

# „Massive Parallel Image Processing“

## Term Project

Department of Computer Science  
University of Applied Science Rapperswil

Fall Term 2010

Authors: Markus Hofmann, Tobias Binna  
Advisor: Prof. Josef M. Joller  
Internal Co-Examiner: Prof. Peter Sommerlad



# Abstract

Massive Parallel Image Processing: How can image data be processed in an extreme parallel manner. In this project, we analyze methods on how image segmentation could be developed with CUDA and give an overview of the advantages and disadvantages by using CUDA in image processing.

Image processing algorithms are more often than not quite complex and a special part of them - the image segmentation task - can become quickly very long-winded because each pixel has to be analyzed and processed repeatedly. NVIDIA has provided a technology called CUDA, based on the C programming language that supports calculations on their graphics cards with thousands of concurrent threads. For this reason the use of CUDA to solve image segmentation algorithm problems is obvious and the applicability of CUDA in this area should be investigated.

We have developed an application that implements an automatic seeded region growing algorithm which divides a given image into color based regions, using the power of NVIDIA's graphics processing unit on most of the partial sub algorithms. The application delivers an output where the regions in the resulting image are colorized with their color mean value additionally to a console output, showing the time required for each algorithm part. The application can be launched with different command line arguments which provides the ability to observe, how the results differ while playing with various threshold values.

Equally important to the implementation documentation we elaborate on the lessons learned from the challenges and performance insights. In addition, we deliver information about the expedient use of the CUDA technology and what definitely should be avoided.





# Declaration

We, Markus Hofmann and Tobias Binna declare

- that this term project and the work presented in it is our own, original work.
- All the sources we consulted and cited are clearly attributed. We have acknowledged all main sources of help.

Rapperswil, December 23, 2010

---

Markus Hofmann

---

Tobias Binna



# Contents

1	Management Summary	3
1.1	Motivation . . . . .	3
1.2	Goals . . . . .	3
1.3	Results . . . . .	4
2	Technical Introduction	5
2.1	Problem Domain . . . . .	5
2.2	Image Processing . . . . .	5
2.2.1	Image Segmentation . . . . .	6
2.2.2	Segmentation Algorithms . . . . .	7
2.3	Automatic Seeded Region Growing Algorithm . . . . .	7
2.3.1	Overview of the automatic seeded region growing algorithm . . .	7
2.3.2	Color Space Transformation . . . . .	8
2.3.3	Automatic Seed Selection Algorithm . . . . .	9
2.3.4	Seed Region Growing Algorithm . . . . .	10
2.3.5	Region-Merging Algorithm . . . . .	11
2.4	CUDA Technology . . . . .	11
2.4.1	The Concept of Parallel GPU Programming . . . . .	12
2.4.2	Existing CUDA Libraries . . . . .	13
3	Method	15
3.1	Architecture Overview . . . . .	15
3.1.1	Two Layer Architecture . . . . .	15
3.2	Logical View . . . . .	15
3.2.1	Package "algorithm" . . . . .	16
3.2.2	Sub-Package "seed selection" . . . . .	16
3.2.3	Sub-Package "seed region growing" . . . . .	16
3.2.4	Package "datatype" . . . . .	16
3.2.5	Package "helper" . . . . .	16
3.2.6	Package "test" . . . . .	17
3.3	Algorithm Implementation . . . . .	17
3.3.1	Interface to the OpenCV library . . . . .	17
3.3.2	Host Image Data Allocation . . . . .	17
3.3.3	Standard Control Flow in CUDA Functions . . . . .	19
3.3.4	Automatic Seed Selection Implementation . . . . .	19
3.3.5	Seed Region Growing Implementation . . . . .	23
3.4	Testing . . . . .	25
3.5	Time Logging . . . . .	25

3.6	Memory Management . . . . .	26
3.6.1	CUDA Memory Allocator . . . . .	26
3.7	Supporting Libraries . . . . .	27
3.7.1	CUDA Data Parallel Primitives Library . . . . .	27
3.7.2	CUDA Utility Library . . . . .	28
4	Results . . . . .	29
4.1	Segmentation Results . . . . .	29
4.1.1	Threshold Selection . . . . .	35
4.2	Performance . . . . .	36
4.3	Known Issues . . . . .	37
4.3.1	Image Dimensions . . . . .	37
5	Conclusions . . . . .	39
5.1	Conceptual Issues . . . . .	39
5.1.1	Memory Management . . . . .	39
5.1.2	Data Representation . . . . .	40
5.2	Implementation Issues . . . . .	40
5.2.1	Code Organization . . . . .	40
5.2.2	C/C++ Mixing/Combining . . . . .	42
5.2.3	Thrust Library . . . . .	43
5.2.4	Reuse device functions . . . . .	44
5.2.5	Structures of Arrays over Arrays of Structures . . . . .	45
5.3	Literature . . . . .	46
5.3.1	Programming Massively Parallel Processors . . . . .	46
5.3.2	CUDA by Example . . . . .	47
A	Development Environment Setup . . . . .	49
A.1	Hardware and Software Tools . . . . .	49
A.1.1	Hardware . . . . .	49
A.1.2	Software . . . . .	49
A.2	Setup CUDA . . . . .	50
A.3	Setup Eclipse CDT with CMake and CUDA . . . . .	50
A.3.1	Add File Type <code>.cu</code> . . . . .	50
A.3.2	Setup CMake and CMake Configuration in Eclipse . . . . .	51
	Bibliography . . . . .	53

## List of Figures

2.1	Difficulties in Image Segmentation . . . . .	6
2.2	Overview of the Automatic Seeded Region Growing Algorithm . . . . .	8
2.3	CUDA Thread Organization . . . . .	12
3.1	CPU/GPU Code Separation . . . . .	15
3.2	Logical View . . . . .	16
3.3	Algorithm Workflow . . . . .	18
3.4	Connected Component Analysis . . . . .	21
3.5	Four Steps in Component Analysis (Source [7]) . . . . .	22
3.6	Block-wise Steps in Component Analysis . . . . .	23
4.1	Seed Regions: Bird . . . . .	30
4.2	Good Case Segmentation . . . . .	31
4.3	Correct Segmentation with unwanted Output . . . . .	32
4.4	Seed Regions: Bridge . . . . .	32
4.5	Failing Segmentation . . . . .	34
4.6	Lena Reference Segmentation . . . . .	34
4.7	Threshold Selection . . . . .	35
5.1	Array of Structures . . . . .	45
5.2	Structure of Arrays . . . . .	46



## List of Tables

4.1	Launch Configuration, Bird . . . . .	30
4.2	Launch Configuration, Bridge . . . . .	31
4.3	Launch Configuration, Gull . . . . .	33
4.4	Launch Configuration, Lena . . . . .	34
4.5	Runtime Example bird.jpg . . . . .	36
4.6	Runtime Example bridge.jpg . . . . .	37
A.1	Project Hardware . . . . .	49
A.2	Project Software . . . . .	49





# Listings

3.1	CUDA Streams . . . . .	19
3.2	CUDPP Plan Configuration and Scan . . . . .	20
3.3	Time Logging Sample . . . . .	25
3.4	CUDA Memory Allocator 1D . . . . .	26
5.1	CUDA Header File host_function.cuh . . . . .	41
5.2	CPU Side Code . . . . .	41
5.3	GPU Side Host Function and CUDA Kernel . . . . .	42
5.4	Use of float4 Operator Overloads . . . . .	44
5.5	float4 "Plus" Operator Overload . . . . .	44
5.6	Array of Structures . . . . .	45
5.7	Structure of Arrays . . . . .	46
A.1	Check for FindCUDA.cmake . . . . .	51
A.2	CMake CUDA Executable . . . . .	51



# 1 Management Summary

In this project, we analyze methods on how image segmentation could be developed with NVIDIA's CUDA technology. The project was initiated by the Institute for Internet-Technologies and Applications at the University of Applied Sciences Rapperswil.

## 1.1 Motivation

The increased complexity of computable problems desires more and more faster hardware. This fact is by far not new, but the needed performance can no longer be achieved by just increasing the clock speed of processors. Hence, the approach to parallelize things has become more and more popular. NVIDIA provides a technology called CUDA, based on the programming language C, where computationally intensive operations can be transferred to a CUDA capable graphics card (almost all new graphics cards from NVIDIA are CUDA aware) where the tasks are calculated in an massive parallel way, using the graphics processing unit (GPU).

Image processing algorithms are more often than not quite complex, because computers are not capable to logically interpret and extract information as seen by human beings. A special part of an image processing task is the image segmentation, which is responsible for the isolation of objects in an image by finding regions based on the color value of its containing pixels. Because each pixel has to be analyzed individually and mostly be compared with its neighboring pixels repeatedly, the algorithm can become quickly very long-winded and performance intensive.

The idea of parallelism and the complexity of image processing tasks leads to the desire to investigate the applicability of CUDA to this domain.

## 1.2 Goals

The goal of this project is to gain an overview of the advantages and disadvantages by using CUDA in image processing. It should give an outline of how and if the CUDA technology can be used for image segmentation algorithms. The resulting work should help to support further implementation attempts using this technology and depict, which barriers have to be conquered.

To demonstrate the feasibility, a specific image segmentation algorithm might be implemented which divides an image into regions, using the CUDA technology. Supplementary, the implementation shall afford a small testing environment where images can be segmented with different input values.

### 1.3 Results

We have developed an application that implements an automatic seeded region growing algorithm which divides a given image into color based regions, using the power of NVIDIA's graphics processing unit on most of the partial sub algorithms.

The application delivers an output where the regions in the resulting image are colorized with their color mean value additionally to a console output, showing the time required for each algorithm part. The ability to launch the application with different command line arguments - namely an image and two threshold values for the similarity and the Euclidean distance - allows an evaluation of the algorithm with different input data. Therefore, this application acts as a small testing environment where it is possible to play with various threshold values.

Furthermore we elaborate on the lessons learned from the challenges and performance insights and deliver information about the expedient use of the CUDA technology and what definitely should be avoided.

## 2 Technical Introduction

### 2.1 Problem Domain

Today people expect their computers to do everything. Indeed everything that is computable can be done by machines and as we start understanding difficult problems better and better scientist and programmers start writing more complex programs to be solved by machines. This is also true for the whole area of image processing as images get larger and hence computationally more complex.

With the increased complexity of the problems the desire for faster hardware increases as well. To be able to solve more complex problems programmers must have possibilities to get more power out of these machines, however this performance gain can not be achieved by just increasing the clock speed of today's processors. As a result the industry is heading towards parallel programming concepts.

### 2.2 Image Processing

Human beings can see and extract information from images that machines are not capable of. Most of the time we can just see the context of an image what it is about, which object it shows or what text is written on it. Machines are not capable of just seeing things they need to process an image, enhance or adjust it to be able to "see" something. As described by [20] these computations can be separated into four types of image processing operations:

- *Pixel Operations:* The output of processing a certain pixel depends only on the input on that single pixel. Typical pixel operations include thresholding, image inverting, addition, subtraction or color space transformations.
- *Local (neighborhood) operations:* The output at a pixel depends on the local neighborhood of that pixel meaning the output is dependent from some of the surrounding pixels. Many filter operations can be classified as this type of operation for instance smoothing filters such as averaging filter or median filter.
- *Geometric Operations:* The output of a certain pixel depends on some input pixels given by a geometric transformation. On the contrary to global operations the input for geometric operations are given by a subset of all pixels. These input pixels do not have to be in the local neighborhood of the considered pixel.
- *Global operations:* The output at a pixel position depends on all pixels in an image. In contrast to local or geometric operations where only a subset of pixels act as input these operations are usually computationally much more intensive.

In fact, all these operations show up in many algorithms in the domain of image processing.

### 2.2.1 Image Segmentation

Image segmentation is a process to extract objects or segments and classify each pixel as either a background pixel or an object pixel. In many situations the system is interested in only some special parts of an image thus the system tries to isolate these parts. Especially in a sequences of images such as video streams the background is usually static and therefore boring. What is of interest are the actors in the scene and those things that change over time. These objects we would like to be extracted by a image segmentation algorithm. Unfortunately correct segmentation is not trivial and actually one of the most error-prone steps in an image processing system [9]. People can segment an image based on the knowledge of its context. For instance most people probably would segment image 2.1 into the segments: water, sky, bridge, the forest in the background and the tree in the foreground. However a segmentation algorithm most likely would segment the image based on color values and thus not recognizing the water as a dedicated segment.



**Figure (2.1)** Difficulties in Image Segmentation

In almost all situations image segmentation is only a part of the overall process of an image processing system. Possibly it provides the knowledge for a following stage of the process with the information of interest, leaving aside non relevant data.

### 2.2.2 Segmentation Algorithms

In general there are four types of segmentation algorithms techniques [20] with different levels of complexity. Namely these four techniques are thresholding, boundary-based, region-based, and hybrid techniques.

Thresholding is one of the straight forward approaches to this problem, and in many situations an adequate method to solve a segmentation problem. Thresholding assumes that background pixels and object pixels are clearly separated by their color or intensity. However in situations when the shadow of an object makes background pixels appearing darker then they are in truth, the thresholding algorithm probably fails and classifies background pixels as object pixels. A common approach to overcome this problem is adaptive thresholding, which uses different thresholds for different regions of the image. In addition to thresholding, other more complex techniques consider their local neighborhood to distinguish between object and background pixels. Boundary-based or edge extraction methods focus on pixel discontinuity in their local neighborhood, whereas region-based methods rely on the similarity of neighboring pixels to make the classification.

