_C(d)$  is approximated through Formula 4.4

<sup>2</sup> $F_C(d)$  is approximated through Formula 4.3

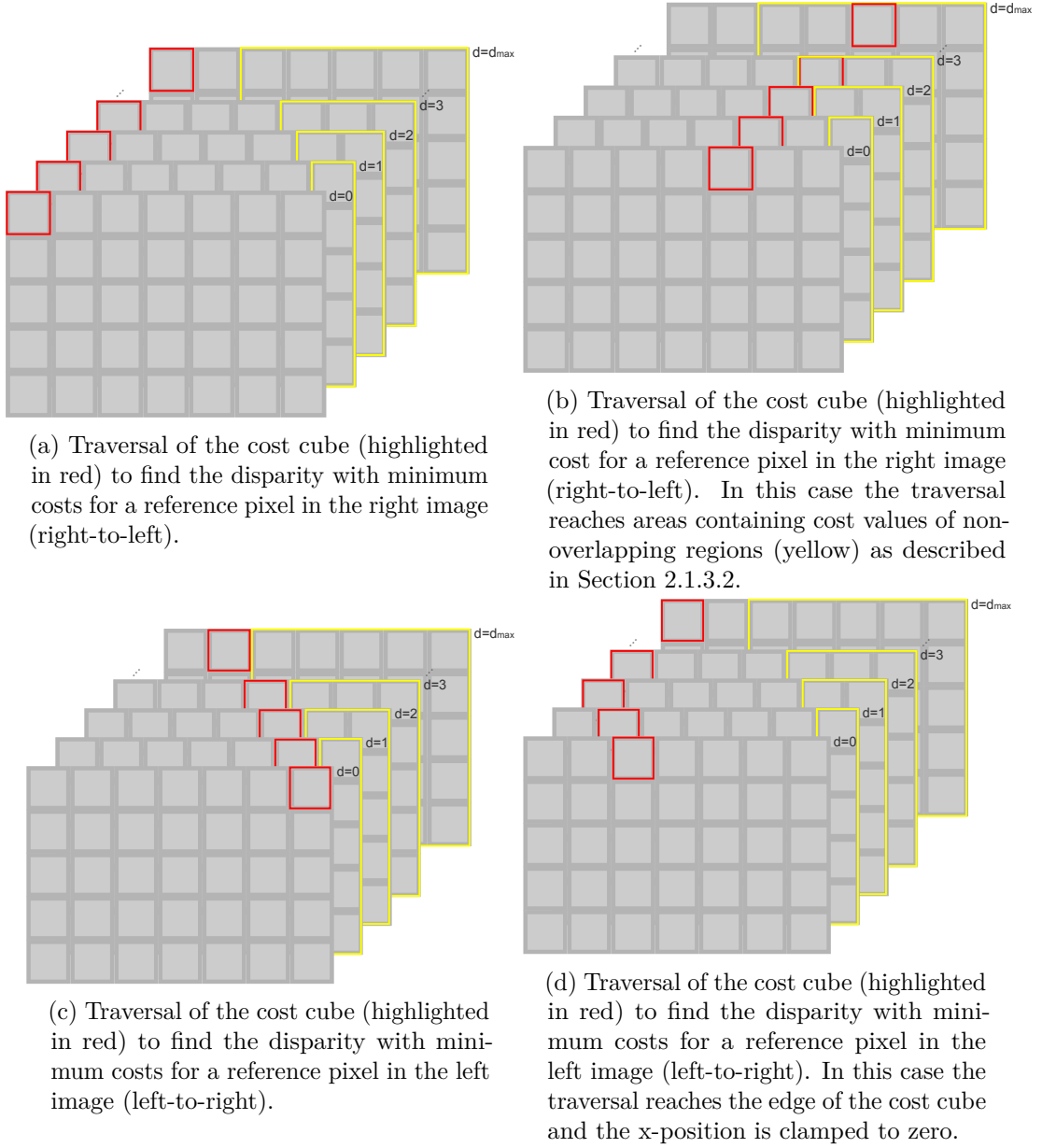


Figure 5.5: Search pattern for disparities with minimum costs in the DSI cost cube in the right-to-left (a,b) and left-to-right (c,d) setting.



used to determine a sub-pixel disparity. The sub-pixel disparity is calculated by fitting a parabola to the three points  $x_p, x_c, x_n$ . This method is described in Section 4.3.

The results are then checked for correspondence. The best matching disparity for one point in the reference image addresses another point in the corresponding image. The best disparity match in reversed search direction for this point should be within an epsilon ( $\epsilon$ ) of the disparity of the reference image for this point. This is a consistency check for the found disparity. If the distance between the disparities is small enough, the disparities are considered to be correct. If the distance between the disparities is too big, the result pixel is marked as invalid. Figure 5.6 visualizes this consistency check.

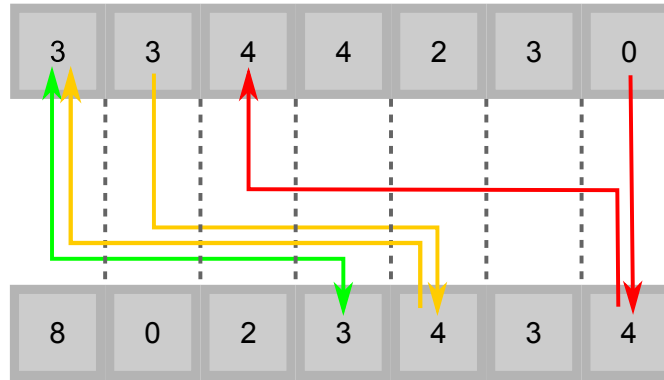


Figure 5.6: Visualization of the consistency check described in Section 4.4. The upper line of values shows one right-to-left disparity result line. The lower line of values shows one left-to-right disparity result line. The perfectly passed consistency check example is highlighted by a green arrow. The not perfectly passed consistency check example is highlighted by yellow arrows. The example for a failed consistency check is highlighted by red arrows.

The kernels described in Section 5.3 to Section 5.10 have a basic set of parameters and setup. The parameters for setup are the image dimensions ( $w, h$ ), the maximum disparity ( $d_{\max}$ ) and the OpenCL device's `CL_DEVICE_MAX_MEM_ALLOC_SIZE` ( $S_{MMA}$ ). One additional runtime constraint is given by the y-dimension of the cost aggregation window ( $S_{AW_y}$ ) which is the minimum for the parameter  $k_{\max}$  which in turn is calculated from these parameters.

The matching process starts by loading the input data into the input buffers in the OpenCL device's memory and then follows the order of kernel calls as outlined in Figure 5.2 and Figure 5.3. In our implementation, we check whether cost processing is complete by checking whether the current iteration number is the maximum iteration number ( $k_{\max}$ ).

The initialisation of the buffers and the calls to start the kernels on the OpenCL device may be implemented in any language that sufficiently exposes the OpenCL API. Our

implementation provides one version that is written in *C++* and one version that is written in *Python 3*.

## 5.2 RectificationKernel

This section will describe the kernel *RectificationKernel* as shown in Figure 5.1.

This implementation rectifies gray scale images with one byte per pixel as described in Section 2.1.2. For this, one input buffer, containing the distorted image, and one output buffer, are needed. Further, two buffers containing the rectification maps have to be provided. The rectification maps contain floating point values.

This results in the following memory consumption  $|RK|$  for rectification of one image:

$$|RK| = (i_1 + f_4 + f_4 + i_1) \cdot w \cdot h = 10 \cdot w \cdot h \quad (5.4)$$

However, the rectification of images is an optional processing step in our pipeline. This is why consideration has to be given to which portion of this memory consumption is optional.

If the provided input images are already rectified, i.e. the rectification step is not required to be applied, then the buffers for the rectification maps and the output buffers for the rectification results are not needed. We rewrite Formula 5.4, by removing the input buffers from the formula, for the optional memory consumption per rectification  $|RK_o|$  as:

$$|RK_o| = (i_1 + f_4 + f_4) \cdot w \cdot h = 9 \cdot w \cdot h \quad (5.5)$$

The rectification maps provide coordinates  $x'$  and  $y'$  at which the intensity values are sampled from the unrectified image for every position  $(x, y)$ . However, the coordinates of point  $(x', y')$  are floating point coordinates. Therefore, bilinear interpolation is used to find the new intensity value for point  $(x, y)$ . The rectification maps can be calculated by the OpenCV command `initUndistortRectifyMap`<sup>3</sup>. Rectification maps have to be calculated for every camera in the system. However, this only has to be done once and may be done prior to the matching process.

The following code sample shows our kernel for rectification. The buffers needed for the rectification operation are provided as *global*<sup>4</sup> pointers of the corresponding type and input buffers in Listing 1 are marked *const*.

```
1 kernel void RectificationKernel(  
2     global const uchar *input,
```

---

<sup>3</sup>see [http://docs.opencv.org/2.4/modules/imgproc/doc/geometric\\_transformations.html#initundistortrectifymap](http://docs.opencv.org/2.4/modules/imgproc/doc/geometric_transformations.html#initundistortrectifymap)

<sup>4</sup>The keyword *global* declares pointers to global memory.

```

3     global const float *mapx,
4     global const float *mapy,
5     global      uchar *output
6 )
7 {
8     //get current pixel position and image dimensions
9     int col  = get_global_id(0);
10    int cols = get_global_size(0);
11    int row  = get_global_id(1);
12    int rows = get_global_size(1);
13
14    //calculate sampling coordinates
15    int y  = floor(mapy[row*cols+col]);
16    int y1 = min(y+1,rows-1);
17    int x  = floor(mapx[row*cols+col]);
18    int x1 = min(x+1,cols-1);
19
20    //calculate interpolation factors
21    float interpol_y = mapy[row*cols+col]-(float)y;
22    float interpol_x = mapx[row*cols+col]-(float)x;
23
24    //get intensity values for bilinear interpolation
25    uchar p00, p10, p01, p11;
26    p00 = input[y *cols+x ];
27    p10 = input[y *cols+x1];
28    p01 = input[y1*cols+x ];
29    p11 = input[y1*cols+x1];
30
31    //bilinear interpolation
32    uchar res =
33        (p00*(1.0-interpol_x)+p10*interpol_x)*(1.0-interpol_y)
34    + (p01*(1.0-interpol_x)+p11*interpol_x)*      interpol_y;
35
36    output[row*cols+col] = res;
37 }

```

Listing 1: RectificationKernel

The basic working steps of this kernel consist of procuring the current work item's id to identify the current pixel, retrieve and process the coordinates  $x'$  and  $y'$  from the rectification maps to get sampling coordinates and interpolation factors, sample the input image at the calculated sampling coordinates and finally use the interpolation factors to calculate the new intensity value for point  $(x, y)$ .

This kernel is invoked for every pixel in the input image. The *global size*, as described in Chapter 2.3, has two dimensions corresponding to the image dimensions and has the vector of the image dimensions  $(w, h)$  as value. The *work group size*, as described in Chapter 2.3, is dependent on the OpenCL device's `CL_DEVICE_MAX_WORK_GROUP_SIZE` parameter ( $S_{WG}$ ) which describes the upper bound of the number of work items per work group. The work group size for any dimension of a problem has to be a whole-numbered divisor of that dimension's global size. The following code sample (Listing 2) shows a strategy to find the biggest possible work group size.

```
1  //get the device's CL_DEVICE_MAX_WORK_GROUP_SIZE parameter
2  int maxWGsize = getWorkGroupSize(device);
3  //get the size of the dimension in question of the data
4  int data_size = getDataSize(data);
5
6  int Swg;
7  //if problem width fits < work item number
8  if( data_size < maxWGsize ) {
9      //then use problem size
10     Swg = data_size;
11 } else {
12     /**
13      * initialize the divisor with int number that is
14      * nearest to the float quotient data_size/maxWGsize
15      */
16     int divisor = ceil((float)data_size/(float)maxWGsize);
17
18     /**
19      * iterate all possible divisors until one is found
20      * that divides data_size without remainder
21      * worst case: if divisor reaches data_size
22      */
23     while((data_size%divisor) != 0)
24         divisor++;
25
26     /**
27      * this is guaranteed to be an integer division
28      * worst case: data_size == divisor -> Swg == 1
29      */
30     Swg = data_size/divisor;
31 }
```

Listing 2: Find biggest possible work group size.

The resulting work group size for the rectification kernel is the vector  $(S_{WG}, 1)$ , which divides the problem space  $w \times h$  into  $\frac{w}{S_{WG}} \cdot h$  equally sized portions.

### 5.3 CensusKernel

This section will describe the kernel *CensusKernel* as shown in Figure 5.2.

This implementation transforms grayscale images with one byte per pixel using the census transform as described in Chapter 4.1. For this implementation one input buffer, containing the rectified image and one output buffer are needed. The output buffer needs to be an array of an eight byte data type i.e.  $i_8$  or  $f_8$ .

This results in the following memory consumption  $|C|$  for census transformation of one image:

$$|C| = (i_1 + i_8) \cdot w \cdot h = 9 \cdot w \cdot h \quad (5.6)$$

The following code sample - Listing 3 shows our kernel for census transformations. The basic steps of this kernel consist of procuring the current work item's id, identifying the current pixel, retrieving the current reference value, iterating through pixel addresses, comparing the values at these pixel addresses with the reference value, appending the comparison result to the census word and finally writing the census word into the output buffer.

```

1  /**
2   * Values defined as compile parameters:
3   * - size_x = 16
4   * - size_y = 16
5   * - gap_x = 1
6   * - gap_y = 1
7   * - centerindex_x = 7
8   * - centerindex_y = 7
9   * - upperrange = 0
10  * - lowerrange = 255
11  *
12  * For explanation on these ranges see Line 57
13  */
14
15  kernel void CensusKernel
16  (
17      global const uchar *input,
18      global          ulong *output
19  )
20  {
21      //get current pixel position and image dimensions

```

```
22  int cols= get_global_size(0);
23  int col = get_global_id(0);
24  int rows= get_global_size(1);
25  int row = get_global_id(1);
26
27  //initialize shift and value
28  uchar value = 0;
29  uchar shift = 1;
30
31  //initialize census word
32  ulong census = 0;
33
34  int img_x, img_y;
35
36  //bitmask is used in Line 97
37  ulong bitmask = 0xfffffffffffffffe;
38
39  //get current reference value
40  uchar center_value = input[cols*row+col];
41  uchar current_value;
42
43  //iterate through pixel addresses
44  for (int y = 0; y < size_y; y+=(1+gap_y))
45  {
46      for(int x = 0; x < size_x; x+=(1+gap_x))
47      {
48          value = 0;
49
50          //clamp pixel address to image bounds
51          img_x = clamp(col - centerindex_x + x, 0, cols-1);
52          img_y = clamp(row - centerindex_y + y, 0, rows-1);
53
54          //sample value from image
55          current_value = input[cols*img_y+img_x];
56
57          /**
58           * bit value is 1 if current_value is
59           * inside window around center_value
60           * 0 otherwise
61           *
62           * 0 to (center_value-lowerrange)  -> 0
63           * (center_value-lowerrange) to
64           * (center_value+upperrange) -> 1
```

```

65      * (center_value+upperrange) to 255 -> 0
66      *
67      * With upperrange = 0 and
68      * lowerrange = 255 the window results in:
69      * 0 to center_value -> 1
70      * center_value to 255 -> 0
71      ***/
72      value =
73      (
74          clamp(center_value + upperrange,0,255)
75          > current_value
76      ) && (
77          clamp(center_value - lowerthreshold,0,255)
78          < current_value
79      )?1:0;
80
81      //shift previous result by 1 (except in center position)
82      census = census << shift;
83
84      /***
85      * if current address is the reference pixel
86      * then set shift=0 for next iteration
87      ***/
88      shift =
89          !(x == centerindex_x &&
90            y == centerindex_y)?1:0;
91
92      //make sure the bit is initialized with 0
93      census = census & bitmask;
94      //set current census bit
95      census = census | value;
96  }
97  }
98  //save census word to output array
99  output[col+row*cols] = census;
100 }

```

Listing 3: CensusKernel

This kernel is invoked for every pixel in the input image. The *global size* parameter, as described in Chapter 2.3, has two dimensions corresponding to the image dimensions and the image dimension vector  $(w, h)$  is used as value for this parameter. The *work group size*, as described in Chapter 2.3, has the value  $(S_{WG}, 1)$ .

Our implementation provides the possibility to influence the census mechanism described by Formula 4.1. An upper and a lower range ( $R_l, R_u$ ) are provided to allow regulation of the intensity range, with respect to the reference value ( $c$ ) that is used to determine if the corresponding bit in the census word for a pixel's value ( $p$ ) will be set to one or zero.

$$b_i = \begin{cases} 1, & \text{if } c - R_l \leq p \leq c + R_u \\ 0, & \text{otherwise} \end{cases} \quad (5.7)$$

To achieve the cost function described by Formula 4.1, the lower bound ( $R_l$ ) has to be set to 255 and the upper bound ( $R_u$ ) has to be 0. This results in the following formula for the comparison of one pixel's value ( $p$ ) with the central value ( $c$ ):

$$b_i = \begin{cases} 1, & \text{if } c - 255 \leq p \leq c + 0 \\ 0, & \text{otherwise} \end{cases} \quad (5.8)$$

## 5.4 DiffCubeKernel

This section will describe the kernel *DiffCubeKernel* as shown in Figure 5.3.

This implementation uses the census words in *census left image* and *census right image* to calculate a DSI containing the Hamming-distances as described in Chapter 2.1.3.2. The parameters for this are the two buffers, containing the census transformed images, one output buffer for the DSI and one four-byte integer variable ( $i_4$ ).

Due to memory restrictions, it is possible that the whole DSI will not fit into the global memory available and the DSI has to be split into multiple parts along the y-axis of the input images. The parameter  $k_{\max}$  that was introduced at the start of this chapter, represents the number of lines that fit into global memory. This means that an iteration of the cost calculation process does not know the full size of the problem space from the parameters provided by OpenCL. Therefore, the four-byte integer parameter contains the height of the input maps ( $h$ ).

The output buffer for this function will contain a slice of the DSI. The size of this slice is  $w \cdot k_{\max} \cdot d_{\max}$ . The Hamming-distance of two 64-bit census words will be a whole-numbered value in the range of  $[0 - 64]$  and the data type therefore only needs to be a one-byte integer ( $i_1$ ).

We define  $k_{\max}$  as a function of the used OpenCL device's `CL_DEVICE_MAX_MEM_ALLOC_SIZE` ( $S_{\text{MMA}}$ )<sup>5</sup> parameter (which is usually smaller than a quarter of total global memory) and the dimension of the problem space  $w \times h \times d_{\max}$ . The following formula (Formula 5.9) shows how the maximum number of lines per iteration is calculated as the maximum memory allocation size divided by the size of a one-row-slice of the DSI.

---

<sup>5</sup>see <https://www.khronos.org/registry/OpenCL/sdk/1.0/docs/man/xhtml/clGetDeviceInfo.html>



$$k_{\max} = \min \left( \left\lfloor \frac{S_{\text{MMA}}}{i_4 \cdot (d_{\max} + 1) \cdot w \cdot 1.5} \right\rfloor, h \right) \quad (5.9)$$

All buffers that are used in the cost processing loop have to satisfy the condition that they have to fit into global memory. In addition to this constraint, multiple buffers have to be allocated, which constrains the value of  $k_{\max}$  even further. For these reasons, the slice size is multiplied by  $i_4$  as the biggest data type used in the loop and by a factor of 1.5. The factor 1.5 was chosen to reduce the total size of the three DSI buffers that are used in the cost processing loop to a maximum of half the total global memory, in order to allow the allocation of the remaining buffers used in the process.

The memory usage of the buffers containing the census words was already accounted for in Section 5.3. This results in the following memory consumption  $|\text{HD}|$  for calculating Hamming-distances:

$$|\text{HD}| = (i_1 \cdot w \cdot k_{\max} \cdot d_{\max}) + i_4 = (w \cdot k_{\max} \cdot d_{\max}) + 4 \quad (5.10)$$

The following code sample shows our kernel for the Hamming-distance calculation. The basic working steps of this kernel are procuring the current work item's id and the work offset to identify the current pixel, retrieve the current values from the left and right census map, calculating the Hamming-distance and finally writing the distance value to the output buffer.

```

1  kernel void DiffCubeKernel
2  (
3      global const ulong *left,
4      global const ulong *right,
5      global      uchar *disp,
6                  int max_rows
7  )
8  {
9      //get problem size and pixel position:
10     int cols      = get_global_size(0);
11     int rows      = get_global_size(1);
12
13     int row_offset = get_global_offset(1);
14
15     int col        = get_global_id(0);
16     int row        = get_global_id(1);
17     int disparity  = get_global_id(2);
18
19     //get values from census maps for current pixel position
20     int row_offset = clamp(row, 0, max_rows-1)*cols;
21     ulong l = left [row_offset

```

```

22         +clamp(col+disparity,0,cols-1)];
23     ulong r = right[row_offset +col];
24
25     //l XOR r
26     ulong diff = l^r;
27     /**
28      * in openCL 1.2+ a function popcount(diff) exists
29      * this implementation should work with opencl 1.0
30      * https://en.wikipedia.org/wiki/Hamming_weight
31      */
32
33     //define masks
34     // m1 = B01010101 01010101 01010101 01010101 01010101 ...
35     const ulong m1 = 0x5555555555555555;
36     // m2 = B00110011 00110011 00110011 00110011 00110011 ...
37     const ulong m2 = 0x3333333333333333;
38     // m4 = B00001111 00001111 00001111 00001111 00001111 ...
39     const ulong m4 = 0x0f0f0f0f0f0f0f0f;
40     //h01 = B00000001 00000001 00000001 00000001 00000001 ...
41     const ulong h01= 0x0101010101010101;
42
43     //calculate the sum of every two bits
44     //two-bit sum, results in range [0-2]
45     diff -= (diff >> 1) & m1;
46     //calculate the sum of every two two-bit sums
47     //four-bit sum, result in range [0-4]
48     diff = (diff & m2) + ((diff >> 2) & m2);
49     //calculate the sum of every two four-bit sums
50     //eight-bit sum, result in range [0-8]
51     diff = (diff + (diff >> 4)) & m4;
52     //calculate the sum of all eight-bit sums
53     diff = (diff * h01) >> 56;
54
55     disp[disparity*rows*cols //disparity slice
56         + (row-row_offset)*cols //row in that slice
57         + col //column in that row
58     ] = (uchar)diff;
59 }

```

Listing 4: DiffCubeKernel

This kernel in Listing 4 is invoked iteratively for every pixel in the input image. The *global size*, as described in Chapter 2.3, has three dimensions corresponding to the image

width  $w$ , the maximum number of lines  $k_{\max}$  and the maximum disparity  $d_{\max}$ . Therefore, the vector  $(w, k_{\max}, d_{\max} + 1)^6$  is used as value for the parameter *global size*. The *work group size* parameter, as described in Chapter 2.3, has the value  $(S_{WG}, 1, 1)$ .

All kernels in our implementation that operate iteratively, due to memory limits, need offset information. This offset information is provided as a vector containing the offset for every used dimension and depends on the iteration number of the current iteration ( $i$ ). The offset for processing sections of the DSI, has to consider overlapping regions. The overlapping region for processing a DSI slice has to allow further processing of the data as if the whole DSI would fit into global memory. In the case of the algorithm by [HZW<sup>+</sup>10] and due to our choice of segmenting the DSI along the y-axis of the images, the overlapping region is the number of lines that are used to aggregate in y-direction ( $Y_{\text{agg}}$ ) in kernel *CostYCubeKernel* as shown in Figure 5.3. Therefore the offset vector has the value  $(0, i \cdot (k_{\max} - Y_{\text{agg}}), 0)$ .

OpenCL adds the offset vector to the vector of every work item's global IDs, i.e. for offset vector  $(l, m, n)$  the global-id-vector will be  $(x_g + l, y_g + m, z_g + n)$  with  $x_g, y_g$  and  $z_g$  representing the global ids of the work items in the case where offset is the zero-vector. This means that global ids can be used to read the full problem space. In this case, this is the whole range of any of the census maps. Corresponding addresses within the DSI buffer have to be reduced by the offset to start at zero. This can be seen in Listing 4 Line 56.

## 5.5 CostXCubeKernel

This section will describe the kernel *CostXCubeKernel* as shown in Figure 5.3.

This implementation calculates the support aggregation in two consecutive steps. The first step of the support aggregation is represented by *CostXCubeKernel* and performs the aggregation along the x axis. The size of the complete aggregation window is defined as  $S_{AW_x} \times S_{AW_y}$ . The aggregation window for this kernel only uses the x component of this window  $S_{AW_x} \times 1$ . The parameters for this are the buffer containing the slice of the DSI described in Section 5.4 and an output buffer that will contain the result of this operation. Figure 5.7 visualizes how this kernel reads data from the *Hamming Distance Buffer* and writes the sums to the *Cost X-Aggregation Buffer*.

The memory usage of the buffer containing the Hamming-distances was already accounted for in the previous section. This results in the following memory consumption for *Cost X-Aggregation Buffer* ( $|AG_x|$ ):

$$|AG_x| = i_4 \cdot w \cdot k_{\max} \cdot d_{\max} + i_4 = 4 \cdot w \cdot k_{\max} \cdot d_{\max} + 4 \quad (5.11)$$

The following code sample shows our kernel for aggregation of costs along the x axis. The basic working steps of this kernel are procuring the current work item's id to identify the

---

<sup>6</sup> $[[0 - d_{\max}]] = d_{\max} + 1$

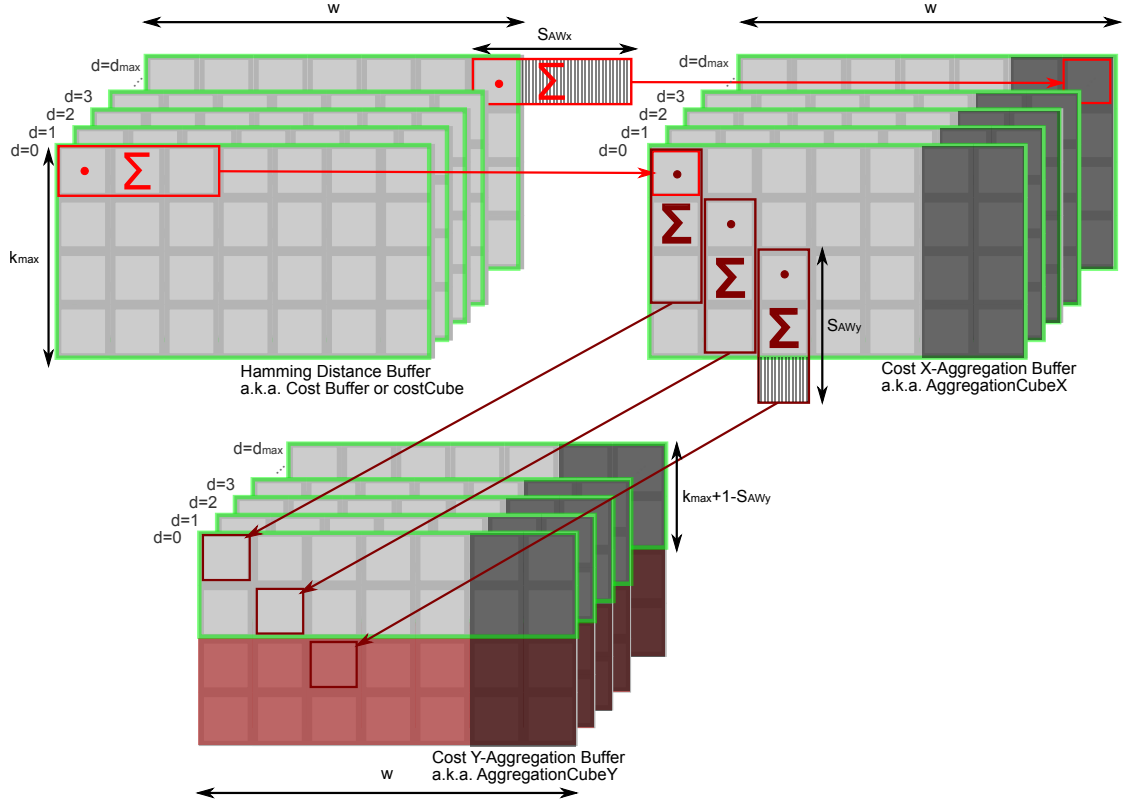


Figure 5.7: Calculation of aggregated cost values by combination of *CostXCubeKernel* and *CostYCubeKernel*. The window function is visualized through light red rectangles. If the aggregation window reaches over the right edge of the DSI, the resulting aggregated cost values are less meaningful. This is inherent to the used method. This less meaningful area is highlighted by a gray overlay in the disparity layers of the *Cost X-Aggregation Buffer*. The *Cost X Aggregation Buffer* is used to calculate sums along the  $y$  direction for every pixel in that DSI in a range of  $S_{AWy}$ . This is visualized by dark red rectangles. If the aggregation window reaches over the lower bound of the DSI, the resulting aggregated costs are invalid. This results in slices of height  $k_{\max} + 1 - S_{AWy}$  of aggregated costs per iteration. The invalid aggregated costs (highlighted by a light red overlay) are discarded. The light green rectangles highlight the regions in the buffers which will be processed by the next kernel in the pipeline.

current pixel, calculate the sum of the cost values in the range of the current position in the DSI to the current position plus the aggregation window size in x direction and finally writing the aggregated value to the output buffer.

```

1  /**
2   * Values defined as compile parameters:
3   * - aggregation_x = 5
4   */
5
6  kernel void CostXCubeKernel
7  (
8      global const uchar *input,
9      global      uint  *output
10 )
11 {
12     int cols = get_global_size(0);
13     /**
14      * In slicing scenario,
15      * starting from 2nd iteration
16      * global ID will be smaller than group ID
17      */
18     int row  = get_group_id(1);
19     int rows = get_global_size(1);
20     int col  = get_global_id(0);
21     int disparity = get_global_id(2);
22
23     uint c=0;
24     int offset = disparity*rows*cols + row*cols ;
25     for(int i=0; i<aggregation_x; i++){
26         c+=input[offset + clamp(col+i,0,cols-1)];
27     }
28     output[offset + col] = c;
29 }

```

Listing 5: CostXCubeKernel

The kernel in Listing 5 is invoked iteratively for every pixel in the input image. The *global size*, as described in Chapter 2.3, has three dimensions corresponding to the image width  $w$ , the maximum number of lines  $k_{\max}$  and the disparity depth  $d_{\max}$  and has therefore the vector  $(w, k_{\max}, d_{\max} + 1)$  as value. The *work group size*, as described in Chapter 2.3, has the value  $(S_{WG}, 1, 1)$ . This kernel does not need an offset vector because every pixel in the input buffer corresponds to a pixel in the output buffer and the input buffer is processed completely.

