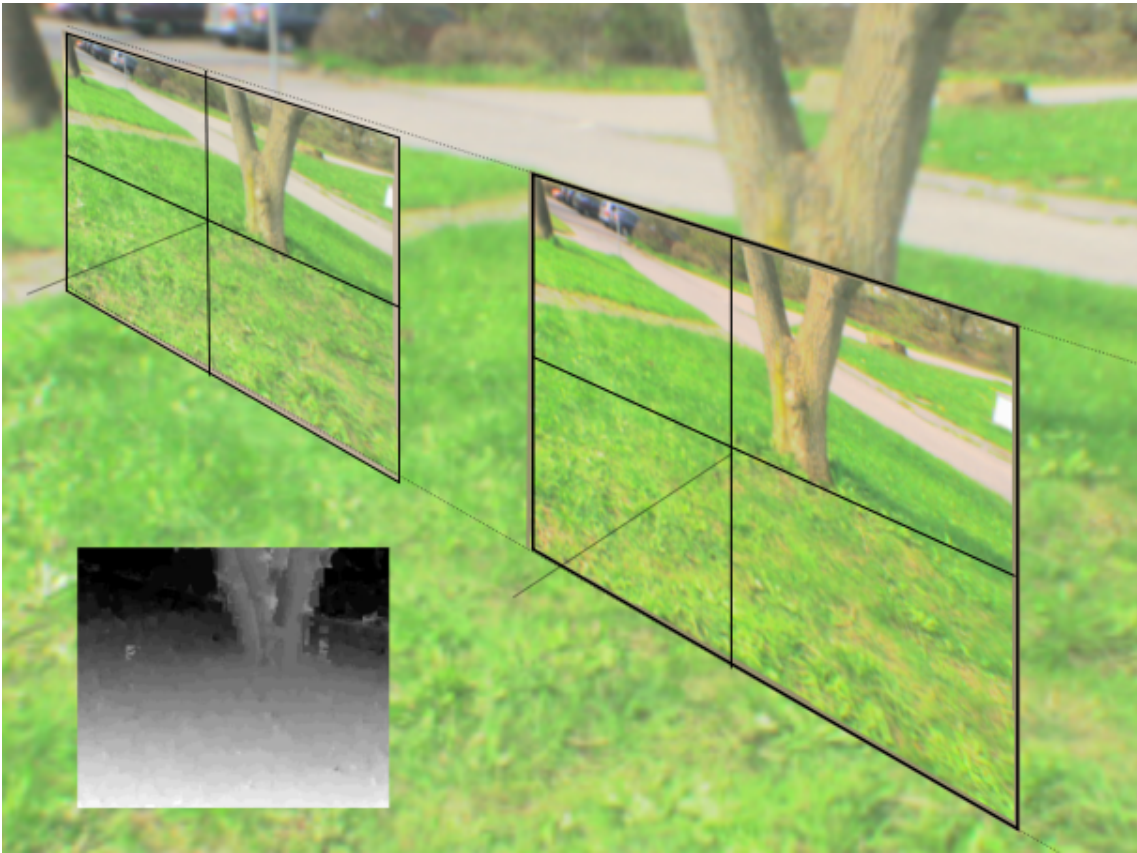


ROBOTICS RESEARCH LAB  
DEPARTMENT OF COMPUTER SCIENCES  
UNIVERSITY OF KAISERSLAUTERN

---

## Project Thesis

---



GPU Stereo Vision

Sebastian Prehn

---

December 6, 2007

---



# Project Thesis

## GPU Stereo Vision

Robotics Research Lab  
Department of Computer Sciences  
UNIVERSITY OF KAISERSLAUTERN

Sebastian Prehn

**Day of issue** : 23. April 2007  
**Day of release** : December 6, 2007

**Professor** : Prof. Dr. Karsten Berns  
**Tutor** : Dipl.-Inf. B. H. Schäfer



Hereby I declare that I have self-dependently composed the Project Thesis at hand. The sources and additives used have been marked in the text and are exhaustively given in the bibliography.

December 6, 2007 – Kaiserslautern

(Sebastian Prehn)



# Preface

I would like to thank everyone at the robotics research lab of AG Berns for providing such a friendly and comfortable working environment.

I thank Tim Braun for supporting me with the integration of Scons and low-level CUDA optimization.

Special thanks go to my tutor Bernd Helge Schäfer for the manifold advice and for helping me out with MCA and C/C++ specific problems.

Both Tim and Helge have guided and mayorly influenced the approaches taken in this work.





# Contents

<b>1</b>	<b>Introduction</b>	<b>7</b>
1.1	Motivation . . . . .	7
1.2	Objectives . . . . .	7
1.3	Structure . . . . .	8
<b>2</b>	<b>Stereo Vision</b>	<b>11</b>
2.1	Binocular Stereo Vision . . . . .	11
2.1.1	3D Reconstruction . . . . .	13
2.2	Stereovision Algorithms . . . . .	15
2.2.1	Dense vs. Sparse . . . . .	16
2.2.2	Global Optimization vs. Window-Based . . . . .	16
<b>3</b>	<b>Concept</b>	<b>19</b>
3.1	GPU Computation Technology . . . . .	19
3.2	Algorithm Selection . . . . .	19
<b>4</b>	<b>CUDA</b>	<b>21</b>
4.1	GPU . . . . .	22
4.1.1	Execution Model . . . . .	22
4.2	Memory . . . . .	24
<b>5</b>	<b>Implementation</b>	<b>27</b>
5.1	Getting started . . . . .	27
5.2	Window Based Approach . . . . .	28
5.2.1	Window Based Kernel . . . . .	29
5.3	Pyramid Based Approach . . . . .	33
5.3.1	Scale Down Kernel . . . . .	35
5.3.2	Pyramid Stereo Kernel . . . . .	37
5.3.2.1	Region of Interest . . . . .	41
5.3.2.2	Confidence . . . . .	41
5.3.3	Post Processing Kernel . . . . .	43
<b>6</b>	<b>Integration</b>	<b>45</b>
6.1	MCA Stereo Vision . . . . .	45

<b>7 Performance / Results</b>	<b>49</b>
7.1 Quality . . . . .	49
7.1.1 Window Based Stereo Kernel Results . . . . .	51
7.1.2 Pyramid Based Stereo Kernel Results . . . . .	54
7.1.3 Quality . . . . .	58
7.1.4 GPU Comparison . . . . .	59
7.2 Timings and Optimization Techniques . . . . .	61
<b>8 Conclusion</b>	<b>65</b>
<b>Bibliography</b>	<b>67</b>

# 1. Introduction

## 1.1 Motivation

The Robotics Research Lab at the University of Kaiserslautern investigates mobile service robots in various environments. In order to navigate through unknown terrain the on site detection of obstacles is a crucial feature. To analyse a scenery obstacles depth information is very valuable. Stereo vision is a powerful way to extract dense range information of a complete area of the environment [Schäfer 04]. On the other hand stereo reconstruction comprises computationally expensive steps which results in high CPU load. The inherent complexity leaves only one logical consequence. The computation has to be moved off the CPU and towards a second processing unit. One solution can be tailor-made stereo vision hardware (see [Kuhn 03]). However, in recent years Graphic Processing Units (GPU) on consumer-level graphics boards have become programmable and increasingly powerful. Furthermore the trend in performance gain is even better than with standard CPUs. Today's off-the-shelf GPUs provide extreme parallel processing capabilities at far less cost than custom made stereo vision hardware. Furthermore this hardware integrates easily with every reasonably modern standard PC. At the same time the PC and the graphics board communicate over a super fast bus beating any network solution by far.

For these reasons GPU implementations of stereo algorithms are a very attractive way to speed up the reconstruction and to unload the main processor freeing resources for further processing.

## 1.2 Objectives

Goal of this work is to development a fast stereo vision algorithm designed to run on a standard NVIDIA GPU of the 8800 and above series. The stereo algorithm needs to be integrated in the MCA framework<sup>1</sup>, which is commonly used for robotics applications at the Robotics Research Lab at the University of Kaiserslautern. One exemplary target application is the lab's research platform RAVON (see figure 1.1).

---

<sup>1</sup>MCA = Modular Controller Architecture (see [Scholl 02, Koch 07] and <http://www.mca2.org>)



Figure 1.1: Robust Autonomous Vehicle for Off-road Navigation (RAVON). Research platform for behavior based motion, localization, and navigation strategies in rough terrain. Equipped with two stereo vision heads.

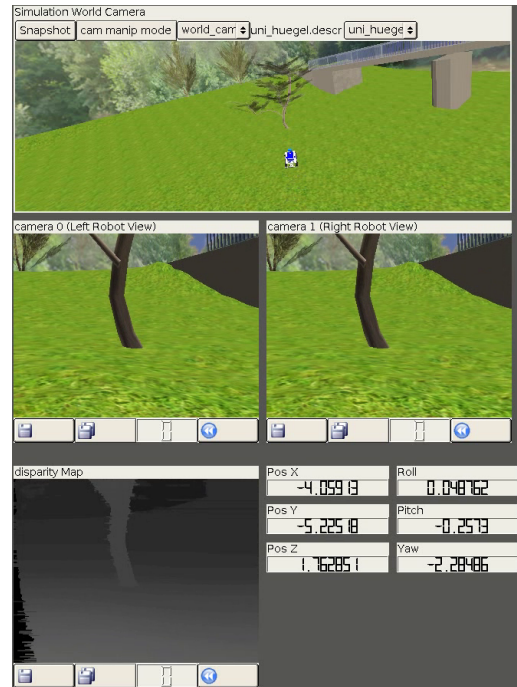


Figure 1.2: Stereo vision simulation: Stereo vision supported navigation on RAVON

Stereo vision has been a research topic for years and is quite well understood. ([Schnabl 03] and chapter 2). The first challenge is therefore to select an approach that works well in the field of outdoor robotics, at the mean time being suitable for a fast GPU implementation.

General purpose computation on the GPU requires a new thinking about algorithm design. Programming strategies that result in a fast CPU implementation do not necessarily produce fast GPU code. Especially control structures, misaligned memory access and any kind of serialization slow down GPU programs. In this work it is investigated how the latest GPU general purpose technology can be applied to the stereo vision problem.

Finally the algorithms need to be benchmarked and optimized for high framerates. Additionally simple and fast forms of post-processing and confidence<sup>2</sup> computation are presented as add-on to the core stereo algorithm.

### 1.3 Structure

This rest of this document is organized into the following chapters

- Chapter 2 explains general stereo vision principles.
- Chapter 3 outlines major design decisions.
- Chapter 4 describes the NVIDIA CUDA framework.

<sup>2</sup>reliability of reconstructed features

- Chapter 5 gives an insight on how the presented stereo vision algorithm evolved.
- Chapter 6 describes the integration of the new algorithm into the MCA framework.
- Chapter 7 illustrates the quality and performance results and concludes the work.
- Chapter 8 concludes this Project Thesis.



## 2. Stereo Vision

This chapter provides a brief overview on stereo vision.

### 2.1 Binocular Stereo Vision

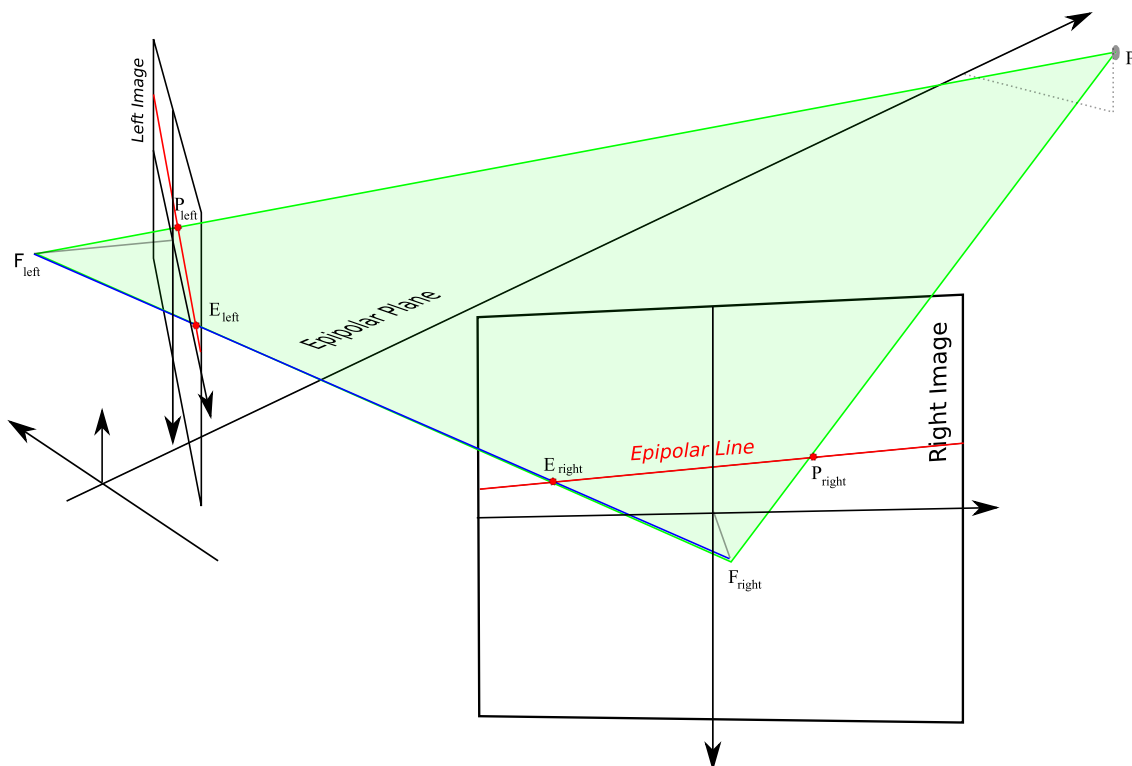


Figure 2.1: General stereo vision geometry (pin-hole camera model).  $E_{left}$ ,  $E_{right}$ : epipolar points;  $F_{left}$ ,  $F_{right}$ : focal points;  $P$ : point in 3D space;  $P_{left}$ ,  $P_{right}$ : projections on image planes

Humans are able to perceive depth information with their two eyes. A distant object is projected onto the retina of the left and right eye. The object's projections differ in their position. The human brain is able to come up with a depth judgment for that object by analyzing the two images.

Computer stereo vision copies this concept. The goal is to reconstruct a depth map from at least two 2D camera images showing a 3D scene from different observation points. The depth information can be inferred by matching a point in both images and looking at the displacement between the matched pair.

Figure 2.1 visualizes the general case of stereo vision geometry with two input images. A 3D scene is projected onto two 2D virtual image planes. The plane defined by point  $P$  and both focal points  $F_{left}$  and  $F_{right}$  is called *epipolar plane*.

The projected line from focal point  $F_{left}$  to  $P$  in the right image is called an *epipolar line*. The right image projection  $P_{right}$  of point  $P$  can always be found on this epipolar line. This is called the *epipolar constraint* and simplifies the correspondence problem. This constraint only holds for the perfectly rectified images of a pin-hole camera. In reality, raw camera images are usually distorted. The images have to be rectified prior to stereo vision processing.

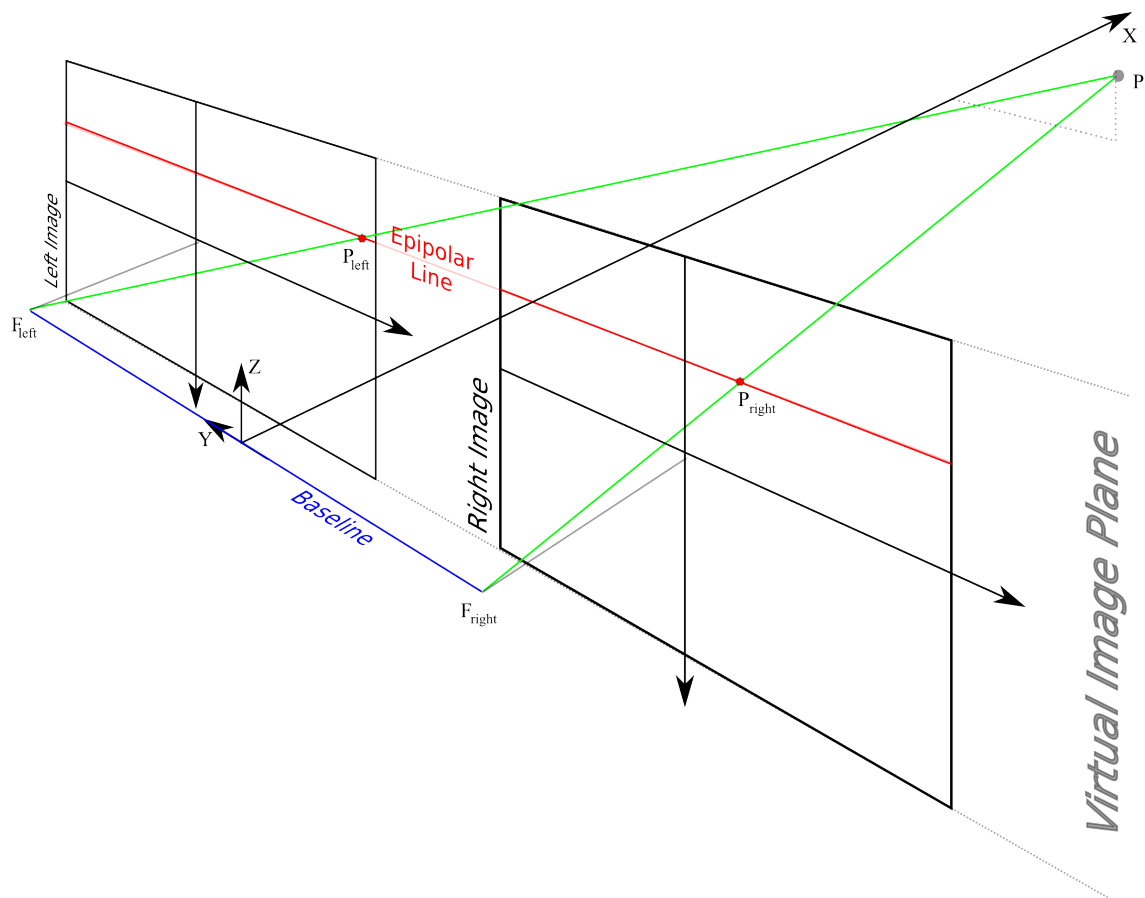


Figure 2.2: Binocular stereo vision setup

A binocular stereo vision system with a *canonical stereo geometry* consists of two identical, parallelly mounted cameras. The cameras are setup such that their virtual



image planes and both epipolar lines fall together. The epipolar lines are aligned parallel to the x-axis of the image planes.