The forth category of techniques are called hybrid methods and are combinations of the approaches described above. For example [18] describes a method to combine boundary detection and region growing to achieve better segmentation results.

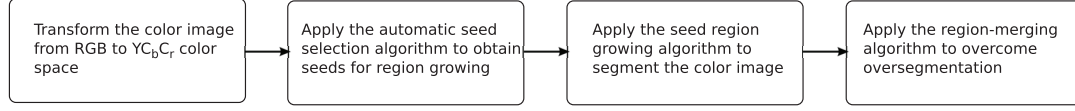
## 2.3 Automatic Seeded Region Growing Algorithm

Seeded region growing is one of the hybrid-methods presented by [5]. It is based on the conventional region growing postulate of similarity of pixels within regions, but it is in its mechanics closer to the watershed method. The watershed method is applied to the gradient of an image, where this gradient can be seen as a topography with boundaries between regions as ridges. Segmentation is then performed by literally flooding the topography from the seed or source pixels. Like the watershed algorithm and in contrast to the boundary-based methods, the seeded region growing technique is guaranteed to produce closed boundaries. As the segmentation is done based on the color values of the individual pixels it is assumed that individual objects or regions are characterized by connected pixels of a similar value. Thus, the technique may not be suitable for highly textured images. Our implementation is based on the automatic seeded region growing algorithm described by [21].

### 2.3.1 Overview of the automatic seeded region growing algorithm

Figure 2.2 presents an overview of the different stages of the automatic seeded region growing algorithm. In the first step the image is transformed from  $RGB$  to  $YC_bC_r$  color space. Secondly, the seed pixels are identified as input for the following region growing algorithm. The seed region growing algorithm is used to segment the image into individual regions. Because these steps usually result in an over segmented image the region-merging algorithm is applied in a final step to merge similar and small regions.

The following sections will describe the individual stages of the algorithm in more detail.



**Figure (2.2)** Overview of the Automatic Seeded Region Growing Algorithm

### 2.3.2 Color Space Transformation

A captured color image is stored in *RGB* values. Even if the *RGB* model is suitable for color display, it is ill-suited for color analysis because of its high correlation among *R*, *G*, and *B* components [20]. In addition, the distance in *RGB* color space does not represent the perceptual difference in a uniform scale. In image processing and analysis, the *RGB* color space is often converted into more applicable color spaces. The *YC<sub>b</sub>C<sub>r</sub>* color space has been widely chosen for color segmentation. The reasons are as follows:

- *YC<sub>b</sub>C<sub>r</sub>* color space is used in video compression standards (e.g., MPEG and JPEG)
- The color difference of human perception is related to the Euclidean distance of the *YC<sub>b</sub>C<sub>r</sub>* color space
- The intensity and chromatic components can be easily and independently controlled

*YC<sub>b</sub>C<sub>r</sub>* is a family of color spaces used as a part of the color image pipeline in video and digital photography systems. *Y* is the luma component (background brightness), where *C<sub>b</sub>* and *C<sub>r</sub>* are the blue-yellow chroma and red-green chroma components. *YC<sub>b</sub>C<sub>r</sub>* is not an absolute color space, it is only a way to encode *RGB* information. The actual color displayed depends on the actual *RGB* colorants used to display the signal. Therefore a value expressed as *YC<sub>b</sub>C<sub>r</sub>* is only predictable if standard *RGB* colorants or an ICC<sup>1</sup> profile (a standardized set of data that characterizes a color input or output device) are used. [4]

If the *RGB* data has a range of 0-255 per channel, (which is common when images exist in digital format), the equations 2.1 to 2.3 and 2.4 to 2.6 can be used to transform the color spaces [8].

#### RGB to YCbCr transformation

To transform the *RGB* values to *YC<sub>b</sub>C<sub>r</sub>*, following equations are used:

$$Y = 0.257 \cdot R + 0.504 \cdot G + 0.098 \cdot B + 16 \quad (2.1)$$

$$C_b = -0.148 \cdot R - 0.291 \cdot G + 0.439 \cdot B + 128 \quad (2.2)$$

$$C_r = 0.439 \cdot R - 0.368 \cdot G - 0.071 \cdot B + 128 \quad (2.3)$$

---

<sup>1</sup> International Color Consortium



### YCbCr to RGB transformation

To transform the  $YC_bC_r$  values back to  $RGB$ , following equations are used:

$$R = 1.164 \cdot (Y - 16) + 1.596 \cdot (C_r - 128) \quad (2.4)$$

$$G = 1.164 \cdot (Y - 16) - 0.813 \cdot (C_r - 128) - 0.391 \cdot (C_b - 128) \quad (2.5)$$

$$B = 1.164 \cdot (Y - 16) + 2.018 \cdot (C_b - 128) \quad (2.6)$$

### 2.3.3 Automatic Seed Selection Algorithm

To select pixels as seed pixels we must ask ourself what are good seed pixels and which criteria must be met by the algorithm to give best results. The proposed seed selection algorithm is based on three criteria that must be satisfied by the selection algorithm:

1. a seed pixel must have high similarity to its neighbors
2. for an expected region, at least one seed pixel must be generated
3. seeds for different regions must be disconnected

The similarity value of a pixel tells us how similar it is compared to its eight neighbors. Considering the  $3 \times 3$  neighborhood of a pixel its similarity is calculated as follows:

1. Calculate the mean value for each of the  $Y$ ,  $C_b$  and  $C_r$  components in the  $3 \times 3$  region as

$$\bar{x} = \frac{1}{9} \sum_{i=1}^9 x_i \quad (2.7)$$

where  $x$  can be  $Y$ ,  $C_b$  or  $C_r$ .

2. Calculate the standard deviations of  $Y$ ,  $C_b$  and  $C_r$  using

$$\sigma_x = \sqrt{\frac{1}{9} \sum_{i=1}^9 (x_i - \bar{x})^2} \quad (2.8)$$

where  $x$  can be  $Y$ ,  $C_b$  or  $C_r$ .

3. Sum the individual standard deviations to calculate the total standard deviation as

$$\sigma = \sigma_Y + \sigma_{C_b} + \sigma_{C_r} \quad (2.9)$$

4. For further calculations the standard deviation is normalized to  $[0, 1]$  by

$$\sigma_N = \frac{\sigma}{\sigma_{max}} \quad (2.10)$$

where  $\sigma_{max}$  is the maximum standard deviation in the image.

5. Finally the similarity of a pixel to its eight neighbors is defined as

$$H = 1 - \sigma_N \quad (2.11)$$

From the similarity in the last equation 2.11 the first condition for a seed pixel is defined as follows:

**Condition 1:** A seed pixel must have the similarity higher than a threshold value  $t_s$ .

In our implementation the threshold value  $t_s$  is given by the user as a command line argument. In production code this threshold should be calculated at runtime e.g. by using Otsu's method [17].

The second condition that has to be fulfilled by a pixel to become a seed pixel is given by the maximum distance to its eight neighbors. The relative Euclidean distance of a pixel to its eight neighbors is calculated as

$$d_i = \frac{\sqrt{(Y - Y_i)^2 + (C_b - C_{b_i})^2 + (C_r - C_{r_i})^2}}{\sqrt{Y^2 + C_b^2 + C_r^2}}, \quad i = 1, 2, \dots, 8 \quad (2.12)$$

where  $Y$ ,  $C_b$  and  $C_r$  represent the reference pixel.

For each pixel the maximum distance to its eight neighbors is determined by

$$d_{\max} = \max_{i=1}^8(d_i) \quad (2.13)$$

Form the maximum distance the second condition on seed pixels is defined as follows:

**Condition 2:** A seed pixel must have the maximum relative Euclidean distance to its eight neighbors, which is less than a threshold value  $t_d$ .

This condition makes sure that a seed pixel is not on the boundary of two neighboring regions. As with the threshold in condition 1 this threshold  $t_d$  must be specified as a command line argument.

Based on condition 1 and condition 2 we get the set of seed pixels. Each connected component of seed pixels now acts as one seed and defines a region. But, in most cases for an expected region several different seeds are detected and therefore the result is likely to be over segmented. This regions can later be merged in the region-merging step.

### 2.3.4 Seed Region Growing Algorithm

After detecting the seed pixels we need to identify the individual regions. This is done by a connected components analysis based on the local neighbor labeling algorithm described in [7]. The implementation details of this algorithm are described in section 3.3.4, Connected Components Analysis and Seed Selection. Later, when each seed pixel

is labeled with a region label the remaining unclassified pixels have to be assigned to one of the identified regions. In this step we do not implement the algorithm as described in [21] because of our focus on parallelizing the problem. For this reason we do not grow the regions pixel by pixel, but try to grow the regions in a faster spreading manner. The method proposed in the original algorithm iterates until all pixels are classified, where in each iteration only one non-seed pixel gets classified. The pixel that is chosen is the one with the least relative Euclidean distance to its adjacent regions. When analyzing the problem from a parallel viewpoint one has always the tendency to process not just one pixel at a time, but many pixels in one iteration. For this reason we tried to find a way to grow the regions faster, knowing that the algorithm may not perform as exact as the proposed one. In our solution we assign a label to all region-outline pixels per iteration. This method is very similar to the watershed algorithm where the image is kind of flooded from the source pixels. The disadvantage of this method is that the region boundaries of the segmented image are not as exact as with a strict sequential region growing algorithm.

Like the proposed algorithm we calculate the value of the relative Euclidean distance  $d_i$  between each unclassified pixel  $i$  and each of its adjacent regions by

$$d_i = \frac{\sqrt{(Y_i - \bar{Y})^2 + (C_{b_i} - \bar{C}_b)^2 + (C_{r_i} - \bar{C}_r)^2}}{\sqrt{Y_i^2 + C_{b_i}^2 + C_{r_i}^2}} \quad (2.14)$$

where  $\bar{Y}$ ,  $\bar{C}_b$ , and  $\bar{C}_r$  are the mean values of  $Y$ ,  $C_b$ , and  $C_r$  components in the region under consideration.

### 2.3.5 Region-Merging Algorithm

It is very likely that several seed pixels split an expected region into multiple small segments. Because of the common problem of over-segmentation in this algorithm a region-merging algorithm is applied to the result of the previous stage. This algorithm checks the size of each region. If the number of pixels is smaller than a given threshold, the region will be merged into the neighboring region with the smallest color difference. This procedure is repeated until no region has a size less than the threshold. Note that this step is not part of our implementation.

## 2.4 CUDA Technology

NVIDIA CUDA is a revolutionary architecture to deliver performance by computing tasks in a massively parallel manner, using NVIDIA's graphics processing units. Applications running under CUDA architecture can achieve dramatic speedups.

The main concept of CUDA is the ability to transfer computationally intensive operations to the graphics card, where the operation would be processed on the GPU (graphics processing unit) with hundreds of thousands of concurrent threads. During this processing time further operations can be computed on the CPU (central processing unit).

### 2.4.1 The Concept of Parallel GPU Programming

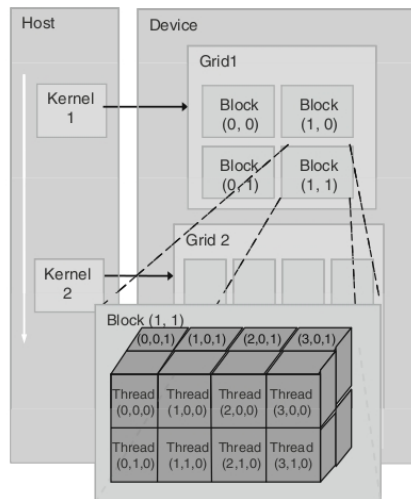
The parallel programming model on a GPU differs from classical task parallelism programming techniques. NVIDIA's basic approach with CUDA is called SIMT (Single Instruction, Multiple Thread) fashion. This approach allows a programmer to write single instructions, which will be executed from multiple threads at the same time.

In CUDA's programming model, the host program launches a sequence of kernels <sup>1</sup> which are organized as a hierarchy of threads. As illustrated in figure 2.3 these threads are grouped into blocks. Blocks themselves are grouped within a grid that represents basically just a kernel launch.

Each thread within a block as well as each block in a grid has a unique index. With the help of the block dimension one can compute the unique global index per thread. This index allows us to assign each thread to e.g. its corresponding element in an array.

Threads in a block can be organized in three dimensions (x, y, z) whereas each block will be executed on a single multiprocessor on the GPU. They can be synchronized and they can share their data with other threads in the same block through shared memory. As multiprocessors can not execute a whole block at once, thread blocks are divided into warp units <sup>2</sup>. So it is best to create thread blocks of size that is a multiple of 32 threads.

When processing large data sets on GPUs one is limited by certain constraints. On graphics cards with CUDA compute capability 1.3 or lower, the maximum number of threads per block is limited to 512. Newer graphics cards with capability 2.x can handle up to 1024 threads per block. The maximum grid size is the same for all devices of all compute capabilities. The grid size is limited to x and y components where each dimension can handle up to 65535 thread blocks.