## 5.6 CostYCubeKernel

This section will describe the kernel *CostYCubeKernel* as shown in Figure 5.3.

This is the second step of the support aggregation and represents the aggregation along the y axis. The parameters for this are the buffer containing the slice of the DSI described in Section 5.5 and an output buffer that will contain the result of this operation. Figure 5.7 visualizes how this kernel reads data from the *Cost X-Aggregation Buffer* and writes the sums to the *Cost Y-Aggregation Buffer*. The aggregation window for this kernel has the size  $1 \times S_{AW_y}$ .

The memory usage of the buffer containing the intermediate results generated by the kernel *CostXCubeKernel* was already accounted for in the previous section. This results in the following memory consumption  $|AG_y|$  for calculating Hamming-distances:

$$|AG_y| = i_4 \cdot w \cdot k_{\max} \cdot d_{\max} + i_4 = 4 \cdot w \cdot k_{\max} \cdot d_{\max} + 4 \quad (5.12)$$

The following code sample shows our kernel for aggregation of costs along the x axis. This kernel procures the current work item's id to identify the current pixel, calculates the sum of the cost values in the range of the current position in the DSI to the current position plus the aggregation window size in x direction and finally writes the aggregated value to the output buffer.

```

1  /**
2   * Values defined as compile parameters:
3   * - aggregation_y = 5
4   */
5  kernel void CostYCubeKernel
6  (
7      global const uint *input,
8      global      uint *output
9  )
10 {
11     int cols = get_global_size(0);
12     int rows = get_global_size(1);
13     int col  = get_global_id(0);
14     /**
15      * In slicing scenario,
16      * starting from 2nd iteration
17      * global ID will be smaller than group ID
18      */
19     int row  = get_group_id(1);
20     int disparity = get_global_id(2);
21
22     uint c=0;

```

```

23     int offset = disparity*rows*cols + col
24     for(int i=0; i<aggregation_y; i++){
25         c+=input[offset + cols*clamp(row+i,0,rows-1)];
26     }
27     output[offset+ row*cols] = c;
28 }

```

Listing 6: CostYCubeKernel

The kernel in Listing 6 is invoked iteratively for every pixel in the input image. The *global size* has three dimensions corresponding to the image width  $w$ , the maximum number of lines  $k_{\max}$  and the maximum disparity  $d_{\max}$ . The vector  $(w, k_{\max}, d_{\max} + 1)$  is used as value for the *global size* parameter. The *work group size*, as described in Chapter 2.3, has the value  $(S_{WG}, 1, 1)$ . This kernel does not need an offset vector because every pixel in the input buffer corresponds to a pixel in the output buffer and the input buffer is processed completely.

## 5.7 MinimumKernels

This section will describe the kernels *MinimumR2LKernel* and *MinimumL2RKernel* as shown in Figure 5.3.

This implementation iterates through the DSI cost cube in the *Cost Y-Aggregation Buffer*. The parameters for these kernels are the buffer containing the slice of the cost cube described in Section 5.6 and an output buffer that will contain the result of the operation. Figure 5.5a and Figure 5.5b visualize how the kernel in Listing 7 traverses the data from the *Cost Y-Aggregation Buffer* and writes the disparity values with minimum cost to the buffer *MinimumR2LBuffer*. Figure 5.5c and Figure 5.5d visualize how the kernel in Listing 8 traverses the data and writes the disparity values to the buffer *MinimumL2RBuffer*.

The memory usage results in the following memory consumption  $|M|$  for the buffer *MinimumR2LBuffer*:

$$|M| = i_4 \cdot w \cdot h = 4 \cdot w \cdot h \quad (5.13)$$

The following code samples (Listing 7 and Listing 8) show our kernels for minimum search in right-to-left and left-to-right direction. The kernels procure the current processing element's working position and traverse the aggregated costs in the buffer *costcube* along the disparity axis to find the disparity with the minimum aggregated costs.

```

1  /**
2   * Needed Compiletime-Values:
3   * - max_disparity = d_max ... commandline parameter
4   */

```

```

5 | kernel void Minimumr2lKernel
6 | (
7 |     global const uint *costcube,
8 |     global      uint *out,
9 |         int max_rows
10 | )
11 | {
12 |     int col      = get_global_id(0);
13 |     int cols     = get_global_size(0);
14 |     /**
15 |      * in slicing scenario
16 |      * starting from 2nd iteration
17 |      * global id will be smaller than group id
18 |      ***/
19 |     int row      = get_group_id(1);
20 |     int rows     = get_global_size(1);
21 |     int row_offset = get_global_offset(1);
22 |
23 |     int slab_size = rows*cols;
24 |
25 |     uint cost      = costcube[slab_size + col];
26 |
27 |     uint cur_cost = 0;
28 |     uint disp = 0;
29 |
30 |     //Global mem-cache optimization
31 |     barrier( CLK_GLOBAL_MEM_FENCE );
32 |
33 |     int offset = row*cols + clamp(col,0,cols-1);
34 |
35 |     for(int i = 1; i <= max_disparity; i++){
36 |         cur_cost = costcube[offset + i*slab_size];
37 |         disp = cur_cost < cost?i:disp;
38 |         cost = cur_cost < cost?cur_cost:cost;
39 |         //Global mem-cache optimization
40 |         barrier( CLK_GLOBAL_MEM_FENCE );
41 |     }
42 |     out[clamp(row+row_offset,0,max_rows-1)*cols + col] = disp;
43 | }

```

Listing 7: MinimumR2LKernel

```

1 | /**

```



```

2  * Needed Compiletime-Values:
3  * - max_disparity = d_max ... commandline parameter
4  */
5  kernel void MinimumL2rKernel
6  (
7      global const uint *costcube,
8      global      uint *out,
9              int max_rows
10 )
11 {
12     int col      = get_global_id(0);
13     int cols     = get_global_size(0);
14     int row      = get_group_id(1);
15     int rows     = get_global_size(1);
16     int row_offset = get_global_offset(1);
17
18     int slab_size = rows*cols;
19
20     uint cost = costcube[slab_size + col ];
21
22     uint cur_cost = 0;
23     uint disp = 0;
24
25     //Global mem-cache optimization
26     barrier( CLK_GLOBAL_MEM_FENCE );
27
28     int offset = row*cols;
29     for(int i = 1; i < max_disparity+1; i++){
30         cur_cost = costcube[
31             i*slab_size +
32             offset +
33             clamp(col-i,0,cols-1)
34         ];
35         disp = cur_cost < cost?i:disp;
36         cost = cur_cost < cost?cur_cost:cost;
37         //Global mem-cache optimization
38         barrier( CLK_GLOBAL_MEM_FENCE );
39     }
40     out[clamp(row+row_offset,0,max_rows-1)*cols + col] = disp;
41 }

```

Listing 8: MinimumL2RKernell

These kernels are invoked multiple times to determine the disparity with minimum costs for every pixel in the current data slice. The *global size*, as described in Chapter 2.3, has two dimensions corresponding to the image width  $w$  and the maximum number of lines  $k_{\max}$ . The value for the *global size* parameter is the vector  $(w, k_{\max})$ . The *work group size*, as described in Chapter 2.3, has the value  $(S_{\text{WG}}, 1)$ . This kernel needs an offset vector because the input buffer does not correspond to the output buffer as a whole but only to a portion of  $k_{\max}$  lines.

## 5.8 CostCacheKernels

This section will describe the kernels *CostCacheR2LKernel* and *CostCacheL2RKernel* as shown in Figure 5.3. This implementation samples the aggregated cost volume at the minimal cost position which was determined by the corresponding minimum kernel. Furthermore, the two neighboring pixels along the disparity axis are sampled. These three cost values are written into a buffer for later parabolic fitting.

The memory usage results in the following memory consumption  $|CC|$  for one cost cache buffer i.e. *CostCacheR2LBuffer*, *CostCacheL2RBuffer*:

$$|CC| = (i_4 + i_4 + i_4) \cdot w \cdot h = 12 \cdot w \cdot h \quad (5.14)$$

The following code samples (Listing 9 and Listing 10) show our kernels for cost caching.

```

1  /**
2   * mirror index at 0 and max disparity
3   * md = max disparity
4   * ...-2 -1 0 1 2 ... dm-1 dm dm+1 dm+2 ...
5   * ... 2 1 0 1 2 ... dm-1 dm dm-1 dm-2 ...
6   * In case of minimum cost at 0 or max disparity, this way
7   * the parabolic fitting results to 0 or max disparity
8   */
9  int mirror_index(int x, int max_disparity) {
10     uint mir = abs(max_disparity-abs(max_disparity-x));
11     return mir % (max_disparity+1);
12 }
13
14 kernel void CostCacheR2LKernel
15 (
16     global const uint *costcube,
17     global const uint *minimum,
18     int max_rows,
19     global uint *costCache,
20     int max_disparity
21 )

```

```

22 {
23     int col          = get_global_id(0);
24     int cols         = get_global_size(0);
25     int row          = get_global_id(1);
26     int rows         = get_global_size(1);
27     int row_offset = get_global_offset(1);
28     int mindisp = minimum[clamp(row,0,max_rows-1)*cols + col];
29
30     //get next, current and previous disparity index
31     int fn = mirror_index(mindisp-1,max_disparity);
32     int f0 = mirror_index(mindisp,0,max_disparity);
33     int fp = mirror_index(mindisp+1,max_disparity);
34
35     int slab_size = cols*rows;
36     int offset = (row-row_offset)*cols + clamp(col,0,cols-1) ;
37     uint akt_s = costcube[
38         clamp(fn,0,max_disparity)*slab_size + offset
39     ];
40     uint akt_m = costcube[
41         clamp(f0,0,max_disparity)*slab_size + offset
42     ];
43     uint akt_l = costcube[
44         clamp(fp,0,max_disparity)*slab_size + offset
45     ];
46
47     offset = cols*3*clamp(row,0,max_rows-1)+col*3;
48     costCache[offset+0] = akt_s;
49     costCache[offset+1] = akt_m;
50     costCache[offset+2] = akt_l;
51 }

```

Listing 9: CostCacheR2LKernel

```

1  /**
2   * mirror index at 0 and max disparity
3   * md = max disparity
4   * ...-2 -1 0 1 2 ... dm-1 dm dm+1 dm+2 ...
5   * ... 2 1 0 1 2 ... dm-1 dm dm-1 dm-2 ...
6   * In case of minimum cost at 0 or max disparity, this way
7   * the parabolic fitting results to 0 or max disparity
8   */
9  int mirror_index(int x, int max_disparity) {
10     uint mir = abs(max_disparity-abs(max_disparity-x));

```

```
11     return mir % (max_disparity+1);
12 }
13
14 kernel void CostCacheL2RKernel
15 (
16     global const uint *costcube,
17     global const uint *minimum,
18         int max_rows,
19     global uint *costCache,
20         int max_disparity
21 )
22 {
23     int col = get_global_id(0);
24     int cols = get_global_size(0);
25     int row = get_global_id(1);
26     int rows = get_global_size(1);
27     int row_offset = get_global_offset(1);
28     int mindisp = minimum[clamp(row,0,max_rows-1)*cols + col];
29
30     /**
31      * upper bound for disparity index:
32      * If we hit the wall of the cost cube,
33      * prevent moving farther to the back of the cost cube.
34      * In this case, values from the back of the cube are bad.
35      */
36     int disp_upper = min(max_disparity,col);
37
38     //get next, current and previous disparity index
39     int fn = mirror_index(mindisp-1,max_disparity);
40     int f0 = mirror_index(mindisp-0,max_disparity);
41     int fp = mirror_index(mindisp+1,max_disparity);
42
43     int slab_size = cols*rows;
44     int offset = (row-row_offset)*cols;
45     uint akt_s = costcube[
46         clamp(fn,0,disp_upper)*slab_size +
47         offset + clamp(col-fn,0,cols-1)
48     ];
49     uint akt_m = costcube[
50         clamp(f0,0,disp_upper)*slab_size +
51         offset + clamp(col-f0,0,cols-1)
52     ];
53     uint akt_l = costcube[
```

```

54         clamp(fp, 0, disp_upper)*slab_size +
55         offset + clamp(col-fp, 0, cols-1)
56     ];
57
58     offset = cols*3*clamp(row, 0, max_rows-1)+col*3;
59     costCache[offset+0] = akt_s;
60     costCache[offset+1] = akt_m;
61     costCache[offset+2] = akt_l;
62 }

```

Listing 10: CostCacheL2RKernel

These kernels are invoked iteratively to sample the cost values at the point of minimum cost and its two neighboring DSI positions along the disparity axis for every pixel in the current data slice. The *global size*, as described in Chapter 2.3, has two dimensions corresponding to the image width  $w$  and the maximum number of lines  $k_{\max}$ . Therefore, the vector  $(w, k_{\max})$  is used as value for this parameter. The *work group size*, as described in Chapter 2.3, has the value  $(S_{\text{WG}}, 1)$ . This kernel needs an offset vector because the input buffer does not correspond to the output buffer as a whole but only to a portion of  $k_{\max}$  lines.

## 5.9 ParabolicFittingKernel

The kernel *ParabolicFittingKernel* is shown in Figure 5.2. This implementation uses the three cost values from one CostCacheBuffer and the minimum disparity in the corresponding MinimumCostBuffer to calculate a subpixel estimation of the assumed parabola described in Section 4.3. This is done for the right-to-left and the left-to-right direction.

The memory usage  $|P|$  for this kernel is the size of two floating point buffers with the image dimensions:

$$|P| = (f_4 + f_4) \cdot w \cdot h = 8 \cdot w \cdot h \quad (5.15)$$

The following code sample shows our kernel for aggregated cost sampling.

```

1 kernel void ParabolicFittingKernel
2 (
3     global const uint   *r2lCostCache,
4     global const uint   *l2rCostCache,
5     global const uint   *l2r_disp,
6     global const uint   *r2l_disp,
7     global float        *l2r_ret_data,
8     global float        *r2l_ret_data

```

```

9           )
10 {
11     int col      = get_global_id(0);
12     int cols     = get_global_size(0);
13     int row      = get_global_id(1);
14     int rows     = get_global_size(1);
15
16     float r2l_best_disp = (float)r2l_disp[col+row*cols];
17     float l2r_best_disp = (float)l2r_disp[col+row*cols];
18
19     float f_disp      = r2l_best_disp;
20     float d_prev      = r2lCostCache[cols*3*row + col*3+0];
21     float d_current    = r2lCostCache[cols*3*row + col*3+1];
22     float d_next      = r2lCostCache[cols*3*row + col*3+2];
23     f_disp += ((d_next - d_prev)
24               / (2.f * (2.f * d_current - d_prev - d_next)));
25
26     r2l_ret_data[row*cols + col] = f_disp;
27
28     f_disp      = l2r_best_disp;
29     d_prev      = l2rCostCache[cols*3*row + col*3+0];
30     d_current    = l2rCostCache[cols*3*row + col*3+1];
31     d_next      = l2rCostCache[cols*3*row + col*3+2];
32     f_disp += ((d_next - d_prev)
33               / (2.f * (2.f * d_current - d_prev - d_next)));
34
35     l2r_ret_data[row*cols + col] = f_disp;
36 }

```

Listing 11: ParabolicFittingKernel

The kernel in Listing 11 is invoked for every pixel in the input image. The *global size*, as described in Chapter 2.3, corresponds to the image dimensions  $(w, h)$ . The *work group size*, as described in Chapter 2.3, has the value  $(S_{WG}, 1)$ .

## 5.10 ConsistencyKernel

The kernel *ConsistencyKernel* is shown in Figure 5.2. This kernel uses a left-right consistency check, as described in Section 4.4, to determine whether a calculated disparity value should be kept or discarded.

The memory usage  $|O|$  for this kernel surmounts to one output buffer for floating point

values containing the subpixel disparities:

$$|O| = f_4 \cdot w \cdot h = 4 \cdot w \cdot h \quad (5.16)$$

The following code sample (Listing 12) shows our kernel for aggregated cost sampling.

```

1 kernel void ConsistencyKernel
2 (
3     global const uint *l2r_disp,
4     global const float *l2r_ret_data,
5     global const float *r2l_ret_data,
6     global float *output
7 )
8
9 {
10     int col      = get_global_id(0);
11     int cols     = get_global_size(0);
12     int row      = get_global_id(1);
13     int rows     = get_global_size(1);
14
15     int l2r_best_disp = l2r_disp[col+row*cols];
16
17     int ad = l2r_best_disp;
18     float a = l2r_ret_data[row*cols + col + ad];
19     float b = r2l_ret_data[row*cols + clamp(col-ad, 0, cols-1)];
20
21     float erg = fabs(a-b);
22
23
24     erg = (erg <= 5.0f)? fabs((a + b) / 2.f) : 0.0f;
25
26     output[col+row*cols] = erg;
27
28 }

```

Listing 12: ConsistencyKernel

This kernel is invoked for every pixel in the input image. The *global size*, as described in Chapter 2.3, corresponds to the image dimensions  $(w, h)$ . The vector  $(S_{WG}, 1)$  is used as parameter for the *work group size*, as described in Chapter 2.3.

### 5.11 Conclusion

The total memory consumption ( $S$ ) of all necessary buffers described in this chapter amounts to the following formula:

$$S = 2 \cdot C + HD + AG_x + AG_y + 2 \cdot M + 2 \cdot CC + P + O \quad (5.17)$$

which in turn results in this formula for memory consumption:

$$|S| = 62 \cdot w \cdot h + 9 \cdot k_{\max} \cdot w \cdot d_{\max} \quad (5.18)$$

For rectification an additional  $|RK_o|$  has to be accounted for.



# Evaluation

In this chapter, we will compare the result quality of our implementation to the results of other algorithms. For this comparison, we will use the *Middlebury Stereo Benchmark* as described in Section 2.2. Using data from this benchmark we will show that the results of our algorithm are reliably reproducible on different devices. We will show runtime differences throughout the used OpenCL devices and how the floating point performance of the used device influences the runtime. Further, we will show how our implementation performs compared to the implementation of [HZW<sup>+</sup>10], which was not available to us.

We will compare the runtime results of our algorithm for *nVidia Quadro 1000M* GPU, *GeForce GTX 750 Ti* GPU and *Mali-T760* ARM-GPU. Further we will estimate the runtime of our implementation for multiple devices and compare the estimated runtime with measured runtime. These devices are an *nVidia GTX 1060 3GB*, an *nVidia GTX 1080 Ti*, an *Intel Iris 6100* and an *Intel i7-5557U*.

Our implementation can be broken down into two steps: buffer setup step and matching step. It is possible to process multiple data sets of the same size in one set of buffers. In order to determine the average setup time per problem space size and the average process time per problem space size, multiple data sets will be processed with the same set of buffers. Since most of the Middlebury image sets have different disparity depths and sizes<sup>1</sup>, we process the same image set multiple times with one set of initialized buffers to calculate average runtimes. How these measurements can be used to determine the setup time and the matching time will be shown in Section 6.1.

In Section 6.1.1 we will show how an upper bound for image dimensions can be estimated in order to achieve a specific matching frame rate on a specific device by use of the Floating point Operations per Pixel Comparison (FLOPC) value.

---

<sup>1</sup>Exceptions are the image sets that are of the same scene but with different lighting or different exposure: (Classroom2,Classroom2E), (Djembe,DjembeL), (Piano,PianoL) and (Motorcycle,MotorcycleE).

Later on, in Section 6.3 we will show the matching results of our implementation. All matching results that are presented in this chapter were generated with the same set of parameters. These parameters are the census filter window dimension ( $S_{CWx}, S_{CWy}$ ), the gap-size-values for the census filter ( $S_{CGx}, S_{CGy}$ ) which define the number of skipped pixels between two transformed pixels, the position of the central pixel relative to the filter window position ( $P_{CCx}, P_{CCy}$ ) and the upper and lower ranges ( $R_l, R_u$ ) to define whether a pixel's value is within a region around the central pixel's value. Further, the aggregation-size-values ( $S_{AWx}, S_{AWy}$ ), which define the size of the aggregation window, and finally the threshold for left-right consistency ( $c$ ) are needed. The census window's size was chosen to be  $S_{CWx} = 16, S_{CWy} = 16$  with a gap size of  $S_{CGx} = 1, S_{CGy} = 1$ . The central pixel was positioned at  $P_{CCx} = 7, P_{CCy} = 7$  and the ranges were set to  $R_l = 255, R_u = 0$  to achieve a census transform as discussed in Section 5.3. The aggregation-size-values were set to  $S_{AWx} = 5, S_{AWy} = 5$ .

The algorithms used for comparison in Section 6.1.2 and Section 6.3 are listed in Table 6.1. Later on in this chapter, we will use the labels from this table to reference these algorithms.

Finally, we will discuss additional results of our evaluation in Section 6.4. In this section, we will discuss runtime behavior of our implementation that is dependent of the problem dimension and the used device's maximum-work-group-size-property.

## 6.1 Timing results

Our implementation offers the possibility of creating a matching structure which holds the buffers needed for a specific problem space (image width, image height, maximal disparity). This allows the streaming of multiple images from one set of cameras, without the need to repeatedly set up buffers.

The measured runtime for the matching of one image set, including buffer setup time, can be divided into two parts, setup time  $t_s$  and matching time  $t_m$ . In a setup where a multitude of  $k$  image sets are matched, using one matching structure, the total time  $T$  is:

$$T = t_s + k \cdot t_m \quad (6.1)$$

Using different values of  $k$  ( $k_1$  and  $k_2$ ) results in different total times  $T_{k_1}$  and  $T_{k_2}$  which let us determine the values of  $t_s$  and  $t_m$  with the following formula:

$$\begin{aligned} t_m &= \frac{T_{k_1} - T_{k_2}}{k_1 - k_2} \\ t_s &= T_{k_1} - k_1 \cdot t_m \end{aligned} \quad (6.2)$$

These can be run multiple times to create the average total times  $\overline{T_{k_1}}$  and  $\overline{T_{k_2}}$ . We can see in the following formula (Formula 6.3) that the average of sums is the same as the sum of the averages of the summands.

Algorithm	Resolution	Description
DF	Q	An anonymous submission to the Conference on Computer and Robot Vision (CRV) 2018 using disparity filtering with 3D CNN.
IDR	H	Real-time stereo Matching on CUDA using an iterative refinement method for adaptive support-weight correspondences by [KPP13].
MCSC	F	An anonymous submission to the Middlebury benchmark. The author claims it to be a method with simultaneous learning of matching cost and smoothness constraint.
MPSV	Q	An anonymous submission to the European Conference on Computer Vision (ECCV) 2016 using morphological processing.
R-NCC	F	A window based matching method by S. Fang and Y. Li.
SGM	Q,F	Stereo processing by semi-global matching and mutual information by [Hir08].
SGM_ROB <sup>2</sup>	H	Stereo processing by semi-global matching and mutual information again by [Hir08]. A submission to the Robust Vision Challenge 2018 (ROB) 2018 in conjunction with Conference on Computer Vision and Pattern Recognition (CVPR) 2018
SNCC	H	Block-matching stereo with Summed Normalized Cross-Correlation (SNCC) measure by [EE10].

Table 6.1: Algorithm Overview.

$$\begin{aligned}
\overline{T_k} &= \frac{T_{k,1} + T_{k,2} + \cdots + T_{k,n}}{n} \\
&= \frac{(t_{s1} + k \cdot t_{m1}) + (t_{s2} + k \cdot t_{m2}) + \cdots + (t_{sn} + k \cdot t_{mn})}{n} \\
&= \frac{t_{s1} + t_{s2} + \cdots + t_{sn}}{n} + k \cdot \frac{t_{m1} + t_{m2} + \cdots + t_{mn}}{n} \\
&= \overline{t_s} + k \cdot \overline{t_m}
\end{aligned} \tag{6.3}$$

<sup>2</sup>This submission, RGB\_ROB - H was chosen in favor of the older submission of this author RGB - H.

With this, we can determine the average matching time  $\overline{t_m}$  and the average setup time  $\overline{t_s}$  per image set for one device. By running this experiment multiple times, we create two values  $\overline{T_1}$  and  $\overline{T_{10}}$  (corresponding to  $k = 1$  and  $k = 10$  for Formula 6.3) for each resolution which can be used to calculate  $\overline{t_m}$  and  $\overline{t_s}$  using Formula 6.2.

We chose 100 as the number of iterations for the calculation of average runtimes. This number, which seems relatively low for statistical evaluation was chosen because of the limitations of our hardware. Using the *GTX 750 Ti* GPU, the total runtime for matching the full Middlebury data set (training and test data with full-, half- and quarter-resolution) 100 times for  $T_1$  and  $T_{10}$  takes more than 15 hours. These 100 runs of the full data set for  $T_1$  and  $T_{10}$  surmounted to 25 days of total runtime on our *Mali T760*.

In order to create a runtime prediction for other OpenCL devices, we multiplied the resulting matching times per megapixel with their corresponding device's floating point performance value in Floating point Operations Per Second (FLOPS) to create a Normalized Calculation Time (NCT) as described by [JS92]. This gives us the number of floating point operations per pixel comparison, that is necessary for the matching task.

The multiplication of the matching time with the floating point performance values given by the different vendors is not the exact value for the normalized calculation time, because it is generated from averaged time measurements and the specification values given by the hardware vendors. The specified floating point performance values are theoretical values because they mark the number of FLOPS that these devices could perform when working at peak efficiency, which is usually not the case.

To be able to compare algorithms, we define the number of Pixel Comparisons (PCs) for an image set:

$$PC = w \cdot h \cdot d_{\max} \quad (6.4)$$

and use the NCT to define FLOPC as an algorithm specific number that is independent of the used hardware:

$$\text{FLOPC} = \frac{t}{PC} \cdot \frac{\text{FLO}}{t} = \frac{\text{FLO}}{PC} \quad (6.5)$$

The value FLOPC is equal to the measured time (t) per PC multiplied by the number of floating point operations (FLO) per second.