A point in the observed scene will be projected to pixels on the very same row in the left and right image (see figure 2.2). Obviously no further calculation for finding the epipolar line is needed. The displacement between left and right projection is called *disparity*  $d$ .

### 2.1.1 3D Reconstruction

In order to reconstruct the 3D scene from disparity values the fixed distance between the cameras called *baseline*  $b$  and the focal length  $f$  of the cameras has to be known.

The stereo vision head coordinate system lies between both cameras on the baseline. Its orientation is defined according to common decisions in robotics as a right-hand coordinate system. The x axis pierces perpendicular through the image plane. (see figure 2.2).

In computer vision image origins usually lie at the top left corner of an image. Here, the image coordinate systems have their origins translated to the very middle of each picture, the closest point to the focal point (see figure 2.2). This is not a restriction. It only simplifies the following derivation and is a common assumption in physics for the pin-hole camera model.

The exact 3D coordinates  $x$ ,  $y$  and  $z$  for two matching pixel  $(x_l, y_l)$  and  $(x_r, y_r)$  can be calculated using triangulation as shown in figure 2.3.

$$\tan(\alpha_l) = \frac{x_l}{f} = \frac{\frac{b}{2} - y}{x} \quad (2.1)$$

$$\tan(\alpha_r) = \frac{-x_r}{f} = \frac{\frac{b}{2} + y}{x} \quad (2.2)$$

Solving 2.1 and 2.2 for  $y$  yields:

$$y = \frac{-x_l \cdot x}{f} + \frac{b}{2} \quad (2.3)$$

$$y = \frac{-x_r \cdot x}{f} - \frac{b}{2} \quad (2.4)$$

Equating 2.3 and 2.4 and solving for  $x$  results in:

$$x = \frac{b \cdot f}{x_l - x_r} \quad (2.5)$$

Solving for  $z$  is even simpler. Since both pixels lie on the same image row  $y_r = y_l$  holds.

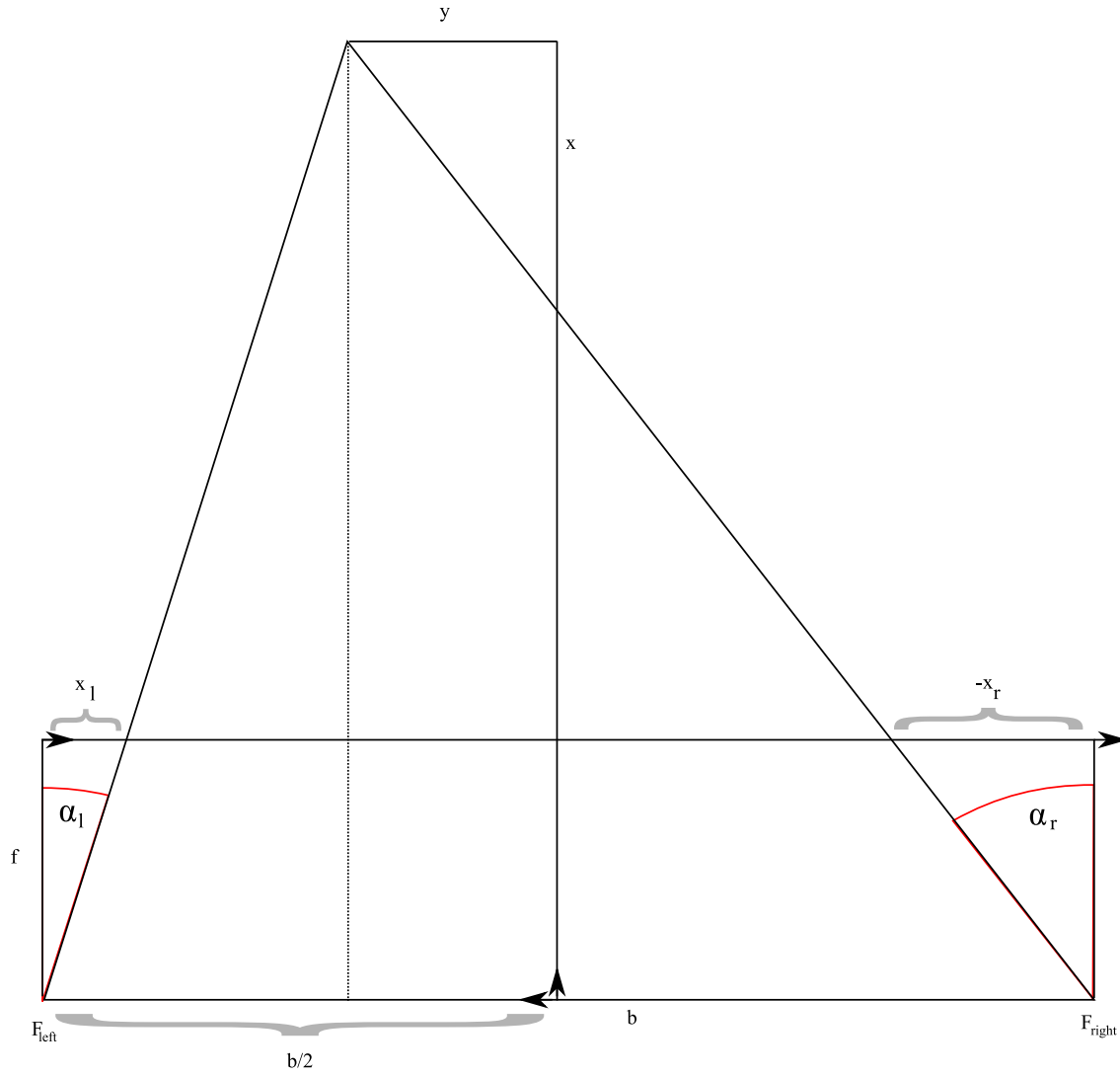


Figure 2.3: X-Y Plane Triangulation (top view on stereo vision head)

$$\tan(\beta) = \frac{y_l}{f} = \frac{z}{x} \quad (2.6)$$

$$z = \frac{x \cdot y_l}{f} \quad (2.7)$$

Substitution of equation 2.5 into 2.7 and 2.3 yields the following formulae for straight forward computation of  $y$  and  $z$ .

$$y = \frac{-\frac{b}{2} \cdot (x_l + x_r)}{x_l - x_r} \quad (2.8)$$

$$z = \frac{y_l \cdot b}{x_l - x_r} \quad (2.9)$$

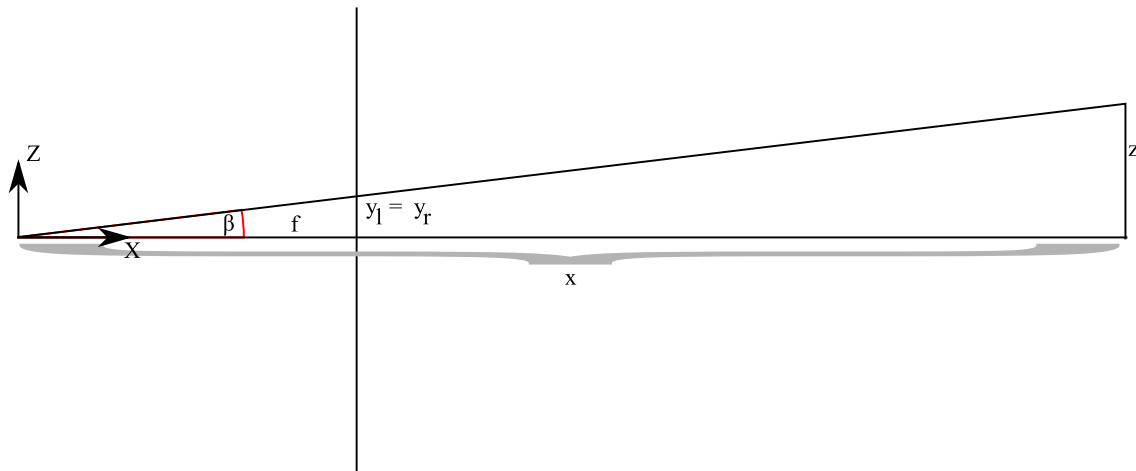


Figure 2.4: X-Z Plane Triangulation (side view on stereo vision head)

The disparity value  $d$  calculated by a stereo vision algorithm already equals  $x_l - x_r$ . Note that the translation of the image origin along the  $x$  axis has no effect on the disparity. Let the left image be the reference image. Then  $x_l$  and  $d$  are known.  $x_r$  can easily be calculated as  $x_r = x_l - d$ .

## 2.2 Stereovision Algorithms

The challenge for a stereo vision algorithm is to solve the correspondence problem, i.e. finding the right match in the left and right image for a given point in the real world. The common approach is to declare one input image as the reference image. For a pixel in the reference image the corresponding match is then searched for in the second input image.

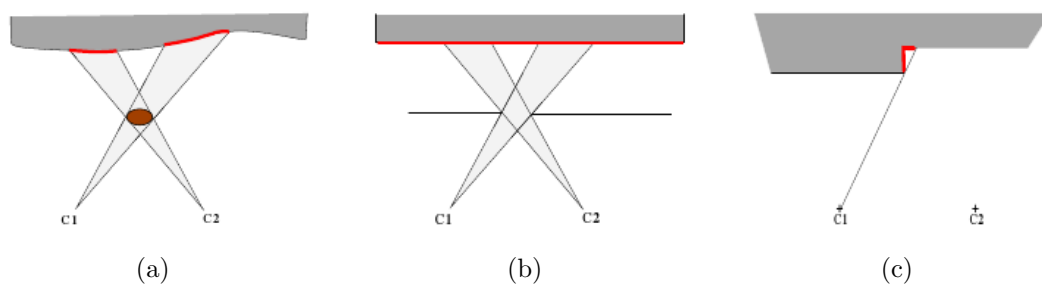


Figure 2.5: Scene configurations with half-occluded regions (red highlighted): (a) occlusions due to thin object at the foreground-scene discontinuity, (b) occlusion due to a small hole at the foreground-scene discontinuity, (c) occlusion due to surface variation-surface discontinuity. (source [Kostkova 02])

However, finding the correct match is ambiguous. Due to occlusion some points may not have a match in both projections (see figure 2.5). Furthermore repetitive textures may produce multiple matches. Detecting occlusion and finding the right match determines the quality of a stereo vision algorithm.

### 2.2.1 Dense vs. Sparse

Stereo vision strategies can be dense or sparse. Sparse stereo vision concentrates on selected feature points only. A feature point can be a point of special interest. Alternatively a feature point may be a point with high likelihood for a good match. This approach can potentially result in a fast stereo vision algorithm due to the reduced amount of analyzed points. Dense stereo vision algorithms find matches for all points in the reference image. (In this work a dense approach was chosen since distance information for the whole image is required for further processing steps.) In a complete run of a dense stereo vision algorithm a disparity value is calculated for each pixel in the reference image. Typically this result is visualized as a gray scale image, also known as disparity map. In this image each pixel's brightness corresponds to a disparity level. High disparity values result in lighter pixels.

### 2.2.2 Global Optimization vs. Window-Based

Recently high quality results have been achieved in CPU stereo vision applying global optimization techniques to the stereo vision problem (see [Scharstein 02]). Such algorithms rank top at the Middlebury evaluation page<sup>1</sup>. However, these approaches tend to be slow and thus are not suitable for our near-real-time task. Yang and Pollefeys [Yang 05] claim that only correlation-based stereo algorithms are able to provide a dense depth map in real time on standard computer hardware. However, a fast optimizing stereo vision algorithm for the GPU could be an interesting future research topic.

Window-based stereo algorithms take two parameters: window width and window height. The window describes a rectangle areas around the reference pixel. Around a match candidate in the second image a rectangle of equal size is compared to the reference window. It is assumed that the areas are most similar for the correct match.

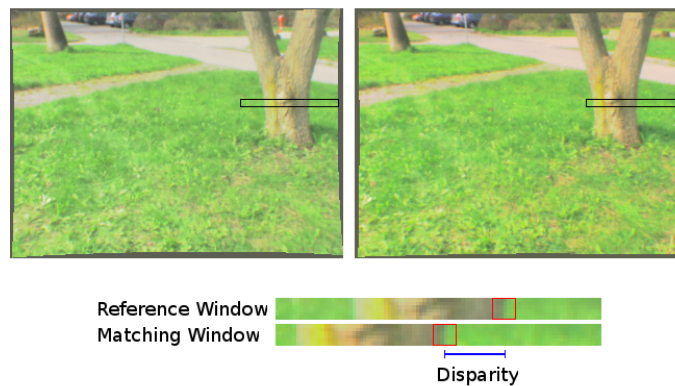


Figure 2.6: Window based stereo vision

For each disparity step a window based strategy therefore shifts the window along the epipolar line in the second image. The region in both images is then compared

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

by a similarity function. The disparity step with the highest similarity provides most likely the correct match. Common similarity functions are cross correlation, sum of squared differences (SSD) or sum of absolute differences (SAD).

The performance and quality of a window based algorithm depends on the window size. Large support regions provide a higher certainty for a correct match even for low textured input images. Small window sizes speed up the computation and produce finer grained results. Window based stereo vision requires good textures in the input images. Luckily in the application field of outdoor robotics such textures prevail.



## 3. Concept

### 3.1 GPU Computation Technology

A CPU is optimized for general purpose computation. In contrast, GPUs are specially designed to perform a set of rather simple operations on a large amount of data. A GPU obtains speed via parallelization. This is achieved by multiple parallel rendering pipelines. From a general purpose computing point of view a GPU is a dedicated parallel processor.

Traditionally the rendering pipelines are programmable with vertex- and fragment programs via a shader language. In this fashion stereo vision systems already have been implemented [Yang 05]. The Robotics Research Lab has used the CG framework on NVIDIA graphics hardware for various general purpose computation [Zolynski 07, Seidler 08] on the GPU.

In all these approaches the application engineers have to reformulate their general purpose problem in terms of these shader languages. Recently, NVIDIA has come up with a new approach to general purpose computation on the GPU: the CUDA<sup>1</sup> framework. In CUDA there is no need to reformulate the application problem (see chapter 4). Therefore this new technology will be used in this work to support the stereo vision implementation.

### 3.2 Algorithm Selection

As explained in chapter 2 a window-based approach is most promising for near real-time implementations. Since the technology is brand new, two algorithms are to be implemented. The first and simple window-based stereo vision algorithm serves as a test and study object. In the second implementation quality improvements are targeted. In order to achieve better results a pyramid approach and simple post-processing are to be incorporated to the implementation.

---

<sup>1</sup>CUDA = Compute Unified Device Architecture





## 4. CUDA

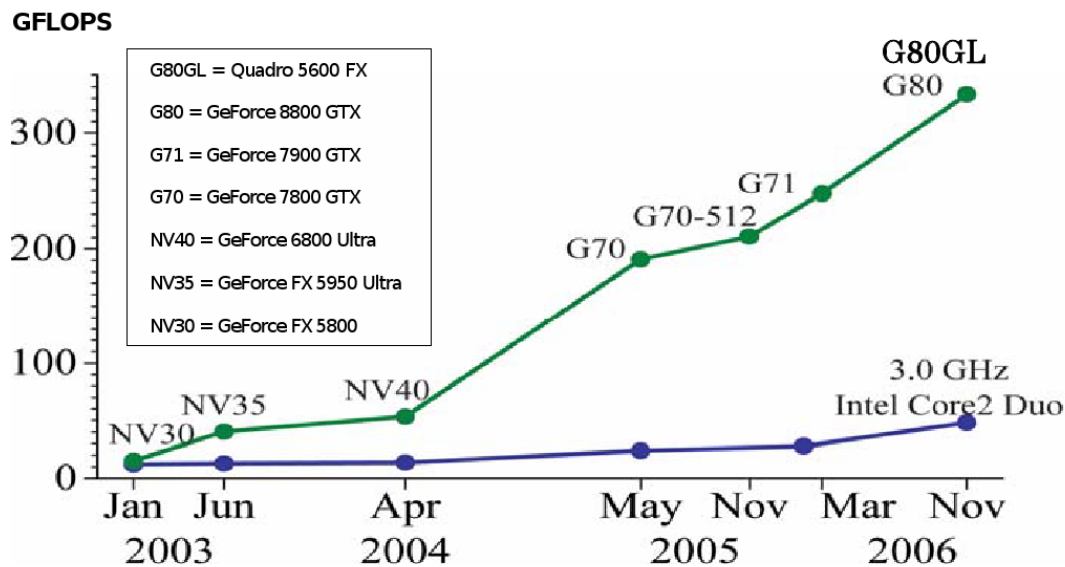


Figure 4.1: Floating-Point Operations per Second for the CPU and GPU [NVIDIA 07]

CUDA stands for *Compute Unified Device Architecture*. It is a new hardware and software architecture for computation on the GPU. CUDA is available for NVIDIA's GeForce 8 Series, Quadro FX 5600/4600, and Tesla solutions [NVIDIA 07].

Programmable graphics hardware has been around for quite a while. Traditionally this hardware had to be programmed through shader languages and graphics APIs like CG<sup>1</sup>. This imposed a high learning curve on developers who were trying to utilize GPU computation devices for non-graphic tasks. Additionally values could be read

<sup>1</sup>C for graphics [http://developer.nvidia.com/page/cg\\_main.html](http://developer.nvidia.com/page/cg_main.html)

from but not written to arbitrary memory locations. In CUDA these problems are overcome.

CUDA provides a standard C programming interface. Inter-thread communication and general DRAM read and write access is possible with CUDA hardware.

This chapter will briefly introduce the main CUDA capabilities and constraints. For more detailed information on the CUDA framework please refer to the *CUDA Programming Guide* [NVIDIA 07].