**Figure (2.3)** CUDA Thread Organization

---

<sup>1</sup> functions that are executed on the GPU

<sup>2</sup> groups of 32 or 64 threads

### 2.4.2 Existing CUDA Libraries

Currently there are four main libraries included in CUDA Toolkit (Version 3.2). In addition, several extra libraries can be found on the Internet, which are supported by NVIDIA.

The CUDA toolkit contains following libraries:

- *CUBLAS* is a library, based on BLAS (Basic Linear Algebra Subprograms) that provides functionality for linear algebra operations on top of the NVIDIA CUDA runtime. The basic model by which applications use the CUBLAS library is to create matrix and vector objects in GPU memory space, fill them with data, call a sequence of CUBLAS functions, and, finally, upload the results from GPU memory space back to the host [13].
- *CUFFT* provides a simple interface for computing parallel Fast Fourier Transformations on an NVIDIA GPU. CUFFT supports in its version 3.2 transformations of complex and real-valued data with up to 128 million elements in any of maximum three dimension (limited by the available GPU memory) [14].
- *CUSPARSE* contains a set of basic linear algebra subroutines used for handling sparse matrices. It is designed to be called from C or C++ host code. Its subroutines can be classified in four categories: Operations between sparse mode vector and dense mode vector, sparse mode matrix and a dense mode vector, sparse mode matrix and a set of dense mode vectors and conversion routines that allow conversion between different matrix formats [16].
- *CURAND* is a library that provides facilities with focus on the simple and efficient generation of high-quality pseudorandom and quasirandom numbers [15].

In addition there are several libraries available on the Internet that are supported by NVIDIA. One popular library that helped us in implementing some of our algorithms is the CUDPP (CUDA Data Parallel Primitives Library), hosted on "google code".

- *CUDPP* is an open source library of data-parallel algorithm primitives such as parallel sort or parallel reduction. Primitives such as these are important building blocks for a wide variety of data-parallel algorithms, including sorting, stream compaction, and building data structures such as trees and summed-area tables [1].

Typically CUDA code is written in standard C language with support for only a subset of C++. An attempt to mimic extended C++ support is taken with the "thrust" project.

- *Thrust* is a CUDA library of parallel algorithms with an interface resembling the C++ Standard Template Library (STL) [3].

### CUDA and image processing

A natural fit for data parallel processing is image processing. Typically (in a CPU implementation) each pixel of an image has to be processed separately in a loop where a lot of data should have been stored for it. One of the biggest advantage in modifying an image on the GPU is that each pixel can be assigned to a thread which causes that a large set of pixels can be processed concurrently. Because of the fact that the data has to be copied between host and device the resulting overhead may compensate the performance gain by the GPU. Therefore two of the main problems in image processing with CUDA is the difficulty to decide how pixels can be well assigned to threads and how the copy operations can be minimized.

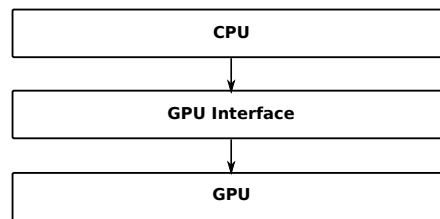
## 3 Method

### 3.1 Architecture Overview

Massive Parallel Image Processing is an implementation of an image segmentation algorithm based on color values. The functions are implemented, using the NVIDIA CUDA framework, where computationally intensive activities are executed on the graphics processing unit (GPU). In addition to the standard CUDA API the application makes use of the CUDPP library, which provides high performance functionalities for several array operations. The implementation of the segmentation algorithm sequentially calls several sub-functions which execute a sub-set of the algorithm.

#### 3.1.1 Two Layer Architecture

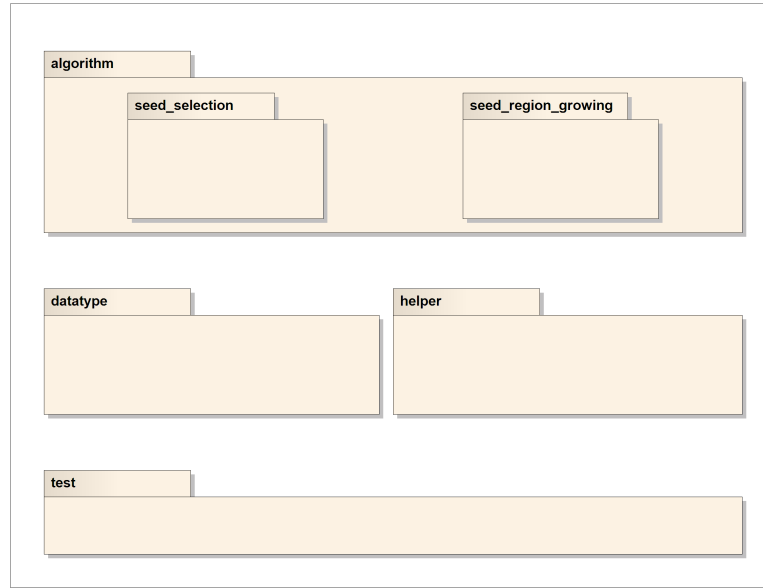
Basically the application is implemented as a two-layer architecture: the host-layer and the device-layer. Each sub-function contains host code which runs on the CPU and device code that will be executed on the GPU. It is responsible for its own memory allocations and CUDA kernel calls, detached from the overall CPU logic. The most important benefit of this architecture is a clear separation of device code and host code and builds a nice and clean interface between the CPU and the GPU side as shown in figure 3.1. Furthermore there is no need to care about device memory allocation or deallocation when calling sub-functions from the host because the device code is not directly visible on the host side.



**Figure (3.1)** CPU/GPU Code Separation

### 3.2 Logical View

The software components are divided in six packages (folders), based on their logical function, illustrated in figure 3.2.



**Figure (3.2)** Logical View

#### 3.2.1 Package "algorithm"

The algorithm package contains the main segmentation algorithm which initially converts the given image into color channel arrays and calls its sub algorithms.

#### 3.2.2 Sub-Package "seed selection"

Members of this package are responsible for the first part of the algorithm where the regions are set, based on a comparison of the color value of each pixel and the following connected component analysis.

#### 3.2.3 Sub-Package "seed region growing"

The second part of the algorithm is implemented with functions that can be found in the seed region growing package. Regions that were found in the prior algorithms will be extended by pixels which are not associated to a region yet.

#### 3.2.4 Package "datatype"

To avoid long parameter lists, we implemented structures that encapsulate the needed data. These structures are located in the datatype package.

#### 3.2.5 Package "helper"

This package contains commonly used helper functions for memory allocation and time logging.



### 3.2.6 Package "test"

The test package includes the unit tests used to verify the correctness of our algorithms.

## 3.3 Algorithm Implementation

In this section we focus on implementation details of the algorithm, its individual components and elaborate on the core concepts in the source code. Initially, figure 3.3 will give an overview of the implemented algorithm. It demonstrates the work flow of the algorithm, shows the borders between the host and device layer and explains, where data is copied from host memory (illustrated in green) to device memory (blue colorized).

### 3.3.1 Interface to the OpenCV library

Above our implementation we used the OpenCV library [2] which acts as the image I/O layer. With the help of OpenCV we read, write and display the input and output images. The function `segment_OpenCV()` provides the OpenCV specific interface and calls the generic `segment()` function. Due to this two phase function call we can easily change to another image I/O library with no impact on the underlying algorithm implementation.

As with most image libraries the image data coming from the OpenCV library is given in an RGB format. This means the image data is given as one array of RGB color triples. Because this data layout is not very useful for a fast CUDA implementation we reorganize the data within the `segment_OpenCV()` function in a GPU suitable form. This means we reorganize the three color channels in three separate sequences of all red, all green and all blue values. Generally the concept of this data alignment is called "Structure of Arrays" whereas the RGB alignment of OpenCV is known as "Array of Structures". In section 5.2.5 we describe why the structure of arrays data layout is faster and better for GPU implementations and why one should aim for such a data organization.

### 3.3.2 Host Image Data Allocation

Note that when we reorganize the image data we allocate the new data with a special type of memory allocation. Namely we use page-locked memory to store the image data on the host side. First of all page-locked memory is a fixed memory buffer (also called pinned memory) where the operating system guarantees that this memory will never be moved to disk, which means the data remains always in the physical memory space. Consequently the GPU hardware can access this host memory via direct memory access (DMA) to move data between host and device memory space and therefore memory transfers are faster than the transfer of pageable memory allocations. However, it is not possible to simply allocate all the host data as page-locked memory because this basically erases the possibility of virtual memory. As a result you should only selectively allocate page-locked memory where it is needed and reasonable.

Now, to the reason why we allocate the image data in page-locked memory. When the host data is in page-locked memory we can access another class of parallelism

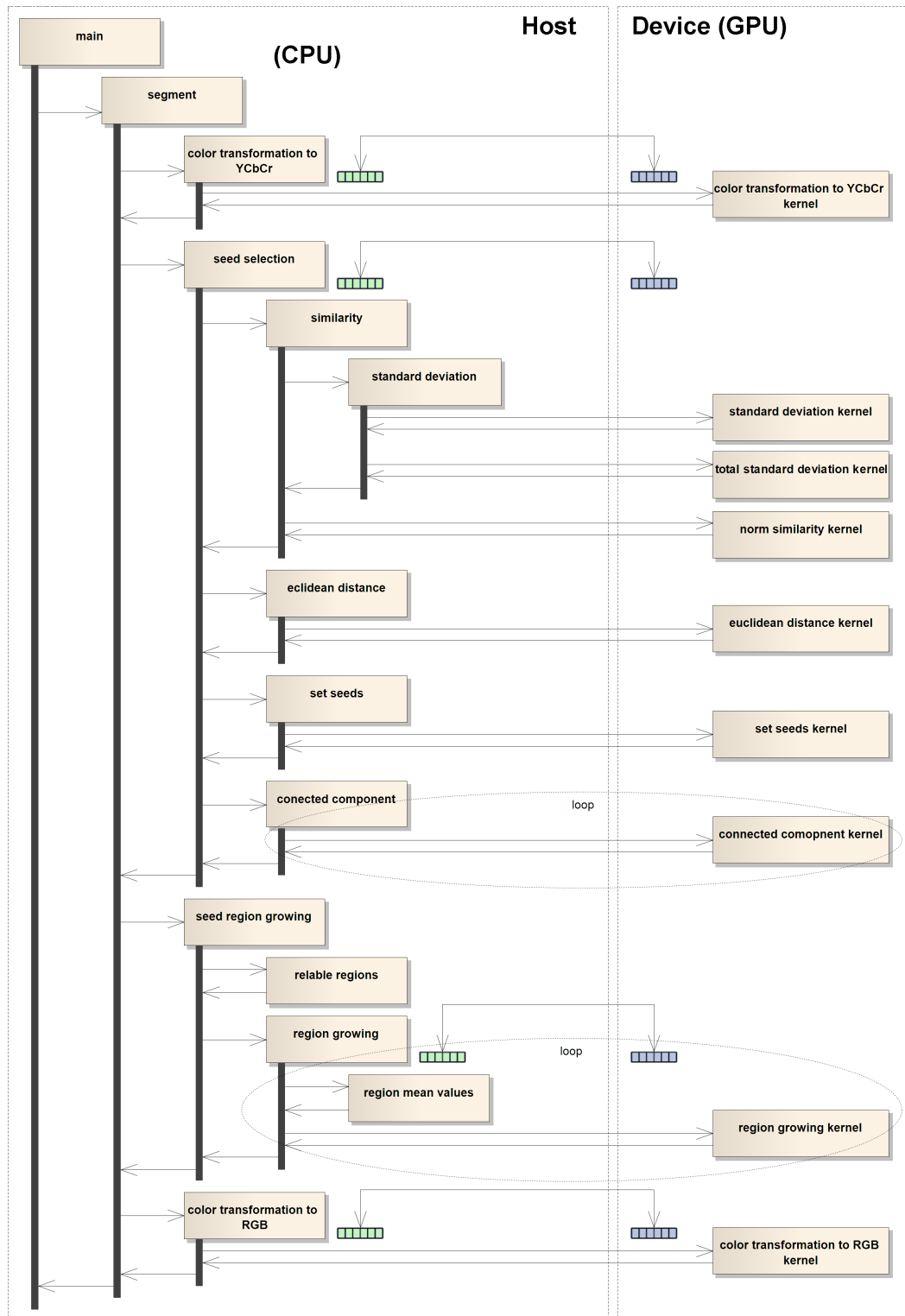


Figure (3.3) Algorithm Workflow

on the GPU. Rather than computing a function kernel on a large set of input data simultaneously we can reach a new type of parallelism known as task parallelism. The way to do task parallelism in CUDA is called "streams". With streams we can do steps (tasks) of the algorithm in parallel however to use streams the host memory buffer must be allocated as page-locked memory. In our case we can exploit this conception in situations when we want to do the same calculation on each color channel. In other words whenever we would like to do the same operation on each channel we can create three streams and compute the three results in parallel.

### 3.3.3 Standard Control Flow in CUDA Functions

In this section we show the way how CUDA functions are most often implemented with the example of the color transformation algorithm. From the host side memory is allocated on the device followed by the memory transfer of the image data. Subsequently the kernel will be called with the predefined launch configuration and the pointers to the data to be modified. Inside the kernel each thread is responsible for one specific pixel which matches the calculated global unique index of the thread. The thread replaces the color values of all channels at his pixel index with the newly calculated values.

