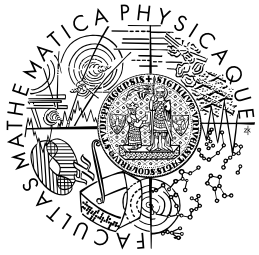


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

DOCTORAL THESIS

Jan Kolomazník

Interactive Processing of Volumetric Data

Department of Software and Computer Science Education

Supervisor of the doctoral thesis: RNDr. Josef Pelikán
Study programme: Computer Science
Study branch: Software Systems

Prague 2018

I declare that I carried out this doctoral thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

Title: Interactive Processing of Volumetric Data

Author: Jan Kolomazník

Department: Department of Software and Computer Science Education

Supervisor: RNDr. Josef Pelikán, Department of Software and Computer Science Education

Abstract:

Interactive visualization and segmentation of volumetric data are quite limited due to the increased complexity of the task and size of the input data in comparison to two-dimensional processing.

A special interactive segmentation workflow is presented, based on minimal graph-cut search. The overall execution time was lowered by implementing all the computational steps on GPU, which required a design of massively parallel algorithms (using thousands of threads). To lower the computational burden even further the graph is constructed over the image subregions computed by parallel watershed transformation.

As a suitable formalism for a range of massively parallel algorithms was chosen cellular automata. A set of cellular automata extensions was defined, which allows efficient mapping and computation on GPU. Several variants of parallel watershed transformation are then defined in the form of cellular automaton.

A novel form of 2D transfer function was presented, to improve direct volume visualization of the input data, suited for discriminating image features by their shape and size.

CUDA Generic Image Processing (CUGIP) library was created to wrap common image processing patterns implemented for CUDA devices and presented algorithms were implemented as part of it.

By combination of all these components a fast responsive system for interactive segmentation was created, where all computationally expensive operations are executed on CUDA-enabled graphics card.

Keywords: minimal graph-cut, segmentation, transfer functions, cellular automata, GPGPU, parallel algorithms, statistical models

Název práce: Interaktivní zpracování objemových dat

Autor: Jan Kolomazník

Katedra: Katedra software a výuky informatiky

Školitel: RNDr. Josef Pelikán, Katedra software a výuky informatiky

Abstrakt:

Interaktivní vizualizace a segmentace objemových dat je v porovnání s dvoudimenzionálním zpracování obrazu omezována složitostí algoritmů a velikostí zpracovávaných dat.

V této práci je popsána metoda pro interaktivní segmentaci, jež je založena na hledání minimálního řezu grafem. Celkový čas výpočtu byl snížen převedením všech výpočetních kroků na GPU. Tento cíl vyžadoval návrh masivně paralelních algoritmů (využívajících tisíce vláken). Ke výraznějšímu snížení výpočetní zátěže je graf konstruován nad množinou regionů vytvořených watershed transformací vstupu.

Jako formalismus pro řadu masivně paralelních algoritmů byly vybrány celulární automaty, které spolu s prezentovanou sadou rozšíření, umožňují efektivní mapování a výpočet na grafických kartách. Několik variant paralelní watershed transformace je následně definováno pomocí těchto celulárních automatů.

Prezentována byla také nová varianta 2D přechodových funkcí pro přímé zobrazování objemových dat. Metoda je vhodná pro odlišování objektů v obraze na základě jejich tvaru a velikosti.

Pro usnadnění implementace algoritmů digitálního zpracování obrazu na grafických kartách byla vytvořena knihovna CUDA Generic Image Processing (CUGIP).

Kombinací všech těchto dílčích komponent byla vytvořená metoda pro interaktivní segmentaci objemových dat, kde jsou všechny výpočetně náročné kroky spouštěny na grafických kartách podporující CUDA rozhraní.

Klíčová slova: minimální řez grafem, segmentace, přechodové funkce, celulární automaty, GPGPU, paralelní algoritmy, statistické modely

At this point I would like to thank to my supervisor RNDr. Josef Pelikán. Without his guidance and support, this work would not have been possible.

I would also like to thank fellows in our department who provided countless advices.

I also dedicate this dissertation to my loving wife Soňa and our two kids Matyáš and Kristýna, who supported me throughout the process and had the patience when I was finishing this thesis.

Last but not the least, I would like to thank my parents who had never lost trust in me and supported me in all possible ways throughout my life.

This thesis and the related research was partially supported by research Grant GAUK 355311.

Contents

Introduction 4
Goals . . . 5
Thesis Structure . . . 6
Contributions . . . 7

1 GPU Architecture 10
1.1 CUDA . . . 10
1.1.1 Kernel Execution . . . 10
1.1.2 Memory Types . . . 13
1.1.3 Asynchronous Computations and Memory Transfers . . . 14
1.1.4 Synchronization . . . 15
1.2 CUGIP - CUDA Generic Image Processing . . . 16
1.3 Concepts . . . 17
1.3.1 Images . . . 17
1.3.2 Views . . . 19
1.4 Lazy evaluation . . . 20
1.4.1 Expression Templates . . . 20
1.5 Meta-algorithms . . . 21
1.5.1 Copy . . . 21
1.5.2 For Each . . . 21
1.5.3 Transform . . . 22
1.5.4 Reduce . . . 23
1.5.5 Scan . . . 23
1.6 Code Example . . . 23
1.7 Conclusions and Future Work . . . 24

2 Voxel Data Preprocessing 26
2.1 Denoising . . . 26
2.1.1 Non-local Means . . . 27
2.1.2 Threading Schemes . . . 29
2.1.3 Optimizations . . . 30
2.2 Coherence Enhancing Diffusion . . . 33
2.2.1 Diffusions . . . 34
2.2.2 Structure Tensor . . . 34
2.2.3 Algorithm . . . 36
2.2.4 Implementation . . . 36
2.3 Conclusions . . . 36

3 Transfer Functions **39**
3.1 Volume Data Visualization . . . 39
3.2 Direct Volume Rendering . . . 40
3.2.1 Discretization . . . 40
3.2.2 Compositing Schemes . . . 41
3.2.3 Renderer Implementation . . . 42
3.3 Transfer Functions . . . 42
3.3.1 Transfer Functions of Higher Dimension . . . 44
3.3.2 Size-Oriented Transfer Functions . . . 44
3.3.3 Size and Shape Oriented Transfer Functions . . . 46
3.4 Conclusions . . . 48

4 Cellular Automata and Watershed Transformation **50**
4.1 Cellular Automata . . . 50
4.1.1 Cellular Automata Introduction . . . 50
4.1.2 Definitions . . . 52
4.1.3 Cellular Automata on GPU . . . 54
4.1.4 Asynchronous Cellular Automata . . . 55
4.1.5 Speed of Convergence . . . 57
4.1.6 Global State . . . 58
4.1.7 Cells with Pointers . . . 62
4.2 Watershed Transformation . . . 66
4.2.1 Terminology . . . 67
4.2.2 Distance Based Watershed Transformation . . . 74
4.2.3 Watershed Transformation by Steepest Descent . . . 75
4.2.4 Steepest Descent with Plateau Handling . . . 78
4.2.5 Automaton with Pointers and Stream Processing . . . 79
4.2.6 Benchmarks . . . 80
4.3 Conclusions . . . 81

5 Minimal Graph-cut in CUDA **88**
5.1 Overview . . . 89
5.1.1 Segmentation . . . 89
5.1.2 CPU Graph-cut Implementations . . . 90
5.1.3 GPU Graph-cut Implementations . . . 91
5.1.4 Other Graph Algorithms for GPU . . . 92
5.2 Graph Topologies . . . 92
5.2.1 Grid Graph . . . 92
5.2.2 Subregion Adjacency Graph . . . 93
5.3 Push-Relabel . . . 94
5.3.1 Min-cut . . . 95
5.3.2 Graph Representation . . . 97
5.3.3 Relabel Step . . . 97
5.3.4 Push Step . . . 98
5.4 Optimizations . . . 98
5.4.1 Algorithmic Optimizations . . . 99
5.4.2 Code Optimizations . . . 100
5.4.3 Heuristics . . . 102
5.4.4 Approximations . . . 102

5.5 Benchmarking . . . 103
5.6 Conclusions . . . 104

6 Statistical models **108**
6.1 Intensity Distribution . . . 109
6.1.1 Formulation . . . 109
6.1.2 Training . . . 110
6.2 Shape Model . . . 110
6.2.1 Formulation . . . 111
6.2.2 Training . . . 111
6.3 Model Usage . . . 113
6.3.1 Alignment . . . 113
6.3.2 Parametric contours/surfaces . . . 113
6.3.3 Minimal Graph-cut . . . 114
6.4 Results . . . 115
6.5 Conclusions . . . 116

7 Complete Workflow **117**
7.1 Preprocessing . . . 117
7.2 Markers Initialization . . . 117
7.3 User Interaction . . . 119
7.4 Pipeline Recalculation . . . 120

Conclusions **121**
Future Work . . . 124

Bibliography **125**

List of Definitions **131**

List of Cellular Automata Rules **132**

Introduction

Most well know volumetric data acquisition methods come from medicine where methods, such as computed tomography (CT), magnetic resonance (MRI), positron emission tomography (PET), are used to diagnose and monitor internal organs in a non-invasive way. Other types of the volumetric data come from industrial CTs, multifocal microscopy, cryo-electron tomography, and lots of others.