## 4.1 GPU

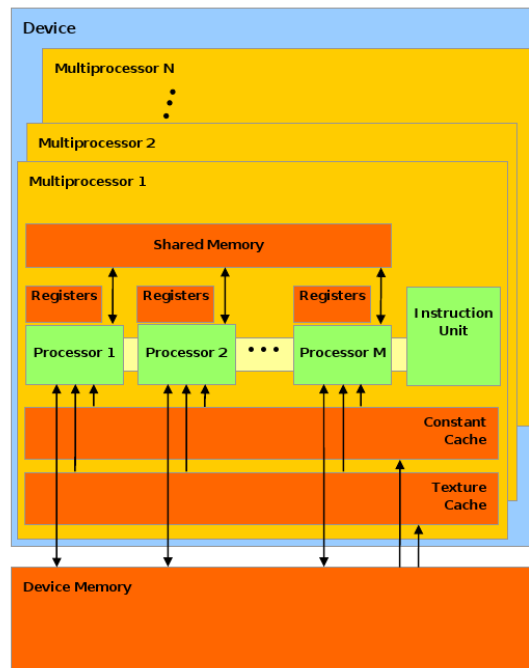


Figure 4.2: CUDA Hardware Model: A set of SIMD multiprocessors with on-chip shared memory [NVIDIA 07].

The GPU is a computation device capable of executing a very high number of threads in parallel. GPUs consist of a set of SIMD<sup>2</sup> multiprocessors (see figure 4.2). Each Multiprocessor consists of a number of processors, each executing one thread. All processors within the multiprocessor execute the same instruction. This means that threads with an identical control flow operating on isolated data can be effectively computed in parallel on one multiprocessor.

### 4.1.1 Execution Model

A C function to be executed in parallel on the device is called a *kernel*. All threads are split up into a grid of blocks. This is called the kernel's execution configuration. Blocks are executed in parallel on the multiprocessors (see figure 4.3). A block is

<sup>2</sup>SIMD = Single Instruction Multiple Data

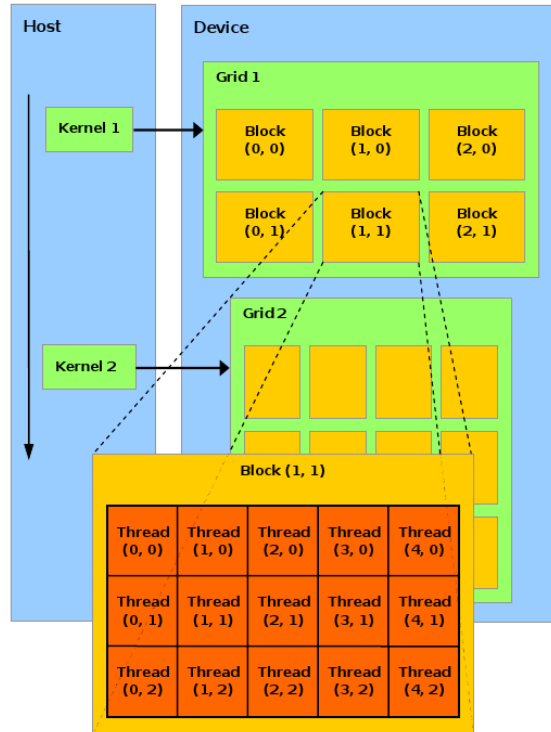


Figure 4.3: CUDA Thread Batching: Each kernel executes as a batch of threads organized as a grid of thread blocks [NVIDIA 07].

processed only on one multiprocessor, but more than one block can be assigned to the same multiprocessor concurrently. Each block is automatically split up into SIMD groups of threads called *warps* which are then scheduled to the multiprocessor.

Threads within the same block can share data and synchronize with each other. This is why it is most convenient to have large blocks. However the block size is limited by the hardware capabilities of the device. Threads in one block have to share limited resources, for example shared memory.

The execution configuration of a kernel call is declared in the code by a special CUDA syntax extension to C.

The configuration parameters are:

- `grid`  
is of type `dim3` and specifies the dimension of the grid, i.e. the number of blocks equals `grid.x * grid.y`. The third dimension `grid.z` stays unused.
- `block`  
is of type `dim3` and specifies the dimension of a block. The number of threads in the block is `block.x * block.y * block.z`.
- `memsize`  
is of type `size_t` and specifies the number of bytes that are dynamically allocated per block in addition to the statically allocated memory.

Take a look at example 4.4.

```

1 // kernel declaration
2 __global__ void Func(float* parameter);
3 ...
4 // kernel call
5 Func<<< grid, block, memsize>>(parameter);

```

Figure 4.4: Cuda kernel declaration and call

Note the `__global__` CUDA keyword. It declares the function to be executed on the cuda device, making it a kernel.

The configuration parameters listed above can be accessed from within each thread through built-in Variables: `gridDim` and `blockDim`.

Through the variables `blockIdx` and `threadIdx` a thread can also access its grid and thread coordinates.

To make all this work CUDA code has to be compiled with the special `nvcc` compiler, which comes with the CUDA Toolkit.

## 4.2 Memory

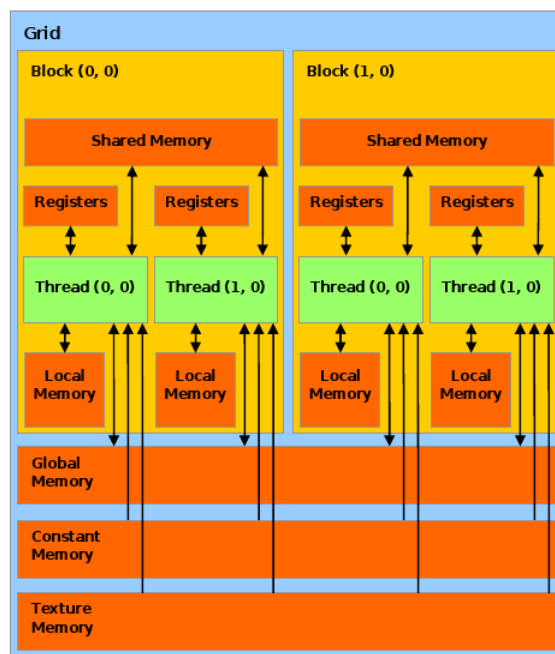


Figure 4.5: CUDA Memory Model: Memory spaces of various scopes [NVIDIA 07].

A kernel thread can access different memory spaces according to table 4.1.

Global, constant and texture memory can be read from and written to by the host and are persistent across kernel launches.

Memory	Access	Scope	Persistence	Cached	Latency <sup>3</sup>
register	read-write	thread	no		0
local memory	read-write	thread	no		4
shared memory	read-write	block	no		4
global memory	read-write	grid	yes	no	400-600
constant memory	read	grid	yes	yes	400-600
texture memory	read	grid	yes	yes	400-600

Table 4.1: CUDA Memory Scopes

Shared memory is extremely fast and enables inter-process communication between the threads in one block.

Due to the latencies shown in table 4.1 it has become a common pattern in CUDA programming to have all threads read data from global, constant or texture memory to shared memory prior to any computation. Once all needed data is loaded into the shared memory read and write access is extremely fast. Once the computation is done the results are written back to the global memory.

For more information please see the *CUDA Programming Guide* [NVIDIA 07].



## 5. Implementation

Prior to implementation it was necessary to become acquainted with NVIDIA's new CUDA technology. In [Yang 05] a stereo vision implementation on commodity graphics is presented using OpenGL. CUDA enables a completely new approach. The CUDA framework explicitly targets non-graphical computation on consumer graphics boards. At the time of writing this novel framework is still under heavy development and experimental. During the course of this work CUDA releases changed three times from version 0.8.2 to 1.0.

### 5.1 Getting started

Prior to any stereo vision specific solution the *CUDA Toolkit* had to be setup on a Linux system. CUDA comes with a set of example applications. Once the toolkit is installed correctly these examples compile. There are two modes, emulation and production mode. In the emulation mode no CUDA supported hardware is needed. The execution is simulated on the CPU. This is quite slow, but useful for debugging. In production mode the CUDA programs are executed on the graphics hardware. At the time of writing this only works with a special NVIDIA graphics driver designed for the CUDA Toolkit available on the CUDA home page<sup>1</sup>.

In addition up to June 2007 the CUDA framework was only available in version 0.8.2. This version did not yet support the graphic card in our development system, a NVIDIA GeForce 8600 GT. Kindly, the department of computer graphics<sup>2</sup> gave us permission to run tests on their GeForce 8800 GTX card. Support for the NVIDIA 8600 card was finally available in the 0.9 and 1.0 releases, which were obtained through the NVIDIA developer early access program.

In order to develop the stereo vision solution IDE, build system, and versioning system had to be setup. As development platform Eclipse CDT and Emacs was used. The stereo vision software had to be integrated into the MCA framework<sup>3</sup>. For that reason the example *make* files could not be used. MCA uses the *Scons*<sup>4</sup>

---

<sup>1</sup><http://developer.nvidia.com/object/cuda.html>

<sup>2</sup><http://www-hagen.informatik.uni-kl.de/>

<sup>3</sup>MCA = Modular Controller Architecture (see [Scholl 02, Koch 07] and <http://www.mca2.org>)

<sup>4</sup><http://www.scons.org>

build system. This required some Python coding to create a CUDA specific *Scons* builder that would integrate *nvcc* into the exiting *Scons* framework. As a versioning system *stereovision cuda* was setup as a sub project within the MCA subversion system.

As a first guide to a stereo vision solution the NVIDIA convolution example provided most useful information to understand the basic concepts in order to solve the stereo vision problem on CUDA hardware. It describes a convolution filter that analyses the surrounding rectangle area for each pixel. A similar approach was then used to obtain the first window based CUDA stereo vision solution.

After three weeks of documentation studies and implementation of prototypes the first runnable stereo vision system was completed. The here presented version of the window based stereo algorithm is the optimized and final version of the simple window based approach. It demonstrates the most important CUDA programming principles to gain most out of the graphics hardware.

## 5.2 Window Based Approach

```
usage: stereovision_cuda_simplewindow left_image right_image
      result_image thread_width thread_height
```

optional arguments are:

```
-dmin          disparity minimum (default: 0)
-dmax          disparity maximum (default: 32)
-width         window_width (default: 5)
-height        window_height (default: 5)
-weight_x      weight of channel x (default 0.3),
               weight sum should be 1
-weight_y      weight of channel y (default 0.3),
               weight sum should be 1
-weight_z      weight of channel z (default 0.3),
               weight sum should be 1
-scale         scale of output image
               (disparity value * scale) (default: 1),
               a power of 2
-showResult    default: 0 (off)
-showInput     default: 0 (off)
```

The window based algorithm takes two input images of equal size as input and computes a disparity map in terms of a gray scale image as output.

The input images can be color images. Internally they are converted to HSV (Hue, Saturation, Value) format. The user can provide weights  $(w_x, w_y, w_z)$  for all three channels. Depending on the image quality it can make a difference of using different weights. If for example the colors of both images are not well calibrated it can make sense to concentrate on saturation or value only.

In addition the user should specify the minimum and maximum disparity values and the window width and height as command line parameters.



The minimum and maximum disparity depend on the camera setup. With increasing distance the disparity values converge towards 0. This is why the disparity minimum is usually assumed to be 0. The maximum disparity value can be identified during camera calibration, by manually calculating the disparity for a point in 3D space with minimal distance to the stereo head system. For benchmark images, such as the Middlebury stereo pairs, the maximum disparity is known.

Depending on the image resolution, the window width  $ww$  and window height  $wh$  are usually: 3x3, 5x5 or 7x7. Larger values result in smooth but blurry disparity maps. Small window sizes produce grainy images, with potentially more outliers.

### 5.2.1 Window Based Kernel

For each pixel  $(p_x, p_y)$  and disparity step  $d$  the algorithm has to compare the window in the reference image to the window along the epipolar line in the second image. For a comparison function the sum of absolute differences (SAD) is used.

$$sad(p_x, p_y) = \sum_{py=p_y-\lfloor\frac{wh}{2}\rfloor}^{py+\lfloor\frac{wh}{2}\rfloor} \sum_{px=p_x-\lfloor\frac{ww}{2}\rfloor}^{px+\lfloor\frac{ww}{2}\rfloor} \begin{pmatrix} w_x \\ w_y \\ w_z \end{pmatrix} \cdot \begin{pmatrix} abs(I_l(px, py).x - I_r(px - d, py).x) \\ abs(I_l(px, py).y - I_r(px - d, py).y) \\ abs(I_l(px, py).z - I_r(px - d, py).z) \end{pmatrix} \quad (5.1)$$

Note that not the whole image can be processed. On the left and right side there are borders of half the window width. On top and bottom there is a border of half the window height. The pixel in this rim are not sufficiently padded. Therefore they are left out in the computation.

The rest of the image is split up into tiles, i.e. rectangular sections of the image (see figure 5.1). In order to process the image in parallel each kernel block is responsible for computing the disparity values for one tile. In order to compute the SAD values for each pixel in the tile a slightly larger rectangular area needs to be analyzed by each kernel. More precisely the area the kernel has to process is the area of the tile plus half window size towards left and right side and half window width towards top and bottom. This extra area is called the apron.

The aprons of two adjacent tiles overlap. This means pixels in the apron will be accessed by multiple kernels. Since interprocess communication is restricted to one block, it is inevitable that blocks of adjacent tiles will perform multiple read accesses to the memory to fetch the same pixel.

Access to the texture memory is 100 times slower than accessing a shared memory location. In order to get the maximum throughput all reference pixels of a block are loaded in parallel. Each thread loads one pixel into shared memory. From then on the apron threads are shut down. In CUDA development it is always important to keep the ration of idle threads to active threads as low as possible. For this reason it is best to have large tiles. In the presented algorithm the number of threads and therefore the tile size is maxed out to 512 thread per block<sup>5</sup>, the current maximum possible.<sup>6</sup> This translates into a constant block width of  $bw = 32$  threads and a

<sup>5</sup>blockIdx.x \* blockIdx.y \* blockIdx.z <= 512

<sup>6</sup>To make this possible the kernel was optimizing to use only 16 registers.

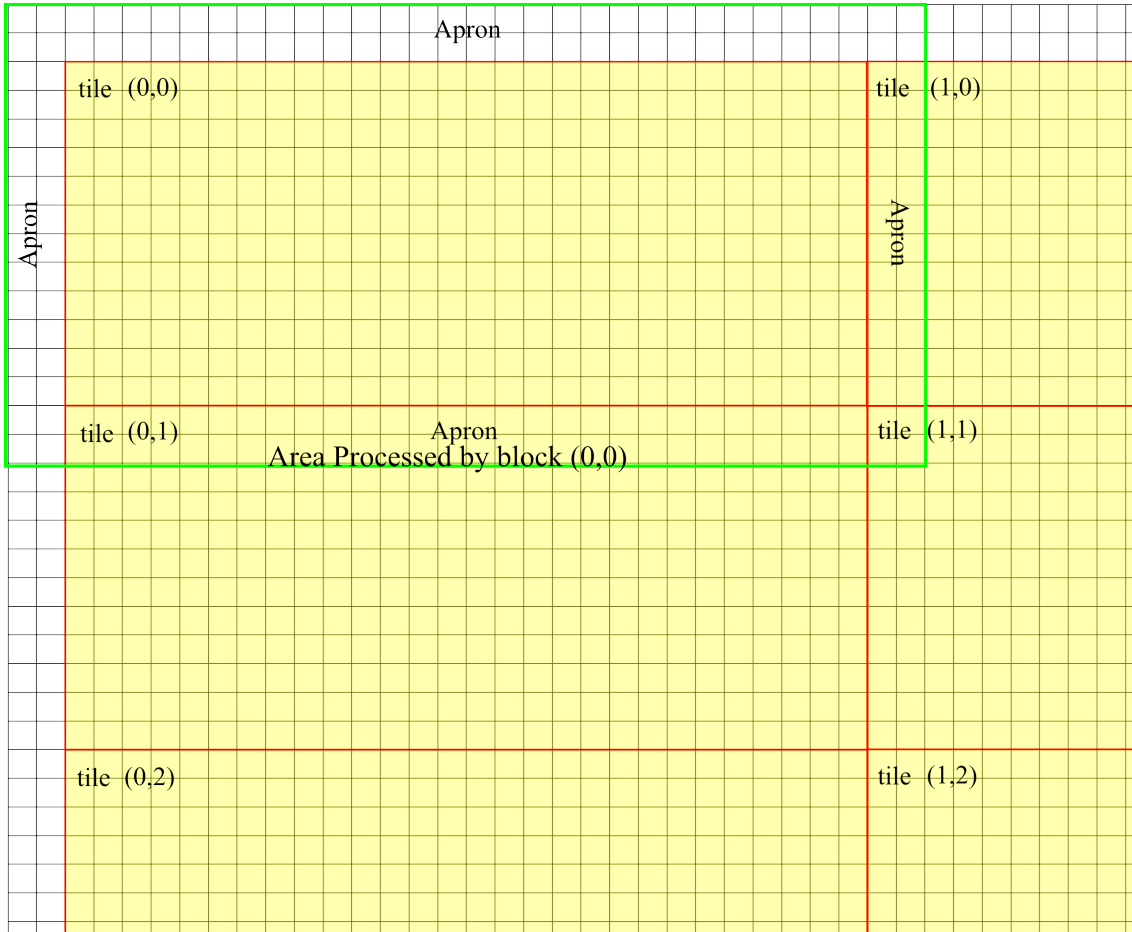


Figure 5.1: Image split up into tiles and corresponding thread blocks.

constant block height of  $bh = 16$  threads. The tile height  $th$  and tile width  $tw$  can be computed as follows:

$$tw = bw - 2 \cdot \lfloor \frac{ww}{2} \rfloor \quad (5.2)$$

$$th = bh - 2 \cdot \lfloor \frac{wh}{2} \rfloor \quad (5.3)$$

Obviously the ration of idle to non idle threads depends only on the window size (see table 5.1). The kernel does not scale with window size. As shown in table 5.1 it has to becomes unefficient with larger windows. At a 7x7 window almost as many threads are put into idle mode as there are active threads calculating disparities. The usage of a 9x9 window does not make any sense at all. However, 7x7, 9x9 or larger window sizes produce blurry images. It is therefore justified to concentrate of 3x3 and 5x5 window sizes.