### 3.3.4 Automatic Seed Selection Implementation

The implementation of the seed selection process follows the explanations in section 2.3.3 and is basically divided into the following four pieces.

1. compute the similarity condition
2. compute the Euclidean distance condition
3. set the seed pixels based on the two conditions
4. identify the regions with the connected component analysis

The output of this step is a set of seed pixels where each seed pixel belongs to a certain region.

#### Similarity Calculations

To determine the similarity of a certain pixel to its eight neighbors the algorithm first has to calculate the standard deviation to these neighbors. This value is computed on each channel separately which opens up the possibility to use streams. As shown in listing 3.1 three streams are created; one for each channel. In this case no memory copies are needed prior to the kernel execution, so we immediately start the three CUDA kernels with specifying the stream in the launch configuration. Note, because a kernel launch is always asynchronous the control flow returns immediately to the host after the call.

```
1  cudaStream_t stream_c1, stream_c2, stream_c3;
```

```

5   cutilSafeCall(cudaStreamCreate(&stream_c1));
   cutilSafeCall(cudaStreamCreate(&stream_c2));
   cutilSafeCall(cudaStreamCreate(&stream_c3));

   std_dev_kernel<<<gridDim, blockDim, 0, stream_c1>>>(d_std_dev_c1.d_ptr
   (), d_img.width, d_img.height, channel1);
   std_dev_kernel<<<gridDim, blockDim, 0, stream_c2>>>(d_std_dev_c2.d_ptr
   (), d_img.width, d_img.height, channel2);
   std_dev_kernel<<<gridDim, blockDim, 0, stream_c3>>>(d_std_dev_c3.d_ptr
   (), d_img.width, d_img.height, channel3);
10  cutilSafeCall(cudaStreamSynchronize(stream_c1));
   cutilSafeCall(cudaStreamSynchronize(stream_c2));
   cutilSafeCall(cudaStreamSynchronize(stream_c3));

15  cutilSafeCall(cudaStreamDestroy(stream_c1));
   cutilSafeCall(cudaStreamDestroy(stream_c2));
   cutilSafeCall(cudaStreamDestroy(stream_c3));

```

Listing (3.1) CUDA Streams

When we do not work with streams and launch the kernels synchronously the compiler will implicitly add synchronization calls between the individual kernel launches. In the case of streams we can decide when the host should be synchronized with the asynchronously launched streams. As a result we have to write the synchronization statements as shown in listing 3.1 after the three kernel launches. Finally, it is important that we do not forget to destroy the streams with a call to `cudaStreamDestroy()`.

### Normalize Standard Deviation

The similarity value is defined in the range of  $[0,1]$  and has to be normalized to this interval as the values of the previous calculations are not yet in this interval. To get values in this range we have to divide all standard deviation values by the maximum standard deviation calculated in the image (see equation 2.10). One possibility to find the maximum value could be to have a synchronized variable where each thread writes his value to this variable in case it is greater than the actual stored value. Specifically we would have to check the value with an atomic test-and-set function. From our experience such an approach does not perform well because of the characteristics of the SIMT architecture of CUDA. In this case nearly all launched threads would like to enter the atomic operation at once and are therefore forced to access the variable almost sequentially. Accordingly we have chosen another approach and search for the maximum value after the values have been calculated. To do so, we rely on the CUDA Data Parallel Primitives Library (CUDPP, see section 3.7.1 and [1]) to scan the data. The CUDPP library selects the best scan strategy based on the configuration plan given by the programmer at compile time.

```

1  CUDPPConfiguration config;
   config.op = CUDPP_MAX;
   config.datatype = CUDPP_FLOAT;

```

```

5  config.algorithm = CUDPP_SCAN;
   config.options = CUDPP_OPTION_BACKWARD | CUDPP_OPTION_INCLUSIVE;

   CUDPPHandle plan;
   cudppPlan(&plan, config, n_elements, 1, 0);

10  cudppScan(plan, d_max.d_ptr(), d_similarity, n_elements);

   cudppDestroyPlan(plan);

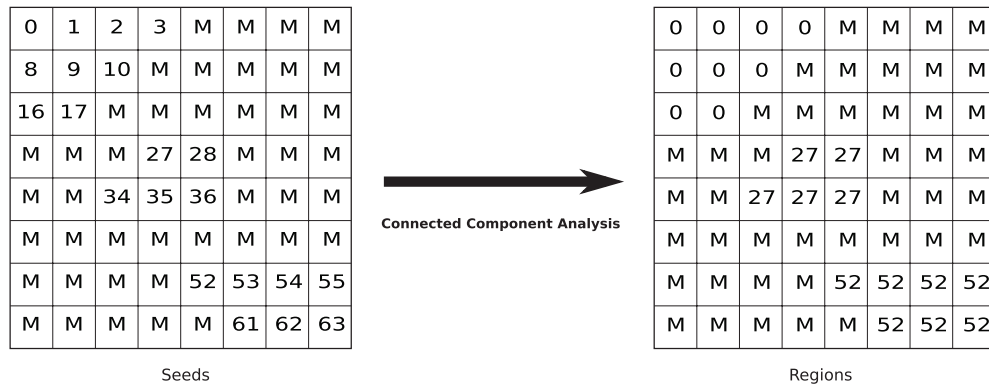
```

**Listing (3.2)** CUDPP Plan Configuration and Scan

Listing 3.2 shows the configuration of the execution plan and the call of the scan algorithm. With this configuration we define the operation to be executed, specify the datatype and select the algorithm to be used. Together with the configuration we create the plan where the last three arguments tell the algorithm how many elements have to be processed and how the data is organized in memory. The "1" means, we would like the function to process one row of data with a pitch of 0, meaning we have no pitched memory allocation.

### Connected Components Analysis and Seed Selection

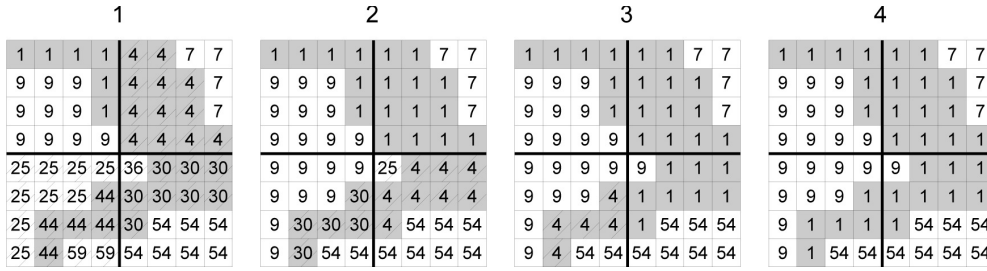
Based on the results of the similarity calculations and the computation of the euclidean distance we set the seed pixels. Only the pixels that fulfill both condition tests become seed pixels. That is, the pixel has to have a similarity value higher then the corresponding threshold given as the first command line argument; on the other side the euclidean distance must be lower than its corresponding threshold value which was given as second command line argument. In case the two conditions are met, the associated thread will write its thread index at the pixel position as illustrated on the left side in figure 3.4. All remaining pixels have to be marked with a special value. We identify these pixels by setting the value at their pixel position to the maximum unsigned integer value (UINT\_MAX) indicated with the M in figure 3.4.

**Figure (3.4)** Connected Component Analysis

When the seed pixels are identified we want to know which of the seed pixels are

connected and therefore form a region. This is done because in further steps we would like to treat connected seed pixels as one seed. To begin one should know that the connected component analysis is a computationally intensive task and due to this challenging to implement it with good performance. Our implementation is based on [7], an iterative multi-pass algorithm with local neighbor propagation. What does this mean?

The basic idea of the algorithm is to do the analysis in iterations where in each loop the launched threads check their neighbors label and compute whether they should update their own region label. The iterations continue until the system reaches a stable state where no thread updates his label and hence the break condition is reached. To go further into the details of the implementation we reconsider the CUDA memory architecture. To minimize global memory reads we do not just iterate the kernel calls, we also loop within the kernel function to make best usage of shared memory. As shared memory is per block memory and therefore accessible by all threads within a block we do the same thing within a kernel as we do outside the kernel, meaning we iterate until the labeling actions of the threads within a block reach a stable state. The squares in figure 3.5 visualize four iterations of the outer iteration loop.

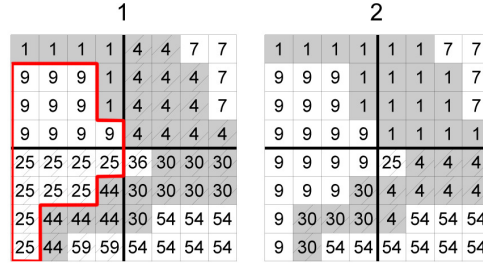


**Figure (3.5)** Four Steps in Component Analysis (Source [7])

The thick lines in the figure indicate the block boundaries meaning we look at four blocks of 16 threads each. The situation shown in the for squares shows the state of the labels after the internal loops. In each of the shown steps the regions within a block have the smallest possible label. Although to get an even smaller label the inner loop has to terminate and the kernel has to be launched again.

To illustrate these steps consider the white region outlined in red in figure 3.6: The pixels in the lower block have the label 25 and although this is not the lowest label in the outlined region, the kernel of step 1 terminates here. In the next kernel launch (figure 3.6, step 2) the threads at the upper boundary of the lower block will read the label 9 as their neighbor label. From now on no labels spill over the block boundaries meaning the label 9 propagates with the inner iterations through the block, until no thread within the block updates his label. The same thing happens with the label 30 that spills over the block boundary at the beginning of the kernel execution. However until label 1 has been propagated to this block it takes three more kernel launches.

As already mentioned we use shared memory to store the image data in a certain block. This way we can do the inner loops on the device by accessing shared memory only. Since all threads within a thread block execute in parallel and all threads within



**Figure (3.6)** Block-wise Steps in Component Analysis

this block access labels that are likely to be updated by other threads, the execution must be synchronized. This is done by simply adding a call to `__syncthreads()` after each inner iteration.

### 3.3.5 Seed Region Growing Implementation

In contrast to the other components of the algorithm, the seed region growing part was the most challenging portion to implement. What makes it difficult to do a good performing implementation on the GPU is that the input data to the kernel call and the output data from the kernel are not the same size. In other words during the kernel computation the system should be able to dynamically allocate memory for the output data.

The region growing algorithm needs to know the mean color value of every region and at this point the dynamic behavior comes into play: Since the growing process repeatedly assigns new pixels the regions the mean values of the regions constantly change. Each thread in the region growing kernel decides based on the different region mean values to which region a certain pixel has to be assigned. As not all the pixels are assigned to a region at once. The kernel gets called multiple times which requires the update of these mean values after each kernel call otherwise the next kernel call operates on obsolete mean values. Besides that all threads assign pixels in parallel which means that we can not calculate the mean values on the fly because a certain thread does not know which pixels have been assigned by other threads.

But why does a thread not know if a pixel has been added to a certain region by another thread? When all threads operate on the same data this data should be up to date, no? While this is true, from an implementation's point of view it is not guaranteed that a thread reads the most current pixel label as the labels are accessed through texture memory fetches. Texture memory is cached memory which means the read operation at a certain pixel location may return a cached obsolete region label.

To sum up: We would like to have a list of region labels which indicate which pixel belongs to which region, furthermore unclassified pixels are marked with `UINT_MAX` (see right square in figure 3.4). The number of regions is a constant that does not change up to this step of the algorithm. A kernel call assigns a defined set of pixels to the existing regions. To update the mean value we need to know how many pixels are in a certain region. As this number constantly changes within a kernel call we can not do this calculation on the fly unless we implement the mean value update as an atomic

function.

In the following two sections we describe the solution we came up with but we know there is still great potential to implement these steps faster.

#### Region Mean Values Implementation

This step is fully realized on the host and this has two reasons: First, the implementation is not as trivial to realize on the device as it may seem. For instance, considering the region labels in figure 3.4. How do we distribute the computation over the individual threads; What should they do? In case of a thread per pixel each thread can add his value to a region sum however this leads to considerable synchronization overhead. Secondly, the serial approach on the host mostly compensates the overhead of the memory copies and synchronization in parallel implementation. For these reasons we have chosen the most trivial approach to simply iterate through the whole array and collect the region statistics within the `region_stats` structure. From our timing experiments this is a reasonable solution as the time consuming task is the following seed region growing step.

#### Region Growing Implementation

The difficulties with this task were already mentioned in section 3.3.5. The idea behind this implementation is similar to the connected component implementation described in section 3.3.4 as it also iterates and calls the kernel multiple times until each pixel is assigned to a region. On the contrary to the connected component implementation this procedure has no inner loop within the kernel function. The basic execution path of the loop consists of the following steps:

1. Compute the mean color values of the regions.
2. Call the kernel and do one iteration of pixel classification.
3. Check if any pixels had been labeled by the kernel. If so start over again with the update of the mean values.

The kernel implementation is very straight forward and follows the algorithm descriptions in section 2.3.4. Note, that we are forced to return to the host with the control flow as the memory locations, read through texture fetches, are also written by the kernel threads. When returning to the host and calling the kernel over again the texture cache gets invalidated and we can be sure that the received data from texture fetches is up to date. Unfortunately there is no possibility to invalidate the texture cache with an API call.

