Real time sparse and dense stereo in an early cognitive vision system using CUDA

Master’s thesis

The Maersk Mc-Kinney Moller Institute
Faculty of Engineering
University of Southern Denmark

handed in by

Jeppe Barsøe Jessen

01.09.2009

Supervisor : Norbert Krüger
Co-supervisor : Lars B. W. Jensen
               Anders Kjær-Nielsen
Abstract

This master’s thesis proceeds the work done in my FORK. Here we investigate how the introduction of a CUDA enabled graphics card can speed up the algorithms involved during the stereo computation of the CoViS framework. Stereo matching and reconstruction in general tend to be computationally heavy, and the support for parallel execution motivates us to transfer known algorithms to the massively parallel platform. The dense stereo algorithms have already proven (e.g. in the FORK) that they often fit the parallel environment nicely, so we investigate how this information can be integrated in an existing vision framework. Sparse stereo is more challenging but in spite of suboptimal memory access patterns and varying levels of parallelism we manage to achieve a moderate speed up.
# Contents

## Contents

<table>
<thead>
<tr>
<th>1 Introduction</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Problem statement</td>
<td>2</td>
</tr>
<tr>
<td>1.2 Report outline</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2 CoViS - an early cognitive vision framework</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Cognitive vision stages</td>
<td>5</td>
</tr>
<tr>
<td>2.2 Representing visual features</td>
<td>6</td>
</tr>
<tr>
<td>2.3 Intrinsic dimensionality</td>
<td>7</td>
</tr>
<tr>
<td>2.4 Using both sparse and dense stereo</td>
<td>9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>3 Computing in parallel</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Choice of hardware accelerator</td>
<td>12</td>
</tr>
<tr>
<td>3.2 FPGA versus GPU</td>
<td>12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4 Dense stereo</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Algorithm in CUDA</td>
<td>15</td>
</tr>
<tr>
<td>4.2 Introduce the texel</td>
<td>16</td>
</tr>
<tr>
<td>4.3 Texel computation</td>
<td>16</td>
</tr>
<tr>
<td>4.4 Test</td>
<td>17</td>
</tr>
<tr>
<td>4.5 Discussion</td>
<td>20</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>5 Sparse stereo</th>
<th>21</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 Algorithm analysis</td>
<td>21</td>
</tr>
<tr>
<td>5.2 Algorithm mapping to CUDA kernels</td>
<td>30</td>
</tr>
<tr>
<td>5.3 CUDA implementation</td>
<td>35</td>
</tr>
<tr>
<td>5.4 Test</td>
<td>39</td>
</tr>
<tr>
<td>5.5 Discussion</td>
<td>44</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>6 Conclusion</th>
<th>47</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1 Future work</td>
<td>48</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Glossary</th>
<th>50</th>
</tr>
</thead>
</table>

| Bibliography                          | 51 |

| Appendices                            | v  |

| List of Figures                       | vii|

---
A  Introduction to CUDA  

B  Integrating CUDA in an existing C++ project
# List of Figures

2.1 Multi-modal primitive template. ......................................................... 6
2.2 Textured box used for testing sparse and dense stereo. ................... 7
2.3 Line segment and contour representations of a small box. ............... 8
2.4 Intrinsic dimensionality examples. .................................................. 9

4.1 Dense stereo algorithms comparison and intrinsic dimensionality. ... 18
4.2 3D features visualisation of the box shown in figure 4.1. ................. 19

5.1 The sparse stereo algorithm. ............................................................. 23
5.2 Matching algorithm. ................................................................. 25
5.3 Epipolar geometry. ................................................................. 26
5.4 Epipolar line constraints. ........................................................... 27
5.5 The sparse stereo algorithm with different levels of disambiguation. ... 29
5.6 Grid of thread blocks for the matching kernel. ............................... 32
5.7 Stereolink data structure ............................................................. 34
5.8 Matching step on CPU or GPU. ..................................................... 43
5.9 Reconstruction step on CPU or GPU. ........................................... 43
5.10 Complete sparse stereo on CPU or GPU. ...................................... 44

A.1 CUDA grid of threads. ................................................................. 55
A.2 CUDA hardware model. .............................................................. 56

B.1 Compiler perspective ................................................................. 59
Chapter 1

Introduction

The human eye and brain solves the fundamental task of visual perception – or just vision – so elegantly, but yet so hard to emulate. Humans perceive and interpret the surroundings using multiple senses and vision plays a key role occupying a huge part of the brain. The needed certainty and completeness for precise actions in the 3D world are achieved with impressive speed and regardless of a rather poor lens, noise, ambiguous structures and missing information.

Thus, for a computer vision system, inspiration has been found in the human visual system. Computer algorithms have been proposed that, starting from the raw image, introduces a series of steps taking us incrementally to higher levels of abstraction. Feedback mechanisms combine several modalities with spatial and temporal interdependencies to disambiguate local image structure interpretation. The ultimate goal is not just a 3D model of our surroundings but also a basic understanding of objects, boundaries and free space in the scene. That is one suggested path towards what could be considered a cognitive computer vision system.

A powerful and fast cognitive vision system would be of great value in many areas, but in particular for guiding vehicles and robots. Whether the objective is to avoid monotonous labour or make automatically guided systems safer, cameras and 3D range scanners can provide the input data. It is however a noisy and incomplete signal which leads us to the problem of finding suitable algorithms and process data fast enough. Some algorithms might even be readily available but are too computationally heavy. The timing aspect is crucial as vehicle guidance and today’s robot cells require real-time decision making for safety and efficiency. Furthermore, the future will bring increasingly challenging environments and timing demands.

This master’s thesis proceeds the work done in my preliminary thesis study (FORK) [1] where different parallel platforms were examined for speeding up specific vision algorithms. Focus will now be twofold on the following subtasks:

1. Try to speed up the intermediate step of an early cognitive vision system named sparse stereo using line segments. The speed up must be attained through reimplementation of an existing algorithm using different levels of parallelism and distributing the work between Central Processing Unit (CPU) and Graphics Processing Unit (GPU) platforms.
2. Use a dense stereo algorithm implemented for the GPU in [1] to introduce a new type of 3D feature – the texel – in an early cognitive vision system. Investigate whether the dense approach can add reliable visual features to the general representation of the scene.

1.1 Problem statement

The following problem statement for this master’s thesis was derived during the FORK project:

Implement the sparse stereo algorithm in CUDA for execution on GPUs as described conceptually in the FORK report. This involves the creation of data structures for efficient storage of Cognitive Vision Software (CoViS) 2D and 3D primitives on the GPU memory, allowing for parallel processing. One or more CUDA kernels should be developed and integrated in CoViS, and the performance effects documented.

Furthermore the dense stereo algorithm used and implemented in FORK should be integrated in CoViS. This will introduce a new type of 3D primitive that can be extracted in highly textured surface areas of the image, which does not resemble a contour. The disparity map holds the necessary information for the integration, but the creation of primitives could also depend on the minimum Sum of Squared Difference (SSD) value used as a confidence, as it indicates the quality of the best match. The target platform for the dense stereo algorithm will also be the GPU as it performs nicely and has better availability than the FPGA – especially as the GPU is chosen for the sparse stereo algorithm.

1.2 Report outline

This report describes the interesting challenges we faced and considerations made during the master’s thesis project. To fully understand the subject, prior knowledge of NVIDIA GPUs and CUDA is an advantage. You find information on this matter in [2, 3, 4, 5]. Simple vision algorithms are also considered basic knowledge and supplemental reading on the CoViS framework [6, 7] can be beneficial. To ease in the understanding of esoteric terms a glossary is appended in the back of the report.

Chapter 1 Introduces the project and the problem statement and provides a report outline.

Chapter 2 CoViS is introduced in the context of this report and our motivation is described.

Chapter 3 Introduces the concept of parallelism and how different parallel architectures have been compared for our vision applications.

Chapter 4 The work done on the integration of texels into CoViS is shortly described and tested.

Chapter 5 The sparse stereo algorithm as analysed, mapped to the CUDA framework, implemented and tested.
Chapter 6 Conclusion of the project and future work

In the appendices you find a brief introduction to common CUDA concepts and an overview of how CUDA functionality has been integrated in the existing C++ application.
Chapter 2

CoViS - an early cognitive vision framework

The Cognitive Vision Lab (CoViL) at the University of Southern Denmark has an ongoing software development project called CoViS, written in C++. It constitutes the backbone of many research activities in the area of cognitive computer vision. One important area of interest is to speed up the core components by utilising parallel processing devices such as multi-core CPUs and many-core GPUs.

A complete introduction to the software project is outside the scope of this report and is better given elsewhere [6, 8]. In this chapter we focus on some basic terminology and concepts which are required to understand the setup in the context of this project.

2.1 Cognitive vision stages

First we list some native stages of a cognitive vision system inspired by the human visual system. For a general introduction we refer to [9] where the stages are introduced and described in detail. Here we map the functionality of CoViS among the different stages and note how the level of abstraction increases:

Preprocessing is the initial image preparation such as undistortion and rectification.

Early vision covers linear and non-linear filter operations as e.g. monogenic filtering that uses the image as a two dimensional signal. In CoViS we compute magnitude, orientation and phase, which is ideal for edge detection. This is then also used for the computation of intrinsic dimensionality as described in section 2.3, which adds for feature classification of image areas. This is all done on a very local scale and is thus highly influenced by noise and ambiguities.

Early cognitive vision is an intermediate step that deals with contextual disambiguation. In CoViS we provide a condensed symbolic representation of complex image features in 2D and scene features in 3D.
Cognitive vision derives a high level understanding of the scene. This is still an area of research and the problem in general appears unsolved.

Quite some work has already been put into the task of speeding up CoViS using different parallel platforms such as multi-core CPUs and many-core GPUs. All preprocessing and early vision functionality in CoViS has thus been successfully speeded up using NVIDIA’s CUDA framework for GPUs. A short introduction to the framework is given in appendix A and more information is found in [2].

The parallel nature of simple image processing and the straightforward pixel operations on rectangular data structures makes the GPU an ideal platform for this purpose. Considering the next step – the early cognitive vision stage – the problems can still be addressed highly in parallel but the data structures are no longer as static and rectangular. Visual features are extracted, requiring dynamic data structures and more divergent execution paths for the linking into groups and matching into stereo. This does not prevent an efficient GPU implementation, but it certainly makes it more challenging to determine whether the GPU is the optimal platform for the computation and to actually implement it and achieve a speed up that justifies the effort.

2.2 Representing visual features

In CoViS the visual features are represented in a compact way using local symbolic descriptors named visual primitives. From this point on just primitives for convenience.

A primitive represents a local image feature and until now only line segment features have been thoroughly investigated and integrated in CoViS. Figure 2.1 shows a template of a primitive and the associated attributes describing a small line segment in the image. The semantic information of the attributes are position \((x, y)\), orientation \(\theta\), phase \(\omega\) and colour \((c_l, c_m, c_r)\) which will be described further in chapter 5. The semantics are subject to change depending on the type of primitive and the context.