The tiles are all of equal size, except the ones in the last row and last column. They may be smaller, since a whole tile might not fit into the image. For this reason the kernel has guard access to pixel coordinates outside of the image. The algorithm

window size	3x3	5x5	7x7	9x9
block size	512	512	512	512
active (tw * th)	420	336	260	192
idle	92	176	252	320
idle to active ratio	0.22	0.52	0.97	1.67
idle [%]	18.0	34.4	49.2	62.5

Table 5.1: Idle to active threads ratio

does not prevent access to negative x coordinates in the texture. This happens when the algorithm analyses pixels on the left side of the reference image. It tries to read the corresponding pixels that is shifted to the left by the current disparity step. This pixel may have a negative x coordinate. The texture unit however is adjusted to return the left-most valid texture value in this case. This special behavior makes sense. In this area not the whole disparity range can be analyzed. Results here can not be reliable, yet some pixels may match.

```

1 __global__ void StereoKernel( unsigned char* disparity_array,
2                               int wx,
3                               int wy,
4                               int tile_width,
5                               int tile_height,
6                               int image_width,
7                               int image_height,
8                               float disparity_min,
9                               float disparity_max,
10                              float weight_x,
11                              float weight_y,
12                              float weight_z
13                              ) {
14     const int wxh = wx >> 1;
15     const int wyh = wy >> 1;
16     const float px = threadIdx.x + MUL( tile_width , blockIdx.x);
17     const float py = threadIdx.y + MUL( tile_height , blockIdx.y);
18     const int result_index = (int)((py * image_width) + px);
19     float4 d0 = tex2D( tex_image0, px, py); // left reference image
20     float old_minimal_difference = FLOAT_MAX;
21     for(float disparity = disparity_min;
22         disparity <= disparity_max;
23         disparity++){
24         // always true.. only to open new var. scope
25         // but compiler's liveness analysis is too bad
26         if(px > 0){
27             // load from right image (cached texture)
28             float4 d1 = tex2D( tex_image1, px - disparity, py );
29
30             /* calc SAD and store result in shared memory */
31             differences[UMUL(threadIdx.y , blockDim.x) + threadIdx.x] =
32                 weight_x * abs( d1.x - d0.x )
33                 + weight_y * abs( d1.y - d0.y )
34                 + weight_z * abs( d1.z - d0.z );
35         }
36
37         /* make sure all shared memory entries for this block
38          * are calculated and loaded */
39         __syncthreads();
40
41         /* make sure px,py are not on the apron or image rim */
42         if(!(threadIdx.x < wxh
43             || threadIdx.x >= wxh + tile_width
44             || threadIdx.y < wyh
45             || threadIdx.y >= wyh + tile_height
46             || px >= image_width - wxh
47             || py >= image_height - wyh
48             )){
49             // sum up window
50             float sum = 0;
51             for(int y = 0; y < wy; y++) {
52                 for(int x = 0; x < wx; x++) {
53                     sum += differences [ MUL(((int)threadIdx.y + y -wyh),
54                                             blockDim.x)
55                                         + threadIdx.x + x - wxh];
56                 }
57             }
58             /* find minimum */
59             if( sum < old_minimal_difference) {
60                 disparity_array[ result_index ] = (unsigned char)disparity;
61                 old_minimal_difference = sum;
62             }
63         }
64         /* make sure all threads have calculated the sum
65          * before altering their difference array location again
66          */
67         __syncthreads();
68     }
69 }

```

Figure 5.2: Window based stereo kernel

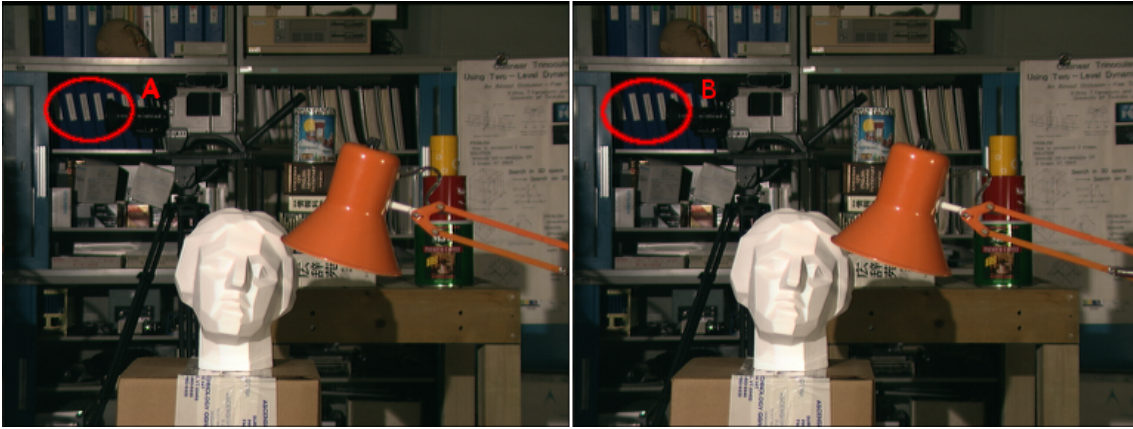


Figure 5.3: Repetitive textures in area A and B in Tsukuba example are hard to match correctly.

### 5.3 Pyramid Based Approach

The straightforward approach presented in the last chapter has two major disadvantages. First of all is the execution time directly dependent on the disparity range processed. In an outdoor setting larger disparity ranges are common. To keep the resulting disparity image crisp and to obtain good performance results it is good to have small window sizes. This however increases the likelihood of matching the wrong minimum, since the support region is small. The effect can be observed best on repetitive textures (see figure 5.3).

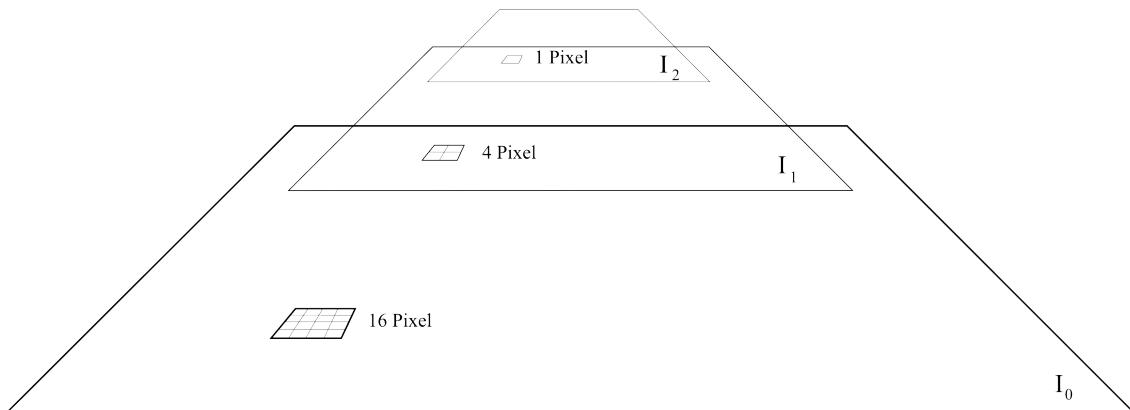


Figure 5.4: Pyramid of scaled down image copies. Factor 1/4

In order to cope with these problems a pyramid based approach can be applied. In a pyramid approach the both input images are scaled down several times. Figure 5.4 illustrates one original input image  $I_0$  and the scaled down copies  $I_{[1..n]}$  as a pyramid. A stereo algorithm with a fixed window size is then applied on each of the scaled copies for a reduced disparity range, starting at  $I_n$ . From that result a rough estimate for  $D_{(n-1)}$  can be calculated. For all  $i \in \{n, \dots, 1\}$  the disparity map

$D_i$  serves as a hint for  $D_{(i-1)}$  in the corresponding region in the next larger pyramid level  $I_{(i-1)}$  (see figure 5.5).

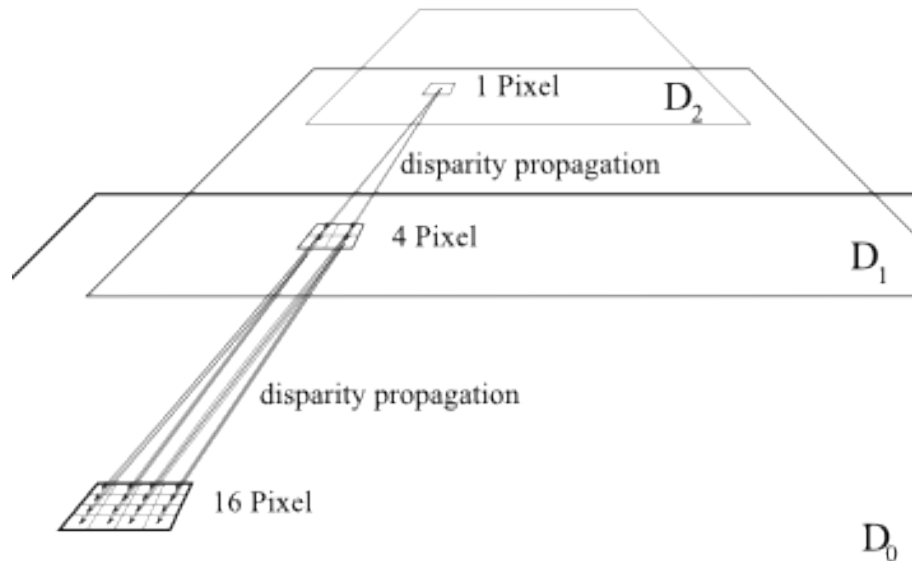


Figure 5.5: Propagation of disparity estimates

During disparity propagation from  $D_i$  to  $D_{(i-1)}$  the disparity values have to be adjusted to the larger image resolution. The resolution of  $I_i$  is exactly double the resolution of  $I_{(i-1)}$ . Therefore the disparity estimates for  $D_{(i-1)}$  are computed by multiplying  $D_i$  times two. The estimates are all even. In order to close the “gaps” the stereo algorithm on level  $I_{(i-1)}$  has to decide between the even and uneven disparity step. In order to make the algorithm more robust against initial errors a slightly larger disparity range around the estimate may be selected. In this fashion the disparity estimates are refined from coarse to finer grained matches with every pyramid level. Finally, the procedure yields the result  $D_0$ .

This algorithm produces finer grained disparity maps at a lesser likelihood of false disparity matches. This is because the disparity estimates on a small copy are based on a virtually larger window than the actual window used, since each pixel in a scaled down copy was obtained by interpolation. Thus, each pixel represents a whole region of the larger copy.

When the image data is loaded from host memory to GPU texture memory a conversion must take place. The pixel data has to be converted from a 3 times 8 bit format to a Float4 CUDA type. This type takes care of the correct memory alignment. As a comparison function this algorithm also uses SAD. The distributive law allows the weights for all three color channels to be applied already during this conversion. This way the weight factors are also applied to all scaled down copies.

The factor by which each level is scaled is one fourth. This factor translate into half window width and half window height. This enables very efficient computation of down scaling as described in chapter 5.3.1.

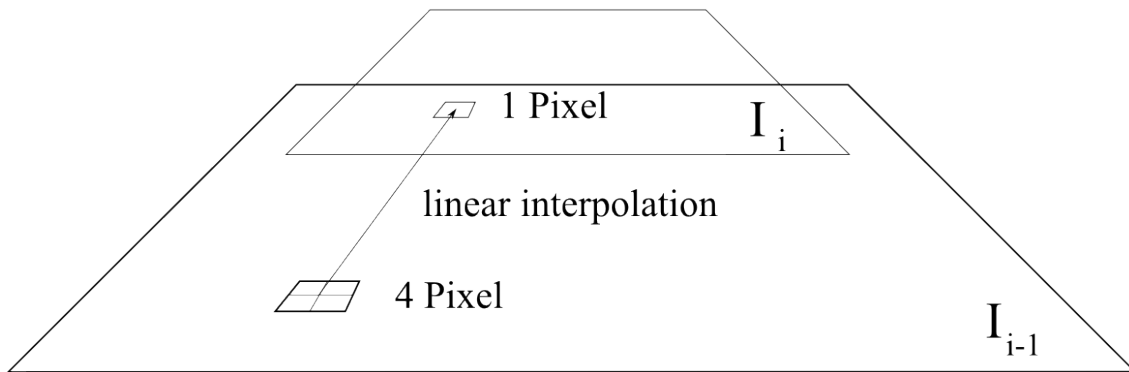


Figure 5.6: Linear interpolation on texture

### 5.3.1 Scale Down Kernel

For input image  $I_i$  a scaled down copy  $I_{(i+1)}$  with half width and half height has to be produced. This means each pixel in the result image is assigned the average value out of 4 pixels in the input image. For an efficient implementation it makes sense to put the scale down function on the graphics hardware as well. On graphics hardware the computation of average values is extremely simple and can be done in parallel. The algorithm (see listing 5.7) takes advantage of the built-in texture unit. By requesting the value of the texture coordinate right between all four input images the texture unit returns the linear interpolation of all 4 pixels values in all three color channels (see figure 5.6 and test results 5.3.1).

Furthermore there is no need to copy each scaled copy to the graphics board memory for each stereo vision kernel invocation. The scale down kernel's result stays in GPU memory and can later be accessed by the stereo vision kernel.

```

1  /*!
2  * reduces the size of an image a x b to a/2 x b/2 using 9bit
3  * precision floatingpoint interpolation
4  * global kernel texture reference tex_image (input image)
5  * width - width of resulting image
6  * height - height of resulting image
7  * pitch - pitch of 2D global memory array from cudaMalloc2D
8  */
9  __global__ void ScaleDownKernel(unsigned int width,
10                                 unsigned int height,
11                                 float4* data,
12                                 unsigned int pitch) {
13      int px = threadIdx.x + blockDim.x * blockIdx.x;
14      int py = threadIdx.y + blockDim.y * blockIdx.y;
15
16      if(px < width && py < height) { // check px,py within boundaries
17          float4* row = (float4*)((char*)data + py * pitch);
18          row[px] = tex2D(tex_image,
19                        2.0f*(float)px + 1.0f ,
20                        2.0f*(float)py + 1.0f);
21      }
22 }

```

Figure 5.7: Scale Down kernel



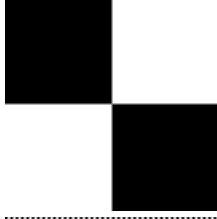

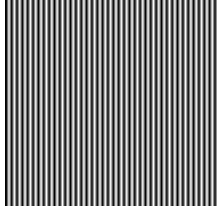

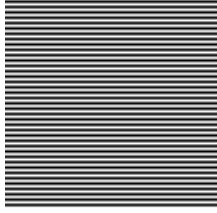

Test	Input Image	Output Image
Black and white checker board and interpolated result. Note the gray line resulting from interpolation on the black and white transition.		
Black and white checker board shifted by one pixel horizontally. No gray line visible in interpolated result.		
Alternating black and white pixel columns. Resulting image is a solid gray <code>rgb(127,127,127)</code> .		
Alternating black and white pixel rows. Resulting image is a solid gray <code>rgb(127,127,127)</code> .		

Table 5.2: Scale and interpolation test images and results

This kernel is quite simple. It uses only 8 multiprocessor registers and 32 bytes of shared memory. Using the *CUDA GPU Occupancy Calculator*<sup>7</sup>, the optimal configuration parameters of 256 threads per block were determined. An occupation of hundred percent for each multiprocessor was hereby achieved. On bandwidth bound kernels a high occupation is the only way to let the CUDA framework hide memory latency by computation.

<sup>7</sup>Spreadsheet in CUDA Toolkit to calculate GPU occupancy



### 5.3.2 Pyramid Stereo Kernel

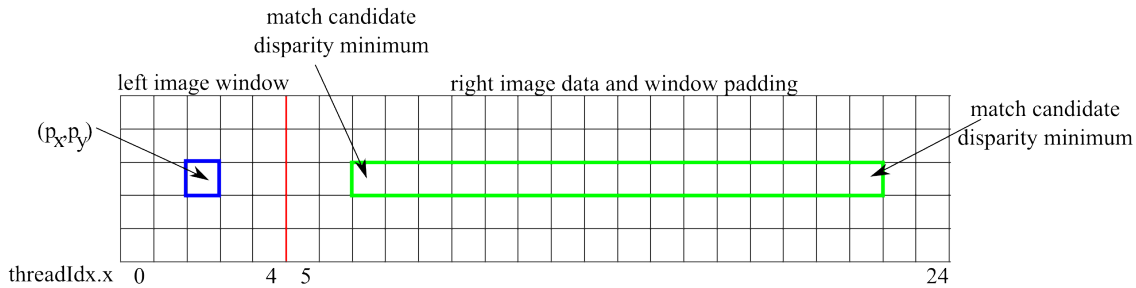


Figure 5.8: Example: Pyramid stereo kernel thread configuration for window size: 5x5 and disparity range 16.

The pyramid stereo algorithm is also window based. However it uses a different approach in mapping the images pixels to threads and blocks. In this approach one block of threads is used to calculate the whole disparity match for one pixel. This means that the maximum disparity range that can be processed by the algorithm is limited by the maximum number of threads in one block (currently 512). This number however is far larger than needed in the pyramid approach. Should there ever be a need for a larger disparity range an additional pyramid level can be introduced. The theoretical maximum disparity range equals  $512 \cdot 2^n$  where  $n$  is the number of pyramid levels above the original level  $I_0$ .

To take a closer look at how each thread block can handle the computation of the whole disparity spectrum it has to be recalled which regions in both input images are relevant for the calculation of a pixel's disparity value. Let  $(p_x, p_y)$  be the pixel's coordinates.

In the left window this is an area of window width times window height around the examined pixel. In the right image all possible match candidates have y-coordinate  $(p_y)$ . Their x-coordinates range from  $p_x$  minus disparity minimum  $d_{min}$  to  $p_x$  minus disparity maximum  $d_{max}$ . Around each match candidate a rectangular area of window width times window height needs to be examined. These areas overlap and form a strip (see figure 5.8).

A block of threads is configured to load all needed image data in parallel. The configuration is set to be one-dimensional. In sum:  $ww + (d_{max} - d_{min}) + 2 \cdot \lfloor \frac{ww}{2} \rfloor$  threads are started on the x component of the block configuration. The first  $ww$  threads load the reference data from the left image. Threads with  $id \geq ww$  load the strip of pixels in the right image. Each thread loads a whole column of window height pixels into shared memory. During the computation of the best disparity match threads with  $id < ww$  are set to idle. All others are active in the computation of the SAD for their disparity candidate. This way a bad idle to active ratio is avoided which can be computed as:  $\frac{ww + 2 \cdot \lfloor \frac{ww}{2} \rfloor}{d_{max} - d_{min}}$ . In the end a distributed minimum algorithm is used to find the correct match (see figure 5.9). This algorithm is very fast, but it relies on disparity ranges to be a power of two.

For each pyramid level the stereo kernel is invoked once. The disparity map is read from and written to global memory, which stays persistent in between kernel calls.