If we look at the elements of this algorithm the time-consuming part is what happens besides the computations - the memory transfers. In the description of the mean value computation we listed the reasons to implement the mean value computation on the host. As a consequence we now have to copy the results of the mean value computation and the relabeling actions of the kernel every loop to device memory and back to the host memory.



### Set Pixel Color

Because we want to display the different regions found in the resulting image (which is only for demonstration purposes), this function sets the color of each pixel to the color mean value of its hosting region.

## 3.4 Testing

To make sure our algorithm returns the correct data, we implemented unit tests for nearly all sub-functions, using the CUTE unit testing framework. The tests are divided up into multiple test suites which match the given packages. The test procedure is practically always the same for all implemented tests:

- *Create input and expected data* Usually we generate an image structure in host memory. Additionally, we create the expected output values.
- *Prepare data for usage in kernel* The next step is the preparation of the input data in device memory which includes the memory allocation on the graphics card and memory copy of the input data from host memory to device memory. In addition we have to allocate host or device memory for the results, depending on whether the function under test copies the result back to host memory or not.
- *Call function to be tested and assert output* Finally we call the function under test and compare the resulting data with our expected result.

## 3.5 Time Logging

To get an idea about the runtime of the individual stages of the algorithm we introduced a time logger class. The time measurement basically happens by static function calls from source code. With the implemented time logger it is possible to log the duration of the whole algorithm, sections and subsections. As shown in listing 3.3 we can simply call the static functions of the `time_logger` class to print the lap times to console.

```

1  void e() {
    // do something...
  }

5  void g() {
    // do something...
  }

10 void f() {
    time_logger::start_section("section f");

    g();
    time_logger::section_lap("section task g");

15  e();
    time_logger::section_lap("section task e");
  }

```

```
    time_logger::stop_section();  
}  
20 void main() {  
    time_logger::start();  
25    f();  
    // and more sections...  
30    time_logger::stop();  
}
```

**Listing (3.3)** Time Logging Sample

To simplify the time logging mechanism even more one might realize this functionality as a Decorator [6]. This way we could have had simply decorated the function calls with the time logging mechanism where time logging would have been desired.

## 3.6 Memory Management

As CUDA is just an extension to the standard C language common CUDA C code suffers from the painful memory allocation and deallocation calls. To get around the numerous calls to `cudaMalloc()`, `cudaFree()` or for the host side `malloc()` and `free()` we implemented memory allocators which guarantee that the allocated host or device memory is freed again. As it is no problem to use C++ code to call a CUDA function on the host side we implemented this logic as template classes to be independent from the data type of the allocation.

### 3.6.1 CUDA Memory Allocator

We provide two CUDA allocator types, one for 1D linear memory allocations and another one for 2D pitched memory allocation. In listing 3.4 we show an extract of the allocator implementation. Note, for simplicity we omitted some implementation details such as the initializer list of the constructors or field declarations.

```
1  template <typename T>  
   class cuda_allocator1D {  
   public:  
       cuda_allocator1D(size_t n_elements) {  
5       cutilSafeCall(cudaMalloc((void **) &_device_pointer, _pitch));  
       }  
  
       cuda_allocator1D(size_t n_elements, T init_value) {  
           cutilSafeCall(cudaMalloc((void **) &_device_pointer, _pitch));  
10          cutilSafeCall(cudaMemset(_device_pointer, init_value, _pitch))  
           ;  
       }
```

```

    }

    virtual ~cuda_allocator1D() {
        cutilSafeCall(cudaFree(_device_pointer));
15    }

    // some more functions and fields are omitted here...
};

```

**Listing (3.4)** CUDA Memory Allocator 1D

The type of the allocation is defined by the template parameter whereas the size of the allocation is given as constructor argument. We provide a second constructor with an optional initialization value. If the second argument is given, the allocated memory will be initialized with a call to `cudaMemset()` to this value. To pass the device pointers to the kernel launches we provide the function `d_ptr()` to get the device pointer of the allocation. Finally we placed the call to `cudaFree()` into the destructor body, to guarantee that whenever the allocator object is destroyed the reserved device resources are released again. The 2D allocator version is basically the same as the 1D version except that we have to tell the constructor the with and the height of the desired allocation instead of the number of elements. Furthermore it exist an additional function to get the pitch of the memory allocation since 2D memory allocations are usually pitched to get better performance in memory access. The implemented host allocators are conceptually exactly the same as their CUDA analogues.

## 3.7 Supporting Libraries

### 3.7.1 CUDA Data Parallel Primitives Library

In this section we would like to look at the internals of CUDPP and introduce the concept of this library. Section 2.4.2 already described what CUDPP is and what functionality it implements. Our goal is not to explain how the library works down to the last item but to illustrate an nice approach how to implement reusable CUDA code.

What makes this CUDA library so interesting? When you write C code it is quite difficult to write nice reusable generic code because the C language has nothing like polymorphism nor supports C++ like templates. At first sight it seems like we have the same problems when trying to write a CUDA library. But CUDA C is a bit different as the NVCC compiler fortunately supports templates which opens a wide range of new possibilities to implement a CUDA library. Now, CUDPP exploits exactly this feature to implement the core library functionality.

With the help of templates CUDPP implements the library core with template meta programming (TMP). For an algorithm there exist usually more then one implementation each of which performs best under certain circumstances. As shown in listing 3.2 we set up an algorithm configuration and create a execution plan. After calling the CUDPP algorithm the library code extracts the configured parameters from the input and selects the most suitable and best performing algorithm via template parameters. This way the library code can be implemented datatype and configuration independent. The vision

of CUDPP is to have a self-tuning library [1] that adapts automatically to the best performing configuration. Note that all these steps happen at compile time meaning that the most suitable algorithm is configured at compile time.

Unfortunately we noticed the concept of the CUDPP library only towards the end of the project. For this reason we had not enough time to make use of these insights in our own implementation.

#### 3.7.2 CUDA Utility Library

The CUDA Utility Library (cutil) is part of the CUDA SDK and checks for errors in CUDA API calls. With this library we can simply decorate the CUDA API function calls with the wrapper `cutilSafeCall()` to catch error states and abort the execution. In case the function call returns an error code `cutilSafeCall()` will print a human readable message which error occurred and where it might had happen. Although this is a sufficient solution for our implementation, you might do a real error handling in productive code.

## 4 Results

In this chapter we would like to analyze the algorithm output and elaborate on performance results. Besides that we do not want to hide the things that could be done better or could be introduced in a further release of this algorithm.

To begin with, let us take an overview of the results of this implementation and then go further into the details about the happy execution paths and failing algorithm configurations. It is important to know that this project does not realize the algorithm to the very last step as described in section 2.3, meaning the region merging step is not part of the implementation. Even though we take this step into account for the interpretation of the results. We ask ourselves what might happen in the last step and how the shown output would get processed by this last step.

The performance section will give an overview about the elapsed time per algorithm part and highlight the insights on this data.

Finally we will elaborate on the known issues in the current implementation and try to show ways how they probably can be eliminated.

### 4.1 Segmentation Results

The following examples are thought to illustrate successful and failing segmentation results. We try to show possible difficulties with our implementation of the algorithm. In each example we will firstly list the configured launch parameters followed by the listing of the input and output image and the corresponding interpretation.

Note that the interpretation of the results hardly depends on the use case and the expectations on the desired precision of the result. Most of the time we have a certain knowledge on how the input image might look and maybe it is reasonable for the intended use that some regions do not get recognized as segments. When trying to get to know what the segments might are we usually rely on the human perception and try to imagine whether the algorithm performs as expected.

#### Good Case Segmentation

At first we show an example where we get a nice segmentation result. The input image 4.2a shows a plant with red blooms on the left and a dark bird on the right on a blurred background. If we look at image 4.2a we can see that it has a lot of different green tones. Whenever an image has low color distribution it could be a sign for a difficult segmentation because it is likely that the objects disappear in the background. Although this is true, in this image the background and the plant are clearly separated by the sharp lines of the leaf even if both are in green tones.

Launch Configuration	Example 1
Image	bird.jpg, 1600 × 1066
Similarity Threshold	0.970
Euclidean Distance Threshold	0.090

**Table (4.1)** Launch Configuration, Bird

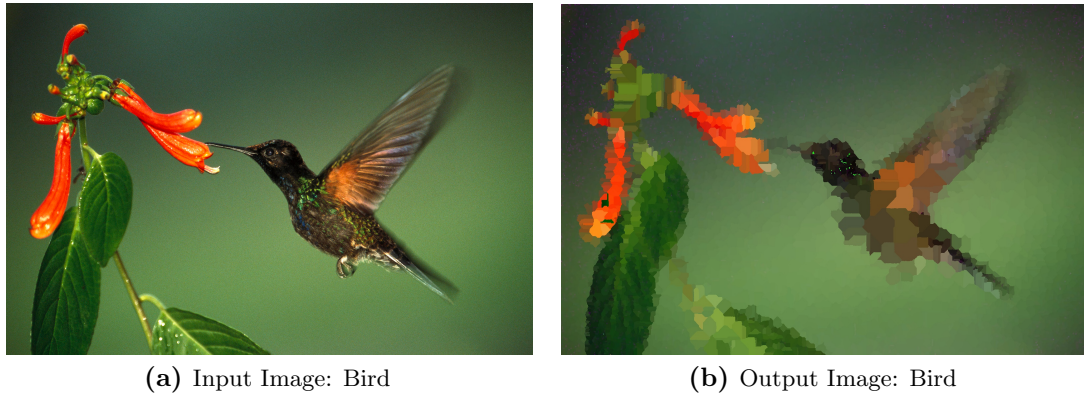
Table 4.1 lists the launch configuration used to produce the output image 4.2b. In this case we noticed that Euclidean distance threshold had a good response to little changes. This is reasonable as the Euclidean distance values say something about the distances between color values in the color space. As we have a lot of green color values that are close together this parameter has good response on the output.



**Figure (4.1)** Seed Regions: Bird

The colorful scattered pixels in figure 4.1 show the seed pixels selected by the seed selection algorithm. The different colors represent the individual regions detected by the connected components analysis. The most important thing is that every desired output segment includes at least one seed pixel which is the case as we have seeds on the bird, the plant and the background. From the number of regions within the same expected segment in this image (e.g. the background) we can see that it is important to merge the numerous regions in the last algorithm step.

If we take a look at the original image in figure 4.2a we expect the algorithm to find two foreground objects: The bird and the plant on the left. From the output image



**Figure (4.2)** Good Case Segmentation

we can clearly see how nice the bird is extracted from the background. Even though everything still looks like a mosaic we are pretty sure that the region merging step of the algorithm would merge the dark and hopefully even the reddish parts of the wing. When considering the plant the situation gets a bit more complicated. The plant has strong orange-red blooms with juicy green leaves which we expect to be merged by the last step of the algorithm.

If we look at the stalk of the flower we can see that the regions towards the lower end become very similar in color to the background. As a result these regions are likely to be merged with the background in the summarizing step.

#### Correct Segmentation with Unwanted Output

In this example the algorithm basically produces the correct output in terms of its computations whereas we as humans would expect something different. As shown in table 4.2 we set both thresholds to a relative strict value as image 4.3a has a large distribution of the color values.

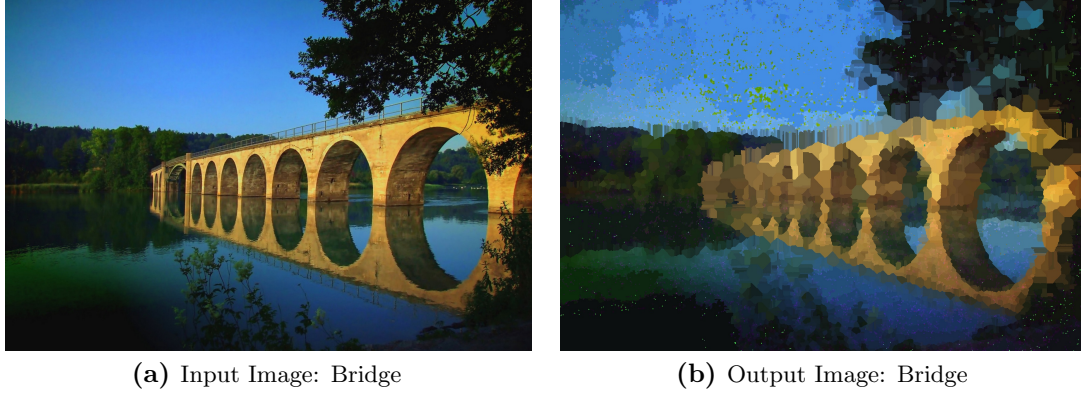
Launch Configuration	Example 2
Image	bridge.jpg, 1800 × 1200
Similarity Threshold	0.985
Euclidean Distance Threshold	0.010

**Table (4.2)** Launch Configuration, Bridge

If we simply look at the regions produced in the output image 4.3b from a computation's viewpoint we get the expected result. The region merging step most likely will produce a segment for the sky, a segment for the tree on the top right and a segment for the bridge. The remaining segments are difficult to assign to a certain region because humans would classify most of them as water. Therefore the algorithm most likely will fail with the mirror effect of the water. Considering the bridge in the output image we can see that the regions of the real bridge and the mirrored one are likely to get merged



to a single segment which is obviously wrong from our logical point of view.