With the steadily increasing computational power of modern computers comes an increased demand for advanced processing of the volumetric images.

Lots of scientific papers were published on automatic or semi-automatic methods for image processing (segmentation, classification, registration, etc.). These methods usually need to find a tradeoff between speed, precision, and generality. Precise algorithms usually require a deep knowledge of solved problem (complex training patterns, statistical modeling) and fast implementations must be tuned for a particular task.

Tuning algorithm for a specific application can be problematic in several situations. In medical imaging scanned subjects often show pathologies in regions of interest – malformed organs due to illness or injury. Tuning algorithms for segmentation or registration for such scenario is often extremely hard.

Semi-automatic algorithms commonly suffer from a usage of unintuitive control parameters (number of iterations, regularization coefficients, etc.), which require an experienced user to set them correctly.

Interactive methods for 2D image processing became widely available with software packages like Photoshop and Gimp. Almost everybody can now edit their photos on mobile devices.

A completely different situation arises when processing 3D volumetric images. The additional dimension brings increased amount of processed data, worse algorithmic complexities and completely new problems coming with different topology. In 2D images each pixel had 4 or 8 neighbors, in 3D images each voxel has 6, 18 or 27 neighbors, region adjacency graphs are no longer plannar, etc.

In such context not many methods exist which can deliver high-quality outputs and also compute the results in such a short time to be usable in an interactive environment, where user tunes the output either by fine-setting input parameters, adding hints, or correcting mistakes.

In recent years GPGPU (general-purpose computing on graphics processing units) became widely used approach to solving large set of computational problems. Due to the specifics of the GPU architectures, some tasks are more suitable than others for implementation on GPUs.

In situations when most of the computation workflow tasks are solved on GPU the ones which are computed on CPU may become bottlenecks due to the neces-

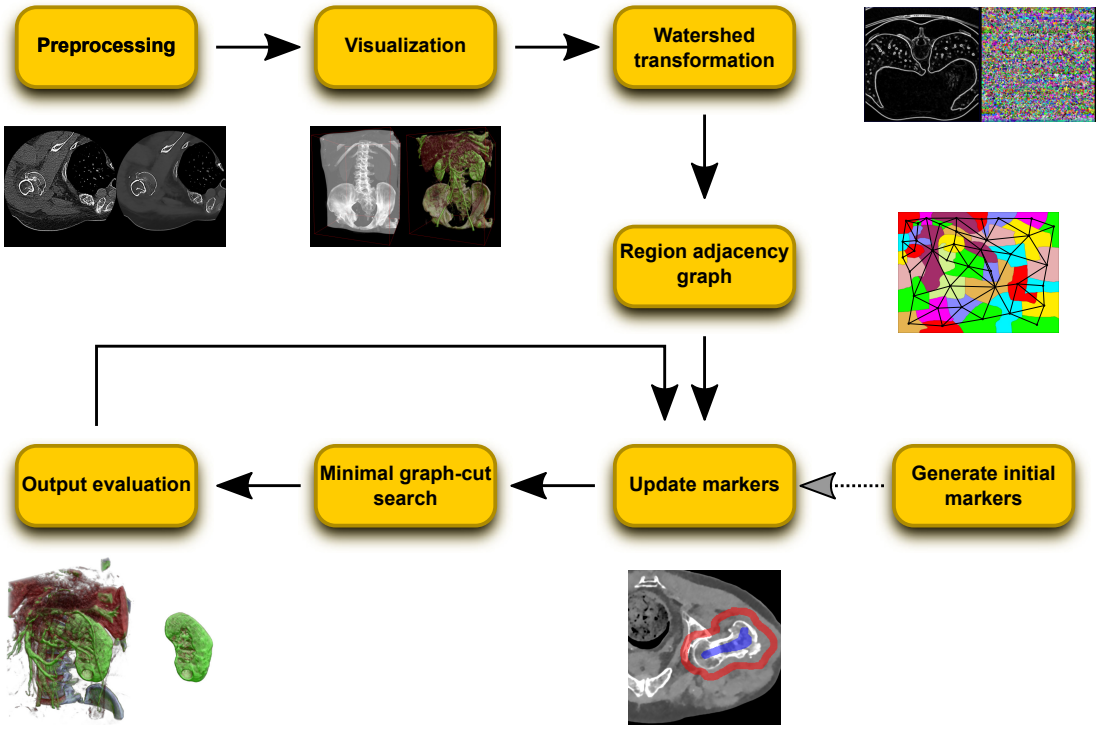


Figure 1: Scheme of the workflow. Each step will discussed in detail in their respective chapters.

sary transfers between GPU memory and RAM. That can enforce implementation of algorithms, which are not usually suitable for computation on GPU. Even in situations, when only small speedup (if any) is gained for the particular algorithm, but the performance of the whole computational pipeline can be improved.

Goals

The main goal of this thesis is to design a complete workflow targeted on interactive segmentation and data visualization of volumetric images. Overview of the workflow can be seen in figure 1.

As most of the steps are computationally expensive, a clear strategy will be finding the fastest approach for solving each step. This will be achieved either by careful selection of suitable algorithms and their approximations, making the task smaller without sacrificing precision (if possible) and by exploiting the overwhelming performance of modern GPUs by reformulating specific tasks, so these can be solved on massively parallel computational architectures (thousands of threads of computation).

The selected algorithms will be studied in broader context to allow their usage in other situations, workflows or on different kind of input data.

Lots of computational patterns emerge when solving image processing problems. So another goal is to follow one of the basic software engineering principles – *DRY (Don't Repeat Yourself)*. The emergent patterns are studied and possible generalizations for the wider set of problems are derived. These generic solutions are gathered into a computational framework, which ease up the development

and experimentation with other variants of the core problems.

Such frameworks also help to hide complexities introduced by APIs of the used technologies, which tend to be quite technical and hardware oriented.

Although this thesis is focused on 3-dimensional data, generic formulations, which works for an arbitrary dimension are used when possible. That is why *image element* will be used instead of *pixel* for 2-dimensional images and *voxel* for 3-dimensional images.

Thesis Structure

Chapter 1 A large portion of this thesis is focused on utilization of graphics processing units (GPUs) for high-performance computations. Because of that, the first half of this chapter serves as an introduction to GPU architectures and CUDA framework, which will be used for implementation of the algorithms presented in following chapters.

In the second half of the first chapter a new library for image processing is introduced. CUGIP (CUDA Generic Image Processing) library is based on CUDA framework and provides basic primitives for image processing designed with speed and simplicity of usage in mind.

Chapter 2 Parallel implementations of two nontrivial preprocessing algorithms are given. Denoising in the form of non-local means algorithm and coherence enhancing diffusion algorithm, which will be used for the improvement of the gradient magnitude images.

Chapter 3 This chapter starts with brief overview of current approaches to 3D volume rendering methods and usage of color mapping and transfer functions to highlight objects of interest inside the voxel image.

A new type of transfer function is presented at the end of the chapter. It is incorporating ideas of 2D and size-based transfer functions, together with a multiresolution analysis of Hessian matrices and objectness measure. The proposed transfer function can highlight or suppress objects based on their size and shape, discriminating between planar, tubular and spherical shapes.

Chapter 4 Lots of image processing algorithms have in their core computations on image element (pixel/voxel) neighborhood. Some of the typical representatives are denoising, edge detection, numerical solutions of differential equations.