The plot in Figure 6.1 shows how the different problems run on the tested devices. We assigned different colors to the devices used to create the data for this plot and used these colors for data points created by these specific devices, e.g. blue data points were created by an *nVidia GTX750Ti*, red data points were the results of a *Mali T760* and green data points were created by an *nVidia Quadro 1000M*. The symbols used in the plot give information about which data set the data points correspond to. Data sets

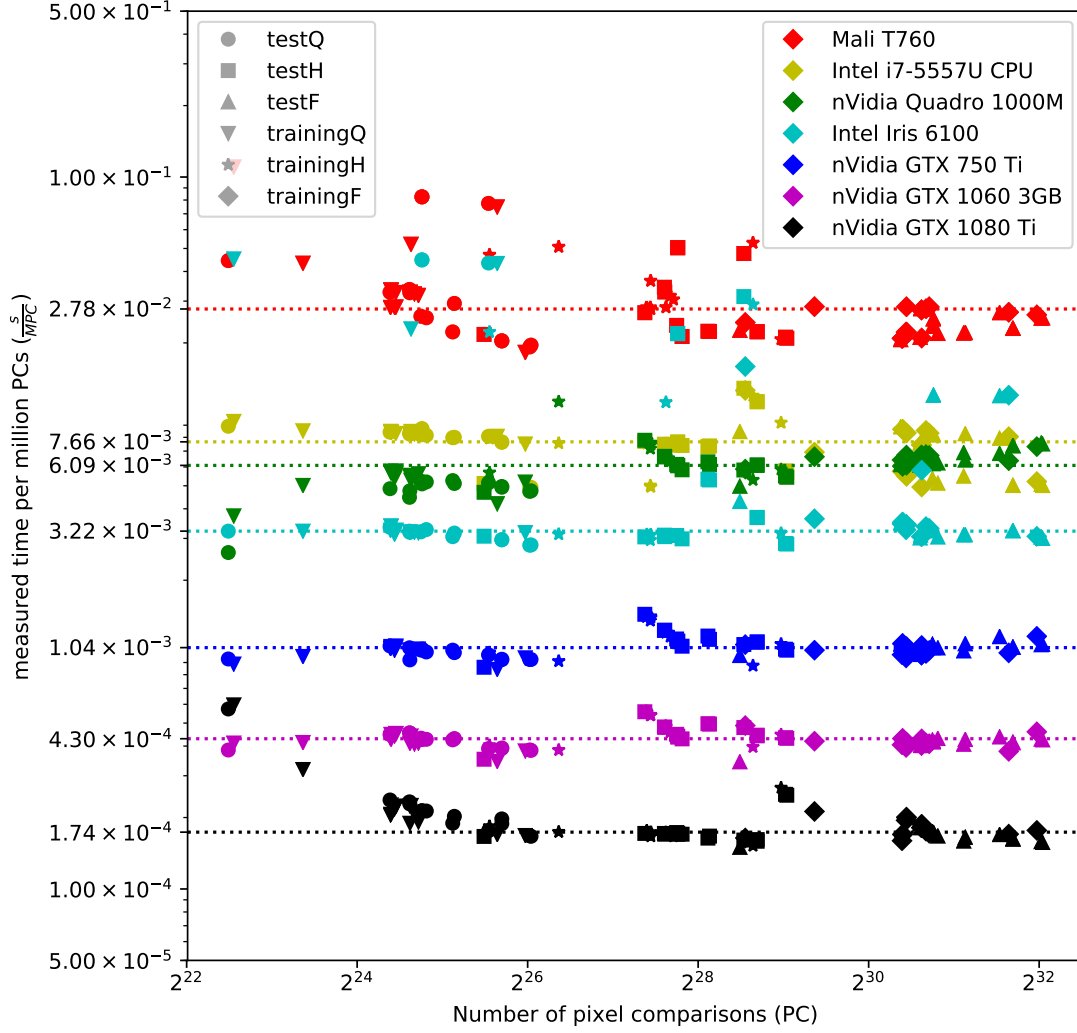


Figure 6.1: Plot of runtime measurements for *Mali T760* (red), *Intel i7-5557U* CPU (yellow), *nVidia QUADRO 1000M* (green), *nVidia Iris 6100* (light blue), *nVidia GTX 750 Ti* (blue), *nVidia GTX 1060 3GB* (magenta) and the *nVidia GTX 1080 Ti* (black) with the median of the data sets marked by dotted lines.

in this context are the Middlebury training and test data sets in full (F), half (H) and quarter (Q) resolution as described in Section 2.2.4.5, resulting in six data sets.

In Figure 6.1, the x-axis of the plot shows the size of the problem space i.e. the number of potential pixel comparisons for the different image sets (PC). Due to the cubic growth of the data sets we chose a logarithmic scale for this axis. The y-axis of this plot shows the measured time per million pixel comparisons( $\frac{s}{MPC}$ ).

From this plot, we conclude that the main parameter for the calculation time per pixel comparisons is device dependent. The parameter with the most influence in our experiments is the number of FLOPS of the used device. Floating point performance values of different devices are listed in Table 6.2. A near linear correlation between the device’s floating point performance and the measured time per pixel comparison can be observed in Figure 6.1.

The resulting values give information about the order of magnitude of the number of FLOPC. In order to get a baseline data set, we calculate the average number of FLOPC for every data point along the problem-space-size axis from the corresponding results of three devices. The device data sets used for this baseline were chosen because of their low variance. The plot in Figure 6.3 shows these values of the *Average* data set. For our evaluation, we also calculated the average FLOPC from this data set. The *Average* FLOPC data set for our implementation is also shown in Figure 6.3 (red). This *Average* data set has a mean value which is marked by a dotted line and has roughly the value:

$$\text{FLOPC}_{\text{avg}} \approx 1.53 \cdot 10^3 \tag{6.6}$$

Using the *Average* data set in Figure 6.3 we try to predict the runtime for all devices available to us, using the floating point performance value from their vendor’s device specification page. In Table 6.2, the FLOPS for these devices and for the devices used by other authors mentioned in this chapter are listed.

Figure 6.1 shows the measured runtime in seconds per mega PC for seven different devices. Figure 6.2 shows our runtime prediction based on the average FLOPC and the devices’ floating point performance values. We can see that the predictions of the runtime per mega PC for the different devices roughly match the order of magnitude of the runtime measurements for these devices.

The results for the *Intel Iris 6100* graphics card were not completely consistent with our prediction. In Figure 6.1, we can see that the median of the results of the *Intel Iris* graphics card, highlighted in light blue, matches the predicted order of magnitude. However, for some image sets, this device consistently had performance issues. In Section 6.4, we will examine this runtime behavior.

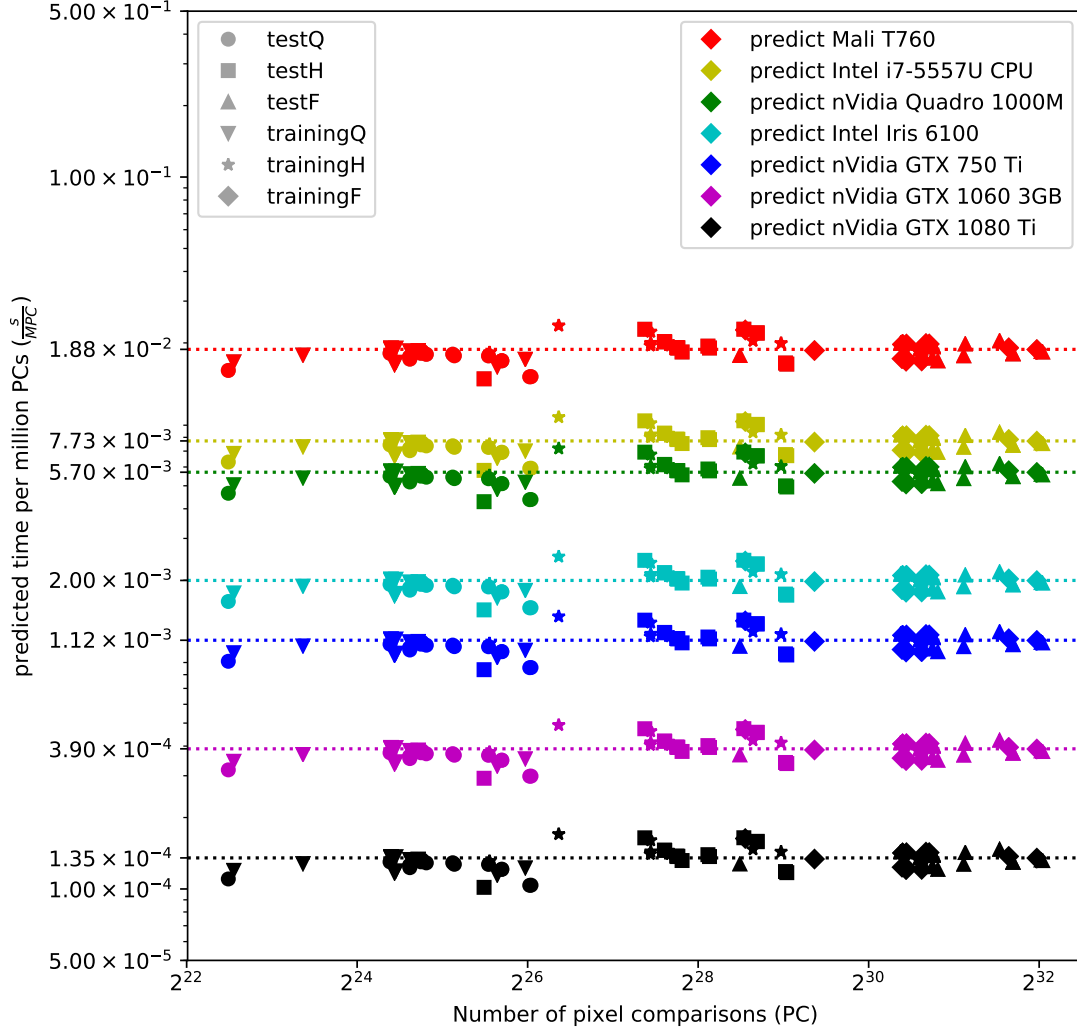


Figure 6.2: Plot of runtime predictions for *Mali T760*, *Intel i7-5557U CPU*, *nVidia QUADRO 1000M*, *nVidia Iris 6100*, *nVidia GTX 750 Ti*, *nVidia GTX 1060 3GB* and the *nVidia GTX 1080 Ti* with the median values for the data sets marked by dotted lines. The predictions are based on the *Average FLOPC* data set in Figure 6.3.

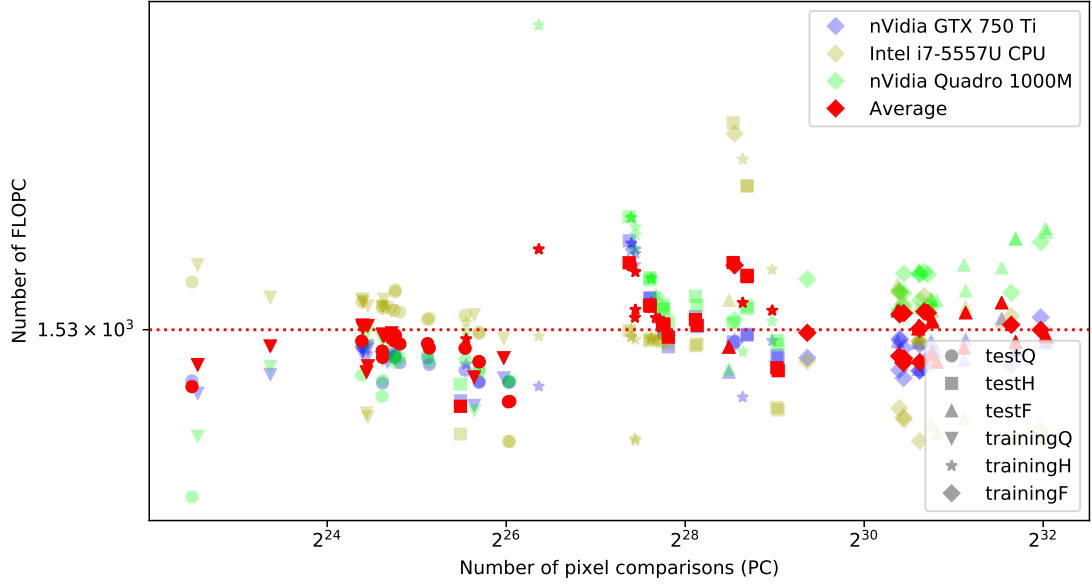


Figure 6.3: NCT normalization (FLOPC) of the *Intel i7-5557U* (yellow), the *nVidia GTX 750 Ti* (blue) the *nVidia Quadro 1000M* (green) and their averaged data set (red). The dotted line marks the mean FLOPC value of the averaged data set.

### 6.1.1 Problem Estimation

In a usual setup, the image ratio of images stays constant across multiple frames. This ratio is given in the form:  $M : N$  e.g.  $16 : 9$  or  $4 : 3$ . Furthermore, the relation  $d_{\max} \approx \frac{w}{10}$  was established in Section 2.1.3.5.

Using these statements we rewrite the problem space parameters  $w$ ,  $h$  and  $d_{\max}$  to be expressed in one variable  $x$  such that  $x$  represents a specific number of pixels:

$$\begin{aligned} w &= M \cdot x \\ h &= N \cdot x \\ d_{\max} &= O \cdot x \approx \frac{M \cdot x}{10} \end{aligned} \tag{6.7}$$

For example, if the image and  $d_{\max}$  ratio  $M : N : O$  is  $4 : 3 : \frac{4}{10}$  and  $x = 150$  pixels, then  $w = 600$  pixels,  $h = 450$  pixels and  $d_{\max} = 60$  pixels. This leads to the size of the problem space to be a function of  $x$  with the constant parameters  $M$ ,  $N$  and  $O$  ( $f_{M,N,O}(x)$ ):

$$w \cdot h \cdot d_{\max} = f_{M,N,O}(x) = M \cdot N \cdot O \cdot x^3 \tag{6.8}$$



Vendor	Device Name	Device Type	Giga FLOPS (GFLOPS)
nVidia	Quadro 1000M <sup>3</sup>	GPU	268.8
nVidia	GTX 750 Ti <sup>4 5</sup>	GPU	1372
nVidia	GTX 980 <sup>6</sup>	GPU	4981
nVidia	TITAN Black <sup>7</sup>	GPU	5645
nVidia	GTX 1060 3GB <sup>8,9</sup>	GPU	3935
nVidia	GTX 1080 <sup>10</sup>	GPU	8873
nVidia	GTX 1080 Ti <sup>11</sup>	GPU	11340
Intel	Iris 6100 <sup>12</sup>	GPU	768
Intel	i7-5557U <sup>13 14</sup>	CPU	198.6
Mali	T760 <sup>15,16</sup>	ARM GPU	81.6

Table 6.2: Overview of different devices mentioned in this chapter.

The size of the problem space is the number of pixel comparisons for one specific problem. In order to achieve real time performance (60 FPS) or any other given frame rate for a certain device, the number of pixel comparisons multiplied by the number of FLOPC has to be smaller than (or equal to) the number of FLOPS this device is able to perform divided by the target frame rate FPS:

$$f_{M,N,O}(x) \cdot \text{FLOPC} \leq \frac{\text{FLOPS}}{\text{FPS}} \quad (6.9)$$

This can be solved for  $x$ :

$$x \leq \sqrt[3]{\frac{\text{FLOPS}}{\text{FLOPC} \cdot \text{FPS} \cdot M \cdot N \cdot O}} \quad (6.10)$$

<sup>3</sup><https://www.techpowerup.com/gpudb/1431/quadro-1000m>

<sup>4</sup><http://www.nvidia.com/gtx-700-graphics-cards/gtx-750ti/>

<sup>5</sup><https://www.techpowerup.com/gpudb/2548/geforce-gtx-750-ti>

<sup>6</sup><https://www.techpowerup.com/gpudb/2621/geforce-gtx-980>

<sup>7</sup><https://www.techpowerup.com/gpudb/2549/geforce-gtx-titan-black>

<sup>8</sup><https://www.nvidia.com/en-us/geforce/products/10series/compare/>

<sup>9</sup><https://www.techpowerup.com/gpudb/2867/geforce-gtx-1060-3-gb>

<sup>10</sup><https://www.techpowerup.com/gpudb/2839/geforce-gtx-1080>

<sup>11</sup><https://www.techpowerup.com/gpudb/2877/geforce-gtx-1080-ti>

<sup>12</sup><https://www.techpowerup.com/gpudb/2627/iris-graphics-6100>

<sup>13</sup><https://en.wikipedia.org/wiki/FLOPS>

<sup>14</sup> $\text{FLOPS} = \text{sockets} \cdot \frac{\text{cores}}{\text{socket}} \cdot \frac{\text{cycles}}{\text{second}} \cdot \frac{\text{FLOPs}}{\text{cycle}} = 1 \cdot 2 \cdot 3.1\text{GHz} \cdot 32$

<sup>15</sup>[https://en.wikipedia.org/wiki/Mali\\_\(GPU\)](https://en.wikipedia.org/wiki/Mali_(GPU))

<sup>16</sup><http://gpu-flops.blogspot.co.at/2015/02/gpu-flops-list.html?m=1>

The combination of Formula 6.7 and Formula 6.9 allows for estimation of the maximum size of a problem space that can be processed on a specific device with a specific frame rate. An example for such a calculation can be found in Appendix 2.

### 6.1.2 Algorithm Comparison

In this section, we will use the formulas shown in this chapter to compare our implementation against algorithms published on the website of the Middlebury Benchmark. For this purpose, we chose the algorithm IDR by [KPP13] which is a census based GPU implementation, the algorithm SGM\_ROB, a census based GPU implementation of Semi Global Matching (SGM) to the ROB and the algorithm MCSC, an anonymous submission to the Middlebury benchmark which involves simultaneous learning of matching cost and a smoothness constraint. These algorithms were chosen, because of the known floating point performance of the used devices. The devices used by the authors of these algorithms are listed in Table 6.3 and the FLOPS rates of the corresponding devices (according to the information given on the Middlebury Benchmark’s website) are listed in Table 6.2.

Column 3 (Average  $\frac{s}{MPC}$ ) of Table 6.3, lists the average of the submitted runtime per image set normalized to those image set’s number of million PCs. These runtime values are reported to the Middlebury Benchmark’s website by the authors of the algorithms, the number of million PCs results from the image dimensions and the maximum disparity of these image sets and average of the normalized values is calculated by us. Column 4 (FLOPC) of Table 6.3, lists the FLOPC values of the listed algorithms, calculated with Formula 6.5. These values were calculated from the Average  $\frac{s}{MPC}$  in Column 3 and the FLOPS values from Table 6.2 for the corresponding devices in Column 2.

The buffer setup time  $t_s$  is roughly constant for different problem space sizes, this can be seen in Figure 6.4. The time  $t_s$  is included in the measured runtimes for different problem space sizes and can be determined by Formula 6.1 and Formula 6.2. In Figure 6.1 and Figure 6.6, the measured runtime is normalized to the problem space size. A normalization of  $t_s$  to increasing problem space sizes results in a general decrease in the values of the plot of the data set.

This decrease of time per PC for the total time ( $t_m + t_s$ ) is shown in Figure 6.5(a). Further, the constant matching time per PC is visualized in this plot. In Figure 6.5(b) the constant buffer setup time is visualized and the increase in matching time corresponding to the increase in problem size.

The plot in Figure 6.6 shows a runtime-comparison of the three algorithms mentioned above (IDR, SGM\_ROB and MCSC), with our implementation. The plot shows the number of seconds per mega pixel-comparison on the y-axis and the size of the problem space in MP on the x-axis. The data shown in Figure 6.6, is taken from the corresponding submissions to the website of the Middlebury benchmark. This plot shows that the algorithm-device combinations are relatively close to each other concerning performance, as indicated by the closeness of the shown average values (represented by dotted lines).

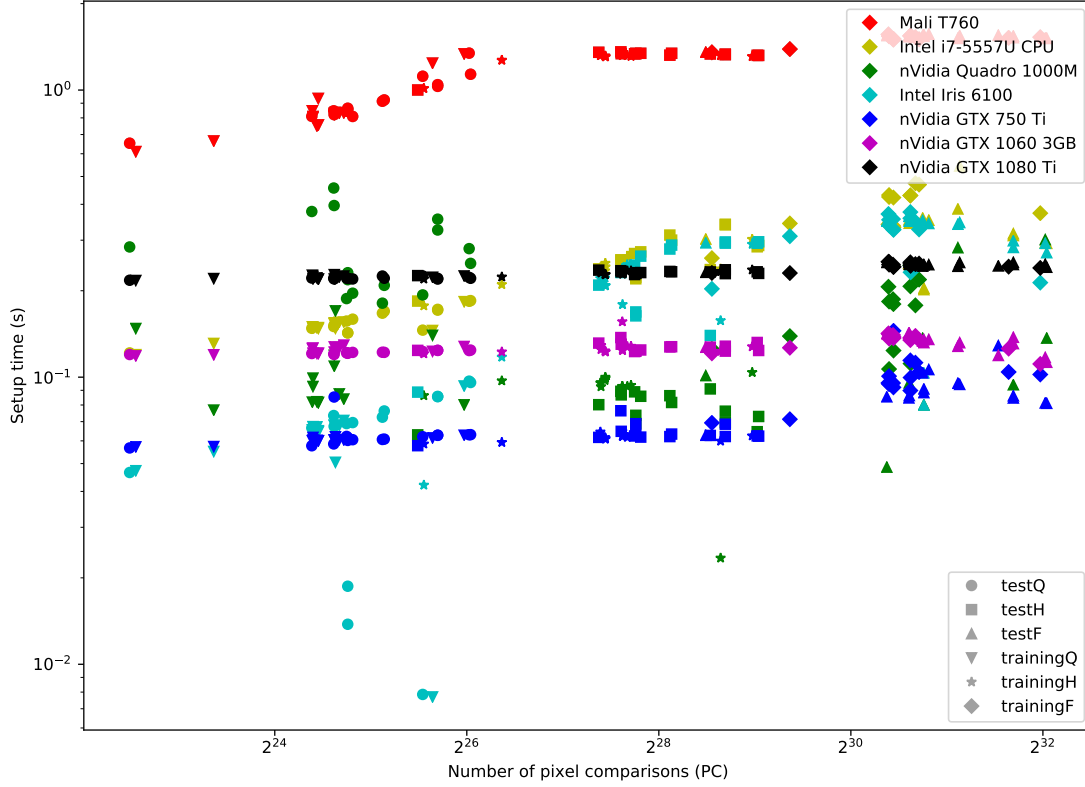


Figure 6.4: Comparison of buffer setup time (s) on the y-axis corresponding to the problem size (MP) on the x-axis. Every data point represents the match of an image set. Colors represent the used devices and symbols (circles, triangles etc.) represent the data set of the used image sets.

All of the compared measurements are situated at a magnitude of roughly  $1.1 \cdot 10^{-3}$  to  $1.7 \cdot 10^{-3}$  seconds per mega pixel-comparison. The plots of the MCSC data set and the SGM\_ROB data slightly seem to decrease in value and the plot of the IDR data set seems to have roughly constant matching time per Mega Pixel Comparison (MPC) (Figure 6.6). We suspect that this is due to different handling of setup times by the various authors. Therefore, we added the performance values of our implementation, once including the setup time and once without the setup time.

It is important to note, that this is purely runtime performance and does not consider any error metric concerning the correctness of the results, a quality evaluation of these algorithms is presented later on in Section 6.3.

The average runtime performance ranking ( $\frac{s}{MPC}$ ) for these algorithm-device combinations can be seen in Table 6.3. These performance values do not take into account the different floating point performance values of the used devices, as listed in Table 6.2. This

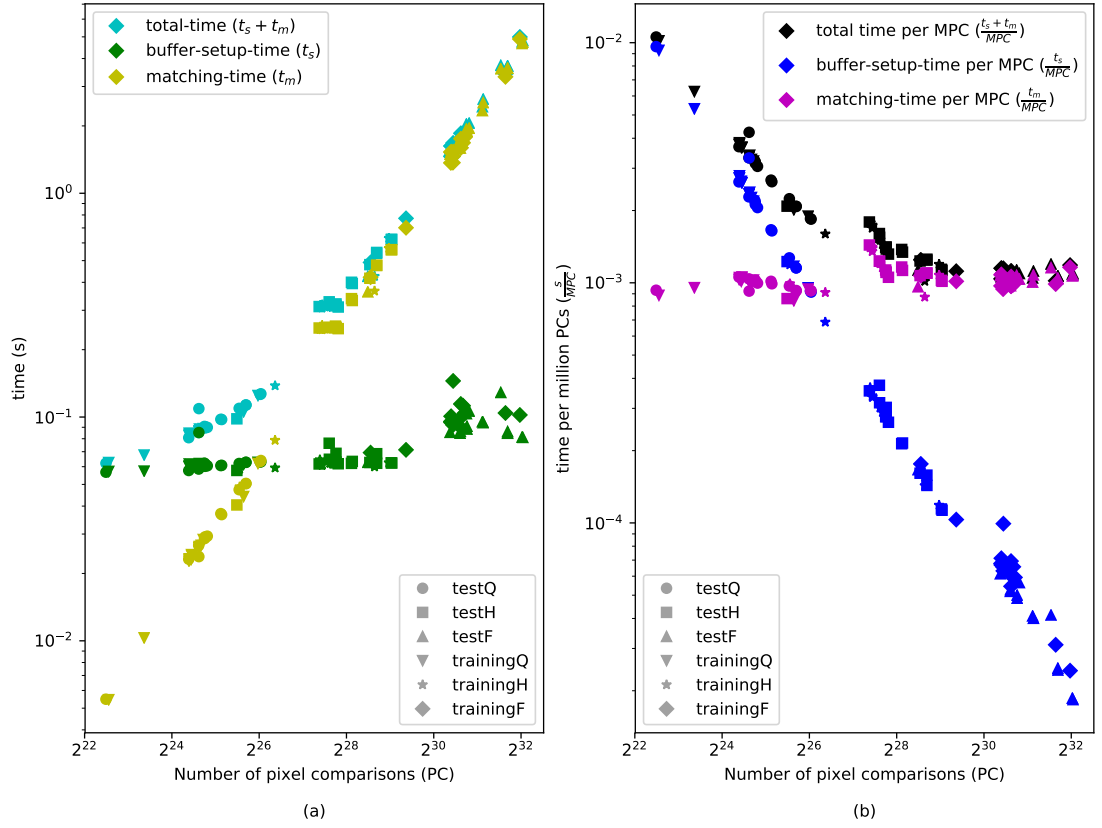


Figure 6.5: Absolute matching time, absolute setup time and absolute total time (a) and matching time, setup time and total time per million PCs (b) for a GTX 750 Ti graphics card.

Algorithm	Device	Average $\frac{s}{MPC}$	FLOPC
<b>OUR IMPL.</b>	GTX 750 Ti	$1.12 \cdot 10^{-3}$	$1.53 \cdot 10^3$
<b>OUR IMPL.</b> (incl. setup time)	GTX 750 Ti	$1.34 \cdot 10^{-3}$	$1.84 \cdot 10^3$
MCSC	GTX 1080	$1.43 \cdot 10^{-3}$	$1.27 \cdot 10^4$
SGM_ROB	GTX 980	$1.56 \cdot 10^{-3}$	$7.76 \cdot 10^3$
IDR	TITAN Black	$1.64 \cdot 10^{-3}$	$9.24 \cdot 10^3$

Table 6.3: Algorithm - Runtime Overview.