**Figure (4.3)** Correct Segmentation with unwanted Output

Interestingly, if we look at the performed seed selection in image 4.4 we can see that the selected seeds are meagerly distributed over the region of the bridge. At this point we could expect that the mirrored part of the bridge gets separated from the real bridge in the region growing step. However as the real and the mirrored parts have very similar color values they are very likely to be merged.



**Figure (4.4)** Seed Regions: Bridge

Considering the number of identified regions in image 4.4 we can see that many



different regions are identified. Comparing this number to the performed seed selection in the bird image 4.1, it is striking that many pixels more are already classified as members of a region. Even if we have chosen stricter thresholds in the bridge image we get more regions and more classified pixels after the seed region step than in the bird image. This fact can be explained with a pixel-level-size look at the images. As the bridge image has many objects and almost no clearly identifiable background, the bird image has a smooth gradient in the background. It seems that this gradient is relatively homogeneous however when we analyze it on pixel level we can see that the pixel color values in a local neighborhood are scattered and grainy. This is the reason why only small regions are identified in the bird image by the first step of the algorithm even if the threshold would allow much more seed pixels.

### Failing Segmentation

We selected image 4.5a with the three gulls as a failing scenario because we expect it to fail with the segmentation. The interesting part of the input image is the lower left corner where we can see a white cloud in the background with a white gull on top of it in the foreground. Even if the wing of the gull is separated from the white cloud by a bright stroke the final step of the algorithm probably will not produce the desired output. We expect it to fail as it probably merges the two objects that we do not want to be merged.

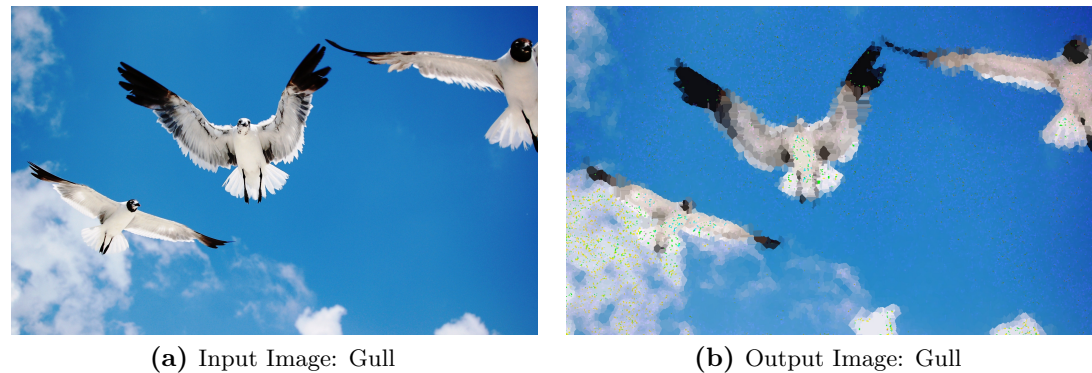
Launch Configuration	Example 3
Image	gull.jpg, 1920 × 1200
Similarity Threshold	0.985
Euclidean Distance Threshold	0.020

**Table (4.3)** Launch Configuration, Gull

The image properties of figure 4.5a correspond to the properties of the bridge image. Therefore the launch configuration listed in table 4.3 is approximately the one of the bridge example.

The two gulls in image 4.5b on the right are nicely isolated from the background and form two separate regions. If we go to the left side the sky has white clouds which are very similar to the white feathers of the gull. As a consequence it is likely that the region merging step will merge the gull with the regions of the cloud. Even if we try to get better results using different threshold values the output is not significantly better. Section 4.1.1 describes why we are limited in output optimization by changing the threshold values.

Unfortunately the algorithm fails in this case whereas the human perception would separate these objects based on meta information about the content of the image. In our case we would construe that the image contains three gulls with some clouds on the blue sky and would extract only the three gulls.



**Figure (4.5)** Failing Segmentation

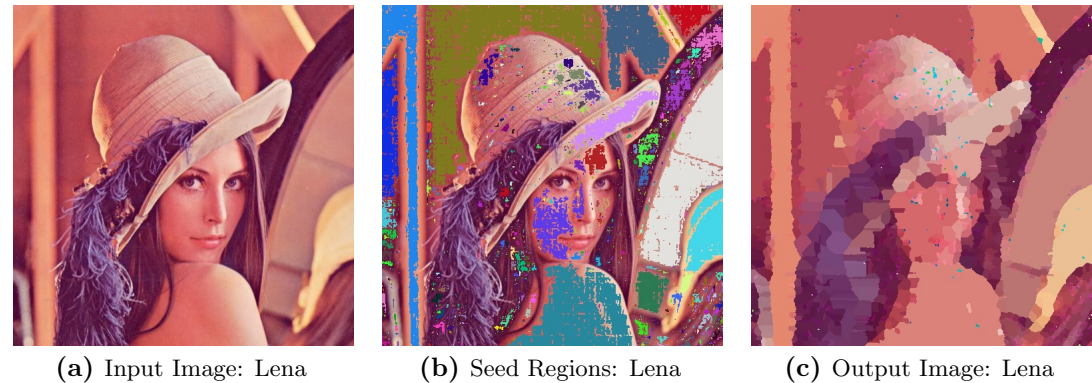
Lena Reference Segmentation

We included this image as a reference to other segmentation algorithms as this is a standard test image across many image processing algorithms. This picture is suitable for algorithm testing as it contains a good mix of flat regions as well as parts with small details.

Launch Configuration	Example 4
Image	lena.jpg, 512 x 512
Similarity Threshold	0.960
Euclidean Distance Threshold	0.060

**Table (4.4)** Launch Configuration, Lena

Table 4.4 shows the command line arguments for the lena picture. Because the image 4.6a has a lot of clearly enclosed segments we set the threshold values relatively moderate.



**Figure (4.6)** Lena Reference Segmentation

The input image 4.6a has many small segments which we expect to be segmented as seen. Compared to the output image we see that all these regions are more or less detected as autonomous regions discernible by identical colorized connected components in image 4.6b. Surprisingly the used threshold values provided already a nice intermediate result as we have quite large connected components for the expected regions.

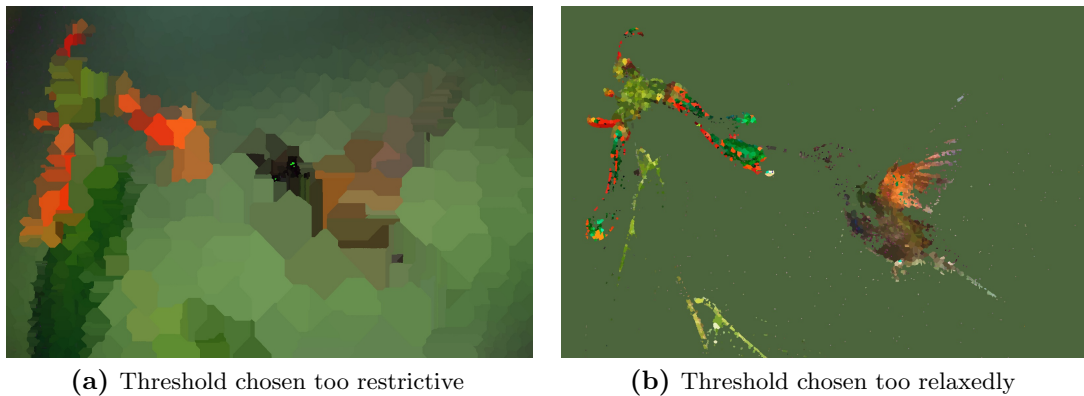
A critical part in the segmentation of the Lena image is the hat which we expect to be separated from the background. If we look at the output image 4.6c we can see that the background region already splashed over the hat's outline and probably is going to be merged with it in the next step. However the region merging step for this image could be quite challenging because of the numerous little regions.

#### 4.1.1 Threshold Selection

The key parameters to get a good set of seed pixels is the choice of a suitable pair of threshold values. Several methods to choose optimal threshold values exist; they may be chosen manually, or can be computed automatically using one of multiple thresholding algorithms.

We decided to not implement an automatic threshold algorithm but let the user choose the threshold values. So the application can be used to test different configurations with various thresholds. This decision has two aspects: On the one hand the user can play with the input values and therefore easily modify the resulting output. On the contrary it can be very hard to find the optimal threshold values.

As illustrated in picture 4.7a and 4.7b the output could be lousy and far from the expected result when the thresholds do not fit the input image.



**Figure (4.7)** Threshold Selection

In image 4.7a the threshold was chosen too restrictive with the result that not enough seed pixels are in the intersection of the two conditions. Therefore we have too few seeds which results in too large regions.

If the thresholds were chosen too weak, too many pixels are marked as seed pixels. As a result the connected component analysis finds too many connected components and classifies most of them to the same region as illustrate in 4.7b. In this case the

misconfiguration leads to an image where the background is one single region. The situation gets even worse because many of the object pixels are also assigned to the large background region.

## 4.2 Performance

The main goal when using the CUDA technology is performance so when we talk about better performance, we usually mean faster executing implementations. Unfortunately we have no references to other segmentation algorithms or CPU-only implementations to compare our results with.

To get an idea about the average runtime of the individual algorithm steps we list a typical runtime log for the bird image 4.2a in table 4.5.

Algorithm Section	Runtime	Share
transform $RGB$ to $YC_bC_r$	11.457 ms	0.07%
Seed Selection		
similarity	1309.669 ms	7.70%
euclidean distance	1.772 ms	0.01%
set seed pixels	1030.335 ms	6.06%
connected component	336.736 ms	1.98%
Seed Region Growing		
region growing	14293.837 ms	83.97%
set pixel color	24.201 ms	0.14%
transform $YC_bC_r$ to $RGB$	11.007 ms	0.07%
<b>Total algorithm</b>	<b>17021.770 ms</b>	<b>100.00%</b>

**Table (4.5)** Runtime Example bird.jpg

First of all, the overall runtime seems to be pretty bad so we have to further analyze each step of the algorithm to see where it spends so much time. It is striking that the time gets lost in the region growing step. Actually we are aware of this fact and know that our implementation of this part is not very supporting to good performance.

The problem with the region growing step is that the implementation includes a lot of memory copies between the host and the device as already stated in section 3.3.5. To overcome this problem we would have to find a better implementation with less memory copies.

On the contrary we would like to show another problem which almost disappears in the previous example: The connected components analysis seems to perform acceptable compared to the other pieces. However if we look at another example we can see that this is also a computationally intensive task that is likely to consume a lot of the overall runtime.

To demonstrate this fact we will run the segmentation on the bridge image 4.3a. When we look again at the runtime log in table 4.6 we can see that the connected component

task consumes about one fifth of the overall runtime while the effort for the region growing step drops from 83% to 65%. These numbers obviously depend on the actual image however we can see from many tests that these numbers are correlated. The correlation is based on the number of selected seed pixels. We know that the number of selected seed pixels depends on the chosen threshold values. When the thresholds are harder to be met less pixels become seed pixels, and conversely more pixels become seed pixels when the conditions are easier to be met. Besides that we know that both the connected components algorithm and the region growing step are performing bad we can see that the seed selection algorithm performs better.

Now we can explain the correlation between the runtime of the two steps: When the thresholds are easy to be met, a lot of pixels become seed pixels and therefore the connected components analysis has a lot to do. On the other hand if the thresholds are harder to be met, less pixels get in the set of seed pixels which means more pixels remain unclassified and have to be processed by the region growing algorithm. As we know that the connected components algorithm is the faster of the two the overall segmentation takes less time when the connected components analysis has more pixels to process.

Algorithm Section	Runtime	Share
transform $RGB$ to $YC_bC_r$	14.845 ms	0.06%
Seed Selection		
similarity	1653.371 ms	7.04%
euclidean distance	2.025 ms	0.01%
set seed pixels	1307.028 ms	5.56%
connected component	5035.615 ms	21.42%
Seed Region Growing		
region growing	15449.903 ms	65.72%
set pixel color	30.320 ms	0.13%
transform $YC_bC_r$ to $RGB$	13.767 ms	0.06%
<b>Total algorithm</b>	<b>23510.029 ms</b>	<b>100.00%</b>

Table (4.6) Runtime Example bridge.jpg

The stated correlation between the runtime of the connected components analysis and the region growing step is supported by the two images 4.1 and 4.4. We can see that in case of the bridge much more pixels are already classified as in the bird image and therefore the region growing step has much more to do with the bird image.

## 4.3 Known Issues

### 4.3.1 Image Dimensions

Because some parts of the algorithm store multiple arrays with the size of the image dimensions in device memory this is likely to become a problem when the input image

is very large. Therefore we are interested in the maximum image dimensions to just fit into memory. Theoretically the maximum possible image dimensions for our algorithm can be calculated as follows

$$\left[ n \cdot \left( \left\lceil \frac{w \cdot s}{t} \right\rceil \cdot t \cdot \frac{p_{\max}}{w} \right) + m \cdot p_{\max} \right] \cdot s \approx G_{total} \quad (4.1)$$

where

- $p_{\max}$  is the maximum image size in pixel,
- $w$  is the image width in pixel,
- $n$  is the number of pitched arrays,
- $m$  is the number of non-pitched arrays,
- $t$  is the texture alignment in bytes,
- $s$  is the size of the allocated datatype in byte (assumed that all used datatypes have the same size),
- $G_{total}$  is the total amount of global device memory in byte.