![Figure 2.1: Multi-modal primitive template representing a line segment in an image.](image)

Figure 2.2 shows an image of a highly textured box. Line segment representations of the image, extracted at different scales, are shown in the top row of figure 2.3. At finer scales smaller details of for example texture appear as line segments, whereas the coarse scale will only allow line segments at the most significant edges. For

\[ (\omega, \theta) \]
\[ (x, y) \]
the information to be useful the small line segments are linked into groups of segments which together describe contours shown in the bottom row of figure 2.3. The contours provide important information of the shapes of the objects in the image.

The line segments extracted in highly textured areas can mostly not be linked to describe contours as they vary too much in orientation and colour. A similar problem arises in stereo as the sparse line segment representation is inadequate for matching points on a textured surface. It is unlikely that the same 2D points are extracted across the images in a sufficient amount to represent for example a curved surface.

Figure 2.2: Textured box used for testing sparse and dense stereo.

2.3 Intrinsic dimensionality

We can conclude that the line segments alone are sufficient for many purposes but inadequate for a complete representation of image information. It is for example straightforward to describe the position of a rectangular box using line segments when the edges are clearly defined. But if an object are highly textured with weak boundaries it is difficult to describe the position and shape of it using line segments. Thus we are motivated to introduce other types of primitives.

First we need a reliable method to determine the likelihood that a given area of the image is best described as for example line segments or texture. This leads to the introduction of intrinsic dimensionality, which can classify small areas of the image into features as described in [10, 11]. The feature classification is local and thus not definitive. A confidence value is assigned to each region which allows for later reclassification of features using the confidence as a cost.

Figure 2.4 shows the intrinsic dimension triangle and some examples of the basic image structures we define here:

Intrinsically zero dimensional are homogeneous areas of the image that do not carry much information

Intrinsically one dimensional are line or edge structures in the image
Figure 2.3: Line segment and contour representations of a small box.
Intrinsically two dimensional are junctions or textured areas of the image.

Figure 2.4: Intrinsic dimensionality examples.

With the line segments CoViS include only intrinsically one dimensional features. A more exhaustive representation would also include intrinsically zero and two dimensional features. The intrinsically zero dimensional homogeneous areas however, do not carry much information that can be extracted locally and matched between different images. The intrinsically two dimensional areas are suited for matching across images and could provide important features. This makes them interesting to study and integrate in CoViS. Junctions have recently been added [12] but textured areas are still not investigated.

2.4 Using both sparse and dense stereo

The matching and reconstruction of features such as line segments and junctions takes the sparse 2D representation of two images as input and computes a sparse 3D representation of the scene. Thus we call it sparse stereo. The result is a sparse representation of significant information that grouped together describes contours.

To process CoViS in real-time a pipeline is introduced where the different steps are computed in parallel. The sparse stereo algorithm also needs considerable speedup to fit into this pipeline, as quite a number of 3D primitives are required for a good scene representation. High resolution cameras make it possible to extract precise and detailed information. But the increase in number of 2D primitives will necessarily also increase the computational load.

For the introduction of a new type of primitive called texel, one of many proposed dense stereo algorithms [13] could be used. Dense stereo has shown good performance in highly textured areas of images, so it might enable us to approximate small surface patches of texture in 3D. At higher level processing stages the texel features can then be combined into larger, smooth surfaces in the scene. Dense stereo in general are computationally heavy algorithms, but a substantial subset of these algorithms will support parallel execution due to low level dense pixel operations. The computational burden can then be offloaded to the massively parallel GPU platform which is already present in the framework.

Using both sparse stereo and dense stereo in the same setup we might get a more complete and reliable description of the scene in 3D. Earlier this dual approach was
out-of-reach for a real-time system at least, because of the amount of computations.
But with the GPU already available and easy to expand in the system, we are highly
motivated to both try to speed up the sparse stereo and introduce the dense stereo
based approach.
Chapter 3

Computing in parallel

Personal computers, supercomputers, game consoles and servers are converging to concurrent platforms mainly because of two reasons:

1. More effective use of chip area and power compared to the traditional CPU.
2. An increasing number of applications that were traditionally hard wired in CPUs are now implemented using concurrent processors to improve functionality and reduce development cost.

In the article *A view from Berkeley: The landscape of parallel computing* [14] a number of people with different technological background sat together and discussed how this new paradigm affects the programming task. The challenge we are facing is how to map sequential algorithms to parallel architectures?

A number of different types of parallelism are identified, such as bit-level parallelism, instruction-level parallelism and data parallelism just to mention some of them. They argue that different parallel architectures do not support each of them equally well. This is interesting in our context, as it gives rise to other fundamental questions: How do we compare the performance of very different architectures? And how do we choose the best platform for a given application?

In *A view from Berkeley* they propose thirteen *dwarfs*, each representing different algorithm characteristics and patterns that can be addressed in parallel applications. They suggest these dwarfs are used for evaluating and comparing the performance of both parallel programming models and architectures.

In my FORK project [1] different architectures (such as the FPGA and the GPU) were compared. A particular dense stereo algorithm of interest were chosen – closely related to a dwarf named *Dense Linear Algebra* – and implemented on both architectures. This provided a good basis for comparison and finally led to the recommendation of the GPU platform as the target for the master’s thesis implementation of the sparse stereo algorithm.

It would be fortunate if the sparse stereo algorithm belonged entirely to the same dwarf, but this is not the case. The sparse stereo algorithm requires a more complex data structure, with less input entities in terms of primitives (compared to pixels in dense stereo) but more computations per entity. Another difference is a relatively large number of memory transactions required for the storage of temporary results.
3.1 Choice of hardware accelerator

Accelerators are special purpose processors frequently used in complex or compute-intensive parts of applications for several reasons:

- to reduce the execution time
- to offload the general purpose CPU
- to offer special easy-to-use features

The broad field of accelerators are ranging from single-threaded processors optimised for throughput at high frequencies, through specialised domain-specific often multi-threaded processors such as DSPs and GPUs, to custom chips often implemented with reconfigurable hardware such as FPGAs. What many accelerators offer is a parallel architecture that enables concurrent execution of tasks. Concurrent execution can give a tremendous speed up compared to sequential execution but different constraints must often be taken into account.

For this FORK [1] and master’s thesis we have limited our focus on parallel architectures to systems on a single chip. Furthermore the devices should be (high-end) mainstream products that are available as either the primary processor or as a co-processor within a personal computer to allow for a simple and compact setup of a CoViS vision system. Clusters and supercomputers are thus disregarded and focus has been on heterogeneous computing in a single personal computer with different types of computational devices supporting different levels of parallelism.

The following parallel architectures were examined during the FORK [1]:

1. The multi-core Central Processing Unit - the CPU - representing coarse parallel execution.
2. The many-core Graphics Processing Unit - the GPU - representing massive parallel execution with many independent multiprocessors.
3. The Field Programmable Gate Array - the FPGA - representing configurable massive parallel execution with high adaptability.

It has been chosen to continue with the implementation of sparse stereo for the GPU for the reasons listed in section 3.2. In this master’s thesis we thus examine how the GPU can be utilised further in the system by practical implementation of the stereo algorithms. The results obtained in the massive parallel framework should be compared with coarse parallelism on CPUs.

3.2 FPGA versus GPU

In the FORK [1] an FPGA and a GPU platform were compared as they were both available in the system as a co-processor. Many issues were discussed based on practical experience, with the following primary conclusions.
First of all FPGA devices are expected to require lower power than both CPUs and GPUs because of the generally lower clock frequency and the designs can be optimised at gate level to use a minimum amount of power. Heat sinks and fans are thus usually unnecessary on an FPGA.

On the other hand a modern FPGA mounted on a PCI Express board is expensive due to limited market appeal compared to GPUs and CPUs. One could argue that GPUs and CPUs are a free resource when not fully utilised as they are both available in most computer systems.

The low price and intuitive programming interface is a clear advantage of the GPU.

### 3.2.1 Memory scheme

An important aspect of parallel execution devices is the incorporated memory scheme. The processing of a data parallel problem makes memory bandwidth and access time a bottle neck. We want to minimise the time in which the accelerator is idle, waiting for data from memory.

On the GPU we ensure that the full memory bandwidth is utilised with coalesced data transfers from device memory to multiprocessor registers. Furthermore we utilise the fast shared memory on the multiprocessors as much as possible to reduce the overall number of device memory transactions. We do this by identifying which parts of the problem can be solved completely in parallel, and where we need fast communication and synchronisation between threads. With some CUDA experience, the programming environment naturally guides us into this line of thought. The task is to represent the necessary data on the device memory, so it can be accessed in an optimal way.

On the FPGA we have the freedom to design the needed memory subsystem, but we lack the prospect of thread execution and high level processor features. We will have to implement everything ourselves in hardware. Specialised pipelines are often used for the data to be available only when required. This limits the set of applicable algorithms, as we assume we can solve the problems with data on the fly. This means that, at all times, we will only have a limited amount of input data available.

### 3.2.2 Level of abstraction

The CUDA environment ensures an intuitive C programming interface with support for a wide range of math functions. The architecture is highly optimised for floating point multiplication and addition which as a result are very cheap operations. A great variety of other math operations and functions are also supported, but more costly in terms of clock cycles. With CUDA and the GPU our primary concern is thus to develop a kernel that solves the problem efficiently using the API and the features for parallel execution offered by the framework. This high level of abstraction simplifies the task of mapping a known algorithm to the platform but comes at a cost. We do not know how the underlying architecture is actually implemented.

On the FPGA we cannot expect the same level of abstraction. Floating point operations are in general not supported directly and thus special components must be developed or found elsewhere to achieve this functionality. Furthermore we need to
address hardware timing issues and pipelined flow of execution which can be quite complex and at a very poor level of abstraction. As a result our focus moves away from the algorithm we intend to implement.

What we gain however is complete control of data flow and use of resources down to each logic gate and flip-flop in the design if desired. Compared to this insight in the FPGA design, CUDA suffers from the closed architecture where the inner workings are a well kept secret from NVIDIA. As a programmer we don’t know exactly what is going on inside the GPU. Some scheduler will invoke the different thread blocks and eventually it will finish, but no guaranteed deadline is given. On the FPGA it is possible to define what the output should be on every single clock flank.

3.2.3 Choice of platform

In the FORK [1] the GPU platform were chosen for the sparse stereo implementation due to the high level of abstraction and intuitive programming interface. We expect this to be an advantage when dealing with the implementation of more complex algorithms. The GPU force us into the parallel way of thinking, and this is considered a good exercise and investment of time. The framework is likely to be similar to future high performance systems. To implement the sparse stereo on FPGAs would be challenging within the time frame for this project. Much time would be spent on low level hardware description work which is irrelevant in this context.

Using the GPU as a co-processor we have learned that the task must be of considerable size before it is feasible to run the problem on the GPU. If this is not the case the overhead will be dominant and a coarse parallel CPU would be just as fast. Thus we do not expect the CPU and GPU to interoperate closely – this might though be possible in the future as the architectures draw closer and someday could end up inside the same chip. For now we will instead try to implement the parts of the algorithm, which support a high level of parallelism, using CUDA on the GPU and then compare the performance with a CPU.
Chapter 4

Dense stereo

4.1 Algorithm in CUDA

The term dense stereo as opposed to sparse stereo covers a large number of proposed algorithms that, given some images of a scene, will compute what we can call a complete depth map. This depth map refers to some function \( f(x, y) \) that associates an estimate of the distance to the original scene point for every pixel \((x, y)\) in a reference image. A comprehensive overview is given by Scharstein and Szeliski [15]. They present a taxonomy of dense two-frame stereo correspondence algorithms, and we will base our selection of algorithm on the thorough comparison and evaluation they provide herein.