These algorithms are usually well suited for parallel computations, but when designing a new rather complex algorithm the proof of correctness in a concurrent environment can be hard if using formalisms normally used for sequential algorithms.

This chapter shows that a large set of mentioned problems is solvable by using cellular automata as a suitable formalism.

Overview of classical cellular automata theory is given at the beginning of the chapter and in the following sections, new cellular automata extensions

are presented. These should ease up the usage of the formalism to design new algorithms and prove their correctness, together with tools, which allow efficient implementations for massively parallel architectures.

The second half of the chapter is devoted to the watershed transformation. Overview of several alternative formal definitions of watershed transformation is given together with sequential algorithms. The theory of extended cellular automata is then used to analyze parallel watershed algorithms present in literature. New methods are presented, and comparison of the studied algorithms is given at the end of the chapter.

Chapter 5 A wide range of applications of minimal graph cut algorithms can be found in the literature. Fifth chapter focuses on their usage in interactive segmentation. The goal is to design algorithm, which can compute minimal graph cut as fast as possible so that it can be used interactively even for 3D volumetric images. To achieve that goal the task at hand is reduced in size by assuming graph of image subregions or super-pixels/super-voxels (watershed regions) instead of direct usage of image elements.

The raw computing power of modern GPUs is again chosen as the primary tool. Although graph-cut algorithms are not well suited for parallelization, especially if the topology of the graph at hand is not constrained (grids for pixel/voxel neighborhood graphs). The final version the algorithm shows satisfying computation speeds as is presented at the end of the chapter, although it does not outperform the CPU algorithms in all cases. For that reason an approximate variant of the min-cut algorithm is introduced, which lowers the computation times significantly without severe impact on the precision of the final segmentation.

Chapter 6 Common problem for broad range of interactive segmentation methods is amount of user work required for satisfying results. In Chapter 6 a lightweight statistical model which is suitable for initialization of segmentation algorithms and thus replacing lots of tedious manual work required. The user can then focus only on the problematic (blurred, noisy) parts of the segmented object.

Chapter 7 All previous chapters focused on different aspects of image processing. In the final chapter will be shown how the presented methods can be combined into the single interactive workflow, which is well suited for wide range of segmentation tasks, but can also be tuned for a particular task, either by training specialized statistical model, and presetting the control parameters.

Contributions

Patterns, which emerged when implementing algorithms in CUDA framework, together with complexity coming from memory management and kernel executions on GPU led to the creation of CUDA Generic Image Processing (CUGIP) library. After experimentation with a range of concepts, the final set allows easy

GPU memory management, zero (or near zero) overhead in comparison to using plain CUDA API.

Usefulness of this library was already proved commercially – a stable subset of the library was already licensed by private companies.

Optimization strategies for computationally intensive algorithms for suppressing noise: non-local means and coherent enhancing diffusion. The optimizations are focused on massively parallel implementations on GPU, which surpasses the performance of CPU implementations by almost two orders of magnitude.

Transfer function suited for highlighting objects by shape, size, and density during direct volume rendering (DVR) in one concise framework based on 2-dimensional transfer functions.

Cellular automata were used as a convenient formalism for the description of a large set of parallel algorithms. New extended definitions of cellular automata were defined to efficiently map the mathematical formalisms to modern GPU architectures. Application of these new cellular automata on the watershed transformation, together with novel lower completion scheme results in extremely fast implementations.

A two pass stream algrorihm is also presented for processing of large datasets in an out-of-core manner.

Algorithm for minimal graph-cut search suitable for segmentation in general graphs, but tuned for the induced minors[1] of the grid graphs. An approximation scheme for the final version of the algorithm was introduced to deal with slow convergence and serialization of the parallel tasks. It was shown that the approximation scheme have only minor influence on the precision of the final segmentation.

A statistical model was created, which describes foreground and background intensity distributions in relation to the position in space. The intensity distribution is then augmented by a vector field, which models average contour normal in the region and constraints thus the possible shape of the 2D/3D contour of the segmented object. Also, gradient modulation based on the presented vector field is developed.

Compilation of all the presented methods into one concise, interactive workflow for visualization and segmentation of volumetric data.

Other Contributions

Most of the standalone GPGPU algorithms presented in this thesis (chapters 2, 5) were implemented as a part of the CUGIP library (section 1.2). The rest of the algorithms, together with the envelope workflow are part of the *MedV4D framework*.

MedV4D is software project targeted on easier development of medical applications, and thesis author is its main maintainer. It consists of several standalone

[1] Graphs generated by edge contractions.

libraries targeting on GUI for applications working with volumetric data, real-time rendering, image processing and manipulation with DICOM format.

Chapter 1

GPU Architecture

Development of powerful Graphic Processor Units or GPUs was driven by market demand for real-time, high-definition 3D graphics. Although the architecture and design of GPUs were focused on real-time rendering of polygonal meshes with a dawn of programmable rendering pipelines, GPUs became quite universal tools providing tremendous computational power with a high degree of parallelism and memory bandwidth. Available computational power currently grows much faster for GPUs in comparison to CPUs as can be seen in figure 1.1.

1.1 CUDA

The CUDA compute platform (NVIDIA Corporation (2018d)) provides parallel computing extensions to a range of popular programming languages – C/C++, Fortran, Python to name a few. It is computation model and API created by NVIDIA to provide developers with access to general purpose processing on CUDA-enabled GPUs – GPGPU.

It is a heterogenous computation platform divided into two parts – *host* and *device*. CPU and its memory for the host side and GPU (GPUs) form the device side.

Top level functions, which will be executed on a device are called *kernels*. CUDA platform allows implementing host and device code next to each other using language extensions, which allows CUDA compiler to separate both parts and compile them for correct hardware architecture.

Current hardware for CUDA implements a massively parallel architecture with the ability to run certain algorithms much faster (even by one or two orders of magnitude) than common CPU. However, such power is not for free, and algorithms must often be completely reworked to fulfill many conditions to enable the hardware arithmetic units to perform with top performance. Care must be taken in the areas of memory access, conditional jumps, and many others.

1.1.1 Kernel Execution

Parallel execution on CUDA-enabled device is done through threads of computation, which are hierarchically organized into 3-dimensional blocks and these are then again organized in 3-dimensional grid[1].

[1]Devices with compute capability lower than 2.x supported only 2-dimensional grids.

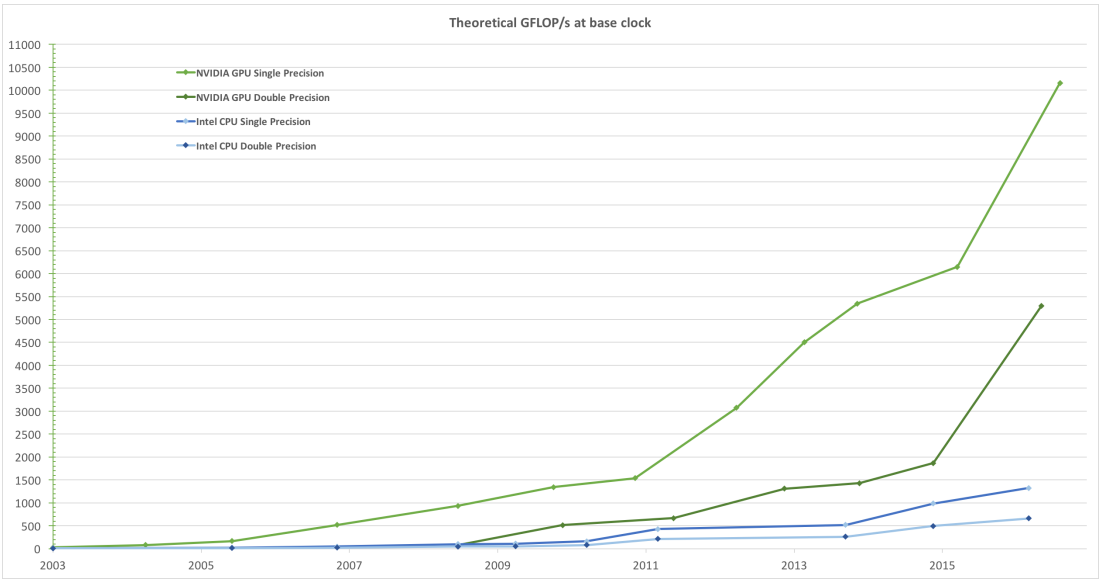


Figure 1.1: Floating-Point Operations per Second for the CPU and GPU (CUDA 9.1 documentation)