By applying equation 4.1 to our hardware configuration and algorithm implementation we try to estimate the maximum possible image dimensions. Our test setup is a NVIDIA Quadro FX 3800 card as listed in table A.1. From these device specifications we get  $G_{total} = 1073020928$  and  $t = 256$ .

From our implementation we know that  $m = 6$ ,  $n = 1$  and  $s = 4$ . Applying these variables to equation 4.1 will result in

$$\left[ 1 \cdot \left( \left\lceil \frac{w \cdot 4}{256} \right\rceil \cdot 256 \cdot \frac{p_{\max}}{w} \right) + 6 \cdot p_{\max} \right] \cdot 4 \approx 1073020928 \quad (4.2)$$

When defining  $w = 5000$  and solving equation 4.2 by  $p_{\max}$  we get a maximum pixel amount of 26705880. Dividing this number by the image width results in an image height of 5341 pixel.

From our own experiments we could handle an image size of approximately  $5000 \times 5000$  pixel. Above this, the application crashed with kernel timeouts or CUDA out of memory exceptions. In fact the calculated number is not an exact result as other memory allocations may limit the maximum possible image size to a lower amount.

## 5 Conclusions

In this chapter we describe the problems that we had to overcome when implementing the described algorithm in CUDA. We would like to focus on the difficulties we had when trying to do something in parallel with the respect to the CUDA architecture and the CUDA library. For this reason we leave the discussion on the segmentation algorithm itself.

### 5.1 Conceptual Issues

#### 5.1.1 Memory Management

The CUDA technology allows you to easily do high-performance parallel implementations nonetheless there are also limitations on the conceptual side of a CUDA program. The limitations come from the fact that we basically work on two separate devices, the CPU and the GPU. These two devices have separate memory spaces at physically different locations which means the data has to be copied between the two devices and the programmer has to tell the system where he would like to allocate memory and how much this should be. In the simplest case `malloc()` allocates memory on the host whereas `cudaMalloc()` reserves memory on the device. These two functions have in common that both are only callable from the host. By the time of this project it was not possible to dynamically allocate device memory from device code. In other words one had to allocate all the required device memory from the host side before the kernel was launched. The conceptual downside of this lack of functionality on the device appears when we would like to do a computation on the device and have no a priori knowledge about the result size. From a conceptual point of view this prevents us from implementing dynamic device code. You may know some fixed boundaries or memory usage limits that you know will not be exceeded.

To give an example from this project: The input data to the seed region growing step described in section 2.3.4 and 3.3.5 are the seed pixels and their region affiliation identified in a previous step. The goal of this step is to assign all non-seed pixels to one of the regions. We know how many pixels we have in the image and therefore how much memory we need for all the pixels in the image. However we do not know how many pixels are added to which regions in one iteration.

Our first attempt was to have an array per region each of which containing its assigned pixels. In the kernel one thread per region should add the newly identified pixels to his region array. In this case the input arrays to the kernel had not the same size as the output arrays from the kernel. This attempt had two major drawbacks: First of all we could not allocate additional device memory within the kernel. Even if the first problem could have been solved, we would not have been able to return the new region

sizes back to the host in an appropriate way.

So one solution we came up with was to rely on the boundary condition given by the image size and allocate memory for the whole image. This way we were sure to have enough memory allocated but knew that we produced an overhead.

Since CUDA Toolkit Release 3.2 (November 2010) it is now possible to allocate device memory within a CUDA kernel or device function with the device function `malloc()`. Unfortunately this is only possible for CUDA devices with compute capability 2.x and even here only with certain restrictions on memory management functions.

### 5.1.2 Data Representation

Another issue coming with the previously described problem is the representation of data that should be processed by the GPU. As described in 5.2.5 the organization of data should preferably be in a linear sequence. There is often not enough device memory available to store data many times in different representations in device memory. Above all the copy overhead between host and device would not make sense anyhow.

In the course of our implementation we came to a point where we needed our image data in a different representation to calculate the mean values of the identified regions. To accommodate the problems explained earlier we tried to implement a special data structure which we named region view. Unfortunately this view could only be used to calculate the region mean values in an appropriate way. The further steps of the algorithm did not match these data structure which forced us to transform it again and this was not acceptable.

These insights brought us to the decision to redesign this data representation.

## 5.2 Implementation Issues

### 5.2.1 Code Organization

Basically neither the NVCC nor the GCC compiler cares about the file extensions. They simply try to compile the input files they get and therefore we could have had named all the files with the common `.c`, `.cpp` or `.h` extension. Although this would perfectly comply with the conventions in C or C++ projects we propose to organize CUDA code differently.

To keep the file structure organized and clearly arranged we propose to separate CUDA code and host code. For our project we came up with the following convention:

- every file that contains CUDA code has a `.cu` extension
- a `.cu` file can contain one or more CUDA kernels
- a `.cu` file may provide a host function to call the CUDA kernel
  - the host functions cares about the device memory allocation and deallocation
  - only this host function calls the CUDA kernel
- one `.cuh` header announces all the public host functions in a `.cu` file



The CPU code includes the `.cuh` header file and calls the host function defined in the CUDA source (`.cu`) file. The code of the host function in the CUDA source file then allocates the device memory and sets the launch configuration for the CUDA kernel. Next the CUDA kernel is executed and as soon as it has finished its work the host code will copy the results back to the host memory and free the device memory allocations. In an abstract notation the code scaffold would look something like illustrated in the following listings.

The header file in listing 5.1 shows the declaration of the public host function implemented in the `.cu` file.

```

1  #ifndef HOST_FUNCTION_CUH_
   #define HOST_FUNCTION_CUH_

   void host_function(float* h_in, float* h_out, size_t size_b);

5  #endif /* HOST_FUNCTION_CUH_ */

```

**Listing (5.1)** CUDA Header File `host_function.cuh`

On the CPU side the code does not contain anything that is concerned with GPU memory allocation, deallocation, or a kernel call. Here we could also use fully object oriented C++ code instead of plain C code. As we can see from listing 5.2 the `host_function.cuh` header file is included and whenever the CPU side wants to do work on the GPU the execution control flow changes to the GPU side, in this case by calling `host_function()`.

```

1  #include "host_function.cuh"

   // do whatever you have to be done first ...

5  size_t size = 10;
   float h_input[size];
   float h_output[size];

   host_function(h_input, h_output, size * sizeof(float));

10 // continue with result h_output ...

```

**Listing (5.2)** CPU Side Code

Note that the arguments taken by the host function in listing 5.3 reside completely in host memory (indicated by `h_` notation). This means no GPU memory is involved in the arguments of the host function. The host function acts as the GPU interface to the CPU side and is responsible for data copies between host and device memory along with the kernel calls. The listing shows how the host function allocates device memory of `size_b` bytes. After the data is being copied to the device memory and the CUDA kernel is launched. As soon as the kernel has finished its operation the result is copied back to the host memory location and equally important, the allocated device memory

is being freed.

```
1  #include "host_function.cuh"

   __global__
5  void cuda_kernel(float* d_in, float* d_out, size_t size_b) {

       // kernel code

   }

10 void host_function(float* h_in, float* h_out, size_t size_b) {

       float* d_in;
       cudaMalloc((void**) &d_in, size_b);
       cudaMemcpy(d_in, h_in, size_b, cudaMemcpyHostToDevice);

15       float* d_out;
       cudaMalloc((void**) &d_out, size_b);

       int nThreadsPerBock = 256;
20       int nBlocks = (size_b / sizeof(float) + nThreadsPerBock - 1) /
           nThreadsPerBock;

       cuda_kernel<<<nBlocks, nThreadsPerBock>>>(d_in, max_value, size_b
           / sizeof(float), d_out);

       cudaMemcpy(h_out, d_out, size_b, cudaMemcpyDeviceToHost);

25       cudaFree(d_in);
       cudaFree(d_out);

   }
```

**Listing (5.3)** GPU Side Host Function and CUDA Kernel

Note, that in this example and likewise in this project the GPU interface and the actual GPU code (CUDA kernels and all `__device__` annotated functions) is implemented within the same file. Of course the kernel could be defined in a separate file.

### 5.2.2 C/C++ Mixing/Combining

To start with, we would like to explain our understanding of the terms mix and combine as we used them in this section. When we talk about mixing the two languages we mean you have C and C++ code within the same layer (CPU and GPU layer, see figure 3.1). On the contrary when we use the term combine we mean you have a C++ layer and typically below it a C layer strictly separated from each other.

With this said, we consider it critical to mix C and C++ code within a CUDA project conceivable on the other hand are solutions where C and C++ is combined. Mixing the two languages is basically a bad idea because it is difficult to keep the data tidy and consistent. The scenario in terms of combination of the two languages is conceivable if you stick to clear interfaces and strictly defined layers.

The basic CUDA Runtime and Driver API provides a C interface. This means implementing something in CUDA with a C++ layer on top of it one usually has to convert all the C++ data in a C conform representation. In case of simple value types we do not need a transformation at all, however when dealing with C++ Standard Template Library (STL) containers or objects the problem gets more complicated. The relevant data has to be mapped to C, then processed on the GPU and finally converted back to the C++ representation. To avoid such data conversions we decided based on our own experiences (see also section 5.2.3) to implement all the data related code in C.

### 5.2.3 Thrust Library

Thrust is a CUDA library with an interface very similar to the C++ Standard Template Library (STL). This means you can have something similar to the algorithms in the STL to work with the CUDA code. The idea behind thrust is nice, especially when working with a C++ layer. Thrust provides a good abstraction and lower representational gap to the problem domain.

Initially thrust seemed a good entry point to CUDA as we were used to the concepts of C++ and the STL. Even though it can be simple to solve a problem in CUDA with the abstraction of thrust after the first few tries we got to know the downside of the thrust concept. Not every problem can easily be solved by a algorithm even with the great range of STL algorithms. To make things worse, by the time of this project the variety of thrust algorithms and thrust data structures is still very limited and thus it is even more difficult to solve as much problems as possible with thrust.

In our case, we came to this conclusion when we tried to implement equation 2.7 of the automatic seed selection algorithm (see section 2.3.3). The central issue in this case is that high-performance CUDA code is still highly dependent on the underling hardware meaning that you have to exploit the different memory hierarchies and the different memory types to get really fast CUDA programs. Hence, the mentioned mean value calculation would perfectly fit to the use of texture memory but when using thrust, unfortunately you have no control on the thrust internals. Therefore we had no chance to use texture memory with thrust. Another issue that comes up particularly when dealing with images is that you can not work with 2D or 3D memory allocations moreover that you have no control over the memory allocation at all. The algorithms (in thrust and in the STL) rely on the fact that the data you would like to operate on is in a linear sequence aligned in memory, which this is not always the case when operating on images. This makes the principle of "Arrays of Structures over Structures of Arrays" (see section 5.2.5) even more important to the work with thrust.

To summarize you can have nice short code with the thrust library for a certain kind of problems. However being on the C++ side you are forced to covert your data to a C compatible form as soon as it is not possible to solve a particular problem with thrust. As a matter of fact this evokes the problem of C and C++ code mixing describe in section 5.2.2. In our case this was the crucial factor to not work with thrust and to do the entire implementation in C.

### 5.2.4 Reuse device functions

Device functions are C functions annotated with the `__device__` tag. This tag causes the effect that these functions are only callable from the device e.g. from a CUDA kernel or another device functions. In some situations we would like to write such functions once and reuse them multiple times in different kernels. In the subsequent text we describe a scenario where it is desirable to provide device function implementations to be reused by many different kernels. Our motivation to show how we solved this problem is because we had some problems when trying to reuse these functions, whether the solution is simple.

The CUDA NVCC compiler supports some of the C++ features such as operator overloading. This allows you to write operator overloads for certain data types and use them in device code. Unfortunately CUDA does not provide any operator overloads on the built-in vector types such as `float4`, `uint3` or `char4` (all vector types can be found in the appendix of [12]). Accordingly we would like to write code that looks as follows:

```
1  __device__
   float4 do_math(float4 a, float4 b, int i) {
       return (a * i + b) / (i + 1);
   }
```

**Listing (5.4)** Use of `float4` Operator Overloads

As `float4` is a struct with four `float` values the math operations shown in listing 5.4 are not trivial. However when implementing operator overloads for the `float4` type we can write such code.

Undoubtedly we want to write such operator overloads only once and reuse them whenever we would like to do math operations on `float4`. Due to this we implemented operator overloads for vector types as shown in listing 5.5:

```
1  inline __host__ __device__ float4 operator+(const float4 &a, const
    float4 &b)
   {
       return make_float4(a.x + b.x, a.y + b.y, a.z + b.z, a.w + b.w);
   }
```

**Listing (5.5)** `float4` "Plus" Operator Overload

The overload itself is not different from a standard C++ implementation however to reuse this implementation within multiple kernels or device functions the keyword `inline` is crucial. We placed all our `float4` operator overloads in a header file and included that file everywhere we needed the overloads. Initially we did that without tagging the function with `inline` which lead us to a compiler error saying something like "multiple definitions of x". The reason for this is clear; the header file with the operator overload implementation is included in multiple CUDA source files and therefore the function is defined multiple times. However when you write the keyword `inline` you request the compiler to in-line the operator overload function. As a result the compiler

error disappears. Note, it is important to write the keyword `inline` when you would like to reuse device functions.