Several approaches have been taken to represent this depth map, but most common is a uni-valued disparity function \( d(x, y) \), where the disparity value is measured in pixels and \((x, y)\) are coordinates of the disparity space, coincident with the corresponding pixel coordinate in the reference image which we choose to be the left.

The correspondence between a pixel \((x, y)\) in the reference image and a matching pixel \((x', y')\) in the right is given by

\[
\begin{align*}
x' &= x + s \cdot d(x, y) \\
y' &= y
\end{align*}
\]

where \( s = \pm 1 \) is a sign chosen so the disparity is always positive. Given a point in the reference image we expect to find the matching point at the same place in the right image, if the object was positioned infinitely far away, or moved to the left within some disparity range. Limiting the search space by introducing a disparity range is a common technique especially when using high resolution images. A disparity range corresponds to some lower threshold of how close objects can be to the cameras in the setup.

Dense stereo matching is often done at pixel level and a subset of the many algorithms thus fit the parallel GPU platform nicely. This was investigated in the FORK [1] which we will refer to here. Other work in the area is presented in e.g. [16, 17]. The dense stereo algorithm investigated in FORK is called SSD and should now be integrated in CoViS. For a better performance test we have also included a phase based dense stereo algorithm running in real-time on GPU. This has been provided.
by Karl Pauwels and the concept of gabor phase based processing is presented in [18].

In this part of the project we are mainly interested in whether the result of the dense stereo algorithm can add to the CoViS representation of image features. Thus our investigation is preliminary and we find inspiration in [19] where small surface patches are computed in 3D.

4.2 Introduce the texel

Having computed the depth map for the reference image using a dense stereo algorithm we can now compute patchlets or what we in this project will call texels. The texel is a small surface element with a position in 3D and an orientation describing a small patch which is approximated over an area in 3D. This area in the scene has been projected onto the image as texture and can thus be anything from distant objects as in a landscape, to nearby carpets with highly repetitive structures. What we hope to describe is the surface of nearby objects with texture.

The texel we introduce exists only in 3D and has the following attributes:

$$\Pi_t = ((X, Y, Z), (N_x, N_y, N_z))$$

This is first the position and second the normal of the patch in 3D. We do not bother with colour or actual pixel information as representing texture from an image is a different task and outside the scope of this project.

4.3 Texel computation

The extraction of texels is still experimental in this project. The purpose is to demonstrate whether dense stereo can add valuable information to the image representation through the computation of texels. Thus we choose a simple approach that is possible to realise within a short period of time. The approach is described in pseudo code in algorithm 4.1. Note that $i0D$ is short for intrinsically zero dimensional, $i1D$ intrinsically one dimensional and so on. This is used in line 4 of algorithm 4.1 to check if the current area has been classified as most likely to be texture or junction. This is the case if the $i2D$ confidence is higher than both $i1D$ and $i0D$.

The implementation is basically a function requiring the disparity map, camera calibration and intrinsic dimensionalities as input. Looping through points on a static grid over the reference image we make use of some math functions that is already present in CoViS as e.g. the reconstruction and the fitting of the plane using a linear least squares method. The choice of dense stereo algorithm is left to the user. Two algorithms are currently available – both running on the GPU – but it can easily be extended with additional algorithms.
Algorithm 4.1 Simplified texel computation algorithm

1. Define a grid over the left input image
2. for all positions in grid do
3. Find the match in right image using disparity value
4. if \( iD < i2D \) and \( i1D < i2D \) for position in both left and right image then
5. Compute reconstructed position of match
6. Add reconstructed points using disparities around match
7. Fit a plane using all added points to get orientation
8. Create and store a texel with position and orientation
9. end if
10. end for

4.4 Test

Figure 4.1 compares the different dense stereo algorithms. In figure 4.1(a) we see the input image with focus on the area we will look at in detail because of the amount of texture.

In the row below we see disparity maps visualised using colours. Blue is far away, green closer and through yellow we end up with red being closest to the camera. In figure 4.1(b) we have the result of the SSD algorithm which is very local and thus not stable in areas with insufficient texture. Below in figure 4.1(e) we have the same disparity image but now only showing the areas where the intrinsic dimensionality combined with the level of energy suggests that this is texture (or a junction). In the middle (figure 4.1(c) and 4.1(f)) we have the results for a phase based dense stereo method (GPU implementation provided by Karl Pauwels [18]) which has a more global perspective and is thus more stable in difficult areas – but with the risk of making wrong assumptions. The phase based method has a consistency option where correspondences are verified searching from the right image in the left. Figure 4.1(d) is white in the areas where the consistency check failed.

We end this preliminary investigation of texels by a visual evaluation of the computed primitives. They are shown in 3D in figure 4.2 using a CoViS display tool called WandererX. At the top – figure 4.2(a) – we see the original CoViS line segments extracted and reconstructed with parameters that gives only very few primitives in the textured area. We see some clear contours of the object but cannot describe the surface inbetween without having a model of the object.

In figure 4.2(b) and 4.2(d) we have the same viewpoint but now showing also the texels extracted with the SSD and phase based algorithm respectively. We are optimistic as we see a clear tendency in the description of the surface.

The side view in figure 4.2(c) and 4.2(e) proves that the texels are so closely aligned that we, especially with more optimisation, will be able to compute good surface descriptions based on dense stereo. We note that the side view based on the SSD algorithm in figure 4.2(c) has more outliers than the phase based side view shown in 4.2(e). Apart from that, the use of intrinsic dimensionality ensures that we only extract texels in textured image areas where a matching is most likely to be possible if we do not encounter repetitive structures.
Figure 4.1: Dense stereo algorithms comparison and intrinsic dimensionality.
Figure 4.2: 3D features visualisation of the box shown in figure 4.1.
4.5 Discussion

By visual inspection the results in section 4.4 are promising. The computation of the orientation of the texels is not as stable as the position. This might be a result of local minima in the least squares computation. However, it should be possible to approximate nice and smooth surfaces in textured areas of the images by weighting the reliable position higher.

The simple approach used here has some obvious drawbacks that could be optimised in future work. First of all discontinuities in disparity values are not handled when adding points around the position. Furthermore the grid is static. The extraction of texels does not adapt to the slope of the surface. This means that frontal parallel surfaces are better covered than surfaces with a steep curve compared to the camera plane.
Chapter 5

Sparse stereo

This chapter first provides an in-depth analysis of the sparse stereo algorithm. We then outline a strategy of how it can be mapped to what is called kernels in the CUDA framework and make some platform specific considerations before the actual implementation is described and tested. The final GPU program will be compared to an equivalent CPU program. The purpose of the work is of course to get a faster system if possible, but just as important; to make some general statements of the type of problems that suits the GPU based on practical experience. This proceeds the FORK project where an FPGA and a GPU were compared in performance and accessibility through practical examples.

5.1 Algorithm analysis

In this section we analyse the stereo method presented in [6]. The approach is called sparse stereo as it is based on 2D line segment primitives which is a sparse image representation. The algorithm computes a 3D representation of line segments in the scene through three steps: most importantly is first the stereo matching step and then the reconstruction step. In-between we have the disambiguation step to enhance the result. The purpose of our analysis is to line out a strategy for a GPU implementation of the exact same matching and reconstruction. The disambiguation methods we will consider in section 5.1.2, but not try to speed up during this project as it is an optional step of the algorithm.

As the stereo algorithm is already implemented in an object oriented framework our analysis focuses on the aim of transferring the functionality to a massively parallel platform. The most conspicuous challenges being the following:

- get an in-depth picture of the original algorithm by studying the source code – this is the only place we find all required math operations
- identify sufficient parallelism to achieve a speed up
- determine how to utilise the highly specialised memory of the GPU
- get around the missing support for an object oriented programming language so the result is still readable code that is similar to the original implementation
The last point is not crucial but will indeed ease the task of maintenance and further development.

The sparse stereo involves the simplified steps shown in pseudo code in algorithm 5.1. The disambiguation step is omitted here.

**Algorithm 5.1 Simplified sparse stereo algorithm**

1: for all primitives in *left* do
2:   for all primitives in *right* close to epipolar line do
3:     Compute multi-modal similarity (confidence)
4:     Discard if similarity is too low
5:     Compute 3D reconstruction
6:     Discard if invalid 3D reconstruction
7:   end for
8: Sort correspondences for left primitive by similarity
9:   (Optional) Keep only the best successful match
10:  (Optional) Discard redundant candidates if links are defined in right image
11: end for

As input the algorithm requires two sets of primitives $I_l$ and $I_r$ which is image representations for the left and right image respectively. Furthermore we need the calibration which is for example projection matrices specifying the stereo camera geometry: $P_l$ and $P_r$ and the optical centres $C_l$ and $C_r$. All are constants for a static setup. Some additional algorithm parameters are also required: $W$ a vector of modality weights used for local confidence computation (relative weighting of orientation, phase, colour and optic flow), $\theta_{min}$ the minimum angle of primitive with epipolar line – if collinear we cannot distinguish the best matches on a line, $d_{norm}$ the maximum distance to epipolar line, $depth_{min}$ the minimum distance from the image plane – objects may not be infinitely close or behind camera. $d_{min}$ and $d_{max}$ is the disparity range for the stereo images and $C_{min}$ is the lowest allowed stereo similarity (confidence).

As we recall from figure 2.1 and [6] the 2D primitive attributes is described by

$$\pi = (x, \theta, \omega, c, \lambda)^T$$

where $x$ is the position, $\theta$ is the angle of orientation, $\omega$ is the phase (edge transition), $c$ is a vector of colours in the primitive and $\lambda$ is the size of the primitive patch. The 3D primitive is defined correspondingly

$$\Pi = (X, \Theta, \Gamma, \Omega, C, \Lambda)^T$$

where the added $\Gamma$ is a required contextual vector that defines a reference plane. The reconstruction step of a plausible match of primitives is then the relation

$$\mathcal{R} : (\pi^l, \pi^r) \rightarrow \Pi$$

Figure 5.1 shows the input stereo images 5.1(a) and 5.1(b) of a robot setup. The extracted 2D primitives are shown in 5.1(c) and 5.1(d) for each image respectively. We notice that some line segments have been extracted in the highly textured surface of the box. Whether image features are best described as lines or simply texture most often depends on scale and in CoViS the scale and level of detail is specified by
5.1. ALGORITHM ANALYSIS

Figure 5.1: The sparse stereo algorithm.
the frequency of the monogenic filtering in the early vision processing as mentioned in section 2.1.

In figure 5.1(e) the 2D primitives of the left image have been matched with the primitives of the right image. The horizontal lines indicate where the best match were found in the right image. The matches have then been reconstructed into 3D using triangulation and the sparse stereo reconstruction algorithm for line segments. The results are then reprojected back onto the two image planes in figure 5.1(f) and 5.1(g) to be visualised in 2D. For true matches this stage would ideally produce the 2D primitives shown in 5.1(c) and 5.1(d) but due to the sparse and discrete nature of the primitives the exact match will rarely occur.