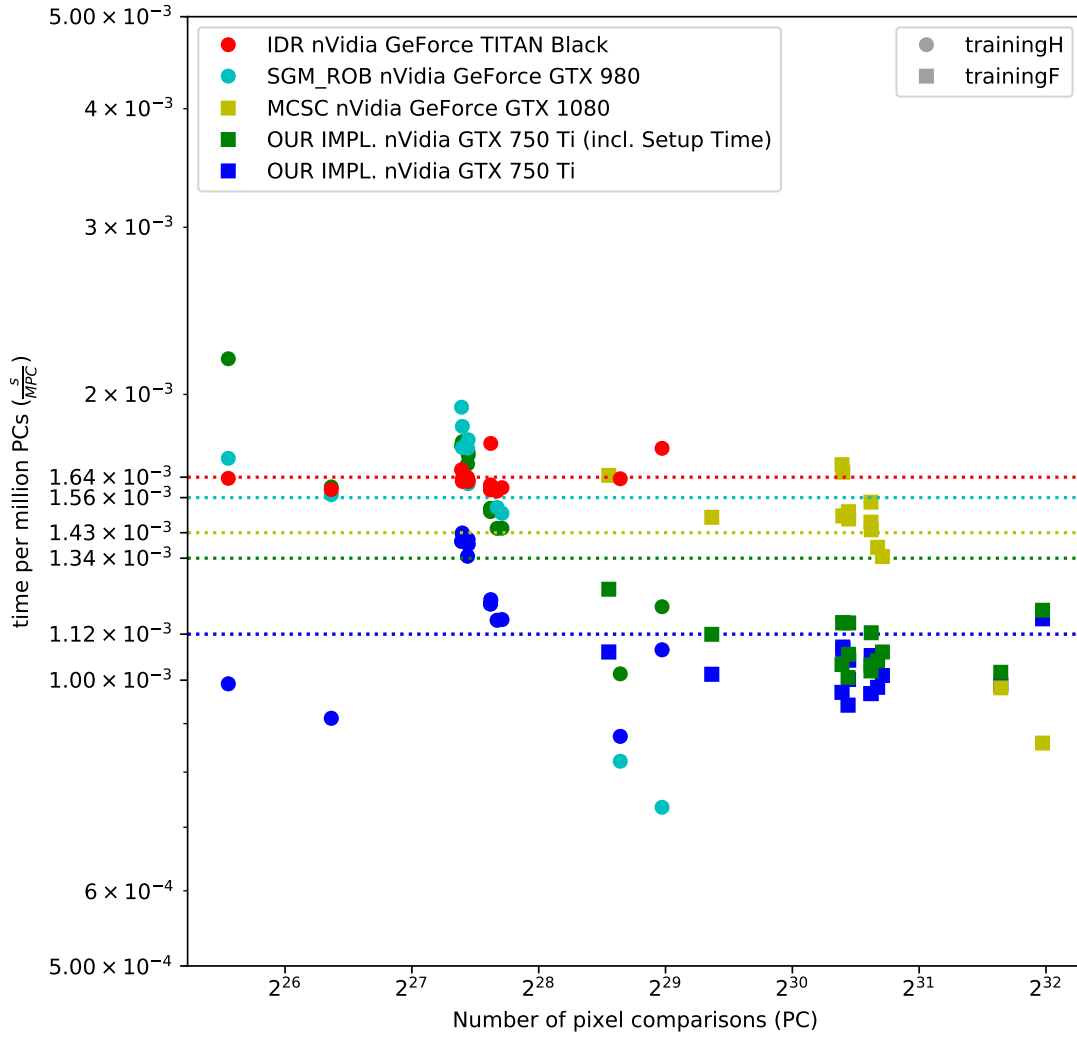


Figure 6.6: Comparison of different algorithms. Seconds per MP comparison (y-axis) to problem size in MP (x-axis): IDR (red), SGM\_ROB (light blue), MCSC (yellow), our implementation (green), our implementation without buffer setup time (blue). Averages are marked by lines.

allows algorithms to have better runtime results than algorithms with similar calculatory complexity by use of faster hardware. Further, we stated in Section 6.1 that FLOPC is an algorithm specific number which makes algorithms comparable. Table 6.3 shows this algorithm specific number in column 4, which also is visualized in Figure 6.7.

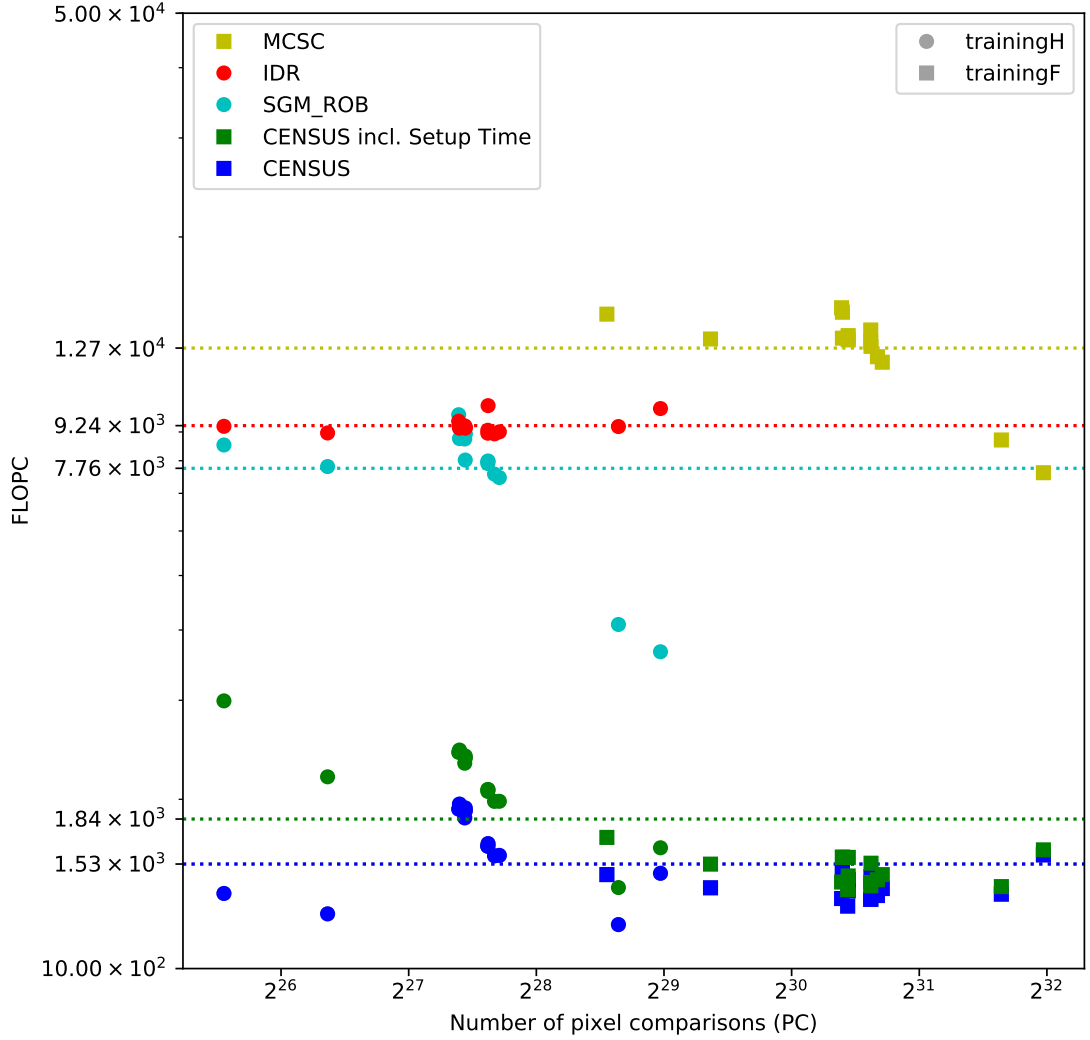


Figure 6.7: Comparison of different algorithms. Number of Floating point Operations per Pixel Comparison (y-axis) to problem size in MP (x-axis): IDR (red), SGM\_ROB (light blue), MCSC (yellow), our implementation (green), our implementation without buffer setup time (blue). Averages are marked by dotted lines.

Comparison of the average FLOPC values (represented by dotted lines) of the algorithms shown in Figure 6.7 and Table 6.3, indicates that the better runtime performance (smaller value for  $\frac{s}{MPC}$ ) of MCSC that lies below IDR's and SGM\_ROB's runtime performance

in Figure 6.6 and Table 6.3 is due to the use of a device with higher floating point performance.

Further, the FLOPC value also allows the use of Formula 6.10 to estimate an upper bound of image dimensions for which a specific stereo matching frame rate is possible for a specific device.

## 6.2 Implementation-Result Comparison

In this section, we will compare our implementation’s result quality with the quality of the results of [HZW<sup>+</sup>10], uploaded to the Middlebury Benchmark’s website in 2010. In 2010, the Middlebury Benchmark was available in *version 2* and the result table is still accessible at <http://vision.middlebury.edu/stereo/eval/>. Due to differences, such as larger data sets, different  $\delta$  values for the bad-pixel-percentage metric and the availability of sparse comparisons, the results in *version 3*’s result table are hardly comparable to the results listed in *version 2*’s result table. The evaluation tools for *version 2* of the Middlebury Benchmark are not available anymore. Further, we did not have access to the implementation of [HZW<sup>+</sup>10], which prevented a re-evaluation of their implementation with Middlebury’s *version 3* benchmark. Therefore, a custom evaluation was implemented to allow comparison of the two implementations.

The results of [HZW<sup>+</sup>10], shown in Figure 6.8, were acquired from the Middlebury Benchmark’s result table *version 2*. The image sets (Cones, Teddy, Tsukuba and Venus) and ground truth disparity maps for these data sets are available on the Middlebury Benchmark’s website.

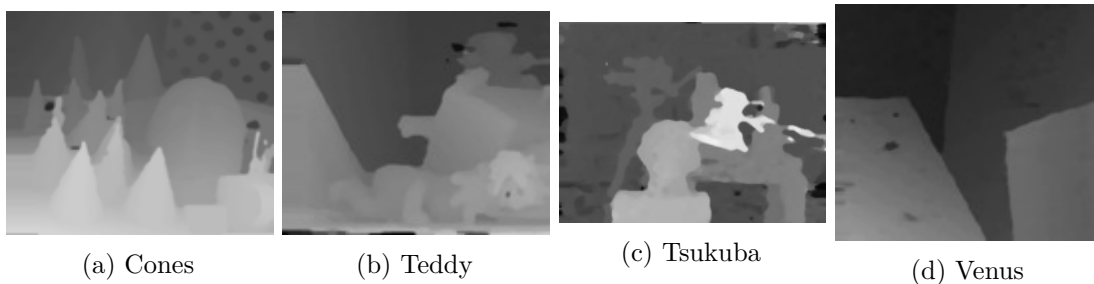


Figure 6.8: Results of [HZW<sup>+</sup>10], from *version 2* of the Middlebury Benchmark.

Further, Figure 6.9 shows a colorized visualization of the results in Figure 6.8. For these colorized results, we created a script which read the input-file (e.g. a Portable Network Graphics (PNG)-file or a Portable Gray Map (PGM)-file). A scaling factor was applied to the values to compensate for the value scaling as described on the Middlebury Benchmark’s website<sup>2,3</sup>. The scaling-factors were 4, for the image sets *Cones* and *Teddy*,

<sup>2</sup><http://vision.middlebury.edu/stereo/data/scenes2001/>

<sup>3</sup><http://vision.middlebury.edu/stereo/data/scenes2003/>

and 8 for the image sets *Tsukuba* and *Venus*. The scaled disparity values were then written to an Portable Float Map (PFM)-file. Finally, the tool *runviz* from the Middlebury Benchmark *version 3* was used to generate the colored version of the results. The *runviz* tool needed a calibration file (*calib.txt*) for each image set. The *calib.txt* files provided the *runviz* tool with the used disparity range for visualization. We chose the range  $[0; 70]$  for all four image sets. This conversion script is provided in Appendix 4.

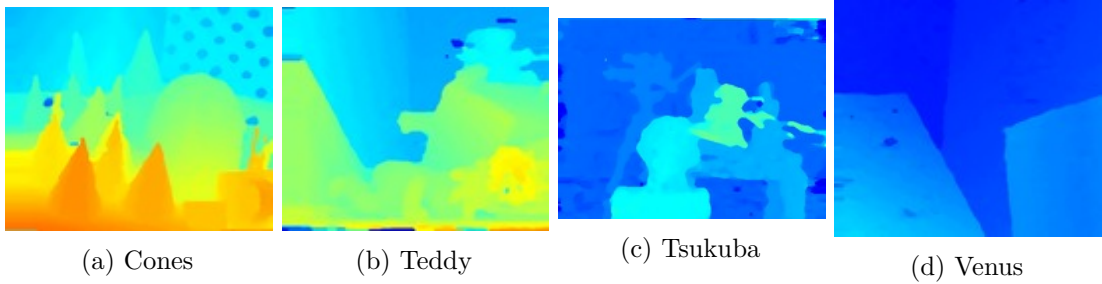


Figure 6.9: Results of  $[HZW^+10]$  from *version 2* of the Middlebury Benchmark. Converted to colored map using the tool *runviz* from *version 3* of the Middlebury Benchmark.

Figure 6.10 shows the ground truth disparity maps of the image sets. To convert the ground truth disparity maps provided as PGM-files, into these colored versions, the conversion script described in Appendix 4 was used.

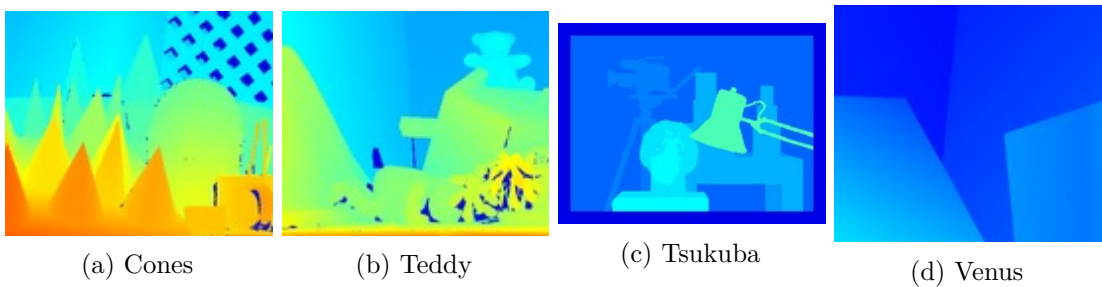


Figure 6.10: Ground truth from *version 2* of the Middlebury Benchmark. Converted to colored map using the tool *runviz* from *version 3* of the Middlebury Benchmark.

Finally, we used our implementation to create disparity maps for these image sets, Figure 6.11 shows these results. The visualization tool from *version 3* of the Middlebury Benchmark was used to generate the colored visualization of these disparity maps.

In order to allow comparison of our implementation and  $[HZW^+10]$ 's implementation we used the *Percentage-of-Bad-Pixels* metric, shown in Section 2.2.2. We created a script that calculated the value of results of the two implementations, for the  $\delta$  values 0.5, 0.75, 1.0, 1.5, 2.0 and 4.0, in this metric. This list represents the conjoined  $\delta$  values, available in the Middlebury Benchmark *version 2* and *version 3*. Figure 6.12 shows



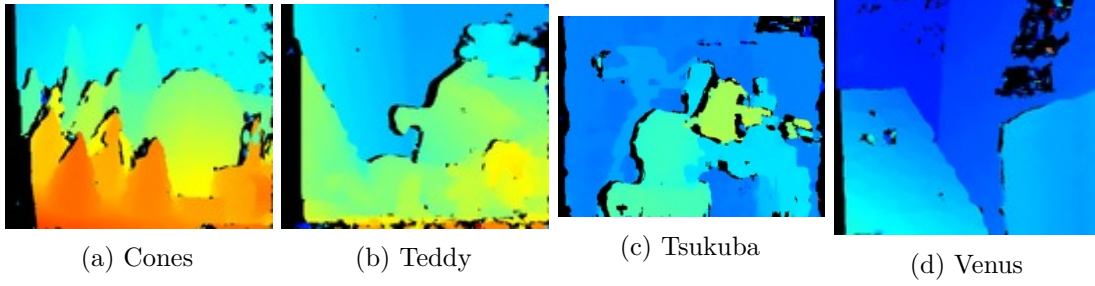


Figure 6.11: Results of our implementation for the image sets of the Middlebury Benchmark *version 2*.

the bad-pixel-percentages for the four image sets calculated by our script. This figure shows that [HZW<sup>+</sup>10]’s implementation (blue) in general produced better results than our implementation.

Moreover, we can see in all four subplots of Figure 6.12 that for [HZW<sup>+</sup>10]’s curves, that the bad-pixel-percentages of some neighboring  $\delta$ -points are the same. We suspect that this is due to the result maps for [HZW<sup>+</sup>10]’s implementation that were available to us in form of PNG-files, which contained integer disparity values. This effect is further amplified in the tsukuba image set, where the ground truth disparity values are provided as integers as well.

The results of our implementation are plotted in red and green. The red curves represent the results of our implementation when all pixels are taken into account. The green curves represent the same results as the red curves, however, for these curves invalid disparities as declared by our consistency check implementation, are not used for bad-pixel-percentage-calculation. Further, the percentage of invalid disparities per image set is displayed in the labels of the green curves.

Differences between the implementations that can be seen in Figure 6.8 and Figure 6.11 are partly missing, due to the handling of occlusions and homogenous surfaces in the scenes. The authors of [HZW<sup>+</sup>10] used bilinear interpolation to estimate missing disparity values. Further, they applied a median filter to smooth the result maps and fill small holes.

We used the parameters described in Chapter 5 for the generation of these results. There may be some quality gains if other values are used, however an in-depth study of result quality was not the goal of this thesis.

We conclude that our implementation is not equivalent to [HZW<sup>+</sup>10]’s implementation. Although they face similar difficulties at the same regions (i.e. occlusions and homogenous surfaces), our implementation misses the bilinear interpolation and the median filter to handle these regions.

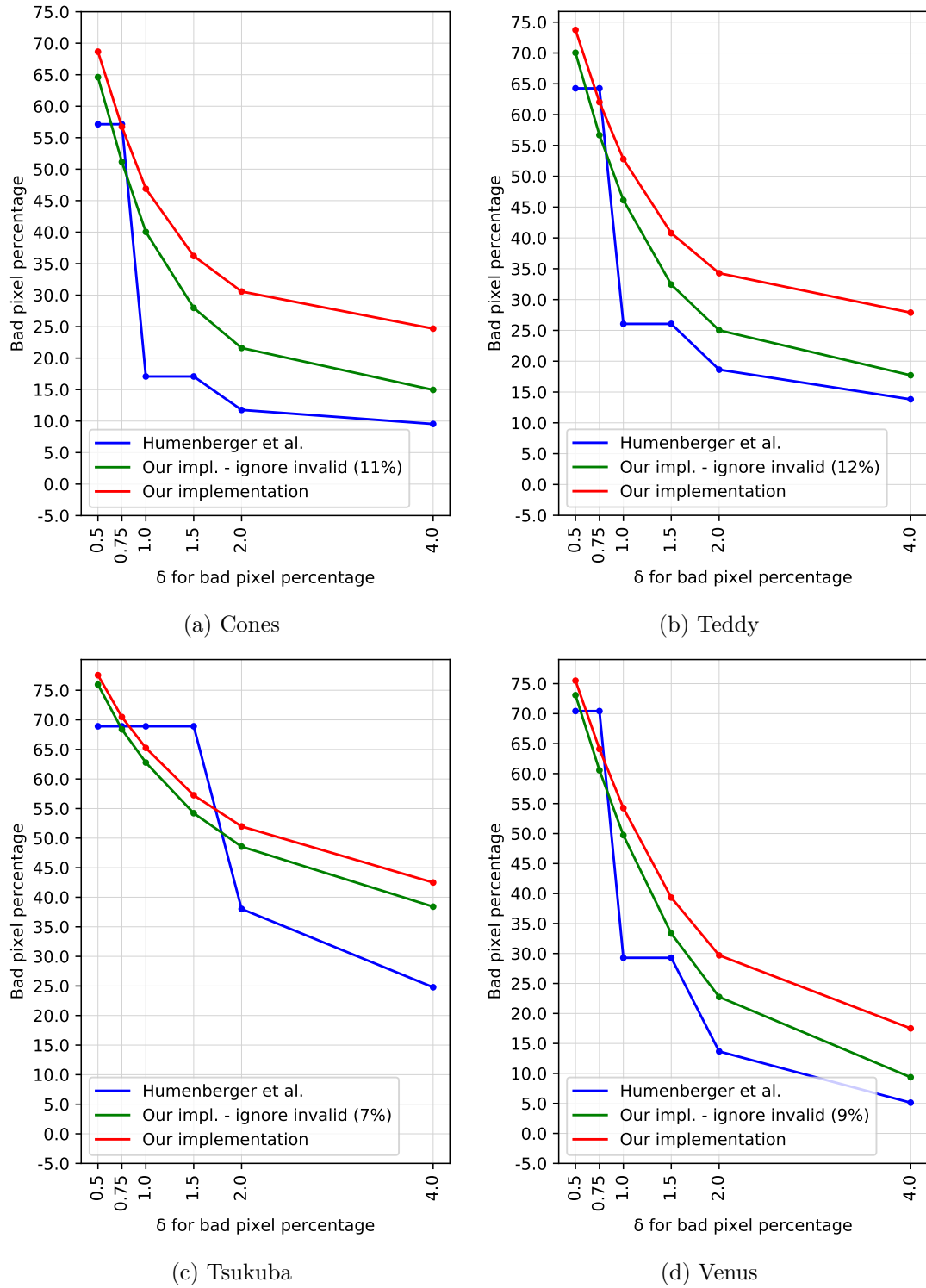


Figure 6.12: Visualization of bad pixel percentage of the four image sets. Dense results from [HZW<sup>+</sup>10]’s implementation and sparse results from our implementation.

## 6.3 Result Errors

In this section, we will show how our implementation performs compared to other algorithms presented on the Middlebury benchmark, in terms of matching accuracy. This is done to show that higher result quality is costly in terms of calculatory complexity. We selected IDR, SGM\_ROB and MCSC because of a certain similarity of these methods to the method of [HZW<sup>+</sup>10]. The results of algorithms, submitted to the Middlebury website are only available in one resolution. The results for IDR and SGM\_ROB are available in half resolution and MCSC's results are available in full resolution.

Furthermore, the density of the results is an important factor when evaluating the algorithms. The Middlebury benchmark provides strategies to make dense and sparse algorithms comparable, as described in Section 2.2.1. Our implementation of the algorithm of [HZW<sup>+</sup>10] and the IDR algorithm are sparse methods. The algorithms MCSC and SGM\_ROB are dense algorithms. In order to compare sparse and dense algorithms, it is important to take the number of invalid pixels, produced by the sparse algorithms, into account. The scripts *runeval* and *runevalF* count a pixel as invalid if the disparity value in the result map is of value positive infinity.

Algorithms purposefully can mark pixels as invalid. Invalid pixels are not counted towards error rates in the Middlebury benchmark's table of sparse results. This can be used by algorithms to mark the disparity of pixels in the result map as "*unknown*" if checks, such as the consistency check described in Section 4.4, fail.

The plots in this section visualize data from the Middlebury evaluation scripts *runeval* and *runevalF*. These scripts evaluate the results produced by stereo matching algorithms in comparison to the ground truth data. The difference between these two scripts is, that *runeval* compares the results to the ground truth available in the corresponding resolution and *runevalF* uses a scaling factor to compare the full resolution ground truth of the low resolution result. The following code sample (Listing 13) shows how the Middlebury evaluation tool works. The original evaluation code can be found in the Middlebury SDK<sup>4</sup>.

```

1      //get width and height for ground truth
2      int width = gt.width, height = gt.height;
3      //get width and height for result map
4      int width2 = data.width, height2 = data.height;
5      int scale = width / width2;
6
7      /** evaluation variable initialisation ***/
8      for (int y = 0; y < height; y++) {
9          for (int x = 0; x < width; x++) {
10             float gt = gtdisp.Pixel(x, y, 0);

```

<sup>4</sup><http://vision.middlebury.edu/stereo/submit3/zip/MiddEval3-SDK-1.6.zip>  
MiddEval3/code/evaldisp.cpp

```
11         if (gt == INFINITY) // unknown
12             continue;
13         float d = scale * disp.Pixel(x / scale, y / scale, 0);
14
15         /** sanity checks **/
16
17         float err = fabs(d - gt);
18
19         /** code for accumulation **/
20     }
21 }
22 /** code for evaluation **/
```

Listing 13: Middlebury Evaluation - pseudocode

The evaluation results on the website of the Middlebury benchmark are generated by comparison to the full resolution ground truth. For all data sets plotted in this section, the mean values of these data sets are visualized as dotted lines in the corresponding colors of the data plots. Further, in plots that show per image set results, lines connecting data points that belong to the same device or algorithm were added, to guide the eye and help locating these data points.

The percentage of bad pixels metric as described in Section 2.2.2 assigns a quality value to the result of an image set which asserts the percentage of how many of the pixels with valid disparity values contain values that are within an  $\varepsilon$  region of their corresponding ground truth value. This  $\varepsilon$ -region was defined in Formula 2.41 by the  $\delta$ -variable. The four thresholds that are used on the website of the Middlebury benchmark are 4.0, 2.0, 1.0 and 0.5. A comparison of algorithms using this metric and the 0.5-threshold, separates the results of sub-pixel methods from methods with integer-value disparity results. If  $\delta$  is chosen to be 0.5, then whole-pixel-method results usually have, due to rounding errors, a higher percentage of bad pixels than sub-pixel methods.

The usage of the full resolution ground truth for evaluation of submitted results with lower resolution is a significant factor in the error rate calculation for a submission, as shown in Figure 6.19. Figure 6.19 visualizes the difference between the evaluation methods. When a disparity map is scaled up to a higher resolution, the maximum disparity for this disparity map changes as well. The resulting disparity values have to be scaled to the new maximum disparity value. A scale up produces for every pixel in low resolution multiple new pixel positions in the higher resolution. For these new pixel positions, disparity values have to be estimated. This estimation introduces an error which results in higher error rates for low resolution results when the full resolution ground truth is used, as compared to when the corresponding low resolution ground truth is used. Detailed plots of the evaluated data sets for Figure 6.19 are provided in Appendix 1.

In Figure 6.19a and Figure 6.19d the average of the bad percentage metric with  $\delta$  being 0.5, 1.0, 2.0 and 4.0 is plotted for the training data set in quarter (Q) resolution. The compared algorithms are SGM (yellow), MPSV (blue), DF (green) and our implementation (red). This plot visualizes the difference between the evaluation tools *runeval* and *runevalF*.

It is important to consider the number of invalid pixels in the results created by the compared algorithms in order to compare sparse algorithms. The rates of invalid pixels are shown in the Figures 6.13, 6.14 and 6.15. These figures show that our implementations percentage of invalid pixels is on average (for all resolutions) around 35%. For quarter resolution the average invalid pixel percentage is close to 38% (Figure 6.13), for half resolution the percentage is near 34% (Figure 6.14) and for full resolution the average is close to 32% (Figure 6.15)

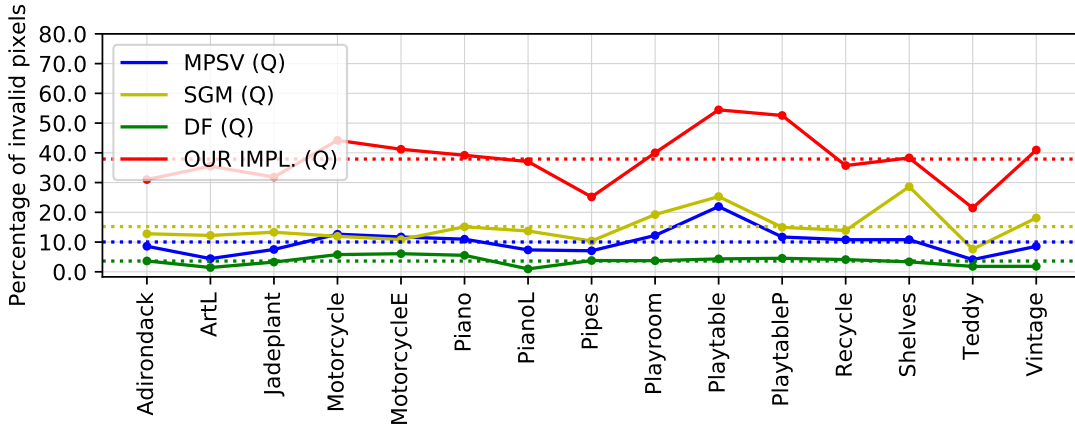


Figure 6.13: Comparison of invalid pixel rates (sparseness) of different algorithms at quarter resolution. Percentage of invalid pixels (y-axis) for each image set (x-axis): SGM (yellow), SGM\_ROB (green), MPSV (blue), DF (green) and our implementation in quarter resolution (red).

The Figures 6.16, 6.17 and 6.18 visualize the average disparity error per image set. For quarter and half resolution the average error produced by *runeval* is, apart from a scaling factor which is 0.5 for half and 0.25 for quarter resolution, the same as the average error produced by *runevalF*. The scaling of the average disparity error is visualized in Figure 6.16 and Figure 6.17. These figures show that our implementation's average disparity error is around 3 or 10 pixels for *runeval* or *runevalF* at quarter resolution which has image widths of 657 to 750 pixels, around 5 or 10 pixels at half resolution (1315 to 1500 pixels image width) and around 8 pixels for both *runeval* and *runevalF* at full resolution (2630 to 3000 pixels image width).