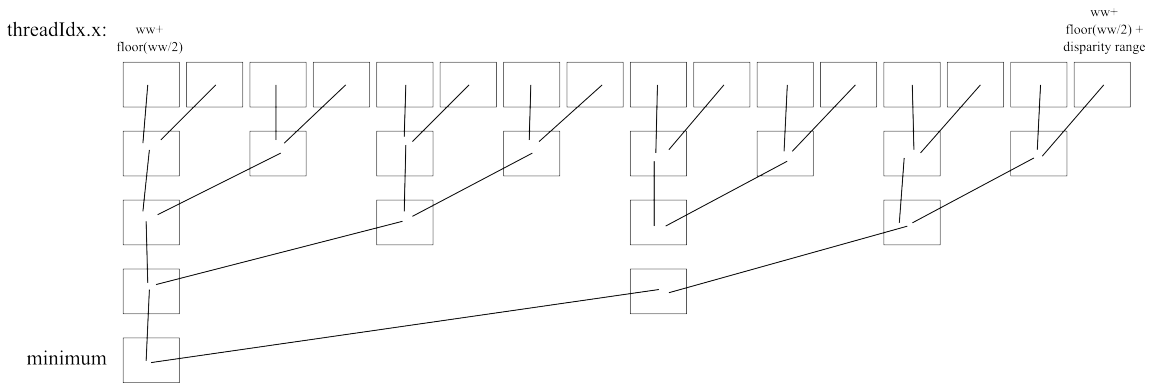


Figure 5.9: Example: distributed minimum calculation for disparity range 16.

After the first run the rough estimates for the disparity values is present in memory. In order to refine the estimates for each following kernel invocation the disparity range can be redefined. Theoretically the minimum disparity range would be 2, since the width is doubled between two pyramid images. This however gave poor results. The optimum would probably be a value of 5 leaving 2 options to either side of the current value. However 5 is not a power of two. This is why 4 was chosen as the disparity range for refinement calls.

```

1
2 __global__ void PyramidStereoKernel(
3     const float roi_offset_x,
4     const float roi_offset_y,
5     const int window_width,
6     const int window_height,
7     int* disparity_array, // guess and result
8     const int roi_width,
9     const unsigned int stride, // >=1.0
10    const float right_image_offset
11 )
12 {
13
14    float4* sm_image = (float4*) sm_array;
15    int disparity_range = (int)blockDim.x - 2 * window_width + 1;
16    int disparity_min = disparity_array[ stride*blockIdx.y * roi_width
17                                       + stride*blockIdx.x ]
18                                   - (int)stride*(disparity_range / 2);
19
20    /** load all columns */
21    const int window_width_h = (window_width / 2);
22    const int window_width_15 = window_width + window_width_h;
23    const int tx = threadIdx.x; // type conversion
24
25    float px = roi_offset_x+(float)(blockIdx.x + threadIdx.x);
26    float py = roi_offset_y+(float)(blockIdx.y)-(float)(window_height/2);
27
28    if(tx < window_width) {
29        /* threads < window_width load reference region in left image */
30        px += (float)( - window_width_h );
31    } else {

```

```

32     /* threads >= window_width load disparity region in right image*/
33     px += right_image_offset - (float)disparity_min / (float)stride
34     + (float)(-disparity_range - window_width_15);
35 }
36
37 // load column into shared memory
38 for(int i = 0 ; i < window_height; i++) {
39     sm_image[window_height*tx+i] = tex2D(tex_image0, px, py + i );
40 }
41
42 // make sure all pixels are loaded into shared memory
43 __syncthreads();
44
45 // create shared memory arrays
46 // (thread configuration reserves device shared memory space)
47 // offset to sm_image array
48 float* sm_sad = (float*) & sm_image[blockDim.x * window_height];
49 // offset to sm_sad array
50 int* sm_disparity = (int*) & sm_sad[disparity_range];
51
52 int column_index = tx - window_width_15;
53
54 /** calc sad */
55 // exclude threads that are not in the disparity range
56 if(tx >= window_width_15
57 && tx < window_width_15 + disparity_range ) {
58
59     // init sad sum
60     float sum = 0;
61     for(int w = 0; w < window_width;w++) {
62         int index0 = window_height * (tx + w - window_width_h);
63         int index1 = window_height * (w);
64         for(int h = 0; h < window_height; h++) {
65             // sad
66             sum += abs(sm_image[index0].x - sm_image[index1].x)
67                 +abs(sm_image[index0].y - sm_image[index1].y)
68                 +abs(sm_image[index0].z - sm_image[index1].z);
69             index0++;
70             index1++;
71         }
72     }
73     // write result to shared memory
74     // no * 0.3333f, weights add up to 1
75     sm_sad[column_index] = sum;
76
77     // initialize disparity array with reverse index * stride
78     sm_disparity[column_index] = disparity_min
79         + (int)stride
80         * (disparity_range - column_index);
81 }
82 __syncthreads();
83
84 /* distributed min */
85 for(int range = 1; range < disparity_range; range += range) {
86     if(tx >= window_width_15
87         && tx < window_width_15 + disparity_range

```

```
88     && column_index % (2*range) == 0) {
89         int column2 = column_index + range;
90         if(sm_sad[column_index] > sm_sad[column2]) {
91             // set winner
92             sm_sad[column_index] = sm_sad[column2];
93             sm_disparity[column_index] = sm_disparity[column2];
94         }
95     }
96     __syncthreads();
97 }
98
99 /* write disparity: */
100 if(tx == window_width_15) {
101     int stride_h = stride / 2;
102     int result_index = stride * blockIdx.y * roi_width
103                     + stride * blockIdx.x;
104     // (+0,+0)
105     disparity_array[ result_index ] = sm_disparity[0];
106     //(+0,+ stride/2)
107     disparity_array[ result_index + stride_h ] = sm_disparity[0];
108
109     result_index += stride_h*roi_width;
110     //(+ stride/2,+0)
111     disparity_array[ result_index ] = sm_disparity[0];
112     //(+stride/2,+ stride/2)
113     disparity_array[ result_index + stride_h ] = sm_disparity[0];
114 }
115
116 }
```

### 5.3.2.1 Region of Interest

In the field of application certain image regions may be of special interest, while others are not interesting at all. Therefore the algorithm was adjusted to allow for a definition of a rectangle area, which represents the region of interest. The disparity values are only computed for this region. If no region of interest is set the largest possible region is set by default. The algorithm can quite easily be adapted for this feature. This is because each result pixel has its own kernel thread instance. Excluding pixel outside of the region of interest is therefore only a matter of starting the kernel with an altered configuration. The kernel configuration parameters width and height had to be adjusted and additional two kernel parameters `offset_x` and `offset_y` had to be incorporated to consider the shift in input coordinates.

### 5.3.2.2 Confidence

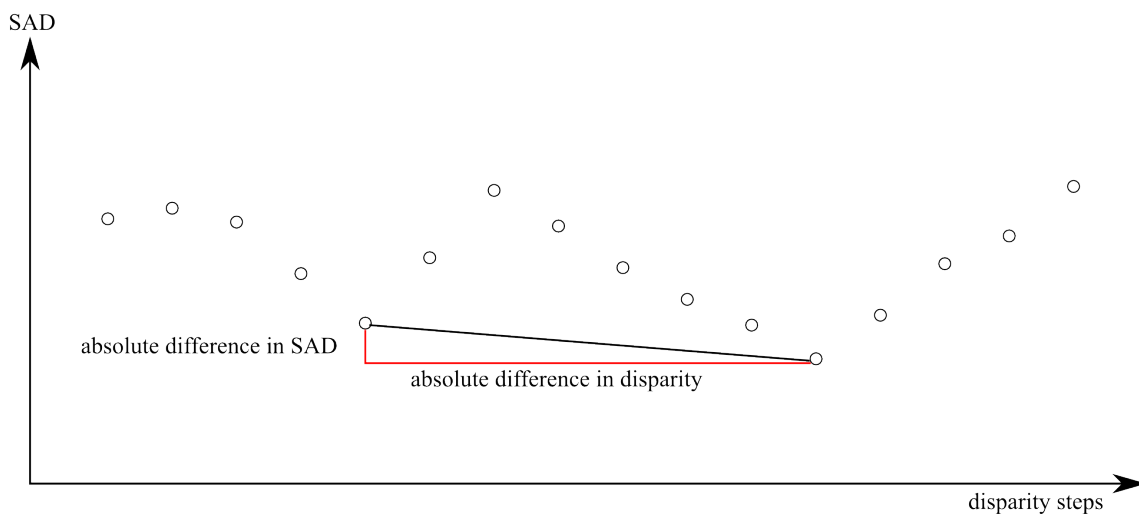


Figure 5.10: Example: Peak Ratio Metric for SAD distribution with two local minima. (disparity range = 16)

The window based approaches presented here run very fast. As a trade-off they do not produce state of the art quality results. For real-time robotic applications speed is more important than quality. However it is still important to have some measure to judge whether the generated disparity map is reliable or not, since further decisions might map on the map. For that reason a confidence map is generated as well. A confidence measure can be calculated using different approaches, of which none is perfect. (cite ...). In this algorithm a combination of three metrics is used:

- Peak Ratio metric (PKR)

During the calculation of the minimum the difference between SAD values of winner and runner up can be put in relation to the distance between both disparity steps. If this slope (see figure 5.10) is very low it indicates that the match is not reliable. This is not very useful to find overall confidence, since only the last pyramid level is examined, yet it is very well suited to determine whether the current distribution is good enough to change the guessed disparity or not.

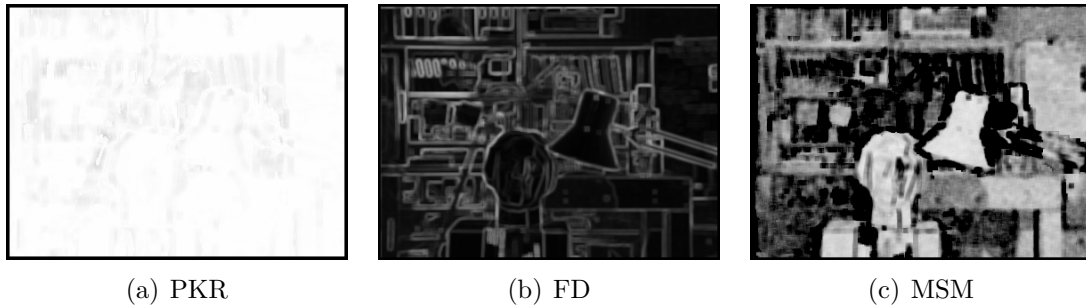


Figure 5.11: Tsukuba example: Confidence Maps

- Feature Density (FD)  
Feature density is a measure of diversity of the texture around each given point in the texture. It is calculated by first determining the average of each color channel in the window. In a second step the absolute difference between average and each point is summed up. A window based approach can not work on featureless textures.
- Matching Score Metric (MSM)  
The SAD value of the winning disparity step directly tells how good the best match actually was. A SAD of zero would indicate a perfect match. The only problem is that this value can not be easily normalized. It is a very straight forward method for overall confidence. Yet on low-textured surfaces this method would give false positives. Therefore it needs to be combined with feature density multiplicatively.

### 5.3.3 Post Processing Kernel

With the confidence values at hand it is possible to do some quick post processing without too much performance overhead. Once a suspicious disparity value is identified by a low confidence value the question is: How can it be repaired?

Given that for a low confidence pixel most neighboring pixels agree on the same disparity level and have a higher confidence, chances are good that an outlier was found and that its true disparity value should be on the very same level.

The GPU algorithm takes advantage of the confidence and disparity values still being in the GPU memory. For each pixel in parallel three passes are run. First horizontally, then vertically, and horizontally again. Each time the algorithm searches 10 pixels to either side for confidence values 10 percent higher than the pixel's own confidence value. Should there be 2 pixels on opposite sides of the current position with high enough confidence values and matching disparity levels the disparity value of the current pixel is corrected and its confidence is increased by ten percent.

On the Tsukuba example (see 5.12) this procedure corrected about two percent of the erroneous pixels. In the example the post processing algorithm closes some of the gaps in the background. Figure 7.11 in chapter 7 shows that the post processing step is very cheap compared to the pyramid kernel execution time.

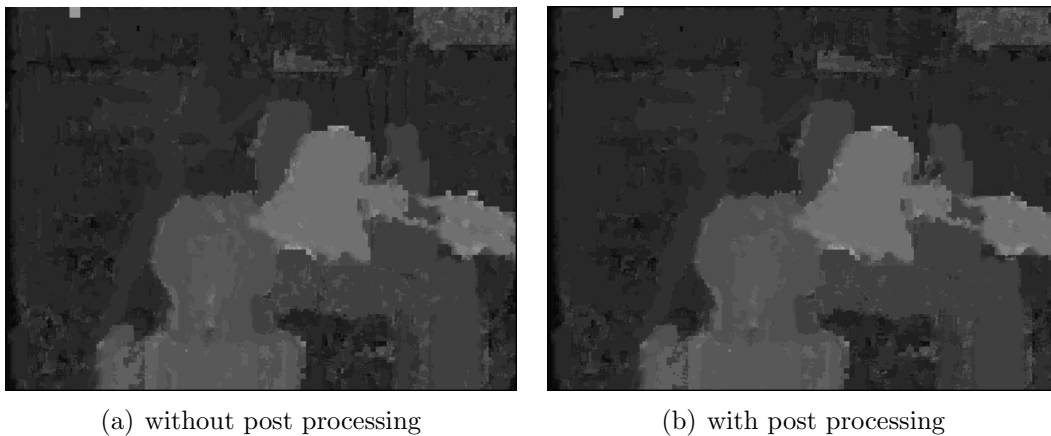


Figure 5.12: Tsukuba example: Post processing





# 6. Integration

## 6.1 MCA Stereo Vision

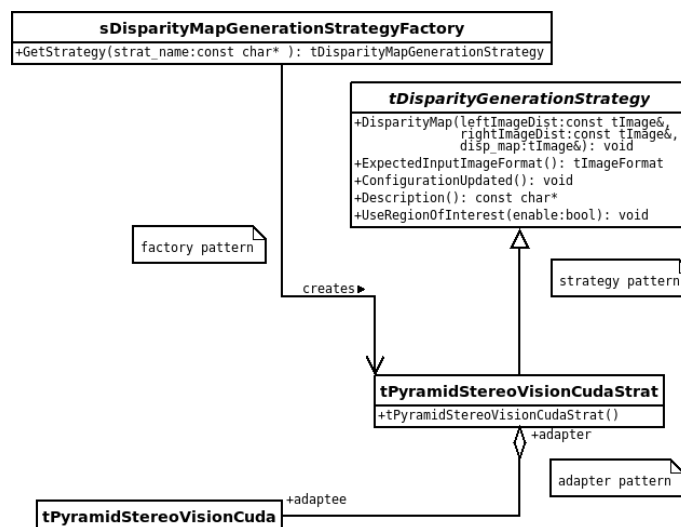


Figure 6.1: UML: Pyramid Cuda Integration into MCA Framework

The stereo vision algorithm under development was built and tested using a stand-alone command line program. Both, stand-alone version and MCA version of the code share the same algorithm core in class `tPyramidStereoVisionCuda` (see figure 6.2).

In the MCA Stereo Vision project there are already several stereo algorithm implementations. These implementations are all exchangeable. This is realized through the strategy design pattern; any stereo algorithm has to extend `tDisparityGenerationStrategy`. Via the factory `sDisparityMapGenerationStrategyFactory` a concrete strategy instance can be created.

According to these patterns *tPyramidStereoVisionCudaStrat* was created. The new strategy receives no command line parameters but is configured via the MCA configuration tree and the MCA Browser(see figure 6.3).

In order to keep the coupling between stereo algorithm and MCA framework loose, the actual core algorithm *tPyramidStereoVisionCuda* was hidden via adapter pattern. This way it was possible to continue testing the implementation separately.

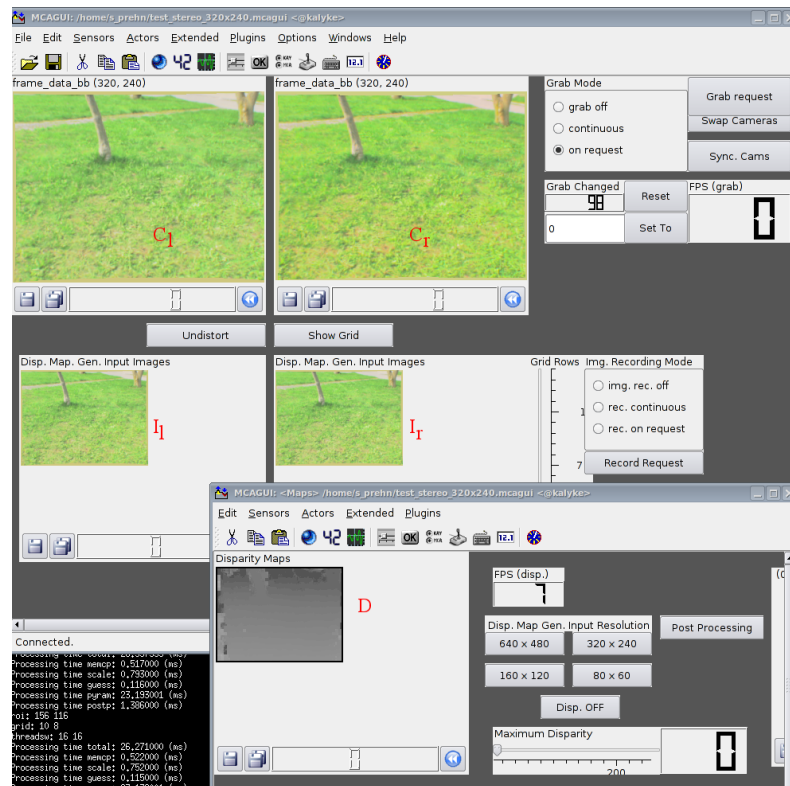


Figure 6.2: Screenshot of MCA Gui: showing input images and disparity map.

With the MCA Stereo Vision project a configured MCA GUI comes along (see figure 6.2). The two camera input images  $C_l$  and  $C_r$  are rectified ( $I_l$ ,  $I_r$ ) and fed into the stereo algorithm, which yields the disparity map displayed as  $D$ .

The MCA Browser (see B in figure 6.3) lets the user choose the input source and configuration parameters on the fly. In the browser the disparity generator implementation can be selected and configured as well. In figure 6.3 a region of interest configuration is shown.