We note that the scattered 2D line segments in the textured area did not provide for 3D reconstructions because of the difficult matching in textured areas. The true match often does not exist and furthermore the neighbourhood constraint applied in the disambiguation step will remove small line segments that is not an obvious part of a larger structure. See section 5.1.2 for more information. A single line segment in 3D does not carry much information alone. But linked together they describe contours and shapes.

Though not explicitly given in the pseudo code, the algorithm can be processed in two separate stages – first the matching then the reconstruction. This separation induces the intermediate data structure called stereolinks, which will be the output of the matching process and the input for the reconstruction. A stereolink – also called a stereo hypothesis – is a potential match of a pair of primitives, one primitive from each image. The semantic information of a stereolink is thus indexes of the originating 2D primitives, a similarity value and some additional parameters as whether geometrical switching is required.

We have defined the input for the algorithm as being 2D line segments, parameters and calibration data. The intermediate data structure consist of stereolinks and the final output is 3D line segments. We will now dig further into the details of the separate stages of the sparse stereo algorithm.

5.1.1 Matching

Figure 5.2 shows the details of the entire matching process.

The original matching process consists of two nested for loops – the outer and the inner loop – where every left primitive is compared to every right primitive. For simplicity we start with a detailed look on the outer loop of figure 5.2 in algorithm 5.2.

First we need to check the angle of the 2D primitive in the left image. If the angle is too close to the epipolar line, which is always horizontal when considering rectified images, then the line segment is most likely a part of a longer horizontal line and it will thus be impossible to find the true match along this line. As the matching will most likely be wrong we discard the left primitive and do not waste more time here. If the angle is accepted we compute the epipolar line as it will be constant for all comparisons with primitives in the right image. The epipolar geometry is illustrated in figure 5.3. Note that the image planes here are not parallel to visualise an epipole e in the image.
5.1. ALGORITHM ANALYSIS

Figure 5.2: Matching algorithm.

Algorithm 5.2 Matching algorithm – outer loop

1: for all $\pi_i \in I^l$ do
2:    if $\theta_i < \theta_{\text{min}}$ then
3:        Discard primitive $\pi_i$
4:    else
5:        Compute the epipolar line $\xi_i = (x_{l\infty}^r, e^r)$
6:        Inner loop - match $\pi_i$ with all primitives in $I^r$
7:          Sort stereolinks by similarity. Best similarity first.
8:    end if
9: end for
The epipolar line connects two points: \( e^r = P^r C^l \) is the epipole, \( x^l,r_{i,\infty} = P^r X^l_{i,\infty} \) is the point \( x^l_i \) reprojected at infinity \( X^l_{i,\infty} \) and then projected onto the right image. We note that the epipole \( e^r \) is constant for a static stereo camera setup and in the case of parallel camera planes it will be a vector pointing at infinity which makes all epipolar lines horizontal. Furthermore in the simple case of coinciding camera planes, it will ensure that \( x^l,r_{i,\infty} = x^l_i \) because a point which is infinitely far away will be projected onto the same point in the two images.

We can now run the inner loop to match the current left primitive with all right primitives. Afterwards we have a list of potential correspondences which just needs to be sorted according to highest similarity.

The inner loop visualised in figure 5.2 consists of the detailed steps in algorithm 5.3. First we have the same angular constraint now for the right primitive. Then we compute the disparity and distance from epipolar line which must not exceed some limits. If all tests are passed we compute whether geometrical switching is required. This is basically a test of the orientation of the two primitives and whether the direction of one of them should be inverted for the colours of the sides to match best. The outcome depends on the type of scene features that produced the primitives. Having determined the boolean switching value we can compute the multi-modal similarity value based on the four modalities and the user specified weighting.

Figure 5.4 illustrates the epipolar geometry constraint used when checking the distance between matching points.

### 5.1.2 Disambiguation of matching

The intermediate steps of the sparse stereo algorithm are used for disambiguation of the matching. It will make use of neighbourhood observations and primitives
Algorithm 5.3 Matching algorithm – inner loop

1: for all $\pi_j \in \mathcal{I}_r$ do
2:     if $\theta_j < \theta_{\text{min}}$ then
3:         Discard match $(\pi_i, \pi_j)$
4:     else
5:         Compute disparity $d_1$ from $x_r^j$ to $x_{i,\infty}^r$ along $\xi_i$
6:         Compute distance $d_2$ from $x_r^j$ to $\xi_i$
7:         if $d_1 < d_{\text{min}} \text{ or } d_1 > d_{\text{max}}$ then
8:             Discard match $(\pi_i, \pi_j)$
9:         else if $d_2 > d_{\text{norm}}$ then
10:             Discard match $(\pi_i, \pi_j)$
11:         else
12:             Compute switching (boolean) using $\theta^r$ and $\theta^l$ compared to angle of $\xi_i$.
13:             Compute weighted multi-modal similarity using $\theta$, $\omega$, $c$ and $f$ from both $\pi_i$ and $\pi_j$
14:             Create stereolink for $(\pi_i, \pi_j)$ and store values $(i, j, \text{similarity}, \text{switching})$
15:         end if
16:     end if
17: end for

Figure 5.4: Epipolar line constraints.
grouped into a 2D contour to compute an external confidence value. The similarity of primitive attributes computed so far is a very local confidence used to find putative matches. Throughout the disambiguation process the stereolink data structure should be sufficient as it stores most importantly the originating 2D primitives and the (combined) similarity values.

In the original CoViS implementation however, the data structures of stereolinks and reconstructed line segments are tightly interwoven. Throughout these intermediate steps both data structures are in use even though only the stereo hypothesis is strictly required at most stages. The result is several reiterations of the reconstruction step to keep the data structures up to date. To achieve an optimal performance these data structures should be separated and the use of them rearranged so the complete reconstruction is only carried out when necessary.

To bind this project within a realistic time frame we will not look into this change of the original functions, but the disambiguation functionality will be disabled for optimal performance during tests. It is however important to be aware of the functionality as the target real-time system of CoViS should deliver the exact same results as the original CoViS, and thus we depend on the intermediate steps and the matching improvements it provides.

The important intermediate steps before the final reconstruction is the following:

**Remove Redundant Hypotheses** can be used if contours have been computed between 2D primitives in the right image. If a left primitive has several putative matches on the same contour in the right image we will then only keep the one closest to epipolar line. If two putative matches in the right image do not belong to the same contour, neither of them will be removed.

**Compute External Stereo Confidence** by examining how well a match preserves the context according to 2D contours in the neighbourhood. The external confidence is introduced and combined with the internal confidence which is based on local attributes comparison. Combining the two confidence values we can rearrange the putative matches for each left primitive so the best match is now chosen on a higher level of abstraction.

**Threshold Confidence** will remove potential matches with too low combined internal and external confidence value. If the confidence is lower than some specified threshold it will be removed even if it is the best or only match for a left primitive.

Figure 5.5 shows reconstructions of the matches at different stages of the matching process. First we consider all combinations between left and right primitives and find putative matches according to the epipolar constraints. Reconstructing these matches gives us the 3D primitives shown in figure 5.5(a). We then apply the internal confidence value using the local primitive attributes and select only the best for each left primitive. This reduces the number of 3D reconstructions and gives us the enhanced 3D model shown in figure 5.5(b). The intermediate steps of disambiguation will now consider the neighbourhood of each match in both images. If we can find matches that according to local confidence are suboptimal but conservers the neighbourhood better, this match will get a high external confidence.
5.1. ALGORITHM ANALYSIS

(a) All putative matches according to disparity constraint

(b) All best matches on a local basis

(c) Best matches according to neighbourhood

Figure 5.5: The sparse stereo algorithm with different levels of disambiguation.
This strengthens the important contours of the scene and by using thresholding we can even remove matches which are not supported by neighbouring reconstructions. This gives the final matches reconstructed in figure 5.5(c) which has a somewhat higher degree of certainty.

5.1.3 Reconstruction

Now we will consider the reconstruction step. The task is to try to reconstruct a 3D line segment using primitives $\pi^l$ and $\pi^r$. Besides the position of the 3D line segment this also includes the computation of orientation, colour and phase.

Considering a 2D point in the image we can back-project it as a line of points which could all be the 3D source of the image projection. The reconstructed 3D point is where two lines, one from each image, meet. In practice it will be where the distance between the lines are minimised as they will never intersect. This transfers to a 2D line in the image in the sense that we can back-project it as a plane of plausible 3D lines. The reconstructed line will then be the intersection of two planes.

A complete review of the math involved in this algorithm, is outside the scope of this section. Instead we refer to [6]. We are interested in challenges regarding the implementation and thus we state that the required linear algebra are mainly dot-products, cross-products and matrix multiplications. Going from the 2D image to the 3D scene we keep track of the covariance matrix as a measure of dispersion and thus the certainty of our computations. This is computationally expensive as we need to update the covariance matrix using a Jacobian matrix for every domain transition we make. We apply the Jacobian $J$, representing the changes, on the original covariance $C_o$ in the following way to get the new covariance $C_n$:

$$C_n = JC_oJ^T$$

The dimension of the covariance matrices changes according to the variables in the current domain and likewise does the dimension of the Jacobian depend on the domains of which we are changing between. Many of the covariance matrices used is of size $6 \times 6$, the largest matrix required is of size $12 \times 12$ and all of them are required for each 3D reconstructed primitive exclusively. We can conclude that the memory requirements for each single reconstruction seem very high, when we have to compute the covariance matrices.

5.2 Algorithm mapping to CUDA kernels

Before starting the actual implementation, we will in this section consider a strategy. It is crucial for both the correctness and the performance of the code that the strategy specifies a level of parallelism for our mapping which is applied in the kernels executing on the GPU. In a regular C++ application the algorithms are traditionally implemented sequentially or at some points divided coarsely into multiple threads that can be processed simultaneously by different CPU cores. For the implementation in CUDA this point is crucial. If the achieved level of parallelism is insufficient performance will drop significantly, as the platform is optimised to handle high levels of parallelism, as for example one thread per pixel in an image.
5.2. ALGORITHM MAPPING TO CUDA KERNELS

But if the level of parallelism is forced too high for a given problem, the performance could also drop for reasons we will elaborate on here.

The GPU platform brings unique constraints to the level of parallelism versus path divergence and memory transfers which introduces a number of trade-offs that must be considered.

First of all, the level of parallelism should be as fine-grained as possible to utilise the many cores and the fast scheduling of the GPU. This low granularity decomposition of the problem is however limited by a number of issues, where the first is of course the minimum task that can actually be solved in parallel. The GPU offers shared memory and fast synchronisation to ease the cooperation among threads but there will always be some minimum task that must be solved sequentially (no matter how many women you assign – it still takes nine months to give birth to a baby).

The CUDA framework puts further constraints on the implementation for high performance execution:

1. As many local threads as possible should work in an SIMD fashion where the same instructions are applied on multiple data. Execution of threads within the basic scheduling unit (the warp) will be serialised whenever the execution path diverges.