In Figure 6.19b and Figure 6.19e the relation of average bad pixel percentage to varying  $\delta$  for half resolution results is visualized. We used the algorithms SGM\_ROB (yellow), IDR

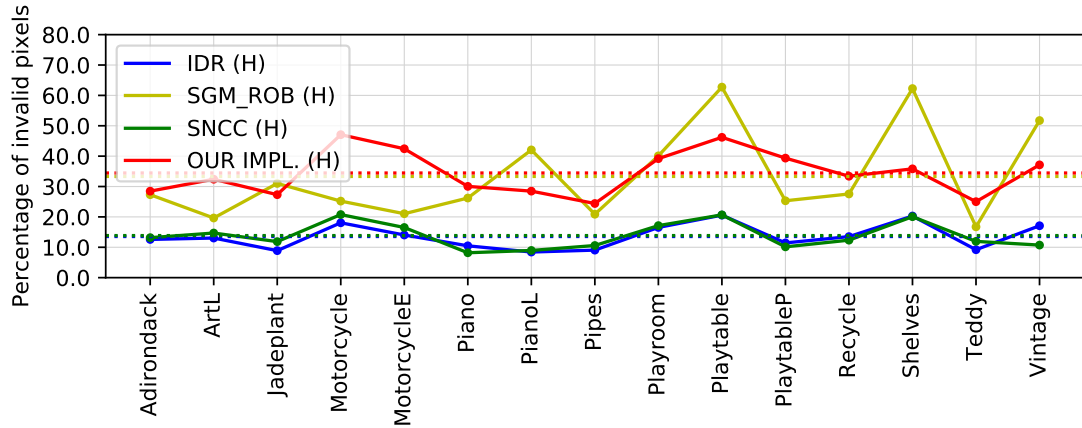


Figure 6.14: Comparison of invalid pixel rates (sparseness) of different algorithms at half resolution. Percentage of invalid pixels (y-axis) for each image set (x-axis): SGM\_ROB (yellow), SNCC (green), IDR (blue), DF (green) and our implementation in half resolution (red).

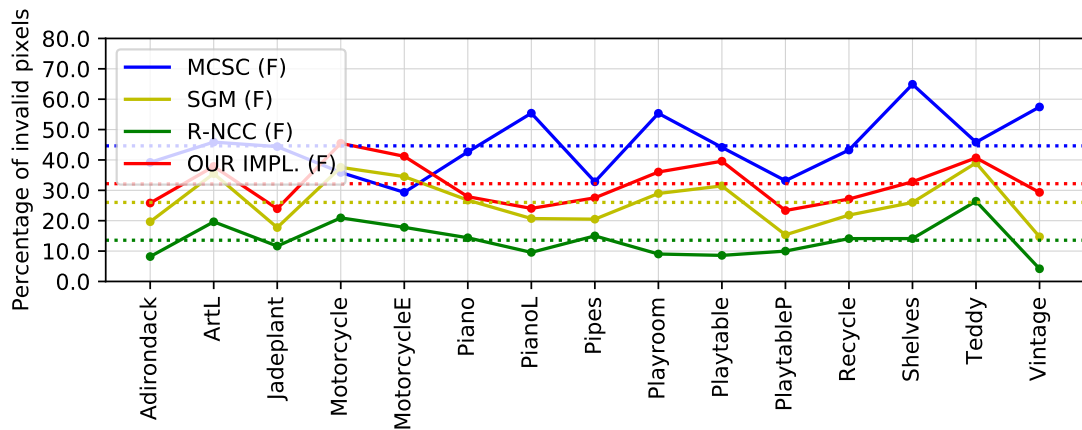


Figure 6.15: Invalid pixel rates (sparseness) of different algorithms at full (F) resolution. Percentage of invalid pixels (y-axis) for each image set (x-axis): SGM (yellow), R-NCC (green), MCSC (blue), DF (green) and our implementation in full resolution (red).

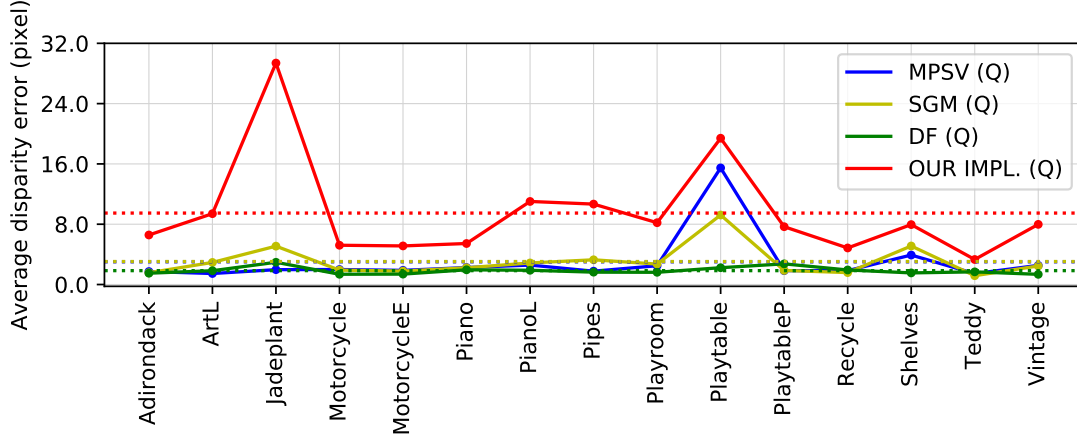
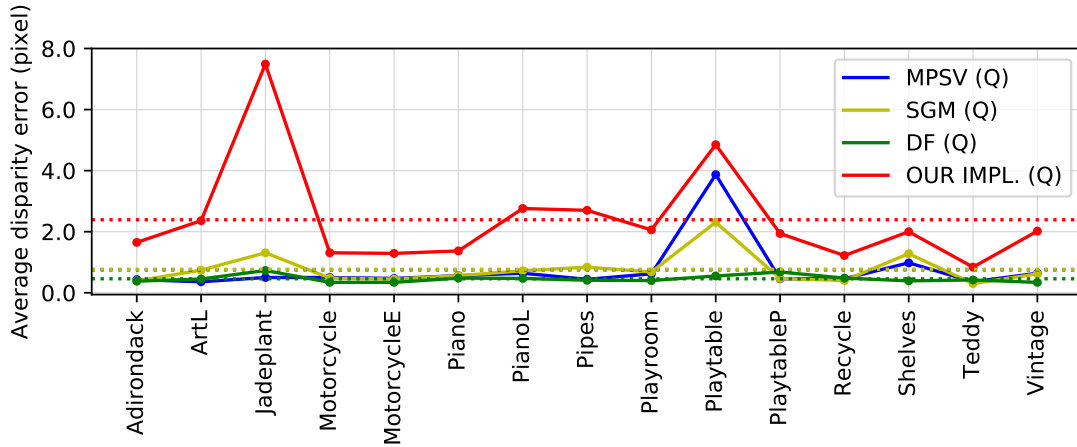
(a) Average disparity error for quarter (Q) resolution data (*runevalF*).(b) Average disparity error for quarter (Q) resolution data (*runeval*).

Figure 6.16: Average disparity error for (a) *runevalF* / (b) *runeval*. Average disparity error (y-axis) for each image set (x-axis): SGM (yellow), SGM\_ROB (green), MPSV (blue), DF (green) and our implementation in quarter resolution (red).

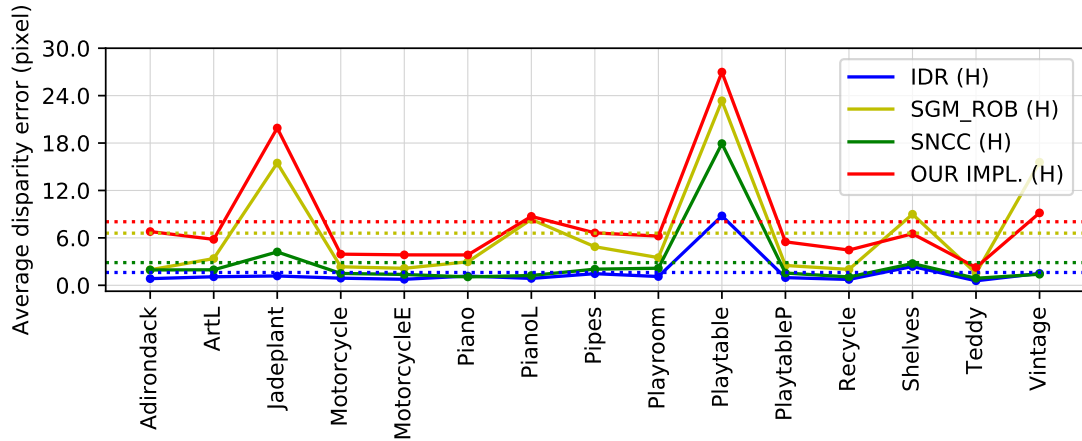
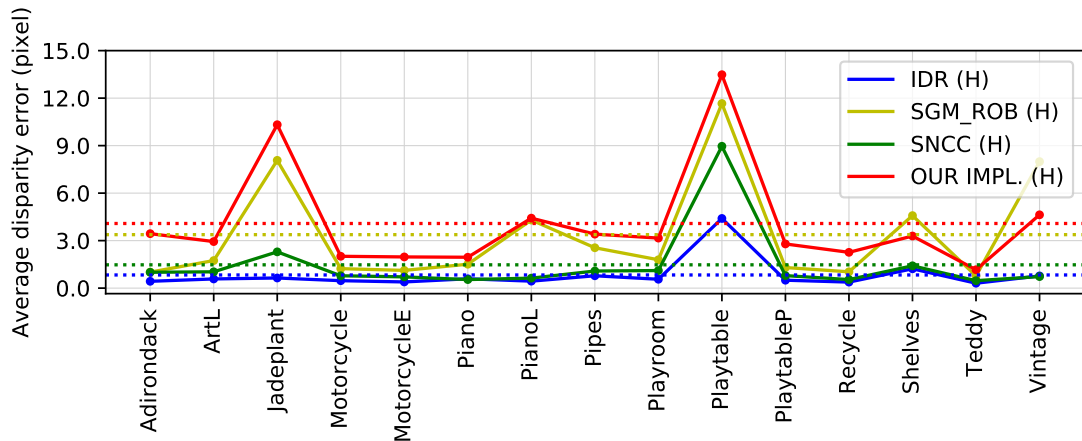
(a) Average disparity error for half (H) resolution data (*runevalF*).(b) Average disparity error for half (H) resolution data (*runeval*).

Figure 6.17: Comparison of different algorithms' average disparity error for (a) *runevalF* / (b) *runeval*. Average disparity error (y-axis) for each image set (x-axis): SGM\_ROB (yellow), SNCC (green), IDR (blue), DF (green) and our implementation in half resolution (red).



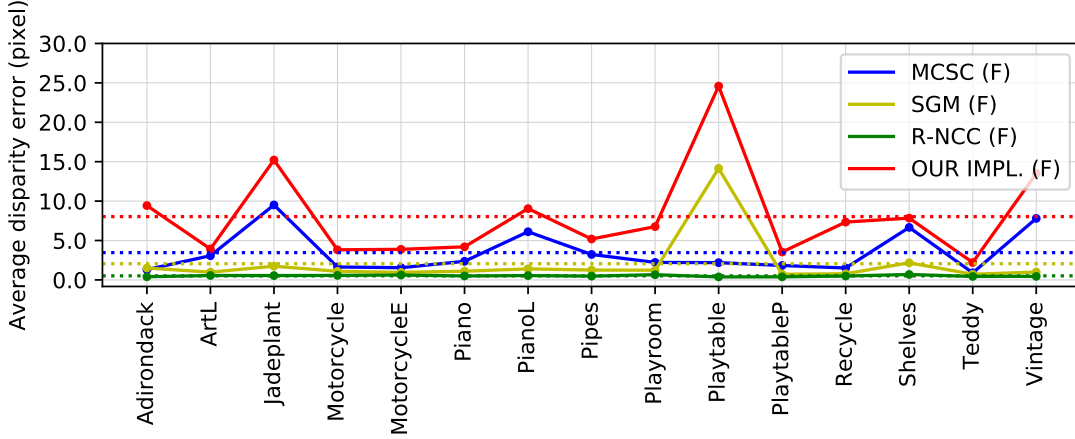


Figure 6.18: Average disparity error for full (F) resolution. Average disparity error (y-axis) for each image set (x-axis): SGM (yellow), R-NCC (green), MCSC (blue), DF (green) and our implementation in full resolution (red).

(blue), SNCC (green) and our implementation for half resolution (red) for comparison at half resolution level.

The Figures 6.19c and 6.19f show the results for algorithms working on full resolution data sets. At full resolution, the results of the evaluation scripts *runeval* and *runevalF* are equivalent. This is because *runeval* uses full resolution ground truth for comparison and *runevalF* does not scale the result disparity map because it is already provided at full resolution. Therefore, both tools use the same input data, error metrics and evaluation techniques in full resolution case. The compared algorithms are SGM (yellow), R-NCC (green), MCSC (blue) and our implementation for full resolution (red).

In Figure 6.19 different resolutions were used to visualize the influence of resolution on the quality of the result. Further, and the influence of the scaling strategy used by *runevalF* is shown in the comparison of Figure 6.19a and Figure 6.19b to Figure 6.19d and Figure 6.19e. Different algorithms were used for the different resolutions, this is due to the Middlebury Benchmark’s regulation that restricts authors’ submissions to one resolution. The restriction to one resolution, results that results for most submitted methods usually are available in only one resolution. An exception is the SGM (SGM\_ROB) algorithm by [Hir08] for which results are available in all three resolution classes (Q, H and F). Therefore, we used multiple algorithms in order to compare our implementation for quarter, half and full resolution. The results for the algorithms in Figure 6.19 were taken from the Middlebury Benchmark’s website.

The *Left-Right Consistency Check* described in Section 4.4 results in a higher percentage of invalid pixels of our implementation of [HZW<sup>+</sup>10]’s method. This is due to the rejection of results that cannot be left-right validated. Furthermore, the *Sub Pixel Refinement* described in Section 4.3 results in good estimations for sub-pixel disparities. This in turn

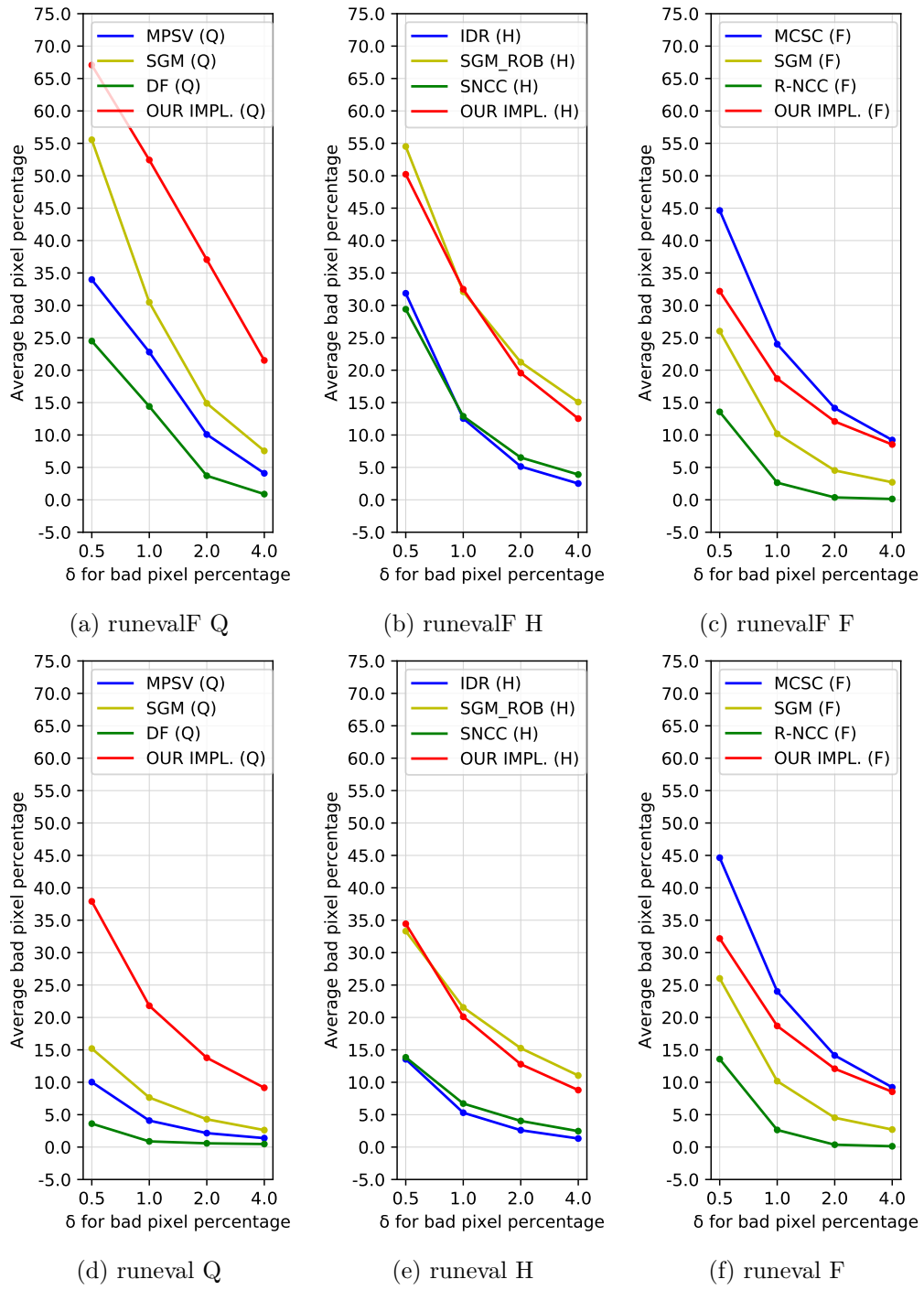


Figure 6.19: This figure shows the average bad pixel percentage of different algorithms. The first row of plots shows results for the *runevalF* script and the second row shows results for *runeval*. The first column of plots shows quarter resolution results, the second column shows half resolution results and the third column shows full resolution results.

is the reason why our implementation of the method of [HZW<sup>+</sup>10] has its best results when it is applied to the full resolution data set. This can be seen in Figure 6.15 where our implementation’s invalid pixel rate is in the range of other sparse, state of the art matching method’s invalid pixel rates. And this can be seen in Figure 6.18 where our implementation’s average disparity error is below 10 pixels.

## 6.4 Additional Results

In this section, we will discuss the runtime behavior of our implementation.

In Figure 6.1, a number of data points with unusually high runtime for the *Intel IRIS 6100* graphics card can be seen. An exploration of the data points of this device (shown in Figure 6.20), affirms that the data points with higher runtime correspond to six image sets along all three available resolution categories. These image sets are Djembe, DjembeL and Newkuba from the test set and Adirondack, ArtL and Jadeplant from the training set. Further on, we will refer to the set of these six image sets as the *P-Set*.

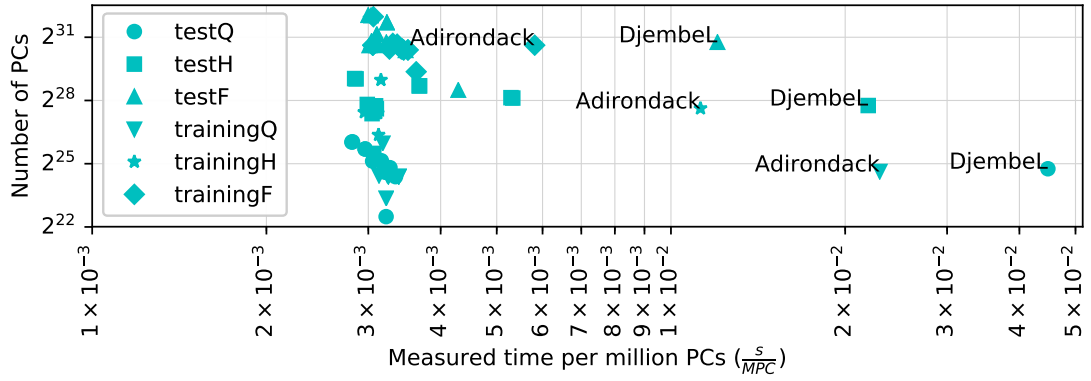
The *P-Set*’s quarter resolution images’ widths are 719px for Djembe, 719px for DjembeL, 701px for Newkuba, 718px for Adirondack, 347px for ArtL and 659px for Jadeplant. Five of these image widths are prime numbers. The image set Adirondack’s image width (718px) is not a prime number, as it is an even number. However,  $359 (= \frac{718}{2})$  is a prime number. Note that for the Middlebury Benchmark’s image sets only the quarter resolution image sets may have prime number image widths ( $w_p$ ). The corresponding half resolution image sets will have widths of  $w_p \cdot 2$  and the corresponding full resolution image sets will have an image width of  $w_p \cdot 4$ .

Figure 6.20 shows the data set of the *Intel IRIS 6100* (also shown in Figure 6.1) with the problem size in PCs on the x-axis and the time per million PCs on the x-axis<sup>5</sup>. The three subplots show the data points of all image sets, without the data points for the *P-Set*, as unlabeled data points. Subfigure 6.20a shows the labeled data points for *Adirondack* and *DjembeL*, Subfigure 6.20b shows the labeled data points for *Jadeplant* and *Djembe* and Subfigure 6.20c shows the labeled data points for *ArtL* and *Newkuba*. This shows that data points corresponding to the image sets in the *P-Set* have a runtime value (shown on the x-axis) close to or larger than  $6 \cdot 10^{-3} \frac{s}{MPC}$ .

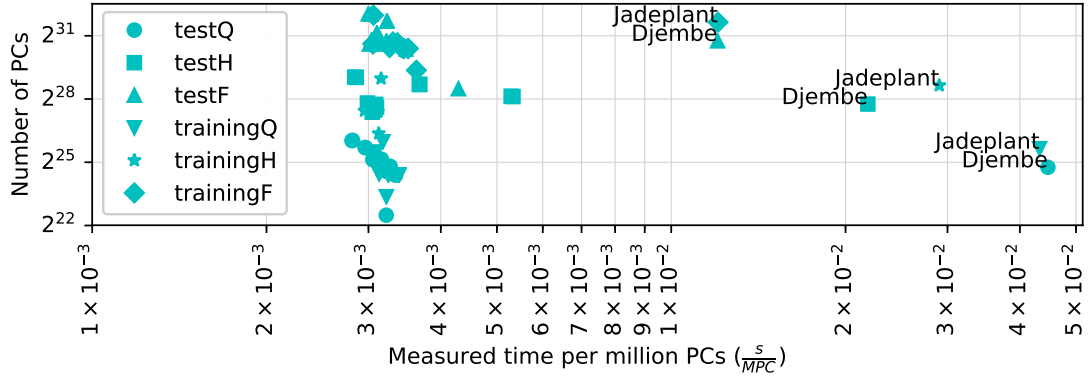
As discussed in Section 6.1, the results shown in Figure 6.1 are the average runtime results from 100 runs i.e. each plotted data point represents the average of 100 runs of the image set corresponding to this data point. This was done to cancel out noise in the measurement data. Therefore, it is unlikely that the behavior, which can be observed for the six image sets mentioned above, occurs through noise.

The higher runtime of our implementation on the *Intel IRIS 6100* for the six image sets in the *P-Set*, is also observable, although less distinct, with the *Mali T760* graphics card. There are two main distinctions between these two devices (*Mali T760* and *Intel IRIS*

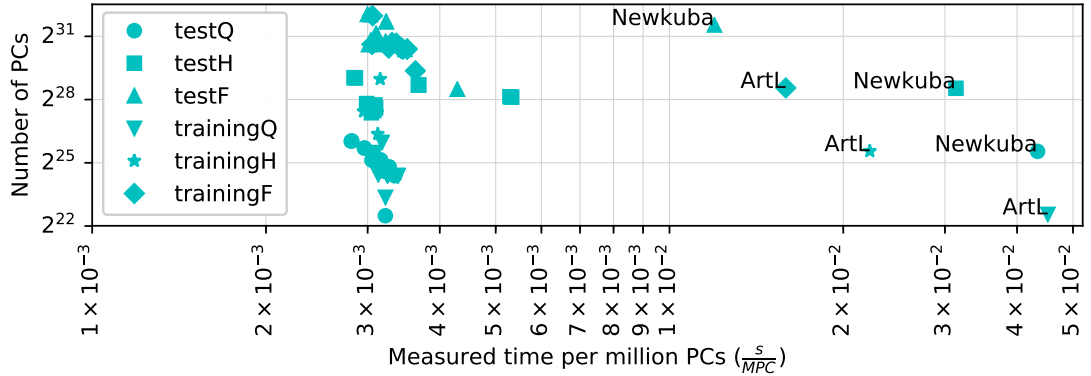
<sup>5</sup>The significance of the x-axis and y-axis in this plot is reversed to the axes in Figure 6.1.



(a) Adirondack and Djembel



(b) Djembe and Jadeplant



(c) ArtL and Newkuba

Figure 6.20: Visualization of the *Intel IRIS 6100* data set shown in Figure 6.1. In the subfigures (a), (b) and (c), the positions of the data points corresponding to the image sets Adirondack, ArtL, Djembe, DjembeL, Jadeplant, and Newkuba are shown. The unlabeled data points correspond to the remaining image sets of the Middlebury Benchmark. These data points visualize the baseline of the runtime performance of our algorithm on this device.

6100) and the other examined devices in Figure 6.1: One is the maximum work group size, which is 256 with these two devices and at least 1024 with the other devices. The other is the size of the work group dimensions, which is (256,256,256) for these two devices and at least (1024,1024,64) for the other devices. We suspect that the maximum work group size, in combination with the problem dimensions as discussed in the following, is responsible for the higher runtime of these image sets on these two devices.

A relation between the work group size and the image sets in this context reveals a weakness in our strategy to find the biggest possible work group size, as shown in Listing 2. If the maximum work group size of a device is smaller than the image width, then the strategy to find an integer divisor for the image width is applied. This integer divisor has to be big enough that the quotient of image width and the divisor is smaller than the maximum work group size. In some cases, this can lead to a work group size value of 1, which leads to bad runtime results. This is the case if the image width is a prime number. Therefore, the behavior of bad runtime is specific to our implementation and will always occur if the image width is a prime number that is larger than the used device's maximum work group size. For this reason, the work group size for five of the quarter resolution images in our *P-Set* is of value 1. The work group size for the quarter resolution image set Adirondack is of value 2. This is further visualized in Figure 6.20a, as the runtime per million PCs of Adirondack in quarter resolution is roughly the same as the runtime of Djembell in half resolution, which is close to  $2 \cdot 10^{-2} \frac{s}{MPC}$ .

Further, Adirondack (H) and Djembell (F) have a work group size of value 4 and the runtime value of these two image sets is roughly  $1 \cdot 10^{-2} \frac{s}{MPC}$ , which is half the runtime of the image sets Adirondack (Q) and Djembell (H). Finally, the runtime of Adirondack (F) at a work group size value of 8, is smaller than  $6 \cdot 10^{-3} \frac{s}{MPC}$ , which is roughly half of the runtime per million PCs of Adirondack in half resolution.

Strategies to fix the problem caused by prime number image sizes would be to clip the input images in such a way that the clipped image widths are integer divisible to avoid this problem or to use a work group size that does not divide the image width into equally sized parts. The latter would result in out of bound addressing of memory buffers, which would have to be fixed in all provided kernels by a clipping of the x-component of the address to the image borders.



## Summary and Outlook

In this thesis, we showed an example of how OpenCL can be used to execute stereo matching algorithms on different devices. We implemented a stereo matching method based on the method of [HZW<sup>+</sup>10], which relies on a Census transform and has demonstrated its potential for embedded real-time implementations. The implemented source code consists of OpenCL kernels that can be used by any SDK that is able to address the OpenCL interface. The code shows how these kernels can be invoked by implementations in C++ and Python.

Using the Middlebury benchmark, different OpenCL devices, the floating point performance values of these devices and the NCT shown by [JS92], we calculated the average number of Floating point Operations per Pixel Comparison (FLOPC) for our stereo matching implementation as shown in Formula 6.6.

Furthermore, we used the estimated FLOPC value to predict the runtime of our implementation on different devices. The predicted runtimes for the examined devices were in general in the same order of magnitude as the measured runtime values of these devices, as shown in Figure 6.1 and Figure 6.2. The predicted runtime value per mega pixel-comparison was usually faster than the measured runtime values. In Section 6.4, we argued that this was due to the fact that floating point performance values are usually peak efficiency values which are seldom reached in practice. Nevertheless, the FLOPC value allows for meaningful runtime predictions of an algorithm for different devices, which can be used to find the best price-to-runtime ratio when selecting a device for a specific algorithm. Further, the FLOPC value allowed the comparison of different algorithms in the context of their runtime complexity.

In Section 6.1.1, we used the FLOPC value to estimate the size of the problem space in order to achieve a certain matching rate for a specific image ratio N:M on a specific device. This can be used for the design of mobile, real-time stereo matching systems, as the floating point performance of mobile devices is usually a limiting factor.