### 5.2.5 Structures of Arrays over Arrays of Structures

Considering the implementation of high-performance CUDA software we notice this very important concept. For CUDA hardware it is desirable to access memory locations in a linear sequence to get highest memory throughput. When threads within a warp or half a warp access the memory in sequence (thread 1  $\rightarrow$  location 1, thread 2  $\rightarrow$  location 2, etc.) the memory access can be coalesced by the hardware (see also [11], section "Coalesced Access to Global Memory"). To support coalesced memory access it is important to keep the data structures as structures of arrays rather than arrays of structures. What does that mean?

First of all let's take a look at the thing we do not want, arrays of structures. Listing 5.6 illustrates this principle in code. The structure `color` contains three floats which represent the RGB color values of one pixel. The complete image then consists of 256 `color` structures.

```

1  struct color {
    float r;
    float g;
    float b;
5  }

color image[256];

```

**Listing (5.6)** Array of Structures

If we consider the memory layout of such an arrangement as shown in figure 5.1 we can see that the structures lay one after another linearly aligned in memory (indicated by the green rectangles).



**Figure (5.1)** Array of Structures

Now, we would like to implement equation 2.7 in CUDA so assume the memory allocation in listing 5.6 is done for device memory and the data has already been copied to the device. To calculate the mean red value within the region of interest we have to read these values from memory (marked with the red arrows). Unfortunately the red values do not lay in successive order within device memory which means the memory access to these values can not be coalesced by the hardware.

Another method to address this problem is to not just save the color data of one pixel within the structure but save the color data of the complete image within a structure in the form of three arrays. The corresponding code is demonstrated in listing 5.7 .

```
1 struct color {  
    float r[256];  
    float g[256];  
    float b[256];  
5 }  
  
color image;
```

**Listing (5.7)** Structure of Arrays

Again assume the memory allocation has been done for device memory and the data has been copied but this time we can achieve a much better implementation of equation 2.7. In figure 5.2 you can see that the red values lay one after another in memory. As a result we can access these memory locations much faster than in the previous example. As illustrated with the red arrows the hardware accesses a series of red values with one read operation which means better performance for the overall runtime of this implementation.



**Figure (5.2)** Structure of Arrays

To sum up it is better for the overall performance of an implementation to organize the data as a structure of arrays rather than an array of structures. This reordering of data is done with regard to the CUDA hardware and hence the performance gain comes from the capability of the hardware to do coalesced memory access. Note, that optimizing coalesced memory access is also listed in [11] as a high priority task when tuning to high-performance CUDA code.

## 5.3 Literature

In this section I, Tobias Binna, would like to provide a review on the two books "Programming Massively Parallel Processors" [10] and "CUDA by Example" [19] I read as an introduction to CUDA.

### 5.3.1 Programming Massively Parallel Processors

Initially I thought this book would be a good introduction to learn how to program CUDA but today I have to say it is not really an introduction. In my opinion the book has a nice start but then gets very fast into details on the GPU architecture and how to tune your program to high performance. I agree, in the end this is all CUDA is about but as an introduction I prefer to first get a global overview and then go further into details. In addition the book does never really says something about any CUDA API functions or what the API actually provides. I missed this stuff all along the reading.

By now it seems like I criticized almost everything written in this book. But I am sure after you have worked out the basic stuff the book has also positive sides. The chapters about memory, performance and floating point considerations will certainly help you when you try to get the last bits of performance out of your code. After many ups and downs during the reading I was satisfied by the last chapter in the book called "conclusion and future outlook". It gives you a nice overview of the current state of CUDA and what is and what is not possible when programming CUDA. Even though it is a nice good finish I thought: "Why couldn't you tell me earlier in the book about all that stuff?"

### 5.3.2 CUDA by Example

First of all, I think Well I have to say, I definitely read them in the wrong order.

This book gives you a very good introduction in the most important concepts of CUDA. Every Chapter covers one or two concepts from the CUDA API such as different Memory Types, Events, Graphics Interoperability with OpenGL or Streams. As an introduction this gives you a very good overview on the most important things provided by the API, and you get a firm understanding on what you have to write on your own and what you can use as predefined.

Usually each topic is introduced by showing a portion of code that solves a particular problem without actually using the dedicated concept that is being introduced. Then this code is transformed step by step into the new solution by introducing the new concept and showing how it is done better. I liked this kind of learning CUDA because after studying this examples I think you know when you should use which concept.

The important thing to say on this book is that you really get an introduction on the API functions and features. When you are reading the book reviewed in section 5.3.1 you get taught the more the general concept of the hardware and the language itself. Unfortunately I read this book after "Programming Massively Parallel Processors" and I have to say I definitely read them in the wrong order.





# Appendix A

## Development Environment Setup

### A.1 Hardware and Software Tools

Table A.1 and Table A.2 summarize the hardware and software tools we used for the development of this term project.

#### A.1.1 Hardware

Tool	Description
Processor	Intel Xeon CPU E5520, 8 x 2.27 GHz
Memory	4049 MB
Graphics Device	NVIDIA Quadro FX 3800 - 192 CUDA Cores - 1024 MB Memory - Compute Capability 1.3

**Table (A.1)** Project Hardware

#### A.1.2 Software

Tool	Description
Operating System	Ubuntu 10.04 LTS - 64-bit
CUDA Driver Version	3.20
CUDA Runtime Version	3.20
Build System	CMake 2.8
Development Environment	Eclipse CDT (Helios)
Unit Test	CUTE (C++ Unit Testing Easier)
Libraries	OpenCV, CUDPP, CUDA Utility Library (cutil)

**Table (A.2)** Project Software

Writing CUDA code is basically programming C. Therefore we used the basic Eclipse CDT development environment with some small adjustments to support the build process and readability of the source code. To build our programs we used the CMake build system which made a good job on a "one-button" build process.

## A.2 Setup CUDA

The simplest way to setup the CUDA development tools is to just follow the "NVIDIA CUDA C Getting Started Guide" for the desired platform. This guide is available from the NVIDIA Developer Zone (<http://developer.nvidia.com>).

Note that in case you are working on hardware that has a built-in graphics chip and a dedicated graphics chip (in our case a MacBook Pro with a built-in Intel graphics chip and a external NVIDIA graphics processor) you may have to force the device to use the NVIDIA hardware to be able to run your programs. If a CUDA program is executed and the hardware is not running on a CUDA capable device one might get an error such as

```
cudaGetDeviceCount FAILED CUDA Driver and Runtime version may be mismatched
```

In this case you have to force your hardware to switch to the NVIDIA capable device before you run the program.

If you are on a Mac you can install a nice free tool called "gfxCardStatus" (<http://codykrieger.com/gfxCardStatus/>) which indicates you with a little icon on which chip one is working and helps you to easily force your Mac to use the NVIDIA chip.

## A.3 Setup Eclipse CDT with CMake and CUDA

This section describes how we setup and configured our Eclipse installation together with the build system CMake to build the CUDA programs.

### A.3.1 Add File Type .cu

Because Eclipse does not know the extension .cu you may configure Eclipse to treat these files as C or C++ source files. To do so do the following steps:

1. go to Window → Preferences
2. open C/C++ → File Types dialog
3. click New...
4. then write as Pattern: \*.cu and select Type: C++ Source File
5. finish the dialog with OK

Now all the .cu files should have code highlighting. The only thing that is not recognized by Eclipse are the kernel launches with the triple brackets (<<<...>>>). Repeat these steps if you would like to have highlighting for .cuh files as well. In this case select as type C++ Header File.

### A.3.2 Setup CMake and CMake Configuration in Eclipse

We assume that you have already downloaded and installed CMake on your system and you are aware of the concepts of CMake. Install the CMakeEd plugin for Eclipse (<http://cmakeed.sourceforge.net>) which supports you when editing CMakeLists files with code highlighting and code completion. Note, the following documentation list just the parts of the CMake files needed to compile CUDA code, the complete files however can be found in the provided source material.

First of all, we check for the module FindCUDA.cmake (see listing A.1). This module should come with your CMake installation as a standard module.

```
1  # find cuda
   IF (INCLUDE_CUDA)
       FIND_PACKAGE (CUDA)

5      IF (CUDA_FOUND)
           MESSAGE ("CUDA has been found")
       ELSE (CUDA_FOUND)
           MESSAGE (FATAL_ERROR "CUDA could not be found")
       ENDIF (CUDA_FOUND)
10  ENDIF (INCLUDE_CUDA)
```

**Listing (A.1)** Check for FindCUDA.cmake

Tell CMake to build a CUDA program by setting the compile instruction as shown in listing A.2, the link instruction is as usual.

```
1  CUDA_ADD_EXECUTABLE (${EXE_NAME} ${source} ${main})
   TARGET_LINK_LIBRARIES (${EXE_NAME} ${LIBS})
```

**Listing (A.2)** CMake CUDA Executable

These are the only special commands used to instruct CMake to build a CUDA program nevertheless the FindCUDA module provides a lot more variables and options to customize the build and set NVCC compiler flags. All these variables can be found on the web in the official CMake documentation. Now that we have covered the special CMake commands we have to configure the Eclipse project properties to work best with CMake.

Follow the steps below to set up Eclipse with CMake.

1. start a new project with  
C++ Projekt → Makefile project → Empty Project
2. create a build folder for out-of-source builds in the project root  
File → New → Folder → Folder Name: build
3. adjust the project properties  
Project → Properties → C/C++ Build

- set the Build location to the just created build folder
  - uncheck Generate Makefiles automatically
  - click OK to finish
4. edit the make targets On the right side of the editor window there should be a tab called **Make Targets**. Click this tab and select your project, then right click and select **New...**
- set the target name to `cmake`
  - uncheck Same as the target name and Use builder settings
  - delete the text in the field Make target
  - write in Build command the text `cmake ..` (you have the possibility to set additional CMake command line arguments right after `cmake`)
5. Now you can start coding. Before the first build and after every change on the CMakeLists files double click the `cmake` target created in the previous step. Afterwards compile your project as usual with the compile button.

## Bibliography

- [1] Cuda data parallel primitives library.  
*<http://code.google.com/p/cudpp/>*, page last visited: December, 2010.
- [2] Opencv library.  
*<http://opencv.willowgarage.com/wiki/>*, page last visited: December, 2010.
- [3] Thrust project.  
*<http://code.google.com/p/thrust/>*, page last visited: December, 2010.
- [4] Wikipedia, ycbcr.  
*<http://en.wikipedia.org/wiki/YCbCr>*, page last visited: December, 2010.
- [5] R. Adams and L. Bischof.  
Seeded region growing.  
*IEEE Trans. Pattern Anal. Mach. Intell.*, 16:641–647, June 1994.
- [6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.  
*Design Patterns: Elements of Reusable Object-Oriented Software*.  
Addison-Wesley, 1995.
- [7] K.A. Hawick, A. Leist, and D.P. Playne.  
Parallel graph component labelling with gpus and cuda.  
*Parallel Computing*, 36:655 – 678, 2010.
- [8] Keith Jack.  
*Video demystified: A Handbook for the Digital Engineer*.  
Newnes, 2007.
- [9] Bahram Javidi, editor.  
*Image Recognition and Classification*.  
CRC Press, 2002.
- [10] David B. Kirk and Wen-mei W. Hwu.  
*Programming Massively Parallel Processors: A Hands-on Approach*.  
Morgan Kaufmann Publishers Inc., 2010.
- [11] NVIDIA Corporation.  
*NVIDIA CUDA Best Practices Guide*, 3.2 edition, 2010.
- [12] NVIDIA Corporation.  
*NVIDIA CUDA C Programming Guide*, 3.2 edition, 2010.
- [13] NVIDIA Corporation.  
*NVIDIA CUBLAS User Guide*, 2010.
- [14] NVIDIA Corporation.  
*NVIDIA CUFFT User Guide*, 2010.

- [15] NVIDIA Corporation.  
*NVIDIA CURAND User Guide*, 2010.
- [16] NVIDIA Corporation.  
*NVIDIA CUSPARSE User Guide*, 2010.
- [17] N. Otsu.  
A threshold selection method from gray-level histograms.  
*IEEE Transactions on Systems, Man and Cybernetics*, 9:62–66, January 1979.
- [18] T. Pavlidis and Y.-T. Liow.  
Integrating region growing and edge detection.  
*IEEE Trans. Pattern Anal. Mach. Intell.*, 12:225–233, March 1990.
- [19] Jason Sanders and Edward Kandrot.  
*CUDA by Example: An Introduction to General-Purpose GPU Programming*.  
Addison-Wesley Professional, July 2010.
- [20] Frank Y. Shih.  
*Image Processing and Pattern Recognition: Fundamentals and Techniques*.  
Wiley-IEEE Press, 2010.
- [21] Frank Y. Shih and Shouxian Cheng.  
Automatic seeded region growing for color image segmentation.  
*Image Vision Comput.*, 23:877–886, September 2005.