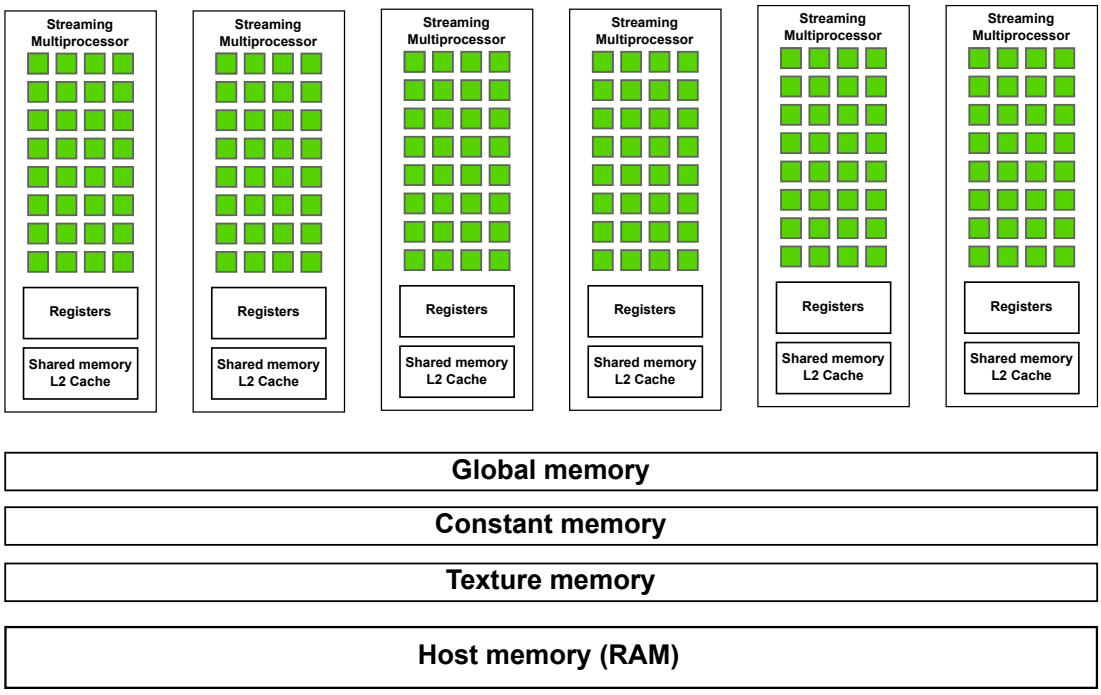


Figure 1.2: CUDA architecture: green boxes are processing units – organized into streaming multiprocessors.

When kernel is executed, it is done by specifying kernel parameters, passed as the regular function parameters (although passed by memcpy instead of copy constructors), and kernel execution parameters, which provide the device driver with sizes of the grid, thread block and shared memory.

As a result, all thread blocks in the grid have the same size. Once the kernel is executed than these parameters cannot be changed. All threads in the grid execute the same kernel function.

Thread Block

Each block is assigned to some streaming multiprocessor (SM) and is executed in its entirety by the SM. Registers and shared memory are allocated for a block as long as that block is active. Once a block is active it will stay active until all threads in that block have completed.

Each thread has a unique ID within the block to distinguish between different threads, and each block has assigned unique ID within a grid. So thread ID and block ID uniquely identify each thread within a grid.

Warp

The threads in a block are further sequentially divided into groups called *warps*. Warp is composed of 32 threads. Each thread in the warp must execute the same instructions, even in situations when a control flow diverges for some of the threads. In such cases, all divergent branches are executed in all threads, but instruction effects are masked, so the behavior looks like the different branches are executed sequentially.

So one of the optimization goals is to prevent thread divergence inside a warp.

Context Switch

Once a block is active, registers and shared memory required by the thread-block are allocated. These resources remain allocated for the block as long as the block is active and cannot be used by another block.

The context switching between active warps is very fast because the resources do not have to be saved and restored. So the goal is to have enough transactions in flight to saturate the memory bus. If a warp needs to wait for a data load another warp can be scheduled.

Occupancy

There is a maximum number of warps which can be concurrently active on a Streaming Multiprocessor (SM). Occupancy is defined as a ratio of active warps on SM to a maximum number of active warps.

Occupancy is limited by resources required by executed kernel (registers, shared memory) and the used block size. It is one of the optimization strategies to have high occupancy so that scheduler can switch between warps, which execute high latency operations.

1.1.2 Memory Types

There are several memory types, which are used by a CUDA device. These types differ in accessibility, speed, optimal load patterns, and caching approach.

Host Memory

Host memory is CPU accessible RAM, which can be allocated in standard way for the used language, or by utility functions provided by CUDA runtime API. It is not directly seen by kernels and as such its contents must be first transferred to another type of device accessible memory.

Data transfers to device memory are internally implemented through the pinned memory (cannot be swapped by OS) and DMA page-by-page copy. There is an option to use pinned memory directly and prevent the additional internal memory transfers.

Linear Device Memory

Linear device (global) memory is one of the memory types visible to CUDA kernels. It must be allocated and deallocated through the CUDA API calls. Because it is a linear memory, value caching is only one dimensional and as such it is not ideal for 2D/3D images and their memory layouts. That is the reason why shared memory is often used to hide memory latency for access patterns typical for multi-dimensional memory buffers.

Each load from global memory loads continuous segment; this is the reason, why the worst access pattern is reading from a random location and the most optimal one is a coalesced memory access.

Unified Memory

With devices of compute capability 3.0 or higher came an option to allocate a so called *unified memory*. It is a pool of managed memory that is shared between the CPU and GPU. The unified memory have shared address space between host and device, so single pointer can be used to access data from either context. This removes the need for explicit manual sync between buffers in device and host memory.

Compute capability 6.0 brought another improvement in the form of possible unified memory oversubscription. It is possible to allocate bigger unified memory buffers than the size of the GPU memory. Driver then takes care of synchronization between host and device as the computation demands. The predictibility of memory access patterns plays important role as it can severely hinder the data preloading via GPU driver.

Constant Memory

Constant memory resides in device memory. It uses constant memory cache which is optimized for read-only access.

CUDA Arrays

Important parts of GPU hardware are texture mapping units which do the texture sampling, interpolation and mapping to geometry primitives. Because textures are usually accessed in a coherent manner, texture caching is adapted to these access patterns, so follow-up reads of neighbor texels are fast. Also, interpolation and access through normalized indices are directly supported by the hardware.

It is useful to get access to these hardware features and for this reason, CUDA supports so-called *CUDA arrays*, which are opaque memory layouts optimized for texture fetching. The value fetching value must be done through specialized functions in contrary to linear memory access which is done through the pointer arithmetics.

Shared Memory

Shared memory is an on-chip data storage of high bandwidth. It is allocated per thread block, so all threads in the block have access to the same shared data. Threads can access data computed or loaded by other threads in a block if it is stored in shared memory buffer.

The usage of the on-chip memory combined with thread synchronization has various advantages. The main one would be preventing slow (in comparison to on-chip memory) global memory reads of data which are used multiple times during computation – shared memory is then used for user-managed data caches.

Shared memory is divided into equally sized memory modules (banks) to achieve high memory bandwidth for concurrent accesses, as these banks can be accessed simultaneously. These banks are assigned cyclically to the shared memory modules (32-bit words and possibility of 64-bit banks for compute capabilities 3.x and newer). If all threads access shared memory through different memory bank, then all these accesses happen simultaneously. The problem is if several threads use the same bank. In such situation the memory requests are serialized, only exceptions are if all threads load from same memory location (broadcast), or several threads load from same memory (multicast on newer architectures).

Aside from memory bank conflicts, there is no penalty for non-sequential or unaligned accesses by a warp in shared memory.

1.1.3 Asynchronous Computations and Memory Transfers

Most of the CUDA calls are asynchronous by default (kernel execution, memory copying, etc.), so these calls return immediately, and GPU (or GPU driver) does work in the background. This is useful because CPU is then available for other work.

When more asynchronous CUDA operations are called their execution happens in the same order, each starts when the previous one finished. This is to ensure that all the operations have valid arguments in case there is dependence between consecutive calls (e.g. memory copy and kernel computation on that particular piece of memory).