Finally, we discussed why our implementation had discernible outliers for six image sets concerning calculation runtime on two devices which had fewer processing units than the other evaluated devices. This was due to our implementation's partitioning strategy which does not work well when the image width is not integer divisible into equal parts smaller than the devices number of processing units.

Future work should include further evaluations of the FLOPC value as algorithm specific and hardware independent runtime complexity value. Another interesting topic for future work would be the translation of our OpenCL kernels into VHDL modules using the SDKs of FPGA vendors. Such an endeavor should include the runtime analysis of these modules and the necessary invocation code, using the FLOPC value.

Furthermore, the use of hierarchical data structures, such as image pyramids, could be subject of further research, as the estimation of disparities for high resolution images from low resolution disparity maps could be used to either increase accuracy or significantly reduce runtime of an algorithm. For the latter, the FLOPC value could again provide a good indicator of the runtime complexity.



# Appendix 1

This chapter contains the full data set plots for the *bad pixel percentage* metric for *runeval* and *runevalF* for the different algorithms and different resolutions.

In plots that show per image set results, lines connecting dots belonging to the same device or algorithm were added, in order to guide the eye and help with locating the data points.

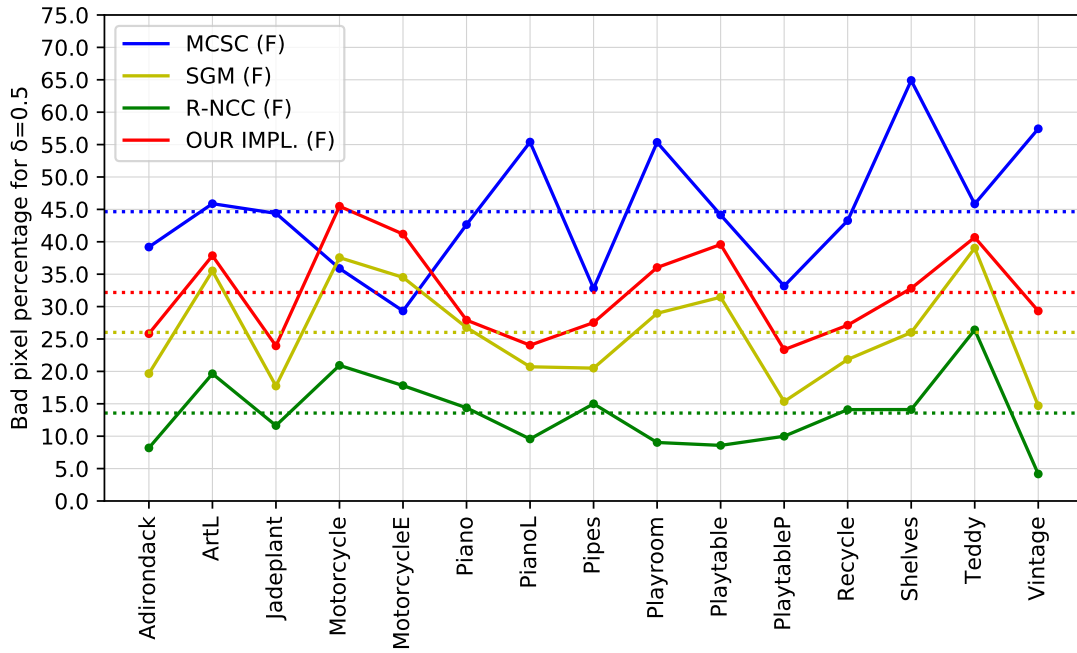


Figure 1: Comparison of different algorithms in full (F) resolution. Percentage of bad pixels at threshold  $\delta = 0.5$  (y-axis) for each image set (x-axis): SGM (yellow), R-NCC (green), MCSC (blue), DF (green) and our implementation in full resolution (red).

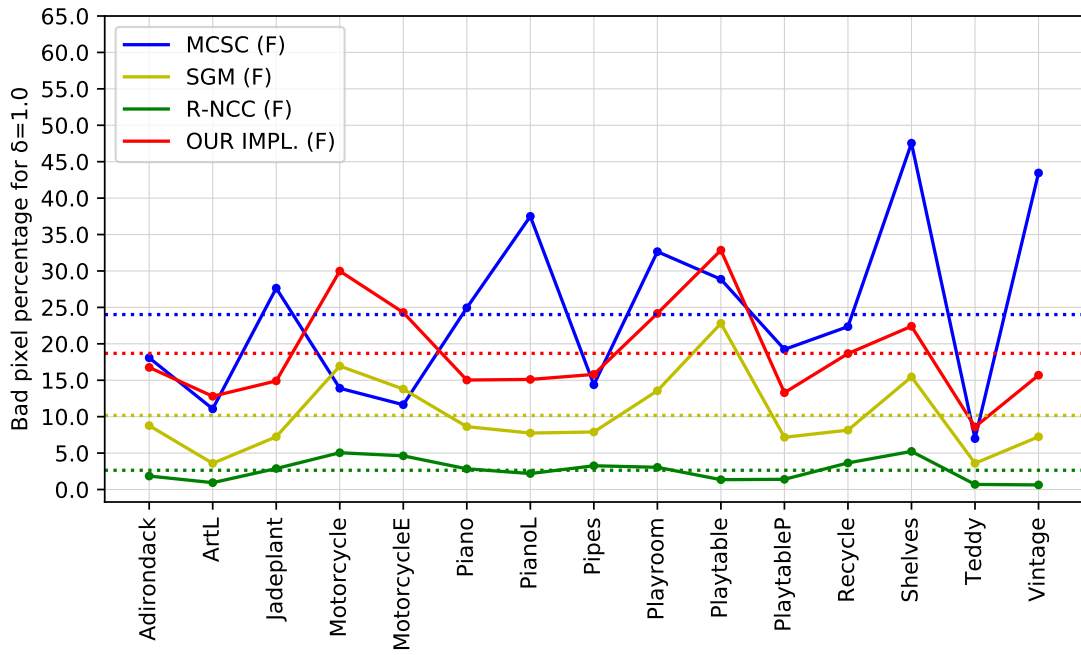


Figure 2: Comparison of different algorithms in full (F) resolution. Percentage of bad pixels at threshold  $\delta = 1.0$  (y-axis) for each image set (x-axis): SGM (yellow), R-NCC (green), MCSC (blue), DF (green) and our implementation in full resolution (red).

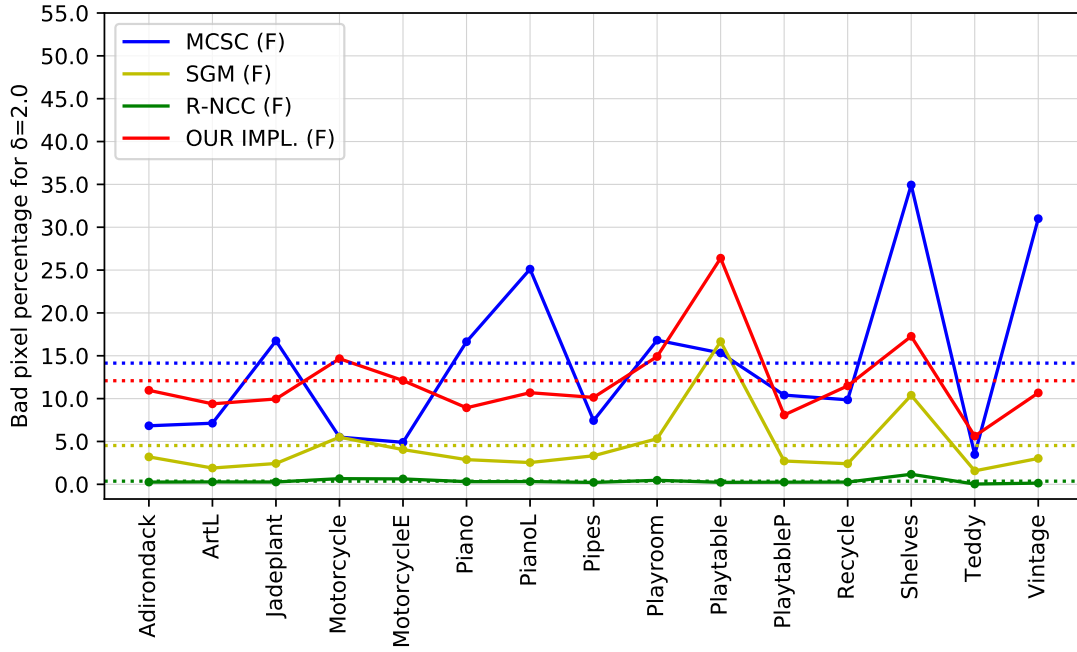


Figure 3: Comparison of different algorithms in full (F) resolution. Percentage of bad pixels at threshold  $\delta = 2.0$  (y-axis) for each image set (x-axis): SGM (yellow), R-NCC (green), MCSC (blue), DF (green) and our implementation in full resolution (red).

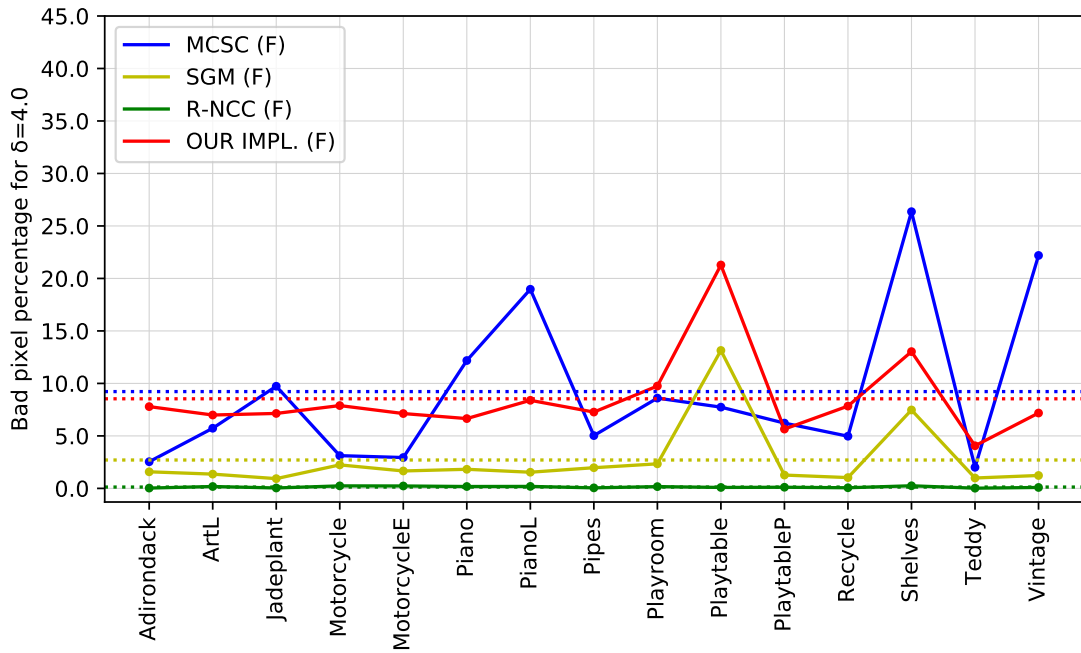
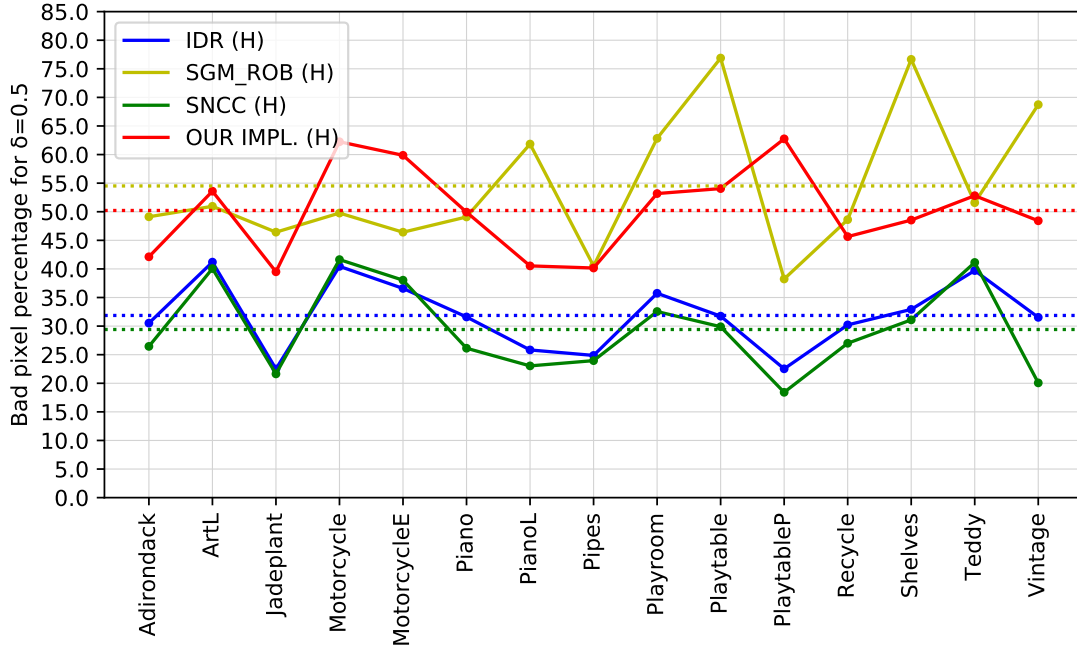
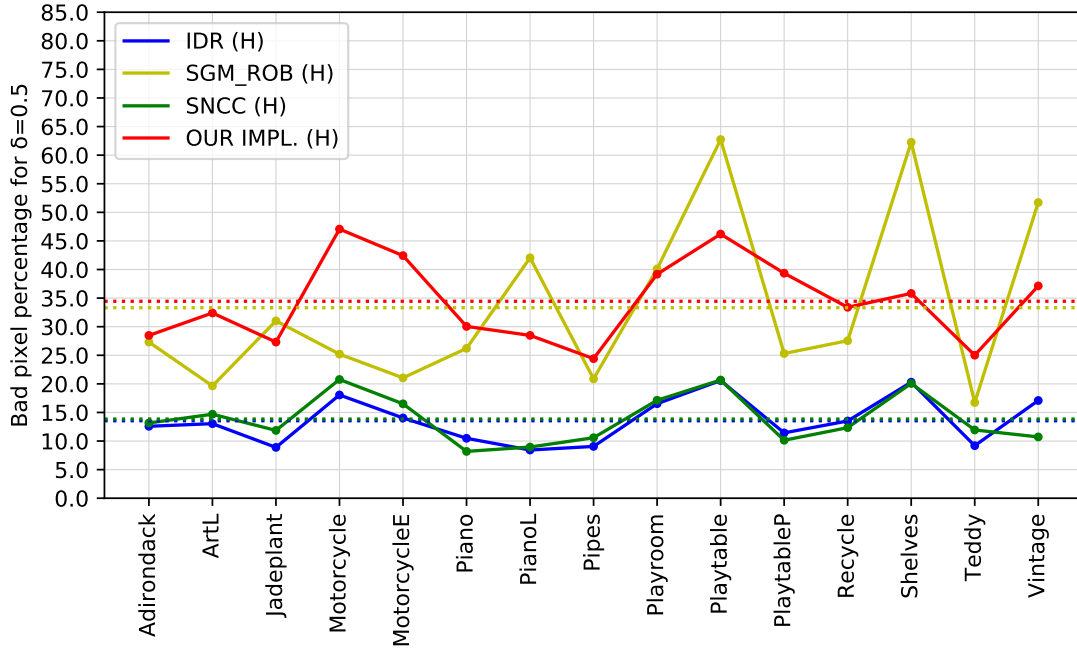


Figure 4: Comparison of different algorithms in full (F) resolution. Percentage of bad pixels at threshold  $\delta = 4.0$  (y-axis) for each image set (x-axis): SGM (yellow), R-NCC (green), MCSC (blue), DF (green) and our implementation in full resolution (red).

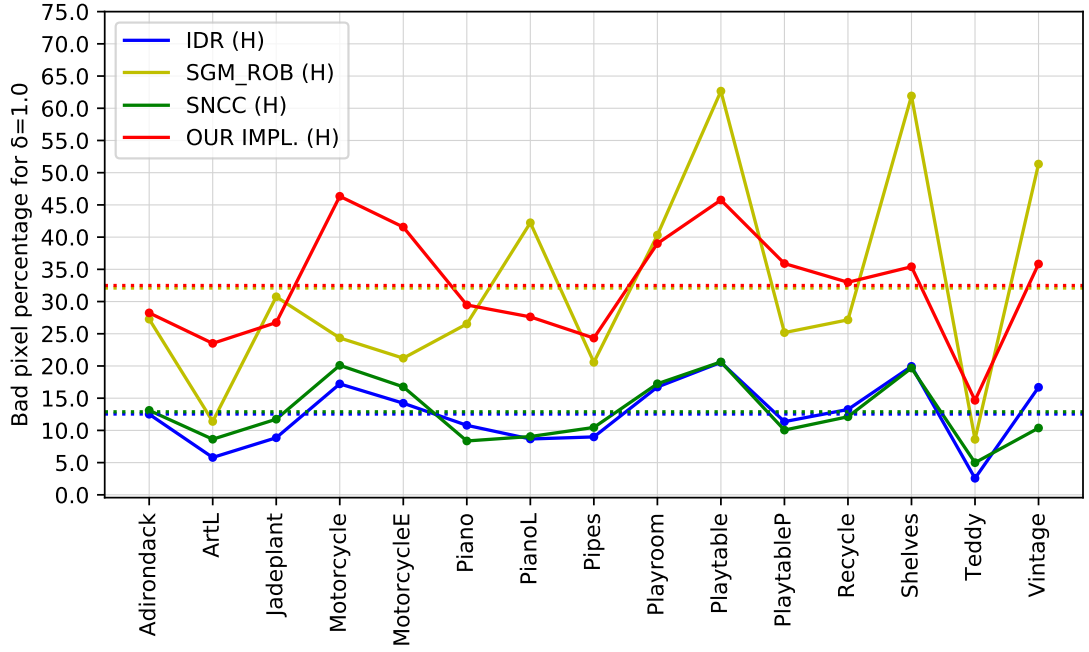


(a) Full resolution ground truth, half resolution data, bad pixel  $\delta = 0.5$ .

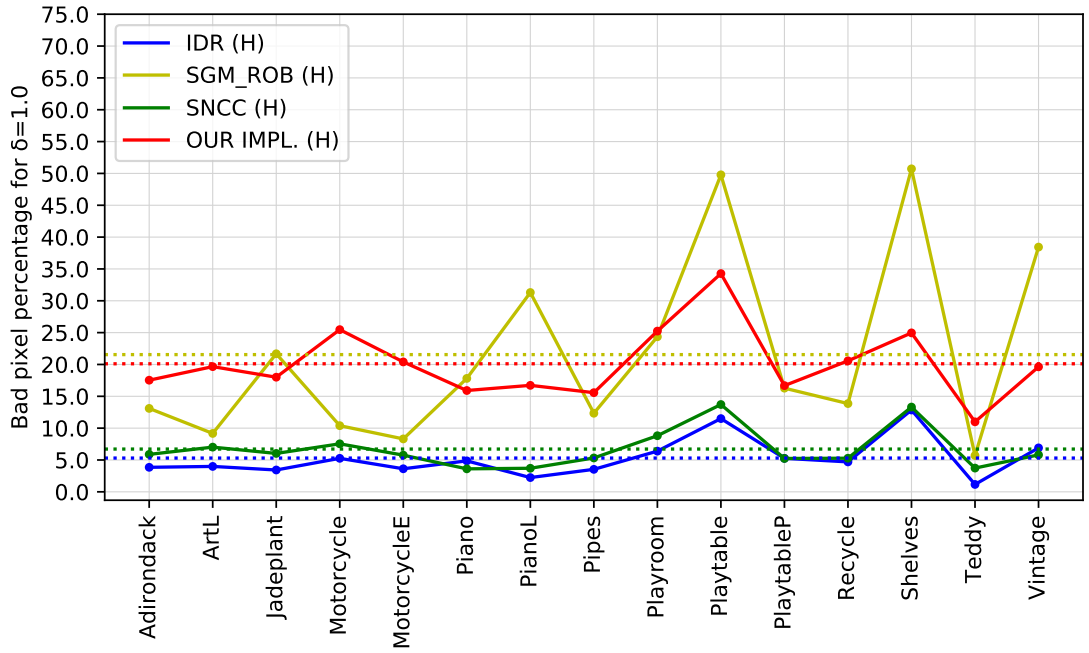


(b) Ground truth in half resolution, half resolution data, bad pixel  $\delta = 0.5$ .

Figure 5: Comparison of different algorithms using (a) *runevalF* / (b) *runeval*. Percentage of bad pixels at threshold  $\delta = 0.5$  (y-axis) for each image set (x-axis): SGM\_ROB (yellow), SNCC (green), I (blue), DF (green) and our implementation in half resolution (red).

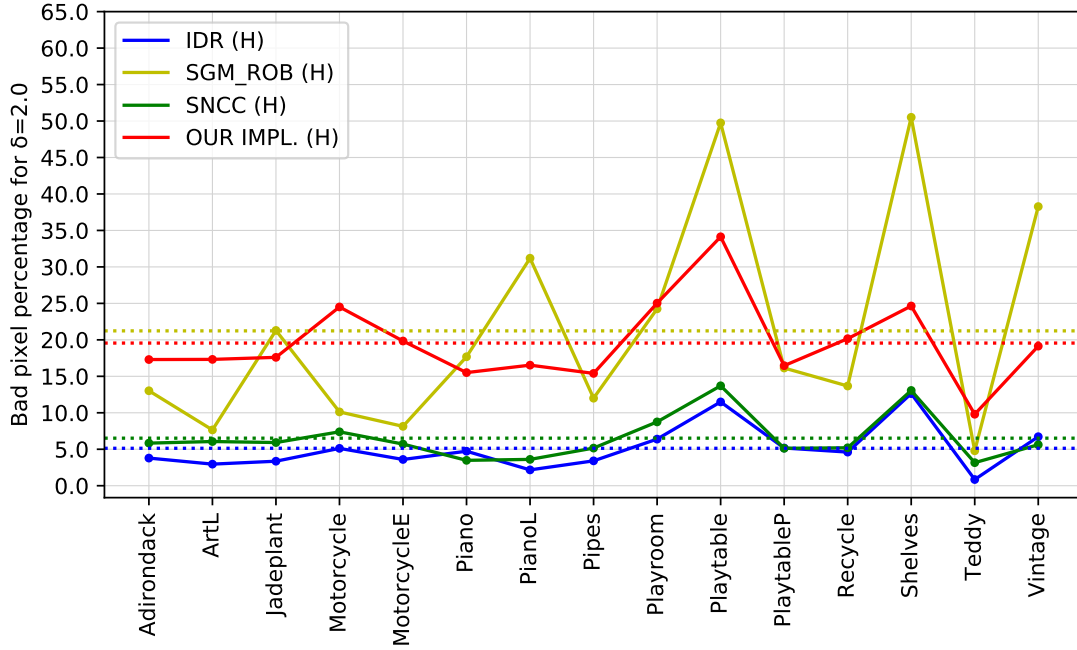


(a) Full resolution ground truth, half resolution data, bad pixel  $\delta = 1.0$ .

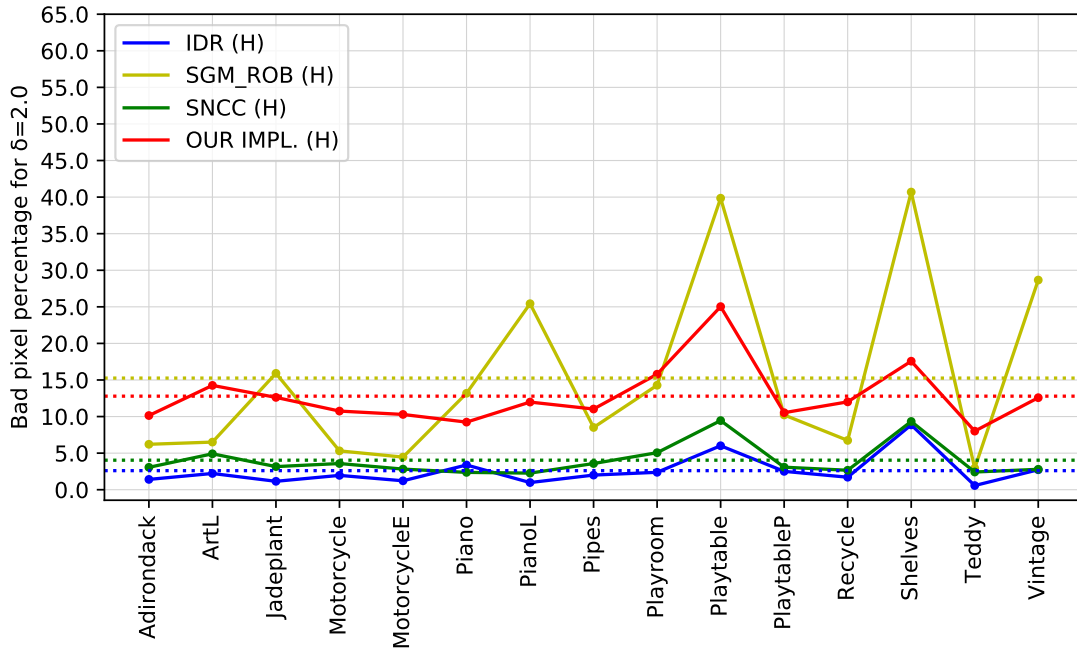


(b) Ground truth in half resolution, half resolution data, bad pixel  $\delta = 1.0$ .

Figure 6: Comparison of different algorithms using (a) *runevalF* / (b) *runeval*. Percentage of bad pixels at threshold  $\delta = 1.0$  (y-axis) for each image set (x-axis): SGM\_ROB (yellow), SNCC (green), IDR (blue), DF (green) and our implementation in half resolution (red).

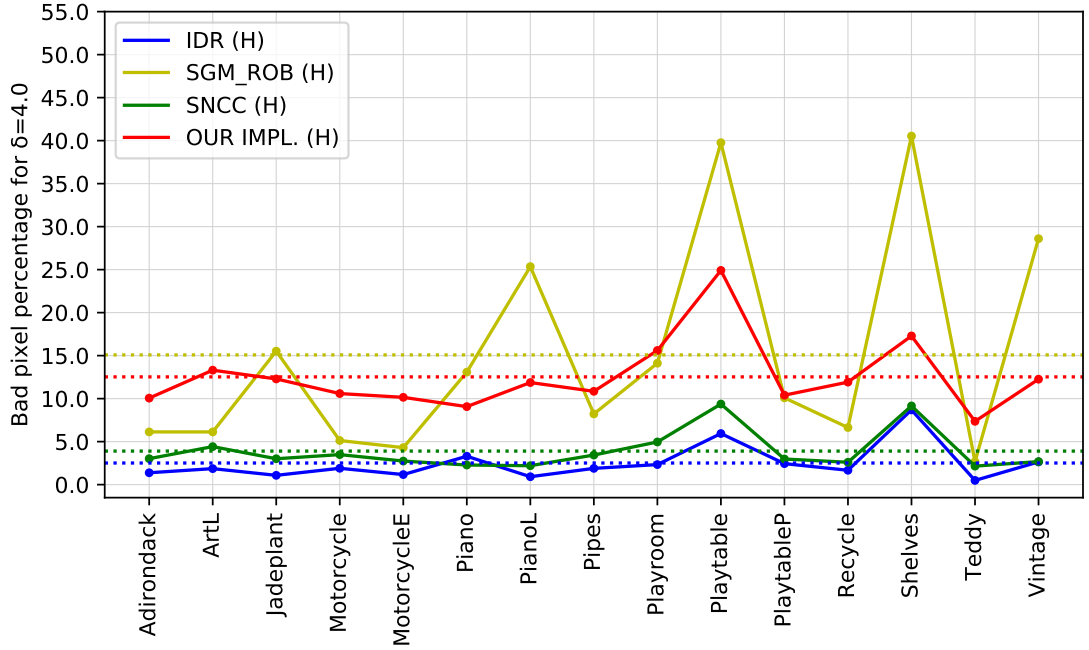


(a) Full resolution ground truth, half resolution data, bad pixel  $\delta = 2.0$ .