2. The memory transfers should be kept at a minimum. Sometimes it might be possible to increase the level of parallelism at the cost of introducing path divergence. Then we could imagine to start a different kernel for the diverging tasks specifically. But if they require the same input data this will often double the required memory transfers as there is no fast memory sharing between different blocks.

Hence there exist a scenario where the level of parallelism could be increased, but not in a feasible way as one of the constraints above would be violated. It would either lead to thread divergence if compromising the SIMD constraint in the warp, or an increased number of memory transfers if the work were divided among more blocks forcing us to introduce more memory transfers.

5.2.1 The matching process in CUDA

The matching process of the sparse stereo algorithm is originally implemented with an $O(n \times m)$ running time ($n$ is number of primitives in left image and $m$ is the number of primitives in the right image). How many primitives we require to represent an image depends on the desired level of detail and the image resolution. Between 500 and 1500 primitives per image is quite common, but for higher resolution images (>2MP) considerably more primitives is an advantage. The stereo images should be a view of the same scene slightly shifted and thus containing approximately the same number of primitives $n = m$ which leads us to the $O(n^2)$ running time. This could be optimised using a bucket algorithm, where the primitives of the right image are sorted into buckets aligned with the epipolar lines. For any primitive in the left image it would then be sufficient to search the contents of three buckets from the right image to find all putative matches. This assumes we have a model of the epipolar constraint that enables us to sort the primitives. It could then reduce the
asymptotic running time at the cost of a slightly higher constant of processing time and memory operations.

For the most simple entry into the GPU programming environment we postpone the usage of a bucket algorithm. At first we simply carry out all comparisons – which by the way fits the massively parallel environment nicely – and if the solution is successful the changes required to benefit from a bucket algorithm are minimal. We just add the sorting into buckets which could be done on the CPU in linear execution time and then rearrange the kernel execution dimensionality.

For now we need to decide on the extent of the task to be assigned to each thread. Two straightforward possibilities are the following:

1. Each thread compares a single combination of a left and right primitive. This will require $n \times m$ threads.

2. Each thread is given one left primitive and is responsible for the comparison of this primitive with all primitives in the right image. This will require only $n$ threads.

We choose option number 1, to get the largest number threads and avoid a huge amount of path divergence. The strategy is to let every thread within a block share the same left 2D line segment data in shared memory. The right primitives cannot be stored in shared memory as they are only used by one thread in each block. Each thread within a block will thus compare the same left primitive to consecutive right primitives. Figure 5.6 illustrates the execution dimensionality as a two-dimensional grid of one-dimensional blocks. The number of right primitives $m$ is rarely a multiplicity of the block width $k$ and thus, a number of threads will be idle in the rightmost blocks.

![Figure 5.6: Grid of thread blocks for the matching kernel.](image)

The suboptimal part of this strategy is that the algorithm basically consists of a series of tests on the combination of left and right primitives. As soon as one of the tests fail, we would on the CPU continue directly to the next series of tests. On the GPU we have an SIMT environment, which means that all threads in the basic scheduling unit – the warp – must do the same operation or be idle when it is scheduled for execution. We use conditional branches in the series of tests and all
threads that fail a test will then be idle until all threads have either failed a test or finished the computation of a stereolink. Fortunately, most of the combinations will fail the test of epipolar distance and disparity, so by arranging the tests in the right order we can minimise thread divergence by finishing the execution of most warps early.

With this strategy we are left with the problem of how and especially where to store the results and afterwards sort the stereolinks according to similarity before the reconstruction can be started. What we have is for each left primitive a number of blocks each containing threads that might have a stereolink to store if the combination of primitives assigned to this thread turned out to be a putative match. To keep this as simple as possible to get the system running we could allocate a two dimensional array of stereolinks according to the number of primitives \( m \times n \). This is possible because the size of the stereolink data structure is expected to be relatively small. The size will be considered in section 5.3.1. Another strategy would be to let the threads communicate within blocks and try to store stereolinks in a compact way. It might even be possible to sort them at the same time. The problem however is branch divergence as the largest number of threads do not have a stereolink to store because the combination of primitives was not a putative match. Thus we choose the first approach and use the large data structure where each thread can store the result of the matching in the relevant position of the array. Discarded matches will just have a negative similarity value and putative matches will have all attributes of the stereolink updated.

Figure 5.7 shows three cutouts of the stereolink data structure at different stages in the matching process. Each square represent a stereolink. The circles and the numbers within are the indexes of the left and right primitive for this stereolink combination. The rectangle below is the associated confidence value. Please note that the real confidence values will be either negative or lie in-between 0.0 and 1.0 but here we use integers for a simple visualisation of the sorting.

Figure 5.7(a) is the stereolink data structure after the matching kernel have finished. The matching algorithm however, has not been fully completed and thus we introduce more kernels to handle the missing functionality. At this stage we have a huge matrix of stereolinks which is sparse in the sense that only few spaces does actually contain putative matches – the cutout does not resemble the true ratio. Thus the task is twofold and we use one kernel for each matter: first condensing the data structure, the result shown in figure 5.7(b) and second a sorting of a fixed maximum number of stereolinks according to local confidence giving us the final result of the entire matching process shown in figure 5.7(c).

For the condensing kernel we will associate one thread to each row in the stereolink data structure. The job is to run through the row in linear time – every time we meet a stereolink with a positive confidence value we move it to the left most free space and thus fill up the matrix from the left side.

In the CUDA community a number of sorting algorithms are available for free. We use bitonic sort which is simple and very effective when sorting a small number of values [20] or in our case structs with confidence values. It uses a fixed parallel network of comparisons that in the end is guaranteed to sort a fixed number of entities. We can get the actual maximum number of putative matches associated to a single left primitive from the condensing kernel that have just gathered the
stereolinks. Normally this number of stereolinks per left primitive will not exceed 32. Thus we apply a generic bitonic sorting kernel provided by NVIDIA with few adjustments to sort the stereolinks in our context. The threads in a block will sort all stereolinks for a single left primitive and thus the number of blocks we initiate will be equal to the number of left primitives. Given the high level of parallelism we expect this sorting algorithm to perform nicely on the GPU.

5.2.2 The reconstruction process in CUDA

This part of the algorithm is known to be memory intensive. The reconstruction of the position and orientation of a small line segment in 3D requires large matrix multiplications to maintain the covariance matrices which is unique for every match. Computing the actual position and orientation is not that expensive – we use the specialised CUDA math functions, denoted with underscore like this: \(_{\text{sinf}}(x)\), whenever it is possible. The precision is slightly compromised in these functions as described in [2] but comparing the final output results against the original CPU implementation we see that it makes no significant difference. The uncertainty in exact location when working with discrete pixels is dominant. Thus we also use floats in the GPU implementation whereas the original implementation uses double precision as performance was not an issue. The recent CUDA versions does support double precision but it affects performance by doubling the number of memory transfers.

The simplest strategy for the mapping of the reconstruction algorithm to the CUDA
platform is to unroll the for loop running through all the matches to reconstruct. Thus each block is responsible for one left primitive, and each thread within a block is responsible for the reconstruction of a single putative match. All reconstructions can thus be transferred back to host memory in one bunch after this kernel have finished. Two options now occur. We can choose to take only the best matches (locally) and instantiate as CoViS primitive objects. But because all putative matches have been reconstructed we can also by use of the external confidence recompute what appears to be the best match.

The task assigned to each thread is rather large. The many multiplications of large matrices requires a large amount of memory transfers, but trying to increase the parallelism further we are forced to either separate the functionality over several blocks, thus introducing even more memory transfers and with no option for cooperation among threads. The other option would be to put the functionality in the same block with increasing branch divergence as a logic consequence unless distributed over several warps. In the end the computational steps of the reconstruction algorithm are not independent. The results of the first computations are required for the latter and thus the minimum size of the problem that can be assigned to a thread is fixed.

5.3 CUDA implementation

A lot of code were written for the implementation of the sparse stereo algorithm in CUDA and it is thus impossible to treat every detail in this report. The source code with a reasonable amount of comments is available in full length on a CD. In this section we introduce data structures and the specific memory types in use on the GPU. Furthermore we highlight the problematic challenges faced during the development. CUDA exposes the different types of memory to the programmer, and it must be used appropriately for optimal performance.

5.3.1 Data structures and memory usage

We use structs with functions associated like methods in CUDA instead of the objects in C++ and CoViS. Listing 5.1 is a simplified version of a line segment struct showing some of the attributes and a function which will return the position of the 2D line segment in homogeneous coordinates. Furthermore it shows the CUDA specific memory alignment keyword (\texttt{align}...) in the first line which define how the data of this struct should be organised and loaded from device memory to registers. In line 9 we see the \texttt{device} qualifier which specifies that this function is intended to run only on the GPU.

Both qualifiers are used in a header file which must be processed by NVIDIA's NVCC compiler which will interpret the input and direct regular C code to the ordinary C compiler. NVCC does not support C++ but the C functions invoking the CUDA kernels can be accessed from C++ using the \texttt{extern "C"} directive.

For the integration in a C++ project this means that the line segment struct must be converted to basic data types in the transition from C to C++ environment as
typedef struct __align__ (16) linesegment2D {
    float x, y;  // position
    float theta;  // orientation
    float phase;  // phase
    primitiveColors colors;  // colour triplets
    float U, V;  // optic flow
    float size;  // size
    _device_ float3 getPosition(void) {
        return make_float3(x, y, 1.0f);
    }
};

Listing 5.1: Line segment as a struct.

the C compiler does not understand the CoViS line segment object and the C++ compiler does not understand the struct with CUDA specific qualifiers.

As we are working with many line segments they should be stored in arrays in global device memory to allow for indexing from the different threads. This is where handling of memory in CUDA can get complex because we want to utilise the full bandwidth when all threads access memory in parallel. What we want is a coalesced memory transfer where parallel threads access memory in a way that utilises the full bandwidth.

The basic scheduling unit is the warp, but when it comes to memory transactions the basic unit is the half-warp. This means that a coalesced data transfer will normally deliver a variable (of up to 32 bit) to all threads in a half-warp (16 threads) in one memory transaction thus utilising the full bandwidth of 128 bytes per memory transaction. The rules of how to get a coalesced data transfer is defined in [2]. For CUDA compute capability 1.3 the constraint is basically that the data being accessed should be in the same memory segment. This is a problem when accessing variables in an array of structs.

When the data is stored in an array of structs we cannot directly achieve coalesced reads. If all threads require for example the $x$-value, from the line segment they are currently processing, these are distributed over a larger part of raw memory. Instead we read in larger chunks of the struct at the time, to make sure we utilise more of the available bandwidth. The value after the __align__ qualifier is the number of bytes that should be transferred from this struct in each cycle. Unfortunately the largest chunk of data that can be directed to one thread this way is limited to either 8 or 16 bytes and, as seen in table 5.1, our structs are much larger.

A better solution to this problem is a more complex data structure: a Struct of Arrays of Structs (SoAoS). The concept is illustrated for a 2D line segment in listing 5.2. According to theory this should align the data so we can utilise the full bandwidth when the threads access contiguous structs. $N$ indicates the number of primitives which is not known in advance. Thus we could instead use pointers to arrays but this is equal to just splitting the original struct up in smaller structs of size either 8 or 16 bytes and then using separate arrays for each of them.