For situations when it is known that some of the calls are truly independent, CUDA provides concept of operation streams. Operations assigned to the

same stream are executed in sequence as described, but operations from different streams can run concurrently.

If no stream is specified, the operation is assigned to the default stream. Although CUDA 7.0 adds an option `--default-stream per-thread`, which creates a default stream for each host thread.

1.1.4 Synchronization

An important property of any parallel processing framework is the set of available synchronization primitives.

Thread Block Synchronization

When sharing data between threads in thread block, correct order of reads and writes is required to prevent race conditions.

CUDA provides several synchronization primitives for thread block memory access coordination, but most commonly used is simple barrier `__syncthreads()`. All threads in a block will stop at that call and will not continue until all threads in thread block execute the `__syncthreads()` call. It is important to prevent calling the function from possibly divergent code – it is an undefined operation and can lead to a deadlock.

Memory Fences

The CUDA programming model assumes a device with a weakly-ordered memory model. It means that order of CUDA thread writes into shared or global memory is not necessarily the same order in which the data is observed written in other threads.

Memory fence functions like `__threadfence_block()`, or `__threadfence()` can be used to enforce some sort of ordering.

Atomics

As the name suggest the atomic operations are performed without interference from any other thread and thus preventing race conditions. Atomic operations in CUDA work on both shared and global memory, although shared memory atomics are much faster. This fact is often exploited by a hierarchical organization of computation when an intermediate result is computed first across the threads in thread block and then calling it on global memory only once per thread block.

Global Synchronization by Kernel Executions

There is no direct support for global synchronization primitives (synchronization of all thread blocks) in CUDA.

The usual approach to implement global barrier is to split the computation at the synchronization points, so the kernels can be stopped at these points and computation can procceed by executing another kernel.

Global Synchronization Primitives

The correctness of inter-block communication cannot be guaranteed unless a memory consistency model is assumed. This can be solved by using `__threadfence()`.

In paper Xiao and c. Feng (2010) authors presented two synchronization schemes. One using atomic operations to operate global lock and second lock-free version, which operates two arrays of the same size as the number of synchronized blocks and does not need expensive atomic operations.

The first locking scheme outperformed the lock-free version for a small number of blocks (up to 8), for synchronization of more thread blocks the lock-free algorithm scaled better. Moreover, with number of blocks increasing the synchronization time increases until the point when it is more efficient to use CPU for synchronization and stop the kernel in the synchronization point.

1.2 CUGIP - CUDA Generic Image Processing

CUDA itself provide only low-level API, and data structures designed only to control the computation.

Set of STL-like data structures and algorithms for developing high-performance parallel applications with minimal programming effort is provided by library *Thrust* (NVIDIA Corporation (2018b)).

Parallelized version of the Fast Fourier Transform is available in *CUFFT*, accelerated image and video processing are provided in *NVidia Performance Primitives – NPP*. All these libraries and lots of others are part of the CUDA toolkit. However, none of these libraries provides tools for developing new parallel algorithms for image processing, as most of the utilities wrap device code and provide only host interface. The only exception is the Thrust library, which provides meta-algorithms allowing a passing of the functors, which are then executed on a device. However, Thrust library is focused only on one-dimensional data structures, moreover, does not know image processing concepts like pixel/voxel neighborhood,.

From this comes the motivation in developing a new library for image processing using CUDA. The *CUDA Generic Image Processing* library (Kolomaznik (2016)) was designed with following goals in mind:

- Simple device $\leftrightarrow$ host transfers
- Minimal overhead in comparison to using plain CUDA
- Cover common algorithmic patterns - kernel execution, image arithmetics, reduce, scan. etc.
- Manage device memory allocations
- Hide the CUDA API complexity
- Error handling using exceptions – strong exception guarantees where possible (plain CUDA uses error codes)
- Resource Acquisition Is Initialization (RAII) wrappers

- Easy interoperability with other image processing libraries (successfully used with ITK and Numpy for example)

1.3 Concepts

A key aspect of the library design is memory management as the amount of GPU memory is usually much smaller then amount of system RAM.

CUDA provides memory swapping in form of the unified memory oversubscription, nevertheless the memory allocations and access patterns are a sensitive aspect of any GPU implemented algorithm.

Algorithm which should work for wide range of task sizes and device configuration should be aware of its memory requirements and if it is possible to do whole computation with memory available for allocation. If there is not enough memory, usually the tasks are split to small subtasks that can be successfully executed.

With that in mind, the library was designed, so that no allocations happen without user knowledge if possible. It means that if some meta-algorithm needs storage for temporary results, the user has an option to pass pre-allocated temporary buffer.

This requirement also enforces that data passed to algorithms are handled by non-owning data structures (views) and owners do not appear inside the library calls.

In CUGIP library most of the processed data are images of some sort. So the terminology is as follows – owning data structures are *images* and non-owning data structures, which provide access to image data are *image views*. The interface implementation is inspired by *Generic Image Library (GIL)* by Adobe Adobe (2009). It follows the basic C++ idiom *Resource Acquisition Is Inititalization* (Stroustrup (1994)) or *RAII*, binding the lifetime of the owning data structures (images) to the stack.

Basic primitive to access data is an image view. Iterators like the ones from STL or more image oriented iterators from *Insight Toolkit* – ITK (Kitware (2016)) are not used because the usual semantics of iterator concept provides sequential access (iteration over range specified by begin and end). Iterators simply do not map well to multi-threaded processing. For a small number of threads the range can be divided into subranges and in each the processing switches back to the sequential processing, as can be seen for n-dimensional ranges in the Intel library *Threading Building Blocks* (Intel (2016)).

All the reasons for these design decisions will be discussed in following sections in more depth.

1.3.1 Images

The image is a container which owns a memory block containing pixel/voxel data. It allocates data in a constructor and deallocates in a destructor. Image data are accessed by provided image views - `const_view(image)` for read-only access and `view(image)` for read/write access.

There are several types of images based on the type of the memory it allocates. All image types are created in the host code (CPU code). Because images are data owning data structures, it would be complicated to prevent unintentional copies to be generated as CUDA uses raw memory copying instead of calling copy constructors.

In general, CUDA kernels and device code can only access images through provided device accessible image views (device code cannot access host memory).

Host Image

For data allocated in host memory, the generic `cugip::host_image` is provided. It is a templated by type of an image element (pixel/voxel) and dimension of the image (2D and 3D images are currently supported). Since data are allocated in the host memory (RAM), they can be easily accessed by the provided image views from host code. The internal memory buffer can be accessed directly by provide pointer, but users must be aware that image owns the data, so the pointer is invalidated in the destructor and possibly during image resize.

Device Image

Similar to HostImage, but the internal data buffer is allocated as a linear device (GPU) memory. Implemented by generic `cugip::device_image`. Because the data live on GPU, it is accessible from device code by provided image views.

Texture Image

Similar to the `cugip::device_image`, it contains data accessible from device code, but data are allocated in texture memory and accessed through bindless texture interface (added in CUDA 5.0). Texture memory has different caching properties as compared to linear memory, moreover, provides additional utilities, such as normalized floating point access, element interpolation, and mipmapping. These features are directly supported by the hardware and thus faster than a manual implementation.

Variant Image

As the CUGIP library is written in C++, which is a strongly typed language, and demand on low overhead led to an elimination of the virtual methods in the CUGIP interfaces. There is often need to use multiple images of different types, which are not mutually compatible. If the images are not needed at the same moment, there would be lots of allocations and deallocations when switching between these image types if there is not enough memory. To prevent this a meta image called `cugip::variant_image` is created, which uses a single internal buffer and provides views of different types (specified at a compile time). This overcomes the problem of the repeated allocation and deallocation and still provides compile time safety (contrary to using only void pointer based interfaces).

1.3.2 Views

CUGIP is designed in a way that all accesses to image data are done through image views. Image views are lightweight memcopyable non-owning containers. They are used in a similar manner as iterators in STL. Since iterators themselves are an not ideal solution in a parallel environment, data views were chosen in CUGIP implementation as the most suitable replacement.

Image view concept is bound to a notion of a zero based n-dimensional interval. For each point from the n-dimensional interval an access to image element (whether read or read/write depends on the type of view) is provided.

Host Image View

Host image views are views returned either by host image instances or by generating subview from already available host view.