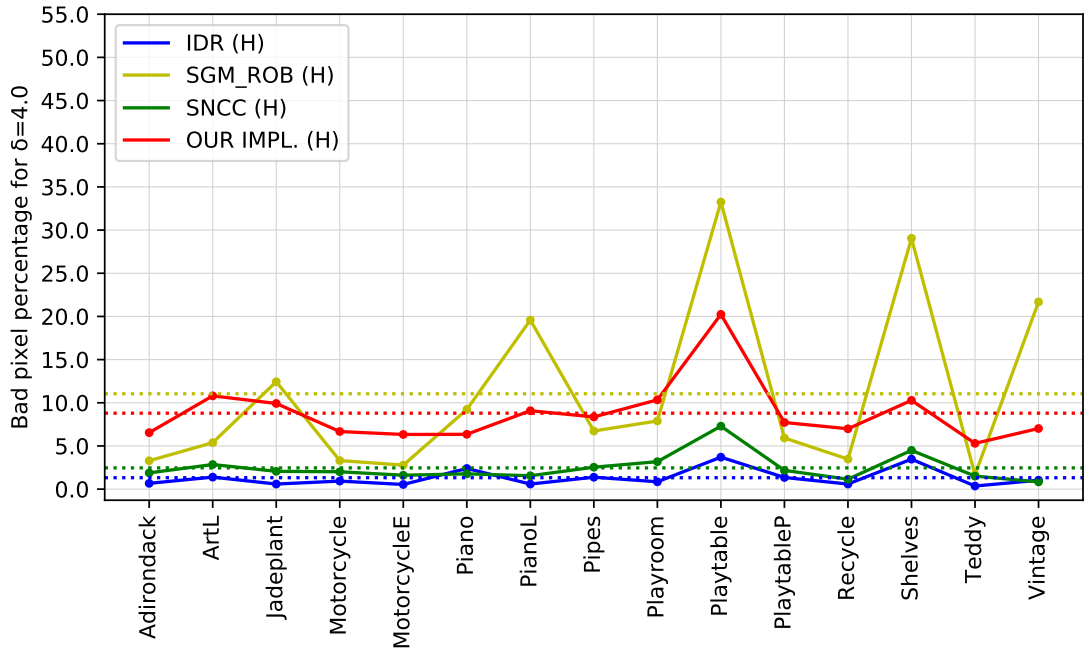


(b) Ground truth in half resolution, half resolution data, bad pixel  $\delta = 2.0$ .

Figure 7: Comparison of different algorithms using (a) *runevalF* / (b) *runeval*. Percentage of bad pixels at threshold  $\delta = 2.0$  (y-axis) for each image set (x-axis): SGM\_ROB (yellow), SNCC (green), IDR (blue), DF (green) and our implementation in half resolution (red).



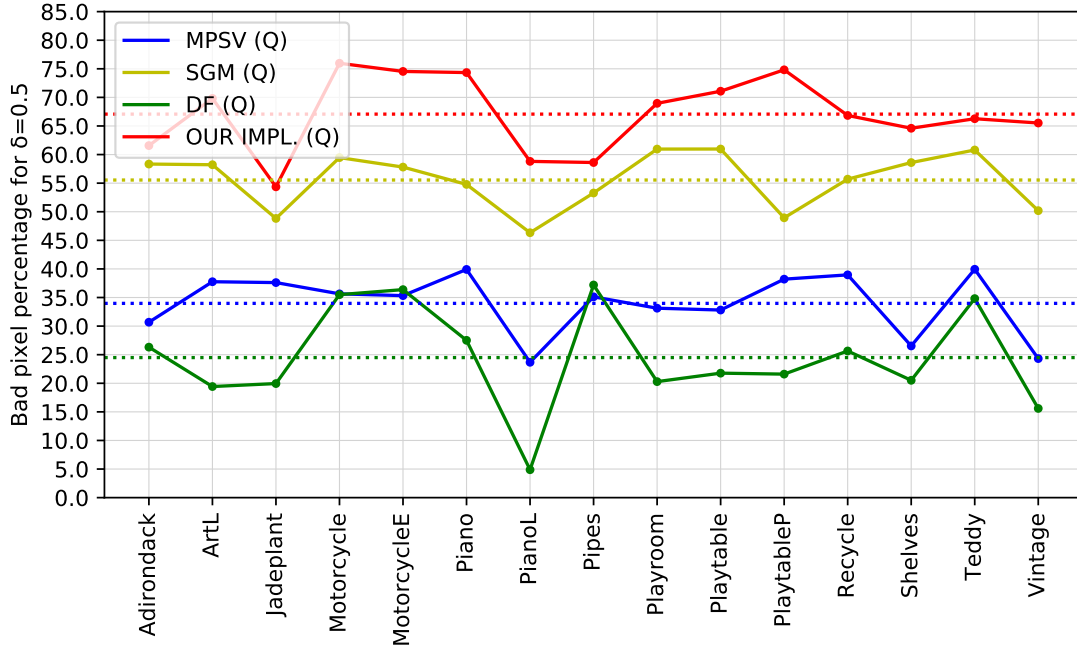
(a) Full resolution ground truth, half resolution data, bad pixel  $\delta = 4.0$ .



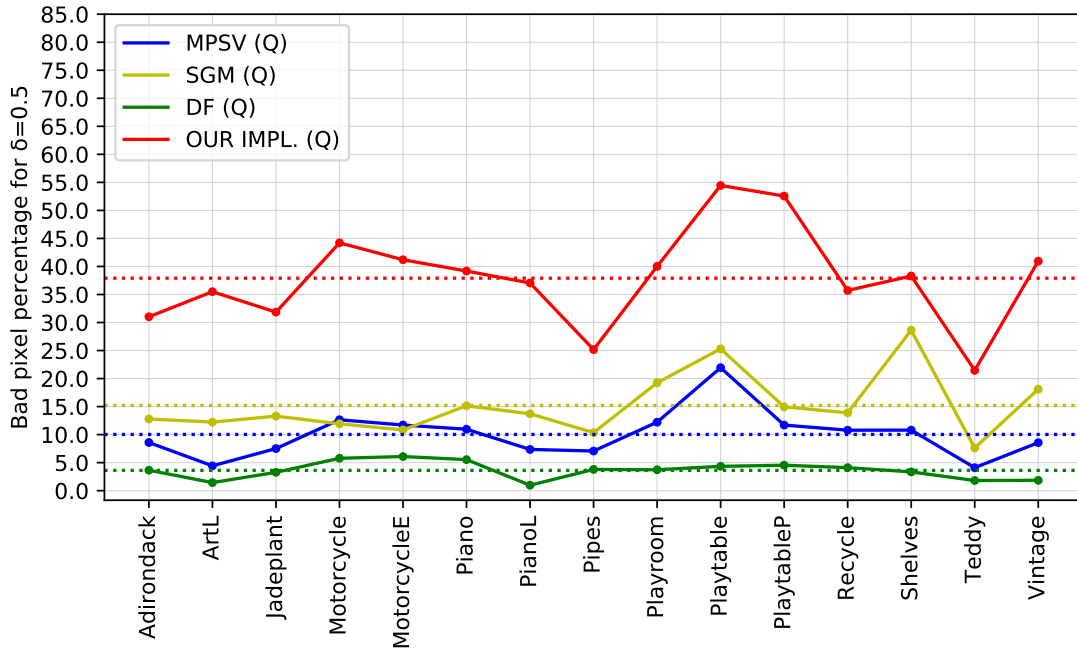
(b) Ground truth in half resolution, half resolution data, bad pixel  $\delta = 4.0$ .

Figure 8: Comparison of different algorithms using (a) *runevalF* / (b) *runeval*. Percentage of bad pixels at threshold  $\delta = 4.0$  (y-axis) for each image set (x-axis): SGM\_ROB (yellow), SNCC (green), IDR (blue), DF (green) and our implementation in half resolution (red).



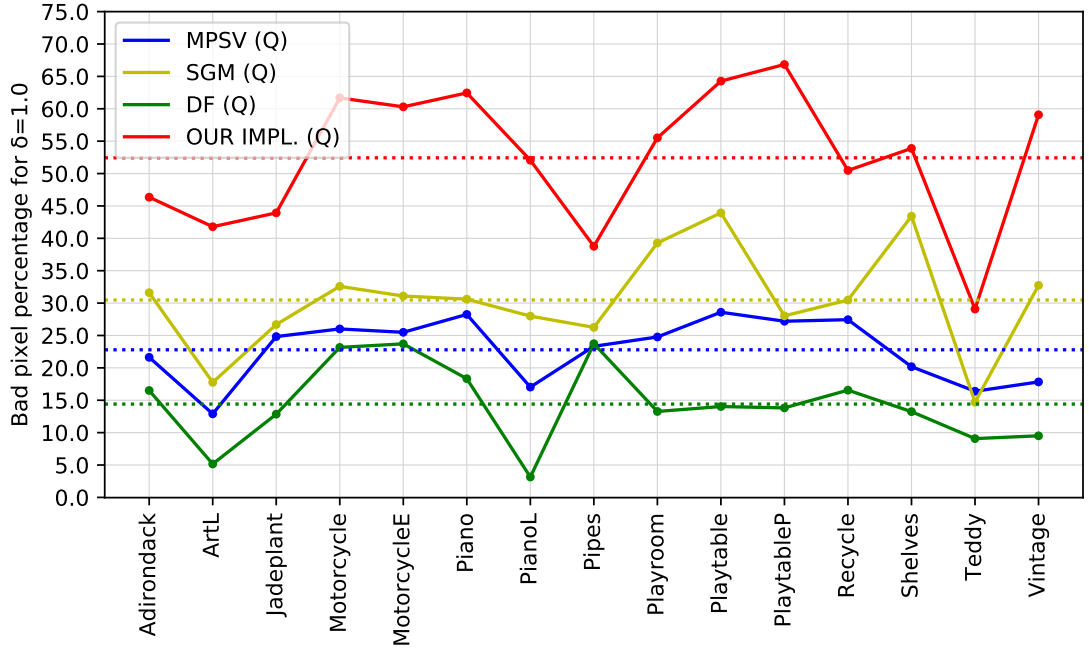


(a) Full resolution ground truth, quarter resolution data, bad pixel  $\delta = 0.5$ .

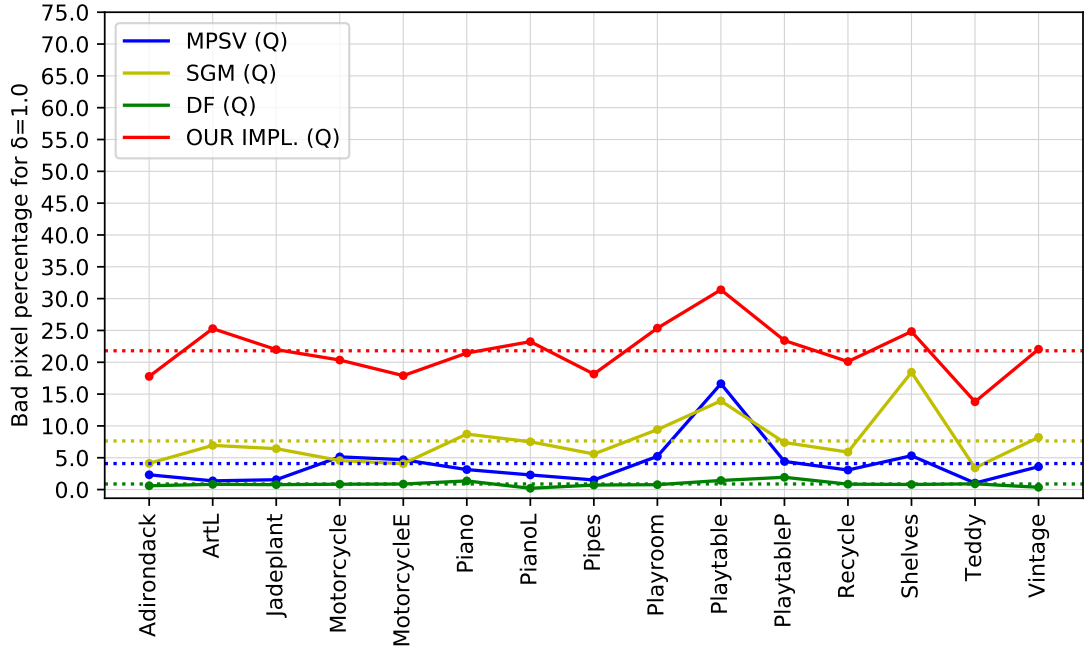


(b) Ground truth in quarter resolution, quarter resolution data, bad pixel  $\delta = 0.5$ .

Figure 9: Comparison of different algorithms using (a) *runevalF* / (b) *runeval*. Percentage of bad pixels at threshold  $\delta = 0.5$  (y-axis) for each image set (x-axis): SGM (yellow), SGM\_ROB (green), MPSV (blue), DF (green) and our implementation in quarter resolution (red).

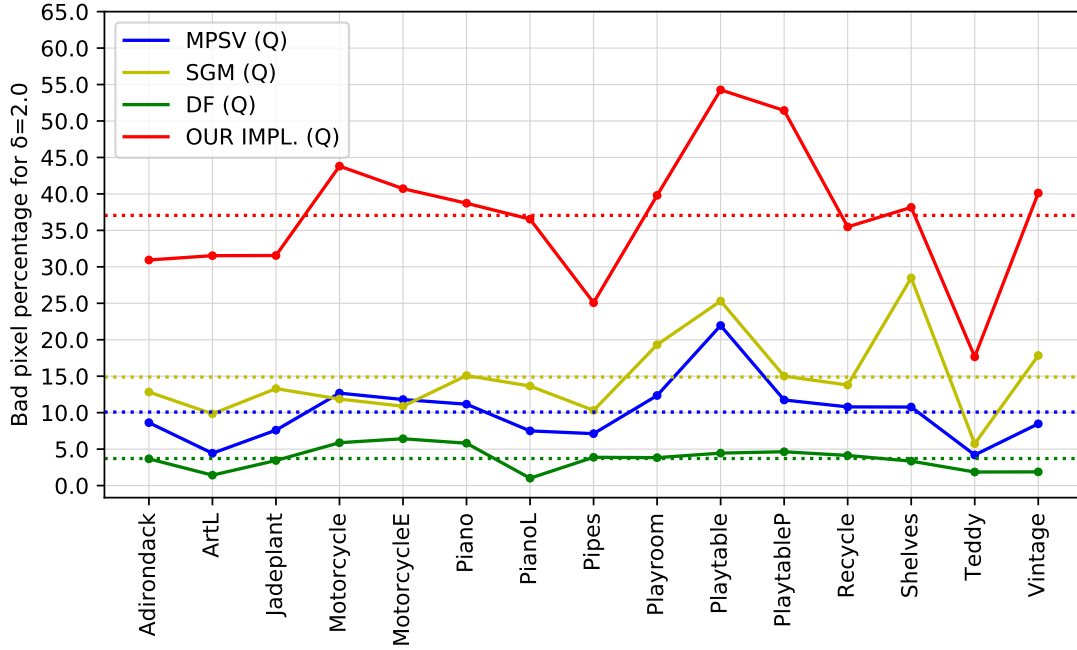


(a) Full resolution ground truth, quarter resolution data, bad pixel  $\delta = 1.0$ .

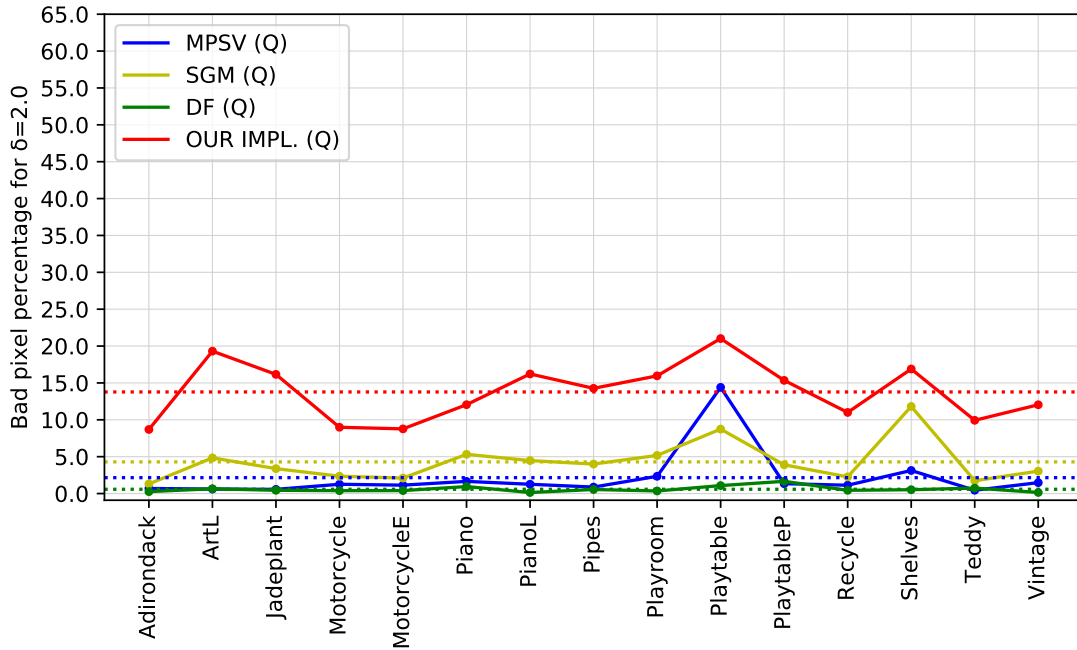


(b) Ground truth in quarter resolution, quarter resolution data, bad pixel  $\delta = 1.0$ .

Figure 10: Comparison of different algorithms using (a) *runevalF* / (b) *runeval*. Percentage of bad pixels at threshold  $\delta = 1.0$  (y-axis) for each image set (x-axis): SGM (yellow), SGM\_ROB (green), MPSV (blue), DF (green) and our implementation in quarter resolution (red).

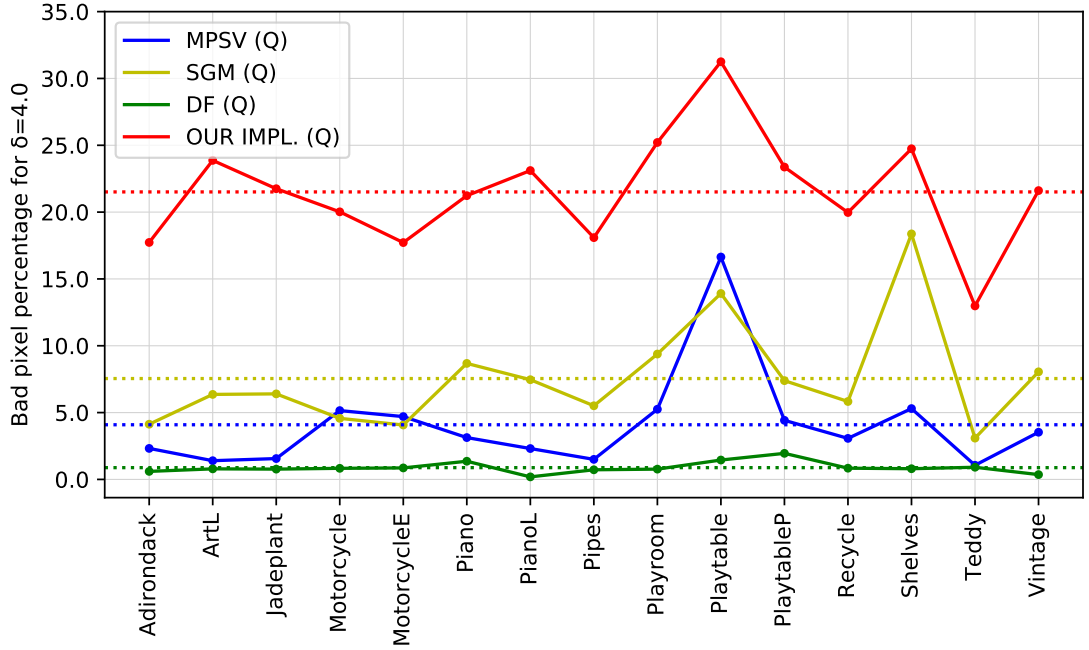


(a) Full resolution ground truth, quarter resolution data, bad pixel  $\delta = 2.0$ .

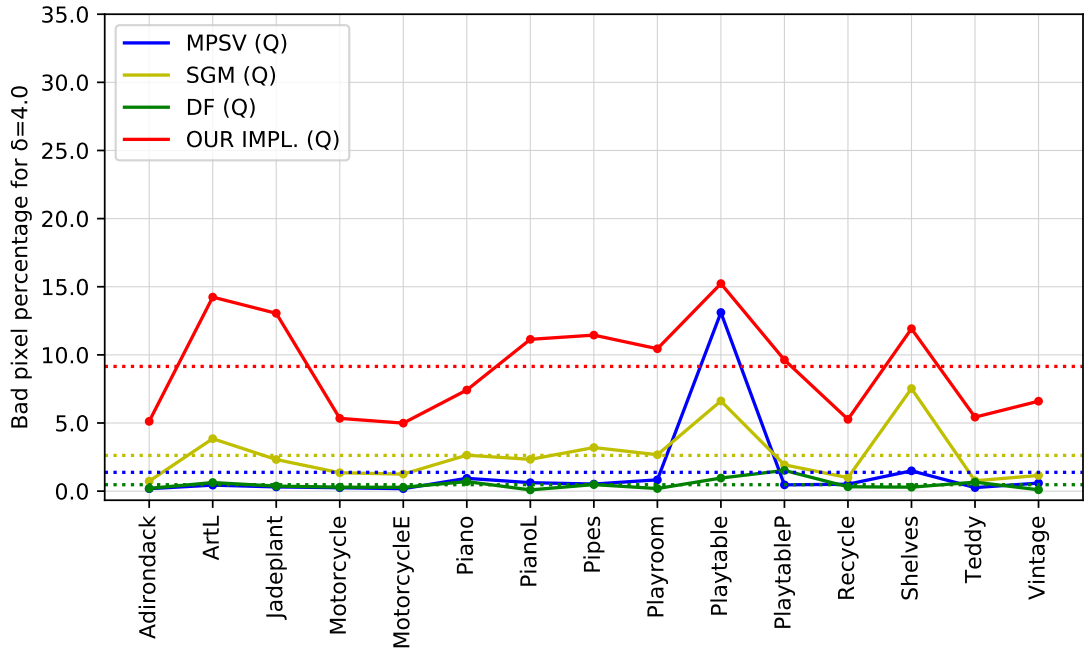


(b) Ground truth in quarter resolution, quarter resolution data, bad pixel  $\delta = 2.0$ .

Figure 11: Comparison of different algorithms using (a) *runevalF* / (b) *runeval*. Percentage of bad pixels at threshold  $\delta = 2.0$  (y-axis) for each image set (x-axis): SGM (yellow), SGM\_ROB (green), MPSV (blue), DF (green) and our implementation in quarter resolution (red).



(a) Full resolution ground truth, quarter resolution data, bad pixel  $\delta = 4.0$ .



(b) Ground truth in quarter resolution, quarter resolution data, bad pixel  $\delta = 4.0$ .

Figure 12: Comparison of different algorithms using (a) *runevalF* / (b) *runeval*. Percentage of bad pixels at threshold  $\delta = 4.0$  (y-axis) for each image set (x-axis): SGM (yellow), SGM\_ROB (green), MPSV (blue), DF (green) and our implementation in quarter resolution (red).

## Appendix 2

Below we will give an example for the estimation of the maximum problem space size for real-time performance on a specific device.

We choose the *nVidia GTX 750 Ti* with a floating point performance of 1372 GFLOPS ( $1372 \cdot 10^9$  FLOPS). Further, we define our image ratio to be  $M : N = 4 : 3$ , the maximum disparity  $d_{\max} \approx \frac{w}{10}$  and in turn  $O = \frac{M}{10}$ . We want to achieve a frame rate of 60 frames per second and use our implementation of the algorithm of [HZW<sup>+</sup>10] which has a FLOPC of approximately  $1.53 \cdot 10^3$  as defined by Formula 6.6.

Using Formula 6.10 we calculate:

$$\begin{aligned}
 x &\leq \sqrt[3]{\frac{\text{FLOPS}}{\text{FLOPC} \cdot \text{FPS} \cdot M \cdot N \cdot O}} \\
 x &\leq \sqrt[3]{\frac{1372 \cdot 10^9}{1.53 \cdot 10^3 \cdot 60 \cdot 4 \cdot 3 \cdot \frac{4}{10}}} \\
 x &\leq \sqrt[3]{\frac{1372 \cdot 10^6}{1.53 \cdot 6 \cdot 48}} \\
 x &\leq 146,02
 \end{aligned} \tag{7.1}$$

Using Formula 6.7, this results in a maximum image size of 584 ( $w$ ) times 438 ( $h$ ) pixels with a maximum disparity of 58 ( $d_{\max}$ ) pixels for the *nVidia GTX 750 Ti*.



## Appendix 3

This appendix contains scripts that were created for the comparison of [HZW<sup>+</sup>10]’s implementation and our implementation.

For these scripts to work as intended, a directory called *humenberger* is required containing the directories *cones*, *teddy*, *tsukuba* and *venus*. Each of these directories must contain the result disparity map for [HZW<sup>+</sup>10]’s submission (*alg55.png*) from the Middlebury Benchmark *version 2* result table. Further, the Middlebury Benchmark SDK (MiddEval3) is needed. In the *trainingQ* directory of the SDK, four directories have to be present, *old\_cones*, *old\_teddy*, *old\_tsukuba* and *old\_venus*. These directories have to contain a calibration file (*calib.txt*), a left and right input image (*im0.png* and *im1.png*) and the corresponding ground truth image provided as PGM-file (*disp2.pgm*)<sup>1</sup>. The calibration file (*calib.txt*) must contain the image dimensions for the image set (width and height), the maximum disparity (ndist) and the disparity range for the *runvis* tool (vmin and vmax).

Listing 14 shows an example *calib.txt* file. The width and height is set to the corresponding image set’s dimensions. The parameters ndist, vmin and vmax are constant for all four image sets.

```
1  #width and height are image set dependent
2  width=384
3  height=288
4  #ndist, vmin and vmax are equal for all four image sets
5  ndisp=70
6  vmin=0
7  vmax=70
```

Listing 14: Example for *calib.txt*-file

---

<sup>1</sup>Note: For the Tsukuba image set provided by [vision.middlebury.edu/stereo/data/scenes2001/data/tsukuba/tsukuba.zip](http://vision.middlebury.edu/stereo/data/scenes2001/data/tsukuba/tsukuba.zip) the ground truth file *truedisp.row3.col3.pgm* provides the disparities for the frame-match *scene1.row3.col3.ppm* and *scene1.row3.col5.ppm*. Further, Portable Pixel Map (PPM)-files can be converted to PNG-files by regular image viewing applications, such as the GNOME Image Viewer.

Listing 15 shows the *Python3* script created to generate the colorized visualizations in Figure 6.9 and Figure 6.10. The script in Listing 15 uses the function *save\_pfm* shown in Listing 16.

```

1 dataset = ["cones", "teddy", "venus", "tsukuba"]
2 factor = {"cones":4, "teddy":4, "venus":8, "tsukuba":8}
3
4 def convertToPFM(inpath, outpath, scale=1):
5     img = scipy.misc.imread(inpath, "F")
6     save_pfm(outpath, img, scale)
7
8     '''
9     Path templates for input files and output files
10    '''
11    baseinput="humanberger/{}/alg55.png"
12    baseoutput="MiddEval3/trainingQ/old_{}/disp0HUMENBconvert.pfm"
13    gtinput="MiddEval3/trainingQ/old_{}/disp2.pgm"
14    gtoutput="MiddEval3/trainingQ/old_{}/disp0GT.pfm"
15
16    for x in dataset:
17        scale=factor[x]
18        convertToPFM(baseinput.format(x), baseoutput.format(x), scale)
19        convertToPFM(gtinput.format(x), gtoutput.format(x), scale)
20
21    subprocess.call(['./runviz', 'Q'], cwd='./MiddEval3')

```

Listing 15: Conversion script from old Middlebury formats to new Middlebury formats.