Unfortunately there has not been time to actually change the code accordingly and play around with the performance tweaks because making it run was higher prioritised to allow for some early performance comparisons. It is a somewhat straightfor-
5.3. CUDA IMPLEMENTATION

```c
typedef struct __align__(16) geometry {
    float x, y;
    float theta, phase;
};

typedef struct __align__(8) opticflow {
    float U, V;
};

typedef struct linesegment2D {
    geometry x[N];
    opticflow y[N];
    ...;
};
```

Listing 5.2: Line segment as a struct of arrays of structs (SoAoS).

ward but still fundamental change involving many lines of code. However, it would have been interesting to see the speed up achieved by this improvement.

Stereo calibration data (288 bytes) and algorithm parameters (64 bytes) are stored in structs in constant memory as these values are used by all threads and does not change during execution. The GPU has 64KB of constant memory in total.

<table>
<thead>
<tr>
<th>Struct</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line segment 2D</td>
<td>80 bytes</td>
</tr>
<tr>
<td>Line segment 3D</td>
<td>32 bytes</td>
</tr>
<tr>
<td>Stereolink</td>
<td>40 bytes</td>
</tr>
</tbody>
</table>

Table 5.1: Memory requirements for each struct.

The 3D line segment is smaller than the 2D line segment because no colour information is currently computed on the GPU. If this information is needed it can be computed by the CPU in linear time. The computed covariance matrices are stored as pointers to the structs which will be located elsewhere.

The size of the stereolink struct is of interest because with the chosen strategy we allocate \( n \times m \) stereolinks in a large two dimensional array. With 1000 2D line segments in each image this will thus require approximately 40MB of device memory. Device memory in total is 1GB and must also store all 2D line segments from both left and right image and the resulting 3D matches.

To save time transferring data back to the host we will not transfer the entire two dimensional \( n \times m \) array of stereolinks or the reconstructed line segments. Instead we note what is the maximum number of putative matches for a single left primitive, and because of the condensing and sorting as described in section 5.2.1 we can then do with a certain number of the left most stereolinks and reconstructions. This drastically reduces the time used on memory transfers and it is easy to realise because of CUDAs specialised 2D memory functions that is originally intended for image transfers. These functions make it possible to define the number of columns to actually transfer but also a pitch which is the entire width of each row, required when some extra spaces are appended to align each row with the physical memory segments.
CHAPTER 5. SPARSE STEREO

Listing 5.3: A simplified matrix struct for the CUDA framework.

typedef struct matrix {
    float * elements;
    uint cols, rows;

    __device__ void init(uint r, uint c, float * data) {
        cols = c;
        rows = r;
        elements = data;
    }
};

5.3.2 Creating a general matrix class

The CoViS implementation in C++ uses a dynamic matrix class that allocates the required memory when the size of the matrix is known. Furthermore the standard math operators are defined for this object making it very easy to for example add, subtract, multiply and transpose matrices. And the code is easy to read and understand.

With the level of parallelism chosen for our CUDA implementation we have a fundamental problem regarding all the matrix operations at thread level. CUDA actually requires every operation to be hardcoded – known at compile time – in the sense that we can have no dynamic memory allocation. It is of course not possible to realise dynamic memory allocation for each thread when hundreds of threads are executing in parallel. Increasing the parallelism so each thread should only process smaller parts of the operands would just increase the complexity of the implementation. Ideally we want to keep it simple and let the CUDA code be as similar to the C++ implementation as possible. In effect this means we want a function that for example multiplies matrices independently of matrix dimensions. In listing 5.3 we show a matrix struct that just contains a pointer to the data and the dimensions of the matrix. This way the size of the matrix struct does not vary and we can make math operator functionality independent of the memory allocation.

The allocation of matrix elements must be explicit and cannot depend on variables. We find inspiration in the CUDA SDK utilities that defines float3 and float4 vector data types of a three and four dimensional float vector respectively. They are allocated using inline functions such as make_float3() and make_float4(). The instructions of an inline function are inserted everywhere it is referenced at compile time replacing every call to this function directly in the code. In a CUDA kernel all functions will automatically be inline because they must be local to the thread. In a similar way we make matrix allocation functions for each single matrix dimension we require. This might not be the optimal solution but it does make the relevant code easy to read and the primary functionality can be implemented very similar to the C++ code. Listing 5.4 shows an example of a matrix allocation function where the size is explicitly defined at compile time.

In the CUDA framework these steps are valid, but it is important to realise what they mean in terms of memory operations and performance. As mentioned when describing the strategy in section 5.1.3 the memory requirements for these matrices will force us to store data in global device memory. This is exactly what happens in
5.4. TEST

This section describes some performance tests of the GPU implementation compared to a modified CPU version. Both the GPU and CPU version could be optimised more however. This is often the case for complex implementations – some compromises

```c
//inline__ device__ matrix * make_matrix3x4() {
    float elements[3*4];
    matrix m;
    m.init(3, 4, elements);
    return &m;
}
```

Listing 5.4: Data element allocation.

```c
device__ matrix * multiply(float a, matrix * m_res) {
    for (int r = 0; r < m_res->rows; r++) {
        for (int c = 0; c < m_res->cols; c++) {
            m_res->set(r, c, a * m_res->get(r, c));
        }
    }
    return m_res;
}
```

Listing 5.5: A matrix struct for the CUDA framework.

line 2 of listing 5.4 and in general this will have a detrimental effect on performance, especially because we cannot expect coalesced data transfers – this is in the hands of the CUDA compiler.

A simple example of a math operator function is shown in listing 5.5 where a matrix is multiplied by a scalar and the result is stored in the operand matrix to save space.

Table 5.2 shows the memory requirements of the different kernels according to the CUDA compiler. Smem is shared memory, cmem is constant memory and lmem is local memory. Lmem is special in the sense that it is stored in global device memory but private to each thread. The reconstruction kernel needs this to store the many matrices used to compute the covariance. This is the more than 16KB in table 5.2 and it will in general prevent high performance and thus we would expect this kernel to run much faster if the computation of the covariance matrices could be omitted. Another problem which could be optimised is the required number of registers which is rather high for the matching and the reconstruction kernel.

<table>
<thead>
<tr>
<th>Kernel description</th>
<th>Registers</th>
<th>smem</th>
<th>cmem</th>
<th>lmem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Match 2D line segments</td>
<td>39 bytes</td>
<td>128 bytes</td>
<td>352+116 bytes</td>
<td>0 bytes</td>
</tr>
<tr>
<td>Condense stereolinks</td>
<td>10 bytes</td>
<td>96 bytes</td>
<td>352+4 bytes</td>
<td>0 bytes</td>
</tr>
<tr>
<td>Sort stereolinks</td>
<td>15 bytes</td>
<td>96 bytes</td>
<td>352+4 bytes</td>
<td>0 bytes</td>
</tr>
<tr>
<td>Reconstruct 3D line segments</td>
<td>39 bytes</td>
<td>128 bytes</td>
<td>352+104 bytes</td>
<td>16984 bytes</td>
</tr>
</tbody>
</table>

Table 5.2: Kernel memory requirements.

5.4 Test

This section describes some performance tests of the GPU implementation compared to a modified CPU version. Both the GPU and CPU version could be optimised more however. This is often the case for complex implementations – some compromises
are made to make the system ready within a limited period of time. Because of this the results of the test are not definitive, but they can give an important indication of the outcome of the effort.

We could anticipate the GPU code to be easy and clean to read as it is reduced to the kernel which should be the essential task to process in parallel. Unfortunately it turns out to be hard to keep clean and easy to read due to this very same parallel scheme. A lot of indexing is required and a lot of extra lines of code are introduced to specify memory types and synchronisation and the data structures have a poor level of abstraction.

The test machine has the following specifications: two Intel Xeon CPUs E5440 each with four cores running at 2.83GHz. It should be noted that all test applications use only one of these cores. For the general speed up of the entire CoViS application eight cores are already highly utilised. It has 8GB RAM and runs a 64 bit version of Ubuntu release 9.04. The GPU is an NVIDIA GeForce GTX 295 supporting compute capability 1.3. It actually embodies two separate GPU devices of which we will only use one in test. The installed version of CUDA is 2.1.

A last thing we must define before we can carry out the tests is the different levels of overhead in the system. Overhead in our case is the extra time required to initiate the processing on the GPU, as e.g. due to memory transfer or data conversion. It goes in two directions; from host to device and device to host. CoViS data structures are object oriented and quite complex due to many years of expanding the code. Overhead is the extraction of relevant class attributes and construction of structs used on the device. Furthermore it includes the time of the physical memory transfer.

The overhead can occur at three stages of the algorithm.

**Overhead I** Relevant data from all the 2D line segment objects in the CoViS framework are extracted and stored in a simple struct suited for the CUDA framework and then transferred to the device.

**Overhead II** The intermediate data structure, called stereolinks, representing stereo hypothesis can be stored on the GPU for later use or transferred back to the CPU. The first option gives no overhead the latter results in a small memory transfer overhead.

**Overhead III** If the reconstructed 3D line segments are on the GPU memory in terms of structs they must be transferred back to the host memory and converted into regular CoViS primitive objects.

Table 5.3 shows the first timing measurements of three execution modes. More than 1200 primitives from each image are used giving a total of 1040 reconstructed 3D primitives. Only the best valid reconstructions are computed. First of all a pure CPU execution mode to the left where no overhead occur. In the middle we have a combined CPU/GPU mode where the matching is processed on GPU and the stereolinks is then transferred to host memory where the CPU computes the reconstruction. Right most we find a pure GPU implementation where both matching and reconstruction is done on the GPU. The GPU implementation for the reconstruction step will always reconstruct all putative matches but they are not
necessarily put back into 3D line segment objects as this takes time. This choice of how many we want to keep is left as an option.

We note that the reconstruction step takes a little bit longer for the combined CPU/GPU procedure in the column in the middle compared to the left most. This is because some of the overhead of the stereolink transfer from GPU to CPU is actually included in this measurement. The stereolink structs loaded back from the device are not converted to CoViS data structures before they are needed, as the number of required conversions depend on the validity of the reconstructions. If the first putative match – the match with highest confidence – gives a valid 3D reconstruction then we are done for that particular left primitive. If it turns out invalid we will proceed to the next best match.

<table>
<thead>
<tr>
<th></th>
<th>CPU only</th>
<th>CPU and GPU</th>
<th>GPU only</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overhead I</td>
<td>0 ms</td>
<td>1.02 ms</td>
<td>0.92 ms</td>
</tr>
<tr>
<td>Matching</td>
<td>139.45 ms</td>
<td>10.99 ms</td>
<td>11.06 ms</td>
</tr>
<tr>
<td>Overhead II</td>
<td>0 ms</td>
<td>6.27 ms</td>
<td>0 ms</td>
</tr>
<tr>
<td>Reconstruction</td>
<td>72.61 ms</td>
<td>74.62 ms</td>
<td>46.44 ms</td>
</tr>
<tr>
<td>Overhead III</td>
<td>0 ms</td>
<td>0 ms</td>
<td>7.57 ms</td>
</tr>
<tr>
<td>Total</td>
<td>212.06 ms</td>
<td>92.90 ms</td>
<td>65.99 ms</td>
</tr>
</tbody>
</table>

Table 5.3: Matching 1248 left and and 1278 right 2D primitives. Only the best matches are reconstructed giving a total of 1040 3D primitives.