Device Image View

Similar to host image views - created either by device image or by getting subview of already created device image view.

Both host and device image views are referenced as memory based views as they both access element through pointers in linear memory.

Memory Views

Essential concept are memory views. These objects provide access to values stored in host or device memory. Passing data between host and device code, and other APIs requiring pointer access to data (CUFFT – NVIDIA Corporation (2018c)) can happen only through memory views, which access the data through a base pointer, size and stride attributes.

Another important feature of memory based views is that generating slice view or subview of the memory based view does not create a wrapper class around the original view, but recalculates base pointer, size and strides to give access to a correct subregion of the original view and creates a new memory-based view with these attributes.

Procedural Views

Procedural views are quite different from memory based views as they do not provide a direct memory access. Instead, they either generate the value based on their arguments and provided an index or do some on-demand operation on a wrapped view of a different type. These views are lazily evaluated – returned value is not computed until the position is accessed.

In lots of situations, the same result can be achieved by using meta-algorithms to compute some output data. However, if an algorithm has several steps, it would mean several kernel executions. By using lazy views, all required operations can be bundled in one procedural view and provide the result in single kernel execution, thus lowering the overhead.

A model situation when this approach pays off is image algebra.

1.4 Lazy evaluation

When a code is optimized for speed one of the first steps is an elimination of the slowest operation. Slowest operation depends on the context. Years back was one of the optimization replacement of multiplications by additions. This kind of optimizations is nowadays done mostly by compilers if necessary (additions and multiplications are instructions of the same length in modern processors), but the principle stays – expensive mathematical functions are replaced by fetching values from precomputed tables or using approximations of sufficient precision.

In CUDA world one of the slowest operations is kernel execution. For complex kernels, the time needed for their execution is often irrelevant. However, if the kernel is lightweight – for example some simple arithmetic operation on input data – its execution can be expensive. So it is a good idea to bundle these simple kernels into one and execute it only once or provide the value on demand.

1.4.1 Expression Templates

Expression templates are C++ technique, which uses principles of template metaprogramming to wrap computations into data structures representing those computations. This happens in compile time. Runtime effect of this approach is that the expression is evaluated on demand (lazy evaluation), or more efficient evaluation approach may be chosen.

Main reasons for expression templates usage are:

- Lazy evaluation – postponing computation to be executed later in the program.
- To create domain-specific embedded language (DSEL)
- To pass an expression (not the result of the expression as a function parameter.

Domain-specific embedded languages usually comes with heavy operator overloading. A good example is parser generator Spirit (de Guzman and Kaiser (2016)), which is one of the Boost libraries. By using Spirit library, a parser can be defined directly in EBNF grammar (little modified to support C++ syntax rules). Moreover, because the grammar is written directly in C++, the parser is generated by the compiler, which can report lots of possible problems and do heavy optimizations, because it "sees" the whole grammar.

Another common application of expression templates is in linear algebra libraries. Operators for addition, subtraction, multiplication by a scalar value, etc. defined on vectors and/or matrices must go through all components of a vector/-matrix and apply the operation on the scalar component. For example, in the expression, each operator will cause a loop over vector components and generates a new temporary value.

$$\begin{aligned} \vec{x} = \vec{a} + k(\vec{b} - \vec{c}) \quad \rightarrow \quad & t_0 = (b_0 - c_0), \ldots, t_n = (b_n - c_n) && (1.1) \\ & u_0 = (kt_0), \ldots, u_n = (kt_n) && (1.2) \\ & x_0 = (a_0 + u_0), \ldots, x_n = (a_n + u_n) && (1.3) \end{aligned}$$

If all the operators are represented by expression template, the looping over components can be done only once at the assignment and the need for temporary vectors disappears. Also, the code can exploit more easily some hardware features like vector instructions (SSE, AVX, etc.). Code with expression templates will do this:

$$\vec{x} = \vec{a} + k(\vec{b} - \vec{c}) \rightarrow x_0 = (a_0 + k(b_0 - c_0)), \ldots, (a_n + k(b_n - c_n)) \tag{1.4}$$

These techniques were used to implement lazily evaluated operations on image views. Simple arithemtics, masking, basic coordinate transformations (mirroring, rotation in 90 degree steps) are very cheap operations on GPU, so the bottleneck is in kernel executions. Evaluating these operations lazily leads to significant speedups as multiple kernel executions are wrapped into one.

1.5 Meta-algorithms

Several STL-like meta algorithms are provided, hiding kernel (or several kernels) execution for common computation patterns. This is similar to what Thrust (NVIDIA Corporation (2018b)) does for 1D data structures.

1.5.1 Copy

`cugip::copy` and `cugip::copy_async` meta-algorithms generally serve for copying data between views of the same size, but there are some restrictions: - When copying between device and host based views, only memory based views can be used. - Procedural and lazily evaluated views can only be copied only in device-device or host-host transactions.

1.5.2 For Each

One of the principal operations on an image is to apply a function on all image elements from provided image view. It has similar semantics as `std::for_each()` from STL. Example can be seen in listing 1.1.

The basic version takes unary callable (function, functor) and applies it to all elements and stores the result in-place. Although the variant without an update (procedure instead of a function) is often used in STL for gathering information about the values from processed iterator range, it is not an efficient approach in CUDA (and parallel processing in general) because it would require intensive synchronization of the threads when updating internal data of the callable. Better approach is to use parallel reduce for this kind of task, which handles synchronization in a hierarchical manner and is thus usually far more efficient.

Another variant of the `for_each()` algorithm takes binary callable, which operates on an element value and its index in the processed image view, making it possible to return value dependent on a position in the image.

As other meta-algorithms, it can be executed on specified stream and its behavior in the sense of multithreading (block and grid sizes) can be modified by providing policy class.

The only variant of the `for_each()` algorithm, which gives access to the neighborhood of image element through locators, works only for read-only functors. The reason is the possible race conditions coming from the concurrent access to the image elements and storing the results in-place. There would be no certainty whether the value returned by locator for some non-zero offset would return original value or newly computed value.

Listing 1.1: Example of for_each and device lambda function usage.

```
device_image<int, 3> deviceImage(256, 256, 256);
// fill deviceImage ...

// apply lambda function on each voxel
// and increase its value by one
for_each(
        view(deviceImage),
        [] __device__(int arg) {
                return arg + 1;
        });
```

1.5.3 Transform

As `for_each()` does in-place processing of the input image view, the `transform()` algorithm and its variants process the input view (or views) and stores the result in an output view. The basic variants works similarly to `for_each()` – it takes unary callable and result of each execution stores as output, but now it can be of different type to input element.

Listing 1.2: Example of transform usage.

```
device_image<int, 3> deviceImage(256, 256, 256);
device_image<float, 3> resultImage(256, 256, 256);
// fill deviceImage ...

// apply lambda function on each voxel
// and store the result in an output image
transform(
        const_view(deviceImage),
        view(resultImage),
        [] __device__(int arg) {
                return arg * 3.141592f;
        });
```

Locator Access and Shared Memory

If the input and output views are different and non-overlapping, it means that values from input view will not be changed during the computation, so it is safe to compute a new value based on the values of the other image elements (typically from the neighborhood). For this case, there is a variant of transform algorithm

called `transform_locator()`, which takes a callable operating on the locator anchored in the processed element (instead of its value).

Without further knowledge about access patterns, the most generic version of the algorithm cannot optimize the global memory accesses. If the memory access pattern is known and especially if the elements accessed through the image locators are from the bounded neighborhood of the anchoring element and each element would be loaded multiple times during the computation, then the region accessed by threads from the thread block can be preloaded into shared memory. The callable object passed to the `transform_locator()` algorithm will then obtain locator which provides access for this preloaded piece of memory instead of access to global memory.

Basic image processing algorithms like convolutions, median denoising, morphological operations, etc. can be implemented using `transform_locator()` meta-algorithm.

1.5.4 Reduce

Reduce (fold, accumulate, etc.) is an operation, which recursively combines values into a single result. A typical example is a reduction with addition operator, which is equivalent to a sum of the values from processed range.

Parallel reductions require the operator to be associative. Otherwise, it would possibly return different results for different schedulings.

The implementation follows optimization steps from Harris (2007) to achieve maximum possible throughput. Slight generalization is needed, because the presented optimized version works on 1D ranges and CUGIP implementation requires image view of arbitrary dimension to be supported.