Listing 16 shows a function to save a numpy-array to a PFM-file.

```

1 def save_pfm(file, image, scale = 1):
2     closeit = False
3     if isinstance(file, str):
4         path = os.path.abspath(file)
5         closeit=True
6         try:
7             file = open(path, 'w+')
8         except:
9             file = open(path, 'w')
10    image = np.flipud(image)
11    image = image[:]/scale
12    scale = 1
13    color = None

```



```

14  if image.dtype.name != 'float32':
15      raise Exception('Image dtype must be float32.')
16  if len(image.shape) == 3 and \
17     image.shape[2] == 3:
18      # color image
19      color = True
20  elif len(image.shape) == 2 or \
21       len(image.shape) == 3 and \
22       image.shape[2] == 1:
23      # greyscale
24      color = False
25  else:
26      txt='Image must have HxWx3, HxWx1 or HxW dimensions.'
27      raise Exception(txt)
28  file.write('PF\n' if color else 'Pf\n')
29  file.write('%d %d\n' % (image.shape[1], image.shape[0]))
30  endian = image.dtype.byteorder
31  if endian == '<' or \
32     endian == '=' and \
33     sys.byteorder == 'little':
34      scale = -scale
35  file.write('%f\n' % scale)
36  image.tofile(file)
37  if closeit:
38      file.close()

```

Listing 16: *Python3* function to save an numpy array to a pfm-file.

# List of Figures

2.1	Epipolar geometry [Sze10]. . . . .	4
2.2	Rectification [Sze10] . . . . .	5
2.3	Pinhole camera sketch by [FP11] . . . . .	6
2.4	Pinhole imaging model [BK08] . . . . .	6
2.5	Extended pinhole imaging model [BK08] . . . . .	8
2.6	Radial Distortion plot (Jean-Yves Bouguet in [Bou]) . . . . .	10
2.7	Barrel distortion [BK08] . . . . .	10
2.8	Tangential distortion (Sebastian Thrun in [BK08]) . . . . .	11
2.9	Tangential Distortion plot (Jean-Yves Bouguet in [Bou]) . . . . .	12
2.10	Fronto-Parallel Assumption (FPA) Problem[EE14] . . . . .	18
2.11	Disparity Space Image . . . . .	20
2.12	Cost aggregation in DSI. . . . .	21
2.13	Minimum search in DSI. . . . .	24
2.14	<i>Middlebury Stereo Benchmark</i> data set: "2001 dataset". . . . .	30
2.15	<i>Middlebury Stereo Benchmark</i> data set: "2003 dataset" . . . . .	31
2.16	<i>Middlebury Stereo Benchmark</i> data set: "2005 dataset" . . . . .	31
2.17	<i>Middlebury Stereo Benchmark</i> data set: "2006 dataset" . . . . .	33
2.18	<i>Middlebury Stereo Benchmark</i> training data set: "2014 dataset" . . . . .	34
2.19	<i>Middlebury Stereo Benchmark</i> test data set: "2014 dataset" . . . . .	34
2.20	Reinterpreted figure from [MGMG11]: Multiple OpenCL devices are available to one host. . . . .	35
2.21	Reinterpreted figure from [MGMG11]. OpenCL device architecture. . . . .	37
4.1	Reinterpreted figure from [HZW <sup>+</sup> 10] Figure (3): Sparse census mask. . . . .	45
4.2	Example for sparse census transform. . . . .	47
5.1	Flowdiagram: optional rectification. . . . .	53
5.2	Flowdiagram: Overview of Stereo Matching Process . . . . .	54
5.3	Flowdiagram: Iterative cost aggregation. . . . .	54
5.4	Visualization of cost aggregation. . . . .	55
5.5	Search pattern for disparities with minimum costs in the DSI cost cube in the right-to-left (a,b) and left-to-right (c,d) setting. . . . .	56
5.6	Basic idea of the consistency check . . . . .	57

5.7	Visualization of cost aggregation performed by <i>CostXCubeKernel</i> and <i>CostY-CubeKernel</i> . . . . .	68
6.1	Plot of runtime measurements for all available devices. . . . .	85
6.2	Runtime prediction based on avg. FLOPC value with median lines. . . . .	87
6.3	FLOPC of Intel i7, nVidia Quadro, GTX 750 Ti and their average. . . . .	88
6.4	Setup time per image set and device. . . . .	91
6.5	Total matching and setup time. . . . .	92
6.6	Comparison of different algorithms . . . . .	93
6.7	Comparison of different algorithms in FLOPC. . . . .	94
6.8	Results of [HZW <sup>+</sup> 10], from <i>version 2</i> of the Middlebury Benchmark. . . . .	95
6.9	Colorized results of [HZW <sup>+</sup> 10] . . . . .	96
6.10	Colorized ground truth for Middlebury 2003 data set. . . . .	96
6.11	Results of our implementation for the image sets of the Middlebury Benchmark <i>version 2</i> . . . . .	97
6.12	Visualization of bad pixel percentage of the four image sets. Dense results from [HZW <sup>+</sup> 10]’s implementation and sparse results from our implementation. . . . .	98
6.13	Comparison of invalid rates of different algorithms for Q resolution. . . . .	101
6.14	Comparison of invalid rates of different algorithms for H resolution. . . . .	102
6.15	Comparison of invalid rates of different algorithms for F resolution. . . . .	102
6.16	Comparison of different algorithms’ average disparity error in quarter resolution. . . . .	103
6.17	Comparison of different algorithms’ average disparity error in half resolution. . . . .	104
6.18	Comparison of average error rates of different algorithms for F resolution. . . . .	105
6.19	Average bad pixel percentage per resolution, algorithm, evaluation script and $\delta$ value. . . . .	106
6.20	Data point positions of the image sets Djembe, DjembeL and Newkuba, Adirondack, ArtL and Jadeplant for the device <i>Intel IRIS 6100</i> . . . . .	108
1	Comparison of different algorithms in Bad-Pixel ( $\delta = 0.5$ ) metric for half (H) resolution. . . . .	113
2	Comparison of different algorithms in Bad-Pixel ( $\delta = 1.0$ ) metric. . . . .	114
3	Comparison of different algorithms in Bad-Pixel ( $\delta = 2.0$ ) metric. . . . .	115
4	Comparison of different algorithms in Bad-Pixel ( $\delta = 4.0$ ) metric. . . . .	116
5	Comparison of different algorithms in Bad-Pixel ( $\delta = 0.5$ ) metric for H resolution. . . . .	117
6	Comparison of different algorithms in Bad-Pixel ( $\delta = 1.0$ ) metric. . . . .	118
7	Comparison of different algorithms in Bad-Pixel ( $\delta = 2.0$ ) metric. . . . .	119
8	Comparison of different algorithms in Bad-Pixel ( $\delta = 4.0$ ) metric. . . . .	120
9	Comparison of different algorithms in Bad-Pixel ( $\delta = 0.5$ ) metric for Q resolution. . . . .	121
10	Comparison of different algorithms in Bad-Pixel ( $\delta = 1.0$ ) metric. . . . .	122
11	Comparison of different algorithms in Bad-Pixel ( $\delta = 2.0$ ) metric. . . . .	123
12	Comparison of different algorithms in Bad-Pixel ( $\delta = 4.0$ ) metric. . . . .	124

## List of Tables

6.1	Algorithm Overview. . . . .	83
6.2	Overview of different devices mentioned in this chapter. . . . .	89
6.3	Algorithm - Runtime Overview. . . . .	92

# List of Listings

1	RectificationKernel . . . . .	59
2	Find biggest possible work group size. . . . .	60
3	CensusKernel . . . . .	63
4	DiffCubeKernel . . . . .	66
5	CostXCubeKernel . . . . .	69
6	CostYCubeKernel . . . . .	71
7	MinimumR2LKernel . . . . .	72
8	MinimumL2RKernel . . . . .	73
9	CostCacheR2LKernel . . . . .	75
10	CostCacheL2RKernel . . . . .	77
11	ParabolicFittingKernel . . . . .	78
12	ConsistencyKernel . . . . .	79
13	Middlebury Evaluation - pseudocode . . . . .	100
14	Example for calib.txt-file . . . . .	127
15	Conversion script from old Middlebury formats to new Middlebury formats.	128
16	<i>Python3</i> function to save an numpy array to a pfm-file. . . . .	129



# Bibliography

- [Arn83] R. D. Arnold. *Automated Stereo Perception*. PhD thesis, Stanford University, California, Dept. of Computer Science, 1983.
- [Bar89] S. T. Barnard. Stochastic stereo matching over scale. *International Journal of Computer Vision*, 3(1):17–32, 1989.
- [BCPG08] M. Bleyer, S. Chambon, U. Poppe, and M. Gelautz. Evaluation of different methods for using colour information in global stereo matching approaches. In *Congress of the International Society for Photogrammetry and Remote Sensing*, pages 415–420. ISPRS, July 2008.
- [Bel96] P. N. Belhumeur. A bayesian approach to binocular stereopsis. *International Journal of Computer Vision*, 19(3):237–260, 1996.
- [BG05] M. Bleyer and M. Gelautz. A layered stereo matching algorithm using image segmentation and global visibility constraints. *ISPRS Journal of Photogrammetry and Remote Sensing*, 59(3):128–150, 2005.
- [BGM14] T. Botterill, R. Green, and S. Mills. A decision-theoretic formulation for sparse stereo correspondence problems. In *International Conference on 3D Vision*, pages 224–231. IEEE, December 2014.
- [BI99] A. F. Bobick and S. S. Intille. Large occlusion stereo. *International Journal of Computer Vision*, 33(3):181–200, 1999.
- [BK04] Y. Boykov and V. Kolmogorov. An experimental comparison of min-cut/max- flow algorithms for energy minimization in vision. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(9):1124–1137, September 2004.
- [BK08] G. Bradski and A. Kaehler. *Learning OpenCV: Computer vision with the OpenCV library*. O’Reilly Media, Inc., 2008. ISBN: 978-0596516130.
- [BM17] K. Bae and B. Moon. An accurate and cost-effective stereo matching algorithm and processor for real-time embedded multimedia systems. *Multimedia Tools and Applications*, 76(17):17907–17922, 2017.

- [BN98] D. N. Bhat and S. K. Nayar. Ordinal measures for image correspondence. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(4):415–423, 1998.
- [BNT07] S. T. Birchfield, B. Natarajan, and C. Tomasi. Correspondence as energy-based segmentation. *Image and Vision Computing*, 25(8):1329–1340, 2007.
- [Bou] J. Y. Bouguet. Camera calibration toolbox for matlab. [http://www.vision.caltech.edu/bouguetj/calib\\_doc/index.html](http://www.vision.caltech.edu/bouguetj/calib_doc/index.html) accessed on 2017-09-13.
- [Bro64] D. C. Brown. An advanced reduction and calibration for photogrammetric cameras. Technical report, Instrument corp. of Florida Melbourne, 1964. Accession Number: AD0431886.
- [Bro66] D. C. Brown. Decentering distortion of lenses. *Photogrammetric Engineering*, 32(3):444–462, 1966.
- [Bro71] D. C. Brown. Close-range camera calibration. *Photogrammetric Engineering*, 37(8):855–866, 1971.
- [BRR11] M. Bleyer, C. Rhemann, and C. Rother. Patchmatch stereo-stereo matching with slanted support windows. In *Proceedings of the British Machine Vision Conference*, pages 1–11. BMVA, September 2011.
- [BT99] S. Birchfield and C. Tomasi. Depth discontinuities by pixel-to-pixel stereo. *International Journal of Computer Vision*, 35(3):269–293, 1999.
- [BVZ98] Y. Boykov, O. Veksler, and R. Zabih. A variable window approach to early vision. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(12):1283–1294, 1998.
- [BVZ01] Y. Boykov, O. Veksler, and R. Zabih. Fast approximate energy minimization via graph cuts. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23(11):1222–1239, 2001.
- [CAD<sup>+</sup>12] T. S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. P. Singh. From OpenCL to high-performance hardware on FPGAs. In *International Conference on Field Programmable Logic and Applications*, pages 531–534. IEEE, August 2012.
- [CHRM96] I. J. Cox, S. L. Hingorani, S. B. Rao, and B. M. Maggs. A maximum likelihood stereo algorithm. *Computer Vision and Image Understanding*, 63(3):542–567, 1996.



- [CLT<sup>+</sup>07] N. Chang, T. Lin, T. Tsai, Y. Tseng, and T. Chang. Real-time DSP implementation on local stereo matching. In *Conference on Multimedia and Expo*, pages 2090–2093. IEEE, July 2007.
- [Col96] R. T. Collins. A space-sweep approach to true multi-image matching. In *Conference on Computer Vision and Pattern Recognition*, pages 358–363. IEEE, June 1996.
- [Con18] A. E. Conrady. The five aberrations of lens-systems. *Monthly Notices of the Royal Astronomical Society*, 79(1):60–66, 1918.
- [Con19] A. E. Conrady. Decentred lens-systems. *Monthly Notices of the Royal Astronomical Society*, 79(5):384–390, 1919.
- [CZYS14] F. Cheng, H. Zhang, D. Yuan, and M. Sun. Stereo matching by using the global edge constraint. *Neurocomputing*, 131:217–226, 2014.
- [DRR03] J. Davis, R. Ramamoorthi, and S. Rusinkiewicz. Spacetime stereo: A unifying framework for depth from triangulation. In *Conference on Computer Vision and Pattern Recognition*, pages 359–390. IEEE, June 2003.
- [EE10] N. Einecke and J. Eggert. A two-stage correlation method for stereoscopic depth estimation. In *Conference on Digital Image Computing: Techniques and Applications*, pages 227–234. IEEE, December 2010.
- [EE14] N. Einecke and J. Eggert. Block-matching stereo with relaxed fronto-parallel assumption. In *Intelligent Vehicles Symposium*, pages 700–705. IEEE, June 2014.
- [Fau93] O. Faugeras. *Three-dimensional computer vision: a geometric viewpoint*. MIT press, 1993. ISBN: 978-0262061582.
- [FP11] D. Forsyth and J. Ponce. *Computer Vision: a modern approach*. Upper Saddle River, NJ; London: Prentice Hall, 2011. ISBN: 978-8131709368.
- [FRT97] A. Fusiello, V. Roberto, and E. Trucco. Efficient stereo with multiple windowing. In *Conference on Computer Vision and Pattern Recognition*, pages 858–863. IEEE, June 1997.
- [Gat55] J. W. Gates. The measurement of comatic aberrations by interferometry. *Proceedings of the Physical Society. Section B*, 68(12):1065, 1955.
- [GB06] M. Gerrits and P. Bekaert. Local stereo matching with segmentation-based outlier rejection. In *The 3rd Canadian Conference on Computer and Robot Vision*, pages 66–66. IEEE, June 2006.
- [GFGC17] T. Gong, T. Fan, J. Guo, and Z. Cai. Gpu-based parallel optimization of immune convolutional neural network and embedded system. *Engineering Applications of Artificial Intelligence*, 62:384–395, 2017.

- [GLU12] A. Geiger, P. Lenz, and R. Urtasun. Are we ready for autonomous driving? the kitti vision benchmark suite. In *Conference on Computer Vision and Pattern Recognition*, pages 3354–3361. IEEE, June 2012.
- [Gri85] W. E. L. Grimson. Computational experiments with a feature based stereo algorithm. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, (1):17–34, 1985.
- [GSC<sup>+</sup>07] M. Goesele, N. Snavely, B. Curless, H. Hoppe, and S. M. Seitz. Multi-view stereo for community photo collections. In *International Conference on Computer Vision*, pages 1–8. IEEE, October 2007.
- [GYWG07] M. Gong, R. Yang, L. Wang, and M. Gong. A performance study on different cost aggregation approaches used in real-time stereo matching. *International Journal of Computer Vision*, 75(2):283–296, 2007.
- [Han74] M. J. Hannah. *Computer matching of areas in stereo images*. PhD thesis, Stanford University, California, Dept. of Computer Science, 1974.
- [HBG10] A. Hosni, M. Bleyer, and M. Gelautz. Near real-time stereo with adaptive support weight approaches. In *International Symposium 3D Data Processing, Visualization and Transmission*, pages 1–8. IEEE, January 2010.
- [HBGR09] A. Hosni, M. Bleyer, M. Gelautz, and C. Rhemann. Local stereo matching using geodesic support weights. In *Conference on Image Processing*, pages 2093–2096. IEEE, November 2009.
- [HIG02] H. Hirschmüller, P. R. Innocent, and J. Garibaldi. Real-time correlation-based stereo vision with reduced border errors. *International Journal of Computer Vision*, 47(1):229–246, April 2002.
- [Hir05] H. Hirschmüller. Accurate and efficient stereo processing by semi-global matching and mutual information. In *Conference on Computer Vision and Pattern Recognition*, pages 807–814. IEEE, June 2005.
- [Hir08] H. Hirschmüller. Stereo processing by semiglobal matching and mutual information. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 30(2):328–341, 2008.
- [HMJP92] Y. C. Hsieh, D. M. McKeown Jr, and F. P. Perlant. Performance evaluation of scene registration and stereo matching for artographic feature extraction. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14(2):214–238, 1992.
- [HRBG11] A. Hosni, C. Rhemann, M. Bleyer, and M. Gelautz. Temporally consistent disparity and optical flow via efficient spatio-temporal filtering. In *Pacific-Rim Symposium on Image and Video Technology*, pages 165–177. Springer, November 2011.

- [HS07] H. Hirschmüller and D. Scharstein. Evaluation of cost functions for stereo matching. In *Conference on Computer Vision and Pattern Recognition*, pages 1–8. IEEE, June 2007.
- [HS09] H. Hirschmüller and D. Scharstein. Evaluation of stereo matching costs on images with radiometric differences. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31(9):1582–1599, September 2009.
- [HZW<sup>+</sup>10] M. Humenberger, C. Zinner, M. Weber, W. Kubinger, and M. Vincze. A fast stereo matching algorithm suitable for embedded real-time systems. *Computer Vision and Image Understanding*, 114(11):1180–1202, 2010.
- [JGM14] J. Joglekar, S. S. Gedam, and B. K. Mohan. Image matching using sift features and relaxation labeling technique - a constraint initializing method for dense stereo matching. *IEEE Transactions on Geoscience and Remote Sensing*, 52(9):5643–5652, September 2014.
- [JS92] J. A. Jensen and N. B. Svendsen. Calculation of pressure fields from arbitrarily shaped, apodized, and excited ultrasound transducers. *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control*, 39(2):262–267, 1992.
- [KNM<sup>+</sup>14] D. Kondermann, R. Nair, S. Meister, W. Mischler, B. Güssefeld, K. Honauer, S. Hofmann, C. Brenner, and B. Jähne. Stereo ground truth with error bars. In *Asian Conference on Computer Vision.*, pages 595–610. Springer, November 2014.
- [KO94] T. Kanade and M. Okutomi. A stereo matching algorithm with an adaptive window: Theory and experiment. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(9):920–932, 1994.
- [KPP13] J. Kowalczyk, E. T. Psota, and L. C. Perez. 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*, 23(1):94–104, January 2013.
- [KSC01] S. B. Kang, R. Szeliski, and J. Chai. Handling occlusions in dense multi-view stereo. In *Conference on Computer Vision and Pattern Recognition*, pages 103–110. IEEE, December 2001.
- [KSK06] A. Klaus, M. Sormann, and K. Karner. Segment-based stereo matching using belief propagation and a self-adapting dissimilarity measure. In *International Conference on Pattern Recognition*, pages 15–18. IEEE, September 2006.
- [KZ01] V. Kolmogorov and R. Zabih. Computing visual correspondence with occlusions using graph cuts. In *International Conference on Computer Vision*, pages 508–515. IEEE, July 2001.

- [LQ02] M. Lhuillier and L. Quan. Match propagation for image-based modeling and rendering. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(8):1140–1146, 2002.
- [LQ05] M. Lhuillier and L. Quan. A quasi-dense approach to surface reconstruction from uncalibrated images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27(3):418–433, 2005.
- [LT98] R. A. Lane and N. A. Thacker. Tutorial: Overview of stereo matching research. *Imaging Science and Biomedical Engineering Division, Medical School, University of Manchester*, 1998.
- [MGMG11] A. Munshi, B. Gaster, T. G. Mattson, and D. Ginsburg. *OpenCL programming guide*. Pearson Education, 2011. ISBN: 978-0321749642.
- [MKS89] L. Matthies, T. Kanade, and R. Szeliski. Kalman filter-based algorithms for estimating depth from image sequences. *International Journal of Computer Vision*, 3(3):209–238, 1989.
- [MMP87] J. Marroquin, S. Mitter, and T. Poggio. Probabilistic solution of ill-posed problems in computational vision. *Journal of the american statistical association*, 82(397):76–89, 1987.
- [OBDA11] M. Owaida, N. Bellas, K. Daloukas, and C. D. Antonopoulos. Synthesis of platform architectures from opencl programs. In *International Symposium on Field-Programmable Custom Computing Machines*, pages 186–193. IEEE, May 2011.
- [OK92] M. Okutomi and T. Kanade. A locally adaptive window for signal matching. *International Journal of Computer Vision*, 7(2):143–162, 1992.
- [PBG13] E. Piatkowska, A. N. Belbachir, and M. Gelautz. Asynchronous stereo vision for event-driven dynamic stereo sensor using an adaptive cooperative approach. In *International Conference on Computer Vision Workshops*, pages 45–50. IEEE, March 2013.
- [PKBG17] E. Piatkowska, J. Kogler, N. Belbachir, and M. Gelautz. Improved cooperative stereo matching for dynamic vision sensors with ground truth evaluation. In *Conference on Computer Vision and Pattern Recognition Workshops*, pages 370–377. IEEE, July 2017.
- [Pra85] K. Prazdny. Detection of binocular disparities. *Biological cybernetics*, 52(2):93–99, 1985.
- [SBGB09] F. Seitner, M. Bleyer, M. Gelautz, and R. Beuschel. Development of a high-level simulation approach and its application to multicore video decoding. *IEEE Transactions on Circuits and Systems for Video Technology*, 19(11):1667–1679, 2009.

- [SBSG08] F. Seitner, M. Bleyer, R. M. Schreier, and M. Gelautz. Evaluation of data-parallel splitting approaches for h.264 decoding. In *International Conference on Advances in Mobile Computing and Multimedia*, pages 40–49. ACM, November 2008.
- [Sei57] L. Seidel. *Ueber die Theorie der Fehler, mit welchen die durch optische Instrumente gesehenen Bilder behaftet sind, und über die mathematischen Bedingungen ihrer Aufhebung*. Abhandlungen der Naturwissenschaftlich-Technischen Commission bei der Königl. Bayerischen Akademie der Wissenschaften in München. Cotta, 1857.
- [SHK<sup>+</sup>14] D. Scharstein, H. Hirschmüller, Y. Kitajima, G. Krathwohl, N. Nešić, X. Wang, and P. Westling. High-resolution stereo datasets with subpixel-accurate ground truth. In *German Conference on Pattern Recognition*, pages 31–42. Springer, September 2014.
- [SLKS05] J. Sun, Y. Li, S. B. Kang, and H. Shum. Symmetric stereo matching for occlusion handling. In *Conference on Computer Vision and Pattern Recognition*, pages 399–406. IEEE, June 2005.
- [SP07] D. Scharstein and C. Pal. Learning conditional random fields for stereo. In *Conference on Computer Vision and Pattern Recognition*, pages 1–8. IEEE, June 2007.
- [SS98] D. Scharstein and R. Szeliski. Stereo matching with nonlinear diffusion. *International Journal of Computer Vision*, 28(2):155–174, 1998.
- [SS03] D. Scharstein and R. Szeliski. High-accuracy stereo depth maps using structured light. In *Conference on Computer Vision and Pattern Recognition*, pages 195–202. IEEE, June 2003.
- [SSZ01] D. Scharstein, R. Szeliski, and R. Zabih. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. In *Workshop on Stereo and Multi-Baseline Vision*, pages 131–140. IEEE, December 2001.
- [STH80] C. C. Slama, C. Theurer, and S. W. Henriksen. *Manual of Photogrammetry*. American Society of Photogrammetry, 1980. ISBN: 978-0937294017.
- [Str12] J. Strong. *Concepts of classical optics*. Courier Corporation, 2012. ISBN: 978-0486432625.
- [SZ99] R. Szeliski and R. Zabih. An experimental comparison of stereo algorithms. In *International Workshop on Vision Algorithms*, pages 1–19. Springer, September 1999.
- [Sze10] R. Szeliski. *Computer Vision: Algorithms and Applications*. Springer Science & Business Media, 2010. ISBN: 978-1848829343.

- [SZS03] J. Sun, N. Zheng, and H. Shum. Stereo matching using belief propagation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 25(7):787–800, 2003.
- [TLA16] B. Tippetts, D. J. Lee, K. Lillywhite, and J. Archibald. Review of stereo vision algorithms and their suitability for resource-limited systems. *Journal of Real-Time Image Processing*, 11(1):5–25, 2016.
- [TMDSA08] F. Tombari, S. Mattoccia, L. Di Stefano, and E. Addimanda. Classification and evaluation of cost aggregation methods for stereo correspondence. In *Conference on Computer Vision and Pattern Recognition*, pages 1–8. IEEE, June 2008.
- [TS00] H. Tao and H. S. Sawhney. Global matching criterion and color segmentation based stereo. In *Workshop on Applications of Computer Vision*, pages 246–253. IEEE, December 2000.
- [TWZ08] Y. Taguchi, B. Wilburn, and C. L. Zitnick. Stereo reconstruction with mixed pixels using adaptive over-segmentation. In *Conference on Computer Vision and Pattern Recognition*, pages 1–8. IEEE, June 2008.
- [Vek01] O. Veksler. Stereo matching by compact windows via minimum ratio cycle. In *International Conference on Computer Vision*, pages 540–547. IEEE, July 2001.
- [Vek03] O. Veksler. Fast variable window for stereo correspondence using integral images. In *Conference on Computer Vision and Pattern Recognition*, pages 556–561. IEEE, June 2003.
- [Was57] F. E. Washer. The effect of prism on the location of the principal point. *Photogrammetric Engineering*, 28(3):520–532, 1957.
- [YK05] K. Yoon and I. Kweon. Locally adaptive support-weight approach for visual correspondence search. In *Conference on Computer Vision and Pattern Recognition*, pages 924–931. IEEE, June 2005.
- [YK06] K. Yoon and I. S. Kweon. Adaptive support-weight approach for correspondence search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 28(4):650–656, 2006.
- [YWY<sup>+</sup>06] Q. Yang, L. Wang, R. Yang, S. Wang, M. Liao, and D. Nister. Real-time global stereo matching using hierarchical belief propagation. In *British Machine Vision Conference*, pages 989–998. BMVC, September 2006.
- [ZCS03] L. Zhang, B. Curless, and S. M. Seitz. Spacetime stereo: Shape recovery for dynamic scenes. In *Conference on Computer Vision and Pattern Recognition*, pages 367–374. IEEE, June 2003.

- [ZFM<sup>+</sup>14] K. Zhang, Y. Fang, D. Min, L. Sun, S. Yang, S. Yan, and Q. Tian. Cross-scale cost aggregation for stereo matching. In *Conference on Computer Vision and Pattern Recognition*, pages 965–976. IEEE, June 2014.
- [Zha99] Z. Zhang. Flexible camera calibration by viewing a plane from unknown orientations. In *International Conference on Computer Vision*, pages 666–673. IEEE, September 1999.
- [Zha00] Z. Zhang. A flexible new technique for camera calibration. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(11):1330–1334, November 2000.
- [ZHHL06] L. Zhang, X. Huang, B. Huang, and P. Li. A pixel shape index coupled with spectral information for classification of high spatial resolution remotely sensed imagery. *IEEE Transactions on Geoscience and Remote Sensing*, 44(10):2950–2961, 2006.
- [ZK00] C. L. Zitnick and T. Kanade. A cooperative algorithm for stereo matching and occlusion detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(7):675–684, 2000.
- [ZK07] C. L. Zitnick and S. B. Kang. Stereo for image-based rendering using image over-segmentation. *International Journal of Computer Vision*, 75(1):49–65, 2007.
- [ZK15] S. Zagoruyko and N. Komodakis. Learning to compare image patches via convolutional neural networks. In *Computer Vision and Pattern Recognition*, pages 4353–4361. IEEE, June 2015.
- [ZL16] J. Zbontar and Y. LeCun. Stereo matching by training a convolutional neural network to compare image patches. *Journal of Machine Learning Research*, 17(1-32):2, 2016.
- [ZNPC13] J. Zhang, J. F. Nezan, M. Pelcat, and J. G. Cousin. Real-time GPU-based local stereo matching method. In *Conference on Design and Architectures for Signal and Image Processing*, pages 209–214. IEEE, October 2013.
- [ZS00] Z. Zhang and Y. Shan. A progressive scheme for stereo matching. In *European Workshop on 3D Structure from Multiple Images of Large-Scale Environments*, pages 68–85. Springer, March 2000.
- [ZW94] R. Zabih and J. Woodfill. Non-parametric local transforms for computing visual correspondence. In *European Conference on Computer Vision, ECCV ’94*, pages 151–158. Springer, Springer, June 1994.