Finally the Birchfield post processor, which was already present in the MCA stereo vision project, was integrated. Figure 6.4 shows the post processor being applied to the pyramid algorithm output.

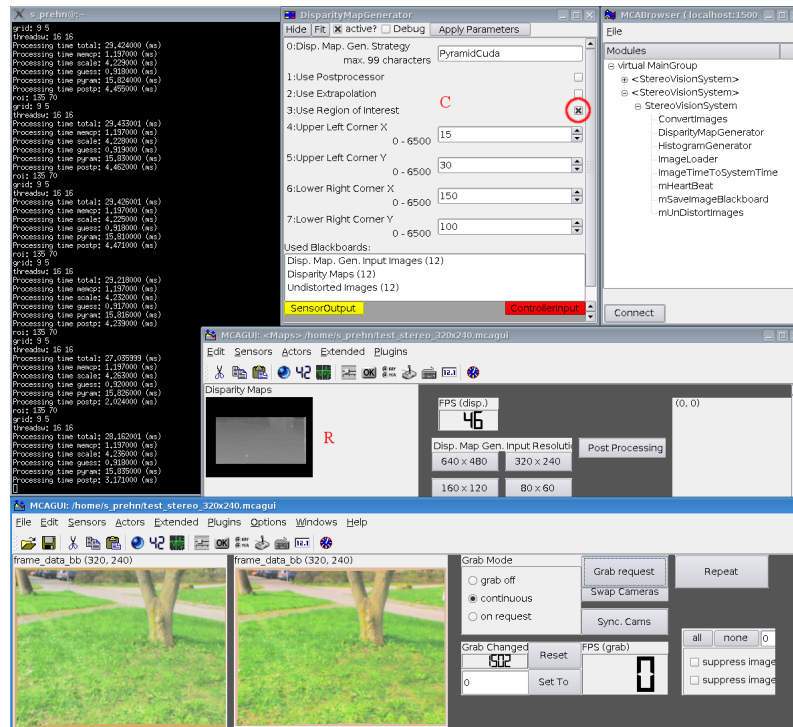


Figure 6.3: Screenshot of MCA Gui: Region of Interest Feature. B: MCA Browser, C: MCA Browser Configuration Window, R: disparity map reduced to region of interest

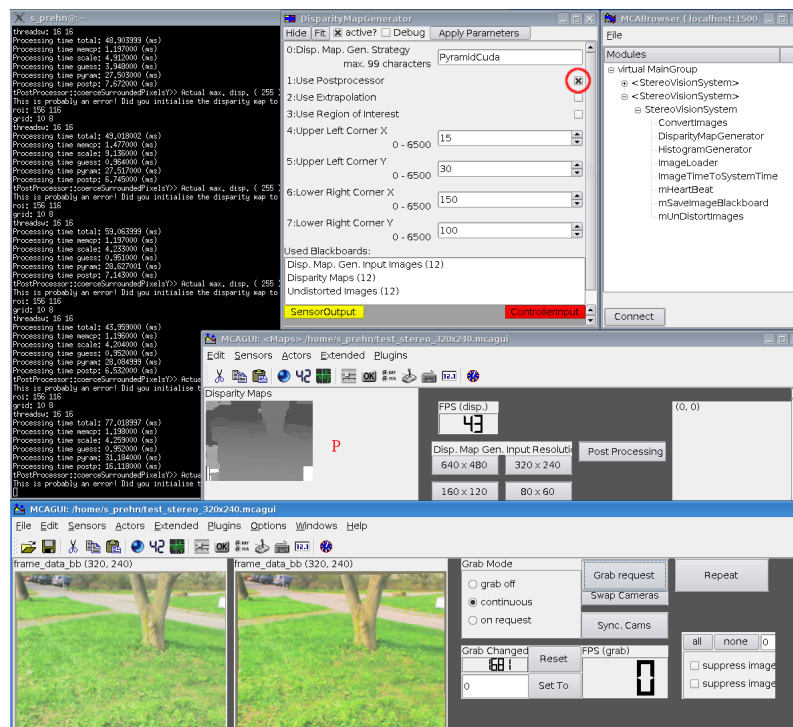


Figure 6.4: Screenshot of MCA Gui: Birchfield Post Processor activated. P: post processed output



# 7. Performance / Results

## 7.1 Quality

In order to evaluate the quality of the two stereo vision algorithms the Middlebury stereo datasets were used (see figure 7.1 and table 7.1). These are image pairs with known ground truth. They serve as a common benchmark for stereo algorithms. The error rates were obtained from the Middlebury evaluation page<sup>1</sup>.

Image Pair	image width	image height	disparity min	disparity max	scale
Tsukuba	450px	375px	0	15	16
Venus	450px	375px	0	19	8
Teddy	384px	288px	0	59	4
Cones	434px	383px	0	59	4

Table 7.1: Characteristics of Middlebury stereo data set.

All tests were run on the color input images with HSV color channel weights 0.2, 0.3, and 0.5. The pyramid version was run with three pyramid levels on top of the original image. These parameters are slightly optimized towards the Tsukuba image pair, since this was the image pair mostly used during the development phase. The Middlebury evaluation page requires that the algorithm is run with unmodified parameters for all four image pairs.

---

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



Figure 7.1: Input stereo pairs: Cones, Teddy, Tsukuba, Venus with groundtruth

### 7.1.1 Window Based Stereo Kernel Results

Window Size	Cones	Teddy	Tsukuba	Venus
3x3	36.5%	45.2%	22.5%	43.0%
5x5	23.0%	33.4%	14.9%	31.9%
7x7	18.2%	27.8%	11.8%	25.3%

Table 7.2: Window Based Kernel: Erroneous pixels in non occluded areas depending on window size

As explained in the previous section. The input parameters were slightly optimized for the Tsukuba image pair. For this reason the Tsukuba error rates are slightly better than in the other input images. Figure 7.2 shows the disparity generated for all three window sizes. In Figure 7.3 the absolute and signed errors are visualized<sup>2</sup>.

Table 7.2 clearly shows how error rates drop with increasing window size. The algorithm simply has a larger area to back its decision on the correspondence problem. The same effect can be observed when looking at the disparity maps generated. In figure 7.2 from left to right along with the direction of increasing window size the images become smoother.

On the other hand low-textured and featureless areas produce many outliers (see figure 7.8 Teddy and Venus example). Subjectively only the 7x7 window examples produces reasonably sufficient disparity maps.

---

<sup>2</sup>Absolute and signed error images are generated by the Middlebury evaluation page

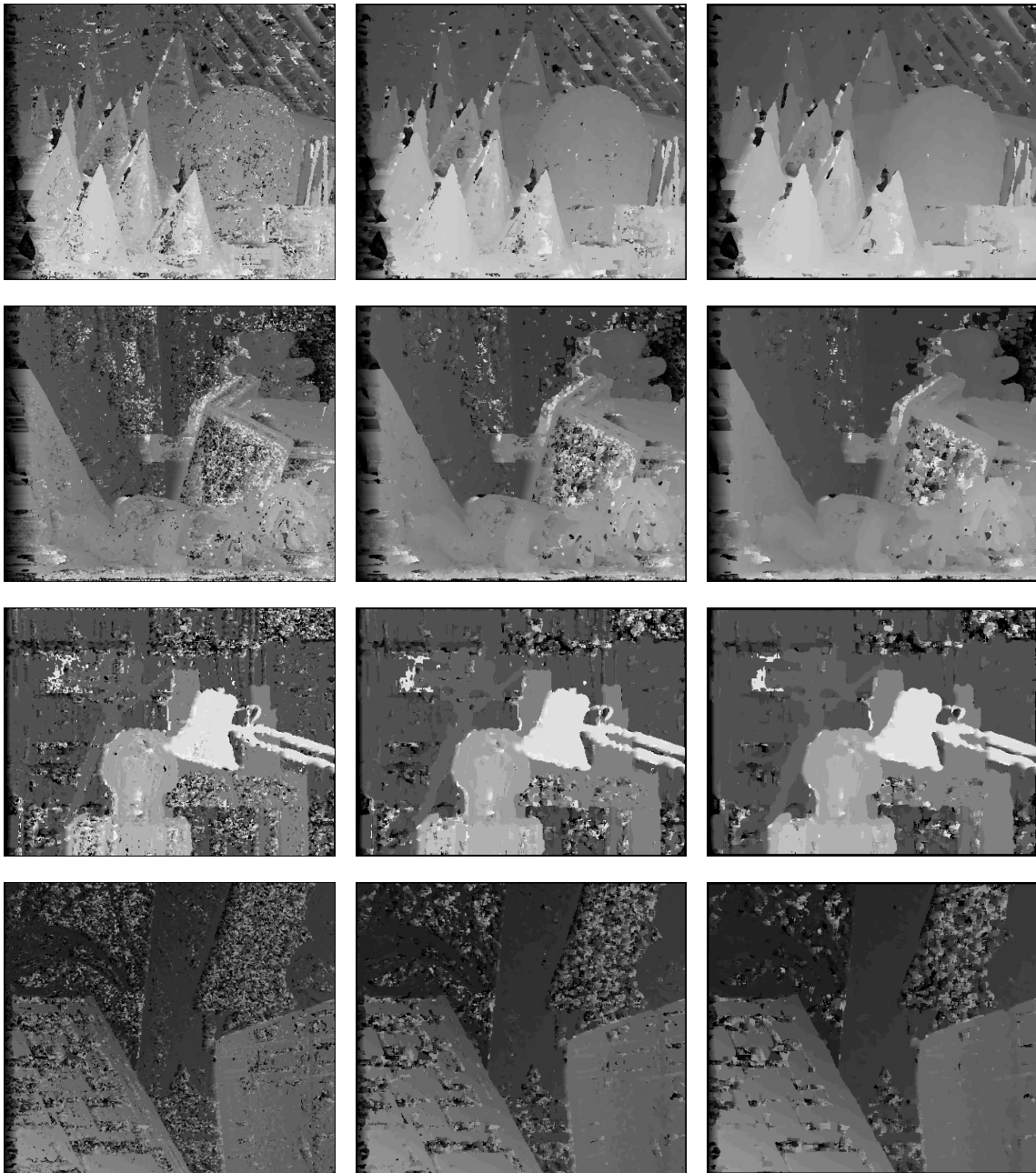


Figure 7.2: Window Based Kernel: Disparity Maps 3x3, 5x5, 7x7



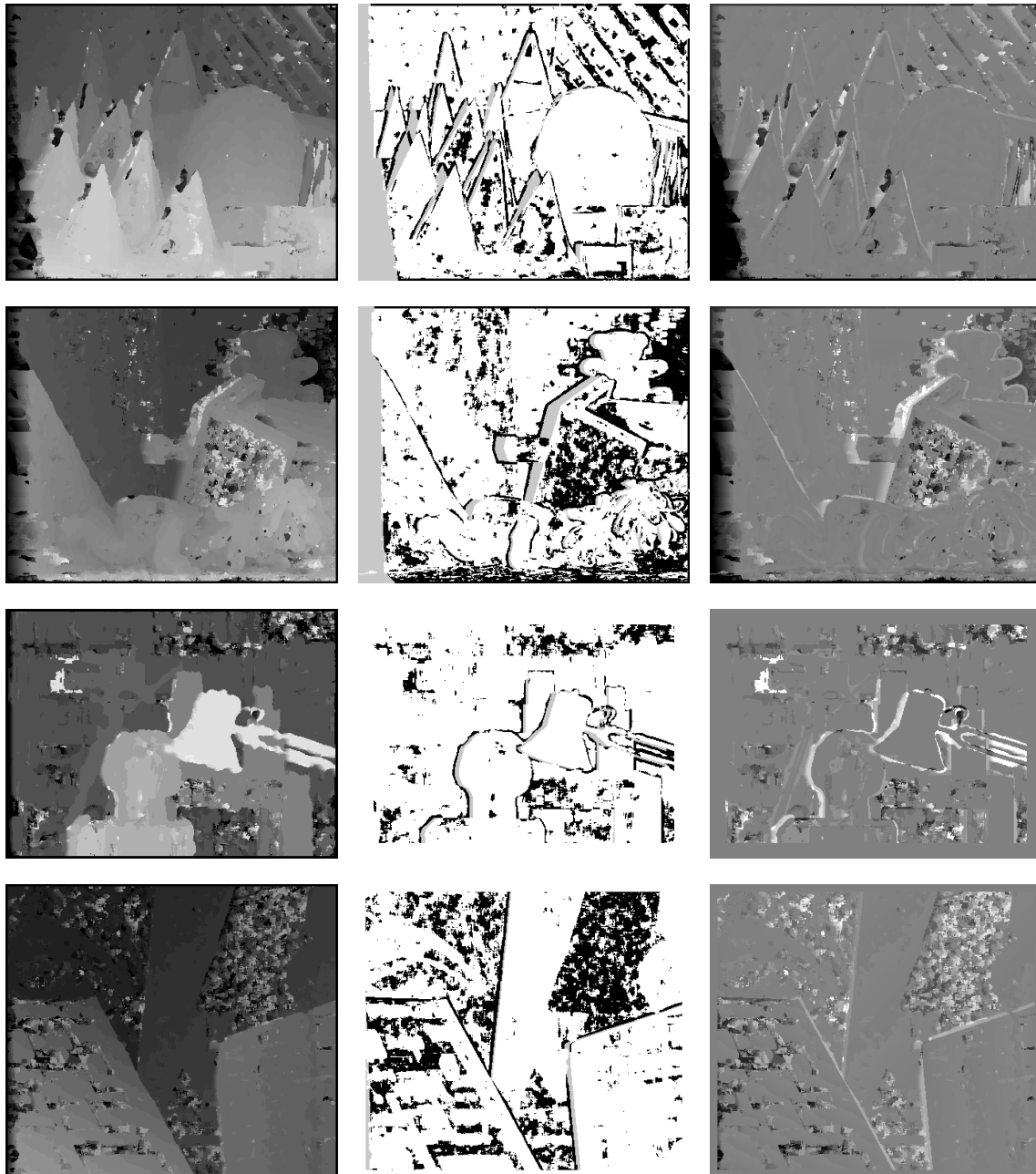


Figure 7.3: Window Based Kernel: disparity map  $7 \times 7$ , absolute error, signed error

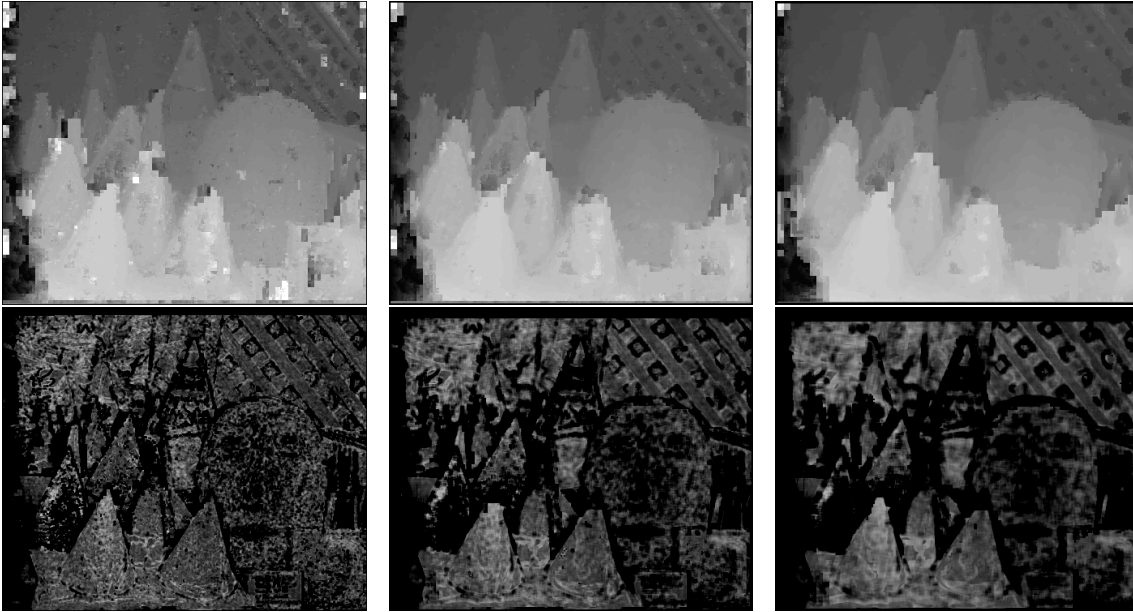


Figure 7.4: Pyramid Based Kernel: Cones - disparity and confidence maps 3x3, 5x5, 7x7

### 7.1.2 Pyramid Based Stereo Kernel Results

For the stereo image dataset in use a “stride max” parameter of eight was determined by experiments to yield the best quality results. This corresponds to three levels of scaled down copies on top of the original image.

Window Size	Cones	Teddy	Tsukuba	Venus
3x3	27.6%	36.0%	14.3%	32.7%
5x5	21.3%	29.2%	10.7%	24.3%
7x7	19.3%	26.5%	9.67%	20.3%

Table 7.3: Pyramid Based Kernel: Erroneous pixels in non occluded areas depending on window size

Due to the post-processor the resulting disparity maps appear to be smoother. Meanwhile, errors in disparity maps are quite well predicted in the confidence maps. Subjectively a 5x5 window already seems sufficient.