1.5.5 Scan

Together with reduce comes in hand another operation called *scan*. Its special case is a prefix sum. The scan operation returns reduction of all items up to the computed one and does it for all elements in the processed range.

Application on image views allows other slight extensions. The scan operation can be applied to the sequence of all image view elements, or separately on slices/scan-lines. The latter approach is useful for computation of integral images.

1.6 Code Example

Listing 1.3: Example of CUGIP code. Computation of the gradient magnitude approximation. Data passed through the pointers and image sizes.

```
void gradientMagnitude(float *aInput, float *aOutput,
    size_t aWidth, size_t aHeight, size_t aDepth, bool
    aNormalize)
{
  using namespace cugip;
  // Wrap the pointers for input/output buffers
  // into host image views.
```

```
  // No strides available  suppose contiguous storage.
  auto inView = makeConstHostImageView(aInput, vect3i_t(
     aWidth, aHeight, aDepth));
  auto outView = makeHostImageView(aOutput, vect3i_t(
     aWidth, aHeight, aDepth));

  // allocate image for input and output
  // on the device
  device_image<float, 3> inImage(inView.dimensions());
  device_image<float, 3> outImage(inView.dimensions());

  // copy the data from host input buffer
  // into its device counterpart
  copy(inView, view(inImage));

  // compute the sobel gradient magnitude
  // approximation (in 3 dimensions)
  transform_locator(
    const_view(inImage),
    view(outImage),
    sobel_gradient_magnitude<3>());

  // if desired, scale all the values
  // into the [0, 1] interval
  if (aNormalize) {
    // find the maximal value by
    // parallel reduction
    auto maxVal = max(const_view(outImage));
    // multiply all the values
    for_each(
      view(outImage),
      multiply_by_factor(1.0f / maxVal));
  }
  // store to host the results
  // computed on device
  copy(view(outImage), outView);
}
```

1.7 Conclusions and Future Work

Abilities of the CUDA devices improve substantialy with each generation, and so is the CUDA framework. The focus of future work on CUGIP library lies in incorporating of the newly available features into existing datastructures and algorithms. Biggest changes would be required by implementing direct support for the *dynamic parallelism* and kernel side memory allocations. These two concepts woul allow afficient hierarchical or adaptive algorithms, such as multiresolution analysis, or algorithms working on spatialy compressed images, to be executed

inside single top-level kernel.

When it was possible the interfaces were designed in such way that if some information was known at the compile time it is possible to pass this information in compile time. It allows early detection of errors, compiler can better optimize (inline, unroll loops), also special cases can be easily identified and more efficient implementation can be provided – different block sizes for different spatial filter kernel sizes, more optimal memory reads for different type sizes.

Benchmarking showed that the meta-algorithms and data-structures are not fully optimal in several cases. So the future developement will also focus on eliminating these suboptimal solutions for these low level algoririthms, which would make also the highlevel algoritms more efficient without modification.

The CUGIP library proved to be useful tool with a potential of futher improvements.

Chapter 2

Voxel Data Preprocessing

Preprocessing data for interactive workflow plays an important role. It is also an opportunity for more expensive operations to be done without disturbing smooth user interaction. Data are transformed into formats needed by later steps usually without user involvement and flaws present in the input data (noise, artifacts) are also handled during the preprocessing phase.

Time expensive preprocessing operations, which cannot be part of the interactive workflow are either direct part of the acquisition phase, done offline in data storage or hidden behind the data loading procedure.

A common property of a large number of scanning methods is a high amount of noise. The quality of the image is the trade-off between an amount of noise and data resolution, with other factors also influencing the result – X-ray dose for living subjects in computer tomography, intensity of the electron beam in cryo-electron tomography, etc.

In this chapter implementations of two algorithms dealing with noise and low contrast data will be discussed. First is *non-local means (NLM) denoising* algorithm (Buades et al. (2004), Buades et al. (2005)), which is shown one of the best denoising methods available. NLM produces high-quality results, but it is quite time-consuming, so tuned version for GPUs will be presented and important factors influencing the speed will be discussed.

The second algorithm which will be shown is an efficient GPU implementation of *coherence enhancing diffusion*, which is used to improve the quality of flow-like structures (lines, ridges, etc.) in the image. This method will be used to improve outputs of gradient magnitude and similar operators needed by watershed transformation in chapter 4.2.

2.1 Denoising

A wide range of denoising algorithms can be found in the literature. There is always a tradeoff between how much noise is actually removed from an image and how well are details and minor features preserved during the denoising process.

Most of the image denoising algorithms fall into one of the following categories:

- Gaussian based linear filtering
- Non-linear neighborhood filters (median filtering)

- Anisotropic diffusions (Perona and Malik (1990))
- Regularization methods based on minimizing total variation of the image (Rudin et al. (1992), GPU implementation Pock et al. (2008)).
- Translation invariant wavelet thresholding (Coifman and Donoho (1995))
- The non-local means algorithm (Buades et al. (2005))

In Buades et al. (2004) and Buades et al. (2005) authors did a comparison of different algorithms. They had selected algorithms for each category of the denoising algorithms and compared how those different approaches affect various image features.

They showed that highest quality results are returned by non-local means filter as it outperforms other state-of-the-art denoising methods. Difference image for non-local means algorithm tends to look like white noise image, so it does not systematically corrupt important features like edges and ridges, which are often present in difference images for other smoothing filters.

Less computationally expensive algorithms can be used for image denoising instead of the NLM especially in context of interactive processing. But as will be shown, NLM denoising can be parallelized and optimized for execution on GPU very efficiently. This approach brings the computation burden to manageable levels. Comparison of such solution with parallel CPU implementation was done in Horacek et al. (2011).

2.1.1 Non-local Means

Non-local means filter (Buades et al. (2004)) computes restored intensity value $NL(u)(x_i)$ of image element on position x_i as weighted average of all image elements.

$$NL(u)(x_i) = \sum_{x_j \in \Omega^3} w(x_i, x_j) u(x_j) \tag{2.1}$$

where u is an image and $u(x_i)$ is the intesity of the element x_i and $w(x_i, x_j)$ is weight assigned to element x_j, when restoring element x_i. Weight $w(x_i, x_j) \in [0, 1]$ and $\sum^{x_j \in \Omega^3} w(x_i, x_j) = 1$. The weight should reflect similarity between neigborhoods N_i and N_j.

In the original paper (Buades et al. (2004)) the Gaussian-weighted Eucleidan $|.|^2_{2,a}$ distance was used to measure similarity between two neighborhoods N_i and N_j. This distance is classical L_2 norm where the respective elements from neighborhoods are weighted by Gaussian with standard deviation a. Thus making the difference between central elements more important than the difference between boundary elements. Given this distance, $w(x_i, x_j)$ will be computed as follows:

$$w(x_i, x_j) = \frac{1}{Z_i} e^{\frac{\|\mathbf{u}(N_i) - \mathbf{u}(N_j)\|^2_{2,a}}{h^2}} \tag{2.2}$$

where Z_i is normalization constant ensuring that $\sum^j w(x_i, x_j) = 1$, and h is a smoothing parametr controlling the decay of the exponential function.

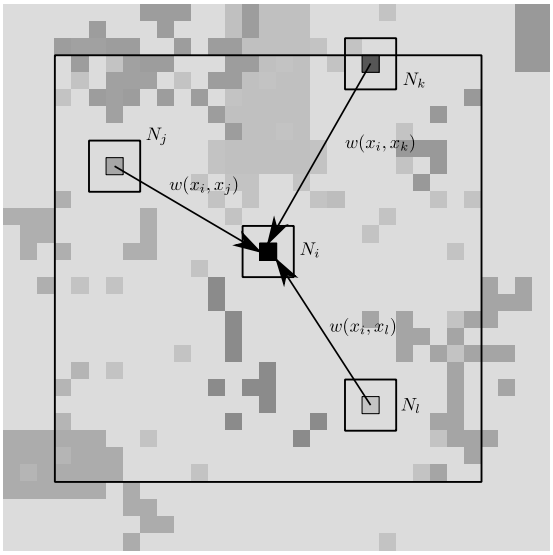


Figure 2.1: Non-local means – illustration of search space and voxel neighborhoods

Computing weights over the whole image domain is extremely computationally expensive, so in practice those weights are computed only for a limited region (*search region* – figure 2.1) around the reconstructed image element.