The matching process on the GPU seems to require approximately 11 ms to process this particular number of 2D primitives. As described in section 5.2.1 our matching implementation actually consist of three CUDA kernels. Table 5.4 shows the time consumption for each of the kernels, as they are invoked serially. We note that the actual matching is the most time demanding followed by the condensing which require approximately half of the time. The sorting of the relatively few number of putative matches is one order of magnitude faster.

<table>
<thead>
<tr>
<th>Kernel</th>
<th>GPU only</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matching</td>
<td>7.33 ms</td>
</tr>
<tr>
<td>Condensing</td>
<td>3.38 ms</td>
</tr>
<tr>
<td>Sorting</td>
<td>0.35 ms</td>
</tr>
<tr>
<td>Total</td>
<td>11.06 ms</td>
</tr>
</tbody>
</table>

Table 5.4: Timing of the three kernels of the matching process.

Table 5.5 shows the same measurements as table 5.3 but with the option of reconstructing all putative matches enabled. We note how the CPU execution time of the reconstruction step rises linearly with the number of reconstructed primitives, whereas the GPU execution time does not change as it already reconstructs all of the putative matches in parallel. We note however that the overhead of transferring the reconstructed primitives back to host memory increases – not because of a larger physical memory transfer, because all 3D line segments are always transferred to host but because of the extra time used for creation of CoViS objects on the CPU.

The reason why we want to reconstruct all is that quite often we have a situation where what appears to be the optimal match locally is not truly the best match.
This can be detected in later stages of processing using e.g. the external confidence computed in the disambiguation step. This is an iterative process that can update the confidence and thus also the relative sorting of stereolinks possibly making another hypothesis appear as the best match. When all hypotheses are already reconstructed in a fast way on the GPU it is no big deal to interchange some of them in an iterative approach.

<table>
<thead>
<tr>
<th></th>
<th>CPU only</th>
<th>CPU and GPU</th>
<th>GPU only</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overhead 1</td>
<td>0 ms</td>
<td>1.04 ms</td>
<td>0.96 ms</td>
</tr>
<tr>
<td>Matching</td>
<td>139.67 ms</td>
<td>11.03 ms</td>
<td>10.85 ms</td>
</tr>
<tr>
<td>Overhead 2</td>
<td>0 ms</td>
<td>6.69 ms</td>
<td>0 ms</td>
</tr>
<tr>
<td>Reconstruction</td>
<td>589.39 ms</td>
<td>657.14 ms</td>
<td>46.43 ms</td>
</tr>
<tr>
<td>Overhead 3</td>
<td>0 ms</td>
<td>0 ms</td>
<td>80.62 ms</td>
</tr>
<tr>
<td>Total</td>
<td>729.06 ms</td>
<td>675.90 ms</td>
<td>138.86 ms</td>
</tr>
</tbody>
</table>

Table 5.5: Matching 1248 left and and 1278 right 2D primitives. All matches are reconstructed giving a total of 8334 valid 3D primitives.

The tables 5.3 and 5.5 show execution times for the different steps of the sparse stereo algorithm and how overhead affects the results. But the execution time is of course also depending on the number of input 2D primitives which is fixed in the tables.

In the following we compare the total execution time of the CPU and GPU sparse stereo implementation while varying the number of input primitives. The number of primitives is changed by adjusting the extraction parameters.

In figure 5.8 we see the timing of the matching. It is clear how the poor strategy of matching primitives with an \(O(m \times n)\) execution time affects especially the CPU version. The GPU uses the high level of parallelism to minimise this effect achieving an almost linear change of execution time with the number of primitives in the image.

The term average number of primitive does not mean that this will be the execution time whenever the number of primitives from the left and right image gives this average value. The variance must of course be small which is also a fair assumption for almost equal stereo images.

Figure 5.9 shows the equivalent measurements for the reconstruction algorithm. Here we do not get a remarkable speed up on the GPU as, for a low number of primitives a single CPU is actually faster. It should however be noted that the GPU computes all reconstructions of the putative matches and transfers all data back to host. Only the best of them is then put into CoViS 3D primitive objects. This means that all reconstructions could be made available for additional processing if required. The CPU version on the other side computes what is locally the best match.

Finally figure 5.10 compares the total execution time of the GPU implementation and the CPU implementation. We show no results of the combined approach using the GPU for the matching and the CPU for the reconstruction but it is expected to lie inbetween the two curves. A moderate speed up has been achieved but it is not more than we could expect from coarse parallel execution on CPUs.
Figure 5.8: Matching step on CPU or GPU.

Figure 5.9: Reconstruction step on CPU or GPU.
5.5 Discussion

The entire sparse stereo algorithm has been speeded up by transferring execution to the parallel platform of the GPU compared to a single threaded CPU version. The two implementations compute the exact same matches and reconstructions. But was it worth the effort? The real-time vision machine where the sparse stereo should be integrated has a high load on the CPUs so the more load we can transfer to the GPUs the better.

An objective comparison however, would conclude that this kind of problem is in a grey zone area. The nature of the algorithm makes it difficult to utilise the full compute capability of the GPU and thus, we do not necessarily achieve a speed up of a factor 100 even though we have hundreds of stream processors at our disposal. More realistic speed ups are smaller and depend on the size of the problem, the amount of parallelism and the resources and work hours we put into it.

We must also consider what could have been achieved if the problem had been addressed differently. For this project we wanted to test an implementation on the GPU, and with more optimisation, and future developments that might resolve some of the limitations, we could still expect superior performance on a large number of primitives. But what if the time had been spent on optimisation for the CPU? We could argue that a multi-core CPU with coarse parallel execution would be just as beneficial for the reconstruction – especially if the memory intensive computations of covariance matrices are required. The algorithm could easily be divided into say four threads and processed in parallel on the increasing number of cores we will most likely see in future CPUs.

The matching process were speeded up the most and the $O(n^2)$ running time on
CPU could really be reduced using the GPU. Better memory optimisation and the introduction of buckets could however increase performance further and allow for real-time processing of even more 2D primitives. The bucket algorithm would require pre-sorting of the primitives in the right image, probably by the CPU, into areas corresponding to the orientation of the epipolar lines from the left image. The sorting did not occupy much of the running time of the matching on the GPU implementation but it has not been compared to the sorting on the CPU.

The reconstruction process proved more challenging as expected, with the bottleneck being the high number of required memory transfers and a relatively low level of parallelism. If we could do without the covariance matrices, meaning that the real-time system would compromise some of the computations to get a faster system, the performance could be drastically improved and allow for many more 3D primitives.

High performance gains were also achieved by cleaning the original algorithm and getting an in-depth understanding of the functionality. The many years of development and expansions of the code have at some points led to redundant reiterations of multiple steps. This is of course optimised for the CPU in the performance comparison and tests in section 5.4.
Chapter 6

Conclusion

The sparse stereo algorithm has been successfully implemented using CUDA as defined in the problem statement. A total of four kernels have been developed for the matching, condensing, sorting and reconstruction tasks and the results are in agreement with the original CoViS implementation.

The GPU implementation has proved to be faster than the corresponding CPU implementation but the overhead for transfer of data and execution control combined with the memory intensive and complex algorithm makes it difficult to achieve remarkable speed ups. The exact performance gain depends on the number of 2D primitives and how they match to 3D primitives.

Further optimisation will always be possible as it is a question of the amount of resources to use. Regarding the GPU implementation especially the development of more complex data structures could provide for better performance as the bandwidth is not fully utilised yet.

Most importantly the work has provided a good basis for an evaluation of which problems suits the massively parallel platform and where the challenges occur in practice.

The dense stereo processing fits nicely as the problem is similar to the task the GPU is originally optimised for. Both cooperation among threads and optimal usage of memory bandwidth can be achieved and it will thus be the CPU far superior.

The matching of the primitives and the following sorting of stereolinks fit the GPU nicely, when the size of the problem is large enough. Memory sharing among threads and many small computations to schedule among compared to the number of memory transfers makes the GPU perform quite well.

The reconstruction of 3D line segments proved to be more problematic as the computation of covariance matrices are memory intensive and the number of threads to schedule among are quite low. A coarse CPU parallel processing has proven to be just as fast on this part. If we could do without the covariance matrices in the real-time system the computation could be offloaded to the GPU with better performance gain.

The sparse stereo CUDA implementation has been integrated in a pipelined real-time system running the entire CoViS primitive extraction at approximately 20Hz on 512x512 image sizes.
An experimental integration of dense stereo algorithms and texel computation into CoViS has also been succeeded even though the time was limited. The texels have been visually evaluated and seem to provide a good basis for surface estimation.

6.1 Future work

The developed CUDA functions are directly compatible with the CoViS framework and the automake based compilation scheme as shown in appendix B. A pipelined vision demonstrator application (based on CoViS) is already using our sparse stereo algorithm implementation that processes the matching on the GPU and the reconstruction on a CPU.

Some of the obvious optimisation possibilities listed in this report could be implemented as for example the bucket algorithm technique and the parallelism could be taken to another level by introducing streams. In a stream the GPU can process a series of data transfers and kernels automatically in a given order and the CPU is free to process other stuff meanwhile. This has been avoided so far because of implementational difficulties regarding the pipelined vision demonstrator.

A dense stereo algorithm running on the GPU has also been integrated in the vision demonstrator but as the texels are still experimental, more work is required here to increase the level of abstraction to larger surfaces and for disambiguation of local image structures.
Glossary

CoViL  Cognitive Vision Lab. 5

CoViS  Cognitive Vision Software. 2, 5, 6, 9, 12, 15–17, 22, 28, 35, 36, 38, 40, 41, 47, 48, 59

CPU  Central Processing Unit. 1, 5, 6, 11, 12, 21, 30, 32, 34, 37, 39–42, 44, 45, 47, 48

CUDA  Compute Unified Device Architecture - GPGPU framework by NVIDIA. 2, 3, 6, 21, 30, 31, 33–41, 47, 59

Dense stereo  Dense stereo refers to a dense pixel based processing of stereo images to find matching points and compute a complete depth/disparity map. Known to be computationally expensive. 2, 9

Disparity  Disparity is the distance a single point moves (along the epipolar line) between a pair of stereo images. Inversely proportional to the distance between the object and parallel image planes. 22

Epipolar line  Given a pair of stereo images, the calibration data and a point in the left image we can compute the epipolar line of which we expect to find the matching point in the right image due to the geometry. This constraint highly reduces the searchspace. 22, 24, 26

FORK  Danish abbreviation for preliminary thesis study project. 2, 11, 12, 21

FPGA  Field Programmable Gate Array - programmable logic using a hardware description language. 2, 11, 12, 21

GPU  Graphics Processing Unit. 1, 2, 5, 6, 9–12, 15, 16, 21, 30–32, 34, 35, 37, 39–42, 44, 45, 47, 48, 59

Junction  In this context a junction is a subtype of primitive that represents a junction feature in an image. 9