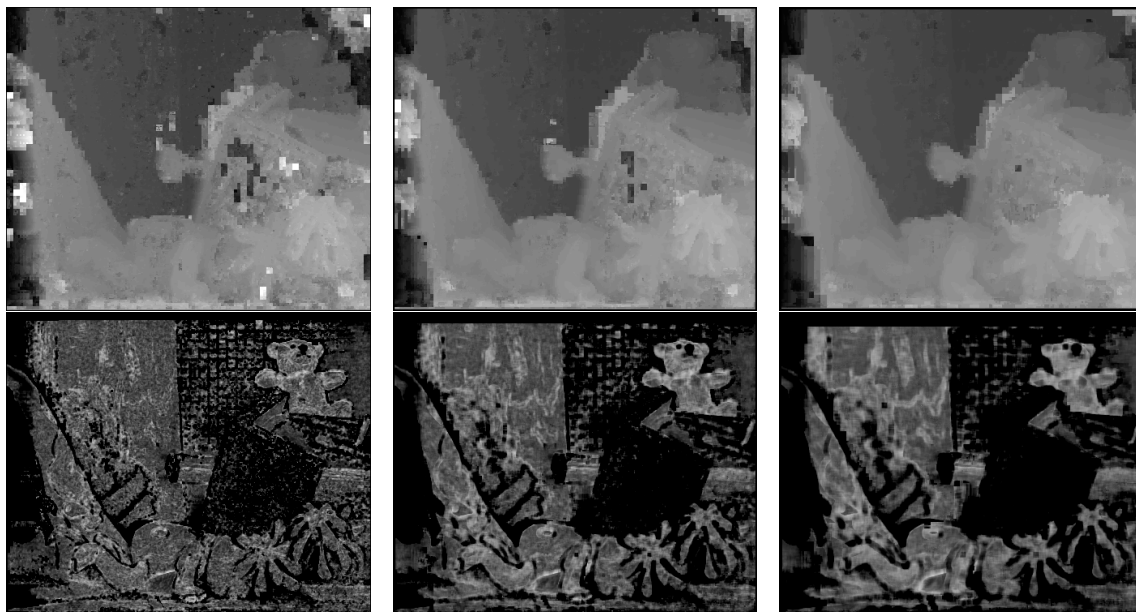


Figure 7.5: Pyramid Based Kernel: Teddy - disparity and confidence maps 3x3, 5x5, 7x7

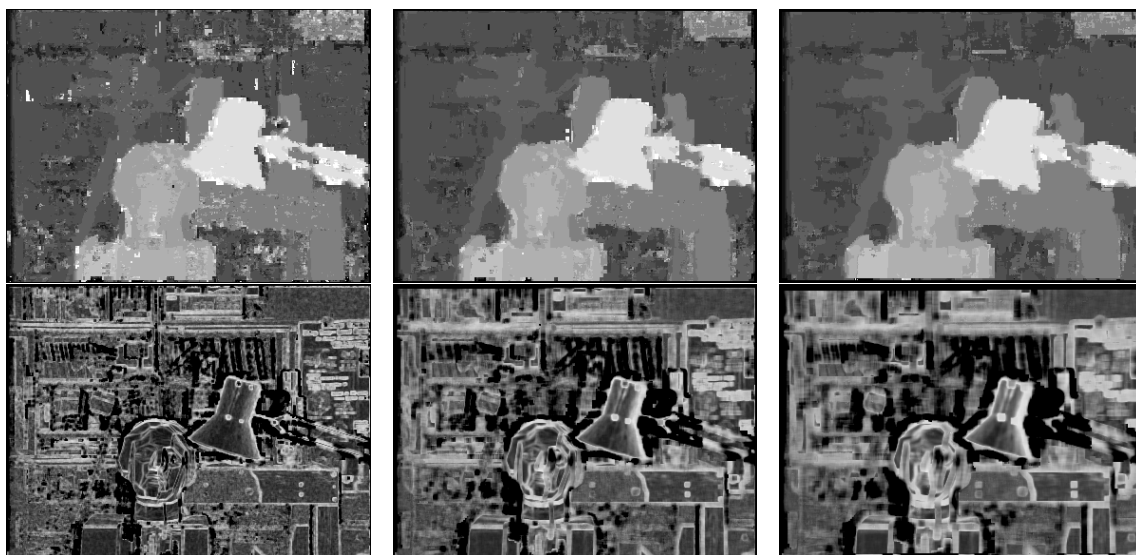


Figure 7.6: Pyramid Based Kernel: Tsukuba - disparity and confidence maps 3x3, 5x5, 7x7

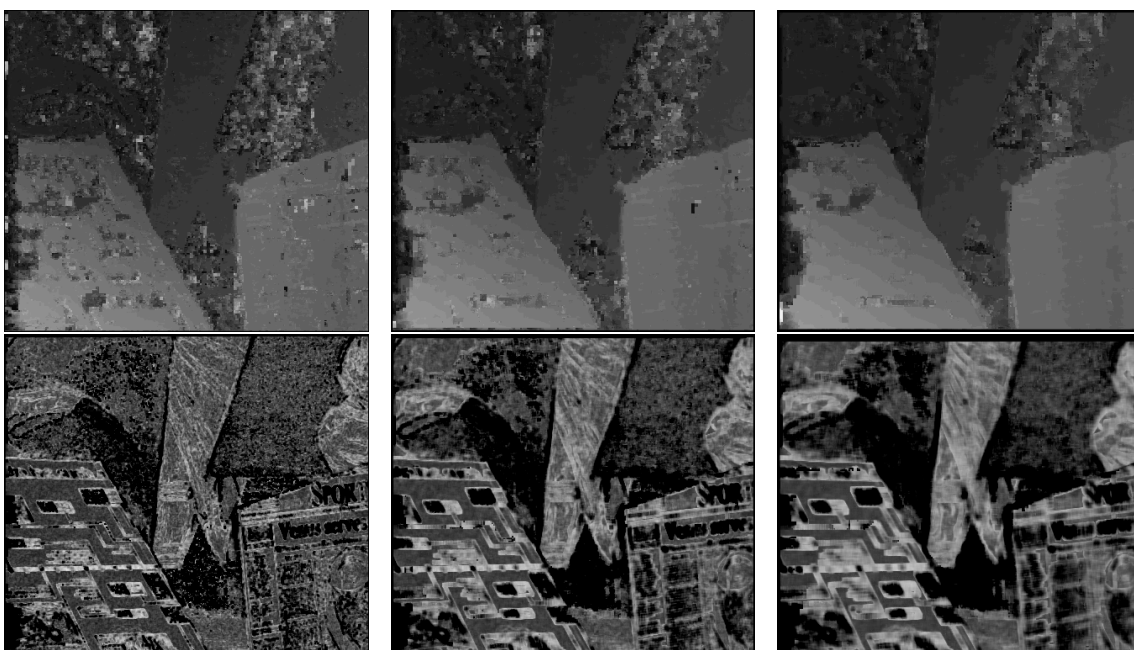


Figure 7.7: Pyramid Based Kernel: Venus - disparity and confidence maps 3x3, 5x5, 7x7

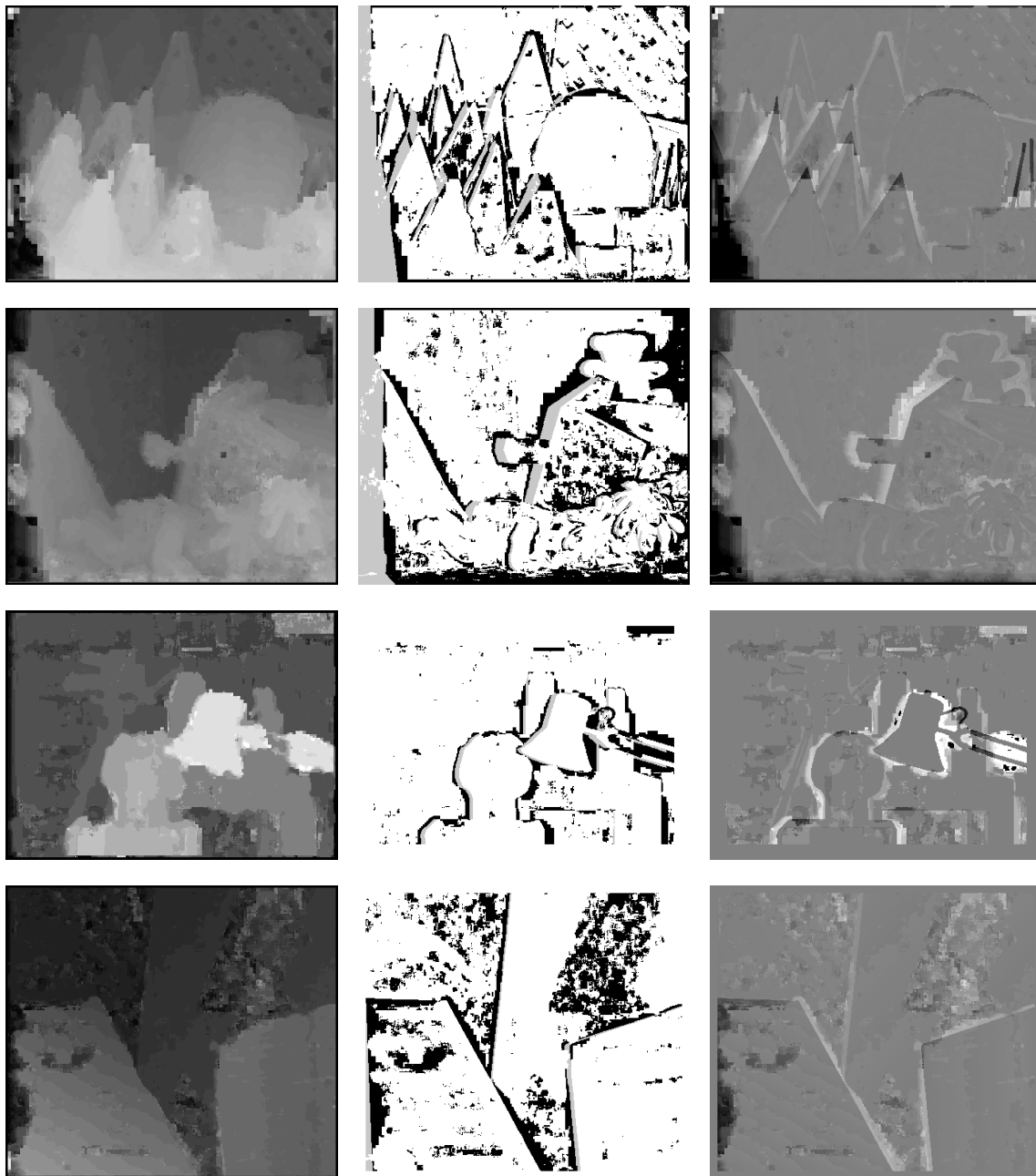


Figure 7.8: Pyramid Based Kernel: disparity map 7x7, absolute error, signed error

### 7.1.3 Quality

The Pyramid based stereo algorithm was expected to produce better quality results, which could be confirmed by the data gathered (see figure 7.9). Especially errors due to repetitive textures and featureless areas are reduced.

The magnitude of how the error rate decreases with window size is higher in the window based version. Due to the virtually larger larger window sizes on the scaled down copies the pyramid version is less dependent on the window size parameter.

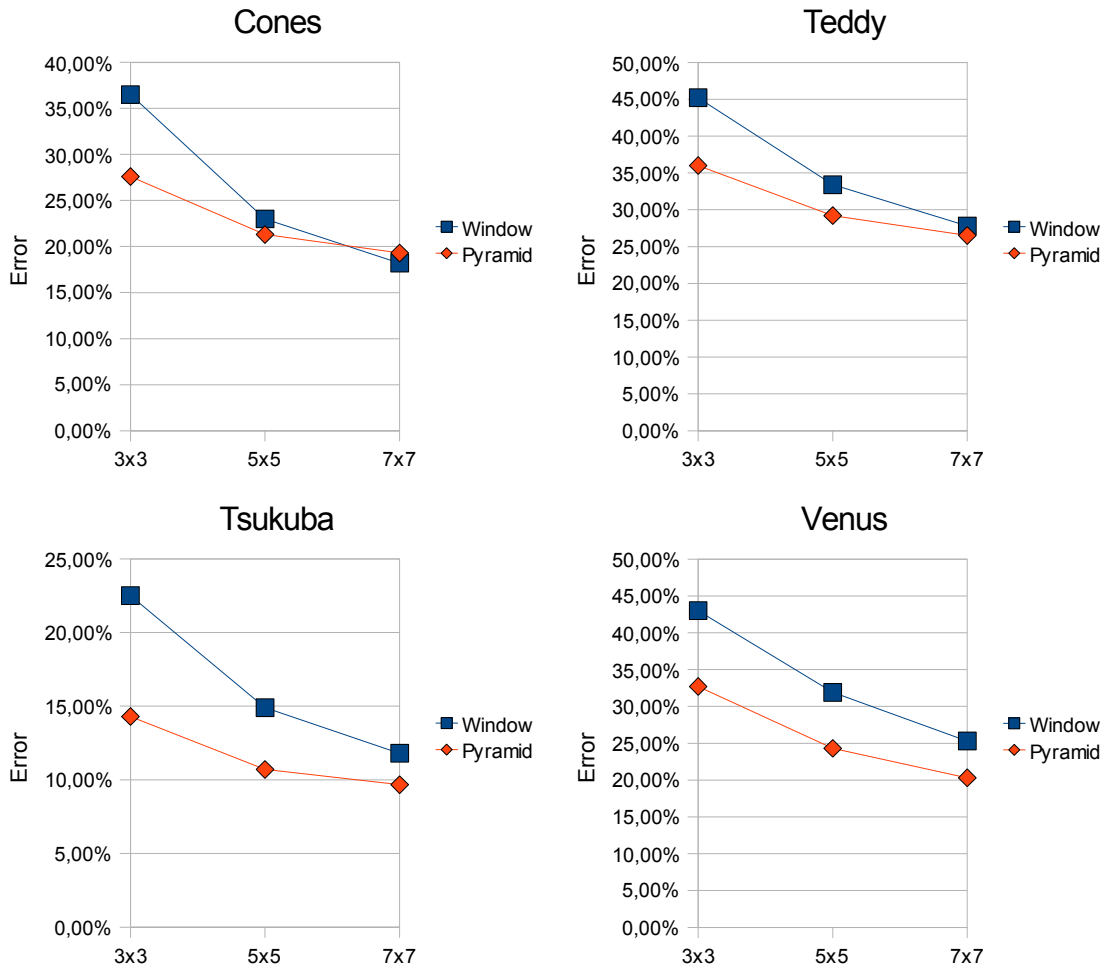


Figure 7.9: Quality of Window and Pyramid Based Algorithm

The

### 7.1.4 GPU Comparison

In order to find out how performance scales with the graphics hardware used the GPU software was tested with two systems.

The scale down performance was tested with a large 1600x1200 Pixel image. It was found that the computation speed is not directly proportional to the number of multiprocessor (see table 7.4). The scale down kernel is memory bound. Therefore the number of multiprocessors has no large impact.

Graphics Board	Multiprocessors	CPU	Scale Down Kernel
8600 GT	2	Intel(R)Pentium(R) D 3.40GHz	4.42ms
8800 GTS	12	AMD Athlon(tm) 64 3200+	3.31ms

Table 7.4: Scale Down Kernel: GPU Comparison (timings: average of three runs)

In comparison of window based and pyramid based kernels (see figure 7.10) the advantage of a higher multiprocessor count has more impact. With increasing window size the computation becomes more calculation intensive. This results in the kernel being computationally bound. A higher number of multiprocessors results in more concurrency and enables CUDA to hide the memory latency better.

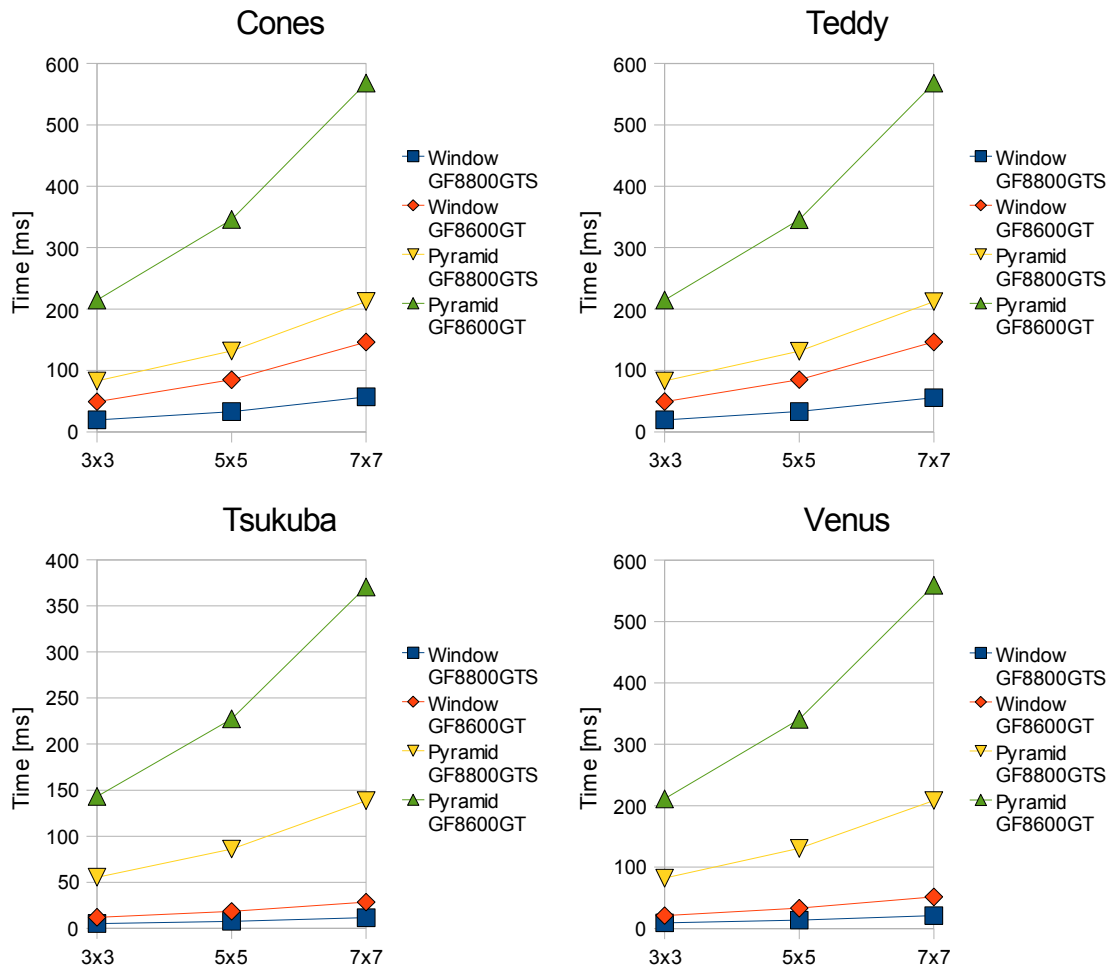


Figure 7.10: GPU Comparison



## 7.2 Timings and Optimization Techniques

Figure 7.11 shows the distribution of execution time for a disparity map calculation. The time for scaling, memory copies, and post processing are not dependent on window size but constant. Besides that the steps taken in these phases are extremely fast. Obviously the pyramid kernel itself is most promising as a subject to optimization.