Kernel  This is the basic function in CUDA written to run only on the device or in emulation mode on host. Each thread in CUDA is said to be an instance of the kernel and the kernel is executed many times in parallel. 2, 21, 31

Line segment  In this context a line segment is a subtype of primitive that represents a line feature in an image. 1, 6
**Monogenic filtering** Filter operations on a multi dimensional signal as e.g. an image. 5

**Primitive** Visual primitives in this context are local image feature descriptors with attributes used in CoViS to represent image information in a condensed way. 6, 21, 40

**Rectification** The process of projecting both images onto the same image plane to make the epipolar lines align horizontally. 5

**SIMD** Single Instruction Multiple Data - data level parallel processing scheme. Data width is exposed to software. 31

**SIMT** Single Instruction Multiple Thread - thread level parallel processing scheme in CUDA. Thread divergence is allowed but should be minimised for overall performance. 32

**Sparse stereo** Sparse stereo refers to a stereo computation based on sparse image representations where significant image features have been extracted. 2, 9, 21, 24

**SSD** Sum of Squared Difference. 2, 19

**Texel** Short for *texture element* which in this context is a primitive subtype describing a textured patch in the image. Is sometimes also called a patchlet. 2, 9, 48

**Undistortion** No camera lens is perfect. It will always distort images and undistortion is the process of making straight lines appear straight in the image. 5

**Warp** This is the basic scheduling unit in CUDA. Each block of threads is divided into warps upto CUDA version 1.3 of size 32. Serialisation of branch divergence occurs inside the warp. 31, 32
Bibliography


Appendix A

Introduction to CUDA

This appendix introduces important concepts of the CUDA architecture with respect to our work. For a comprehensive introduction to CUDA we refer to the programming guide from NVIDIA for CUDA version 2.0 [2]. An introduction to the Tesla architecture which the latest NVIDIA GPUs are based upon is found in [4, 5, 3].

CUDA is designed to overcome the challenge of creating software that scales its parallelism to fit a varying number of processor cores. It is basically an extension of C consisting of three key abstractions:

1. a hierarchy of thread groups
2. shared memory
3. barrier synchronisation

Using these abstractions NVIDIA argue that the programmer is naturally guided into the phase of partitioning a problem into coarse sub-problems that can be solved independently in parallel and then into finer pieces that can be solved cooperatively in parallel. The GPU is especially well-suited to data-parallel problems where the same instruction is executed on many data elements in parallel. High arithmetic intensity is also preferred whereas the memory operations should be kept at a minimum. Branching, where threads running in parallel take different execution paths, can affect GPU performance negatively. All these issues regarding GPU performance relates to the hardware architecture and how it differs from the regular CPU.

In short the GPU devotes more transistors to data processing whereas the CPU has cache and sophisticated flow control. Both architectures use these features as different ways to hide memory latency, which according to A Landscape Of Parallel Computing: A View From Berkeley [14] is a major bottleneck in modern high performance systems.

A.0.1 CUDA programming

From a programming point of view the only thing we need to know about the hardware architecture is how threads are organised into a hierarchy of blocks in a
grid and how this affects the execution. The hierarchy is illustrated in figure A.1 on page 55.

Function type qualifiers specify where a function can be invoked and where it is executed. The _global_ function qualifier specifies a kernel. This is a special type of function which is invoked by the host (the CPU) and executed \( N \) times in parallel by \( N \) threads on the device (the GPU). Listing A.1 illustrates the basic definition of a kernel and how it is invoked from the main program residing on the host. Note the special \(<<<<...>>>\) syntax of the kernel invocation. The first parameter is the specification of a one- or two-dimensional grid of blocks. The second parameter is a specification of a one-, two- or three-dimensional block of threads. The chosen dimensions for a thread hierarchy should depend on the nature of the data elements. In the example of listing A.1 two vectors are added together which is a one-dimensional problem. Each thread within a grid of blocks will execute the same code but on different data elements. In this case two values, one from each of the input vectors are added together. Through the built-in \texttt{threadIdx.x} variable a specific thread gets its own identity and can work on different data elements. It is however also possible for threads to follow independent paths of execution through for example loops or if-statements. This will very likely affect overall utilisation of the device and result in poor performance. So for logical correctness of a program the concept of similar execution paths can be safely ignored, but if optimal performance is desired we need to consider how threads expected to take similar execution paths are organised into blocks to take full advantage of the special hardware architecture as described in section A.0.2.

```c
// Kernel definition
__global__ void vecAdd(float * A, float * B, float * C) {
  int i = threadIdx.x;  
  C[i] = A[i] + B[i];  
}

int main() {
  // Kernel invocation
  vecAdd <<<1, N>>> (A, B, C);  
}
```

Listing A.1: Kernel specification and invocation.

On the Tesla architecture a block can contain a maximum of 512 threads. All threads of a block will reside on the same multiprocessor and is thus able to cooperate through fast shared memory. Furthermore they can synchronise for example memory references by calling the \texttt{syncthreads()} intrinsic function. This function is very lightweight and ensures in a fast way that all threads have reached a certain point in the execution path before any of them are allowed to continue. Threads from different blocks should execute independently as the only means for communication and thus cooperation are via global memory which in this context is considered very slow. Different blocks can thus be scheduled and executed independently across any number of multiprocessors and hence the code is scalable.

Each thread has private registers on the multiprocessor. They are assigned to the thread from a pool of available registers on the multiprocessor. Each kernel has its own register requirements and this determines how many blocks will be able to
CUDA threads may access data from multiple memory spaces during their execution as illustrated by Figure 2-2. Each thread has a private local memory. Each thread block has a shared memory visible to all threads of the block and with the same lifetime as the block. Finally, all threads have access to the same global memory.

There are also two additional read-only memory spaces accessible by all threads: the constant and texture memory spaces. The global, constant, and texture memory spaces are optimized for different memory usages (see Sections 5.1.2.1, 5.1.2.3, and 5.1.2.4). Texture memory also offers different addressing modes, as well as data filtering, for some specific data formats (see Section 4.3.4).

The global, constant, and texture memory spaces are persistent across kernel launches by the same application.

Each block can have shared memory which also resides on the multiprocessor and can be accessed by all other threads in the block. This is used to communicate sub-results between different threads.

All threads have full access to global device memory (DRAM managed by the GPU) and read only access to constant and texture memory spaces which is optimised for different memory usages. This memory is persistent through the entire application. Allocation and deallocation on the GPU as well as transfers between device and host are handled explicitly in the code.

A.0.2 Hardware model

Figure A.2 from NVIDIA's CUDA programming guide [2] illustrates the concept of a set of Single-Instruction-Multiple-Thread - SIMT - multiprocessors with shared memory. The SIMT term is NVIDIA's name for their streaming multiprocessors (SM) which is akin to the well known SIMD (Single Instruction Multiple Data).
classification of Flynn’s taxonomy, where the size of the dataset or vector are fixed and the exact same operation is performed on all elements. When a multiprocessor (the SIMT unit) on the NVIDIA GPU is given one or more thread blocks to execute, it will split them into *warps*, which is the basic scheduling unit. A warp contains 32 parallel threads (so a block should always have at least 32 threads and the maximum is currently 512) and current GPUs with the Tesla architecture has 8 scalar processors (SP) in each multiprocessor (SM). Every instruction issue time the scheduling unit of a multiprocessor selects a warp that is ready to execute and the common instruction unit issues the next instruction to active threads in the warp. Only threads that agree on the current execution path will be active. With eight scalar processors and 32 threads in a warp an instruction issue time will require four clock cycles to execute for instance a bitwise operation, compare or single-precision floating-point add, multiply or combined multiply-add. Many other operations are supported but requires more clock cycles. The CUDA programming guide [2] provides a good overview in chapter 5 about Performance Guidelines.

A block of threads are always split into warps in the same deterministic way and it is thus recommended to try to ensure that threads in the same warp often will follow similar execution paths. They don’t have to however, as the threads are allowed to follow completely different paths, and hence we have the SIMT architecture. The execution of threads in a warp that diverge will be serialised until they converge back to the same path.

![CUDA hardware model](image-url)  
*Figure A.2: CUDA hardware model with a set of SIMT multiprocessors.* [2]
A multiprocessor can contain up to eight concurrent thread blocks. And it is important to ensure a sufficient number of warps on each multiprocessor so the scheduler is able to hide memory latency by invoking other threads with sufficient arithmetic instructions. A memory instruction includes reads and writes to shared memory and global memory and takes 4 clock cycles for a warp. In addition an access to global memory which is not cached takes 400-600 clock cycles of memory latency. This cost of device memory access clearly illustrates the importance of minimising the number of accesses and follow the right access patterns to achieve the desired speed-up.

Memory accesses should be coalesced when possible to ensure efficient use of memory bandwidth. This means that threads in a half-warp (the first 16 or last 16 threads) can access data in a single memory transaction by fulfilling some requirements. With the newest compute capability (which can be thought of as NVIDIA's different generations of CUDA) the only requirement to coalesce memory accesses into a single transaction is that all words must lie in the same segment of size 32 bytes, 64 bytes or 128 bytes. Multiple threads can access the same address and the words don't need to be in sequence.

A multiprocessor has a fixed number of 32 bit registers that the threads currently residing on the multiprocessor must share. During compilation of the code the required number of registers for a specific kernel is reported and thus the programmer can choose a proper execution configuration. Allocating more threads per block will give efficient time slicing, but leaves fewer registers per thread. If a kernel invocation requires more registers per block than is available on a multiprocessor the launch will fail. The same accounts for shared memory and thus registers and shared memory are the resources limiting the number of threads that can reside on a multiprocessor.

On our GeForce GTX 280 supporting compute capability 1.3 we find 30 independent multiprocessors. Each multiprocessor has 16384 32-bit registers and 16 KB of shared memory. The kernel size is limited to 2 million PTX instructions. 32 warps can be active on the same multiprocessor and thus 1024 threads can be active provided that they do not require more shared memory or registers than the multiprocessor offers (an average of for example 16 registers per thread). Shared memory are assigned on a per block basis and a multiprocessor can execute a maximum of 8 concurrent blocks (only if they each have an average of 128 threads - one fourth of the maximum) and they can then get an average of 2 KB shared memory per block.

These issues give rise to the occupancy level of the GPU, which is the ratio of active warps per multiprocessor to the maximum number of active warps.
Appendix B

Integrating CUDA in an existing C++ project

CoViS is compiled using an automake setup and compilation script. The compilation of the additional functionality using CUDA has been integrated in this system with a clear definition of what should be exposed to the different compilers. The basic concept is shown in figure B.1. From CoViS the new functionality is available through interface classes that can handle the object-oriented data structures and prepare for the execution path to jump via the `extern "C"` directive. This is the arrows in the middle of figure B.1.

The C entry functions receive input data and prepare structs. The memory allocation on the GPU is handled before the actual kernel is invoked. The parallel processing of threads of a kernel is shown to the right. When the computation has finished, the data flow will be in the opposite direction. It now goes from the GPU memory to the C functions also compiled by CUDA's NVCC. The data is extracted from the structs and transferred to the interface classes that will prepare the results for the continued processing in CoViS.

![Figure B.1: Compiler perspective](image)

Figure B.1: Compiler perspective