During the course of implementation it was found that a lot of time was wasted due to casts between data types (*unsigned int*, *int*, *float*). On a GPU floating point operations are fairly cheap. A floating point multiplication (4 clock cycles) is even faster than a regular 32bit integer multiplication (16 clock cycles). This resulted in two optimization patterns:

- Avoiding unnecessary casts in favor of more general data types or floating point data types (use of *int* instead of *unsigned int*, use of *float* whenever possible). This resulted in smaller kernels.
- Instead of 32bit integer multiplication CUDA offers a 24bit variant of unsigned and signed integer multiplication that computes within 4 clock cycles. 24bit precision was sufficient in all cases of the presented CUDA code.

During the implementation of the pyramid kernel two textures were used at first, similar to the first window kernel implementation. One shortcoming of the current CUDA framework is that no texture reference variables are supported in the kernel. For that reason the reads from the two textures could not be performed in parallel but needed to be serialized. This being a crucial bottleneck was avoided by putting both input images next to each other into one texture.

Considering the findings of chapter 5.2 it really only makes sense to run the algorithms with window sizes 3x3, 5x5, or 7x7. This also means that the kernels can be optimized for these exact window sizes. Computation on the GPU is only efficient if the control flow of the programs is quite simple. Constant loop variables help the CUDA compiler to predict the execution better and to unroll the loops as required. Therefore three new CUDA kernels were written. Each calling the old kernel, which was marked `__device__` instead of `__global__` with constant window size parameters. This has the effect that the old kernel gets inlined into the three new kernels. For each new kernel the compiler produces an optimized kernel.

The same trick was applied to the disparity range parameter. While the disparity range of the first pyramid kernel run on the smallest pyramid level can not be predicted, the disparity range of all following pyramid steps is already known (constant 4). In total there are now six kernels, three for dynamic disparity ranges and three for the constant disparity range four.

The optimization results are depicted in figure 7.12. Timing measurements on window and pyramid based runs are compared in figure 7.13 which is based on data tables 7.5, 7.6, 7.7, 7.8.

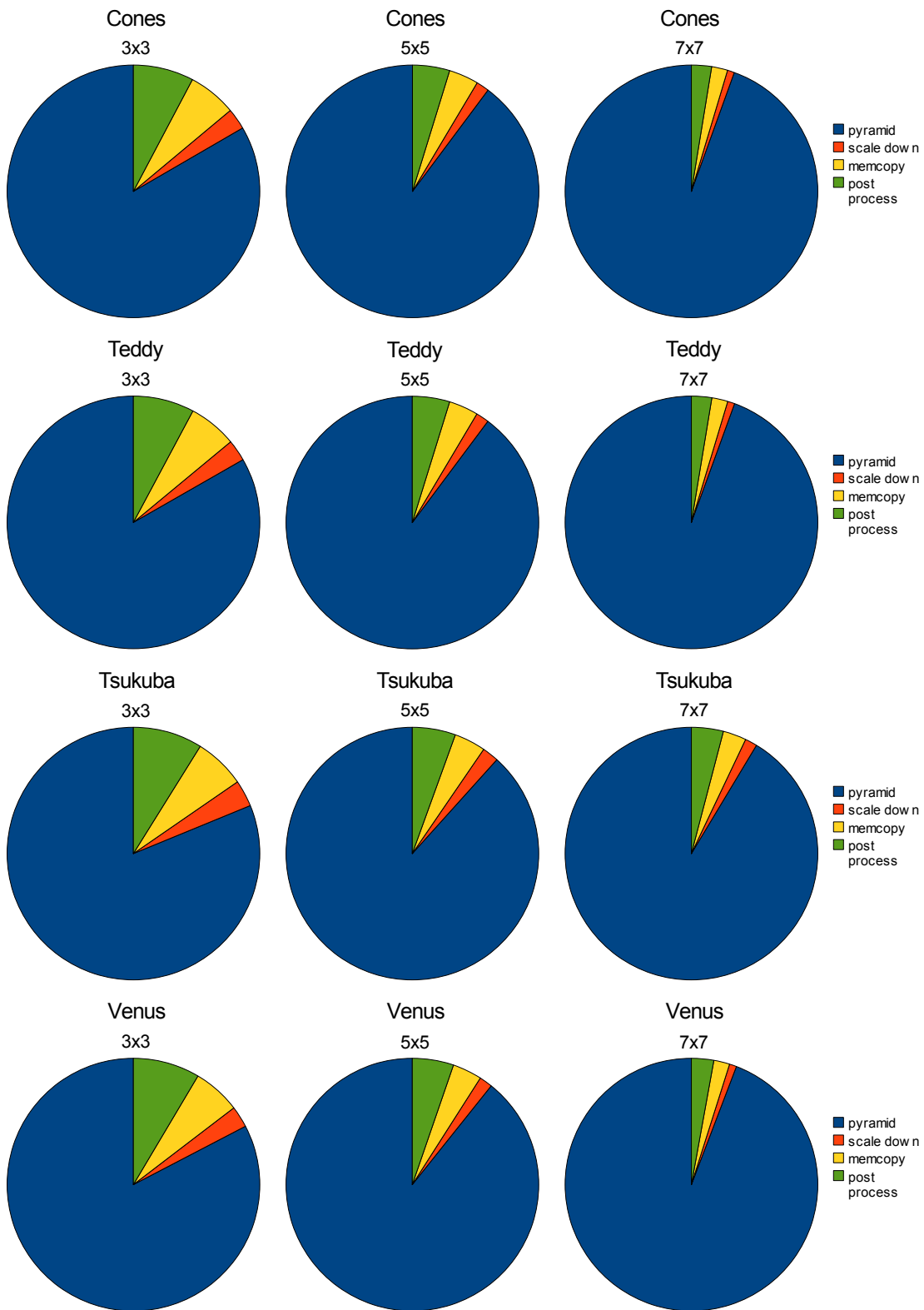


Figure 7.11: Distribution of execution time on algorithm phases. (run on GeForce 8800 GTS)

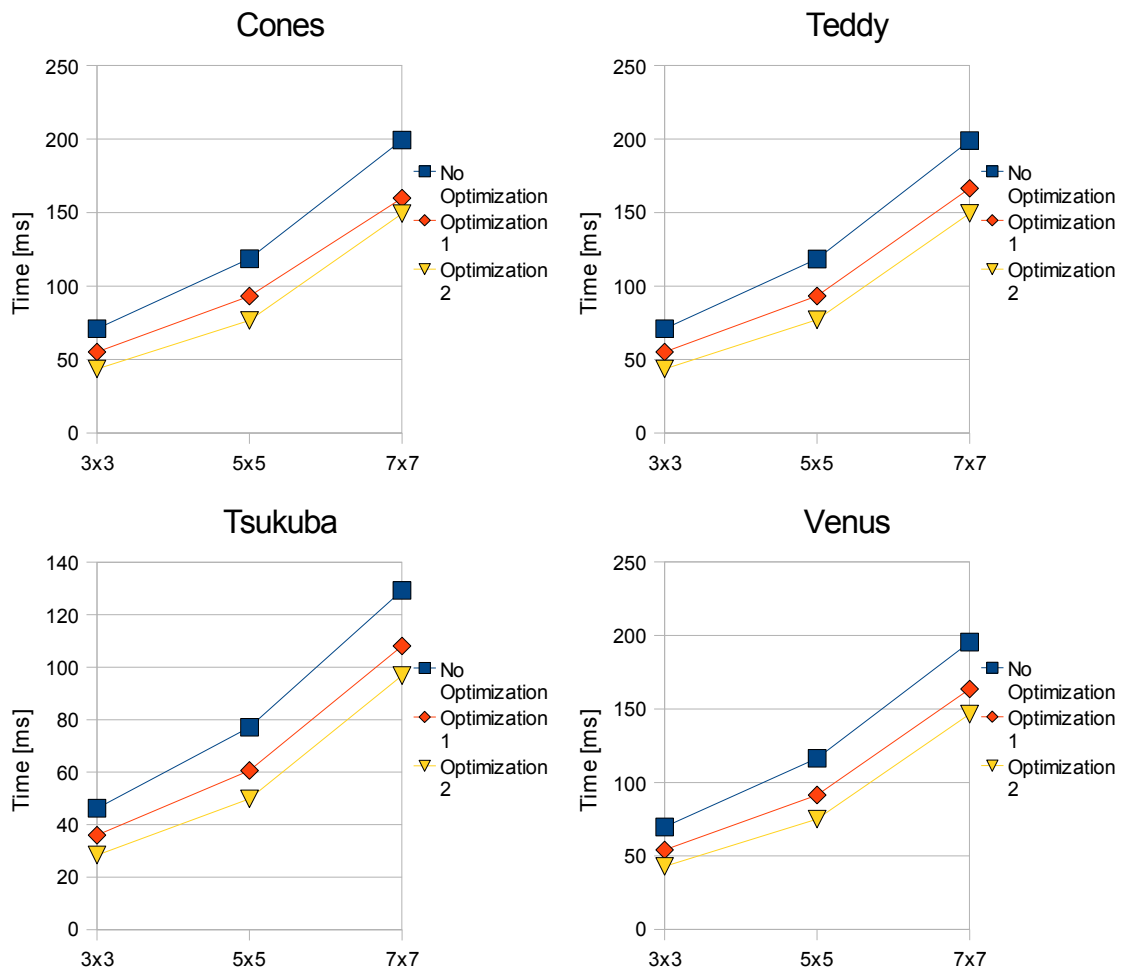


Figure 7.12: Optimization (run on GeForce 8800 GTS)

Cones	Window	Opt. Pyramid Kernel	Opt. Pyramid Total
3x3	19.54ms	43.64ms	55.50ms
5x5	32.93ms	76.65ms	89.23ms
7x7	56.94ms	149.28ms	161.48ms

Table 7.5: Averaged execution timings (run on GeForce 8800 GTS)

Teddy	Window	Opt. Pyramid Kernel	Opt. Pyramid Total
3x3	19.48ms	43.64ms	55.59ms
5x5	32.98ms	77.09ms	93.55ms
7x7	55.54ms	149.42ms	162.52ms

Table 7.6: Averaged execution timings (run on GeForce 8800 GTS)

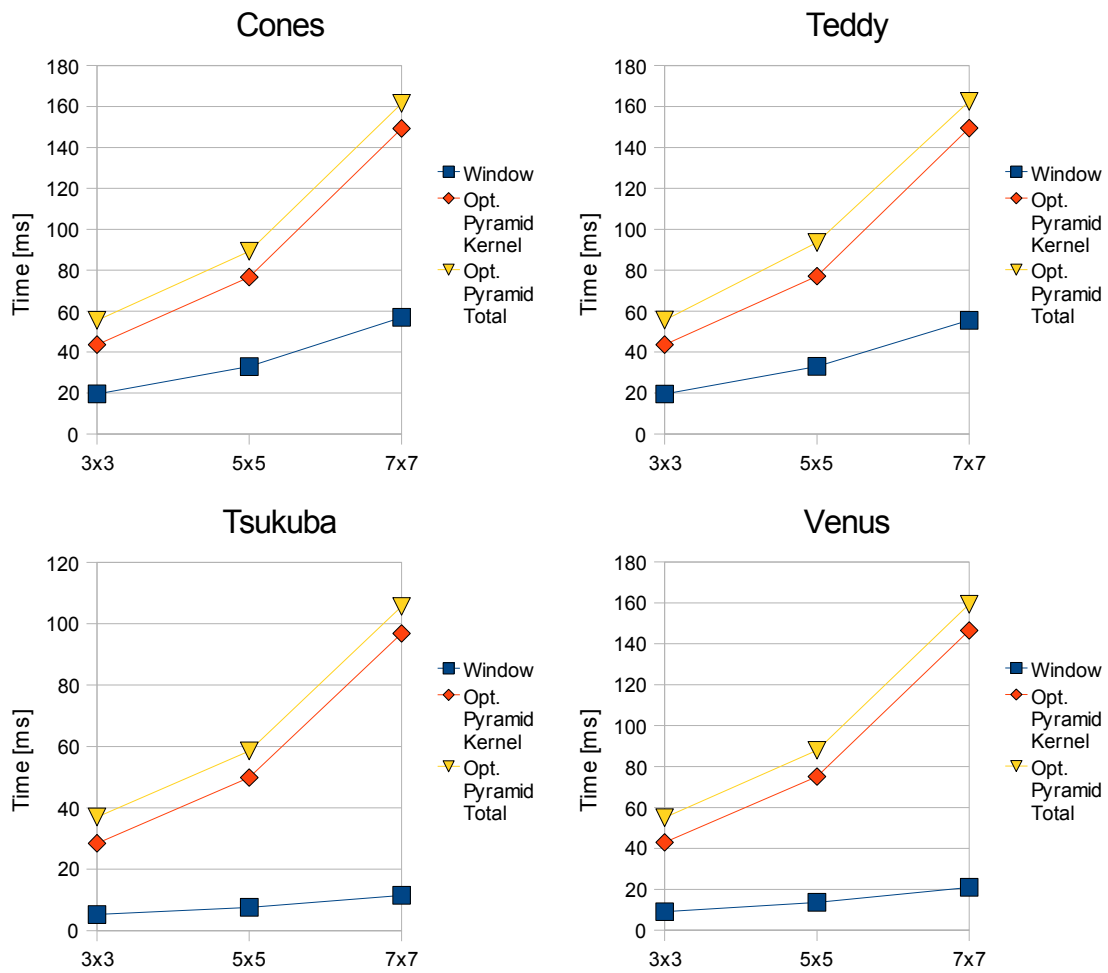


Figure 7.13: Execution speed (run on GeForce 8800 GTS)

Tsukuba	Window	Opt. Pyramid Kernel	Opt. Pyramid Total
3x3	5.22ms	28.44ms	37.00ms
5x5	7.56ms	49.84ms	58.52ms
7x7	11.48ms	96.81ms	105.62ms

Table 7.7: Averaged execution timings (run on GeForce 8800 GTS)

Venus	Window	Opt. Pyramid Kernel	Opt. Pyramid Total
3x3	9.11ms	42.94ms	54.99ms
5x5	13.68ms	75.12ms	87.98ms
7x7	20.98ms	146.53ms	159.34ms

Table 7.8: Averaged execution timings (run on GeForce 8800 GTS)

## 8. Conclusion

Besides the implementation of a stereo vision algorithm on the GPU, goal of this work was to evaluate CUDA. It was found that once everything was installed it was quite easy to get started with the CUDA example programs (especially the convolution example [Howes 07]). Leveraging the tools right, e.g. using the nvcc compiler and the debugging and simulation features were much more complicated. Debugging and simulation modes must be used carefully. Without debugging mode kernels will fail silently. A kernel succeeding in simulation mode is not guaranteed to work on the card. Most often the failure on the real hardware was due to concurrency issues, which were impossible to discover in simulation. Most issues however could be resolved by reading the *CUDA Programming Guide* ([NVIDIA 07]) thoroughly or by seeking help at the NVIDIA developer forums. Once everything was setup and ported from *make* to *Scons*, development in CUDA worked like a charm, because of the seamless transition between CPU and GPU C-programming.

The new technology enables near real-time performance. The stereo kernels have been proven to run fast and scalable. However it is not easy to predict performance ahead of time. Many unknown variables are in play: memory access, kernel configuration parameters, kernel register count, etc. Optimal results had to be found by empirical testing.

Looking at the results it was found that the new CUDA framework is very promising for the stereo vision field of application.

Due to the window-based principle, the stereo-vision algorithms presented here can not compete against the state of the art stereo algorithms presented at the Middlebury evaluation page in terms of error rates. However, these algorithms do not have to fulfill the same speed requirements.

In a near real-time approach it is more important to generate disparity map fast, than to elaborate on a perfect match to long. Via the confidence map it is then possible to decide whether the match was sufficiently good or the result is to be thrown away.

The window-based implementation is fast, but produces low quality results. The pyramid version is slower, but scales better with window size. It also produces better results and generates the a confidence map.

Depending on speed and quality requirements it may be advisable to choose the simple window-based approach over the pyramid approach. The experiments showed that on larger window sizes the difference in quality results becomes smaller. However it is not fair to compare both algorithms, since the loss in speed is due to the generation of the confidence map.

Since all top ranking stereo algorithms at the Middlebury evaluation page use color segmentation and optimization approaches, further work in this field should be conducted porting a global optimization approach to the CUDA hardware.

# Bibliography

- [Howes 07] Lee Howes. Image Convolution with CUDA, Mar 2007.
- [Koch 07] J. Koch, M. Anastasopoulos, K. Berns. Using the Modular Controller Architecture (MCA) in Ambient Assisted Living. 3rd IET International Conference on Intelligent Environments (IE07), Sep 2007.
- [Kostkova 02] Jana Kostkova. Stereoscopic Matching: Problems and Solutions, 2002.
- [Kuhn 03] Michael Kuhn, Stephan Moser, Oliver Isler. Stereopsis: Depth mapping using stereo vision, 2003.
- [NVIDIA 07] NVIDIA. NVIDIA CUDA Compute Unified Device Architecture Programming Guide 1.0. [http://developer.download.nvidia.com/compute/cuda/1\\_0/NVIDIA\\_CUDA\\_Programming\\_Guide\\_1.0.pdf](http://developer.download.nvidia.com/compute/cuda/1_0/NVIDIA_CUDA_Programming_Guide_1.0.pdf), 2007.
- [Schäfer 04] Bernd Helge Schäfer. Security Aspects of Motion Execution in Outdoor Terrain. Mastersthesis, Kaiserslautern Institute of Technology, Sep 2004.
- [Scharstein 02] Daniel Scharstein, Richard Szeliski. A Taxonomy and Evaluation of Dense Two-Frame Stereo Correspondence Algorithms, 2002.
- [Schnabl 03] Thomas Schnabl. Geometrie der Zweikamera- und Mehrkamera- Stereovision, 2003.
- [Scholl 02] K.-U. Scholl, B. Gassmann, J. Albiez, J.M. Zöllner. MCA - Modular Controller Architecture. Robotik 2002, VDI-Bericht 1679, 2002.
- [Seidler 08] Benjamin Seidler. Texture-based Terrain Traversability Estimation for Outdoor Robot Navigation (preliminary title). Mastersthesis, Kaiserslautern Institute of Technology, 2008.
- [Yang 05] Ruigang Yang, Marc Pollefeys. A Versatile Stereo Implementation on Commodity Graphics Hardware, 2005.
- [Zolynski 07] Gregor Zolynski. GPULBP: Implementing the Local Binary Pattern Operator on Graphics Processing Hardware, Oct 2007.