The important step when using the non-local means filter is the choice of the smoothing parameter h. The correct value depends on the standard deviation of the noise σ. There is also dependence on the size of the neighborhood $\|N_i\|$ (equation 2.2). According to Coupé et al. (2008), the automatic tuning of the parameter h can be done by estimating the noise standard deviation σ. In the case of an additive Gaussian noise, the standard deviation can be estimated via pseudo residuals ϵ_i.

$$\epsilon_i = \sqrt{\frac{6}{7}} \left(u(x_i) - \frac{1}{6} \sum_{x_j \in P_i} u(x_j) \right) \tag{2.3}$$

where P_i being the six-neighborhood at voxel x_i. The constant $\sqrt{\frac{6}{7}}$ is used to ensure that $\mathbb{E}[\epsilon_i^2] = \hat{\sigma}^2$.

$$\hat{\sigma}^2 = \frac{1}{|\Omega^3|} \sum_{i \in \Omega^3} \epsilon_i^2 \tag{2.4}$$

To make the distance independent from neighborhood size it should be normalized by the factor $\|N_i\|$:

$$\frac{1}{|N_i|} \|\mathbf{u}(N_i) - \mathbf{u}(N_j)\|_2^2 = \frac{1}{|N_i|} \sum_{p=1}^{|N_i|} \left(u^{(p)}(N_i) - u^{(p)}(N_j) \right)^2 \tag{2.5}$$

Final version of the weight equation is obtained by combining $\hat{\sigma}^2$ and the normalized neighborhood distance:

$$w(x_i, x_j) = \frac{1}{Z_i} e^{\frac{\|\mathbf{u}(N_i) - \mathbf{u}(N_j)\|_2^2}{2\beta\hat{\sigma}^2 |N_i|}} \tag{2.6}$$

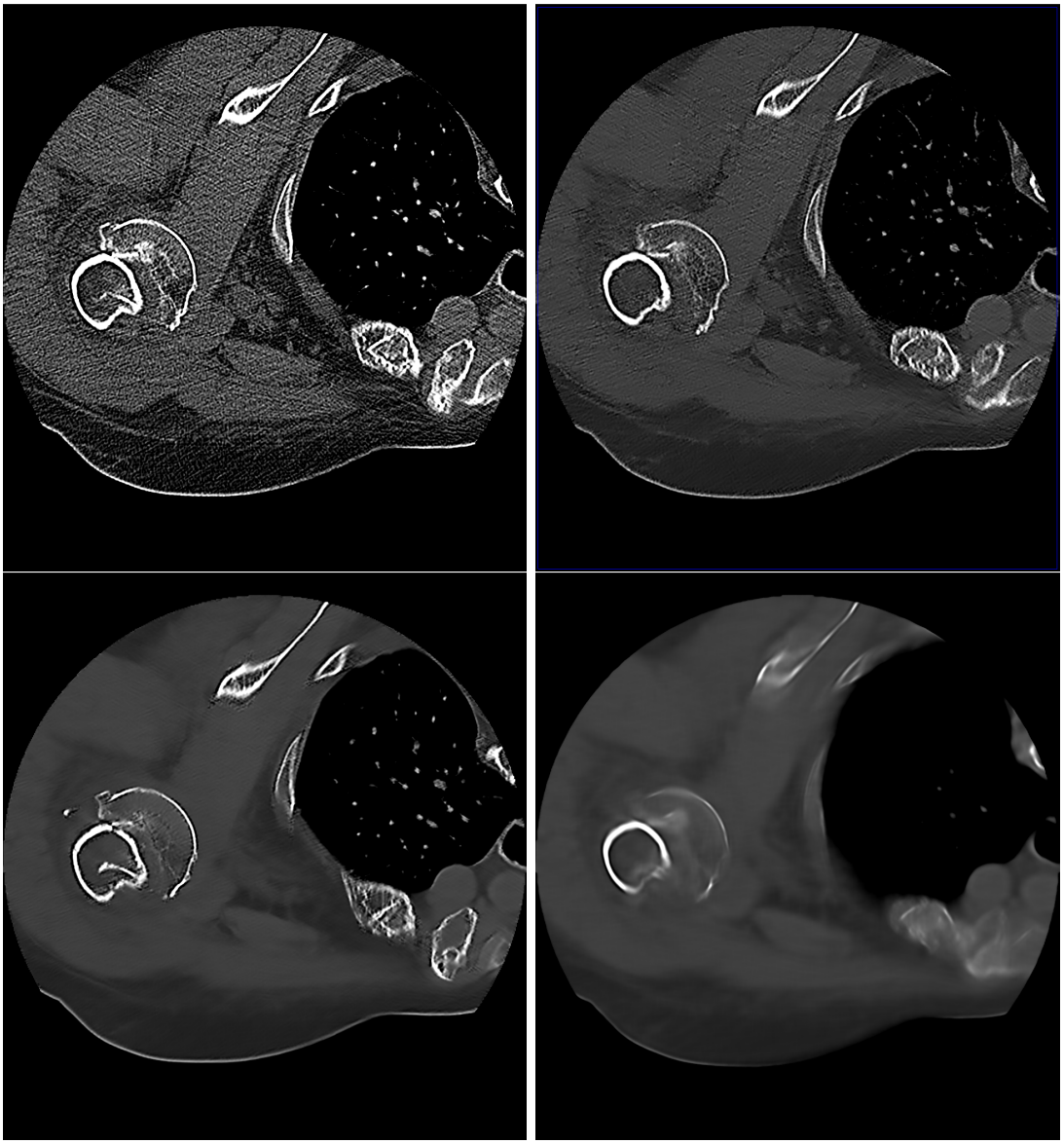


Figure 2.2: Non-local means outputs with increasing smoothness parameter β.

Only constant β to be adjusted manually. For Gaussian noise and correct estimation $\hat{\sigma}^2$ it should be close to 1.

Algorithm 1: Non-local means - host code

Data: Noisy image
Result: Denoised image
Compute pseudo-residuals e_i;
Compute variance estimation $\hat{\sigma}^2$ from pseudo-residuals;
Execute NLM kernel;

2.1.2 Threading Schemes

The basic decision to be made is how the work is divided between threads of computation on CUDA device. The two direct approaches are either that each thread computes the new value of the voxel by processing the patches influencing the voxel, or each patch has its thread which updates all the affected voxels.

The second scheme is not practical because it must handle the possible race conditions during voxel updates (more threads update single voxel) – this can be handled by using atomic operations or parallel reduction to gather the final value. However, these are all expensive operations on GPU. Thus the first threading scheme seems to be the best option.

Algorithm 2: Non-local means kernel (device code)

```
Data: Noisy image input;
Result: Denoised image output;
buffer ← Shared memory buffer;
index ← map thread and block index to buffer coordinates;
globalIndex ← map thread and block index to image coordinates;
// Compute the preloaded region position and size from block
   index and sum of search and patch radius
region ← ComputeRegion(blockIdx, p_radius, s_radius);
buffer.LoadData(input, region, index);
__syncthreads();
acc = 0;
value = 0;
if mapped output index is valid image index then
    foreach index2 in radius_s from index do
        weight ← PatchWeight(buffer, index, index2, p_radius);
        acc += weight;
        value += weight · buffer[index];
        output[globalIndex] = value / acc;
    end
end
```

2.1.3 Optimizations

When computing the new value of an element the algorithm needs to access a large number of other image elements. It means that the CUDA kernel will have to do a large number of global memory reads. That is a problem because accessing global memory from a kernel is slow operation even if it is coalesced access. As a result, naive implementation may not bring any speedup in comparison to CPU version (it can be even slower) as described in Horacek et al. (2011).

Shared Memory Preloading

Important property of these global memory accesses is that each element is read multiple times, so it means that preloading the values from search region into shared memory can be used to prevent multiple global memory reads of a single element.

The amount of shared memory available limits the size of the search region. This limitation can be alleviated by dividing the search space into disjoint regions and executing the kernel multiple times, processing different part of the search space in each run. However, as discussed in Horacek et al. (2011), Horacek et al. (2015) and Coupé et al. (2008) the search space of radius 4 and pach radius 2 is adequate for CT and MRI medical imaging and most probably for other

applications. So the rest of this chapter will be focused on executing NLM with 4-2 configuration as the most extreme case, which would require shared memory for $(w + 2 \cdot (4 + 2))(h + 2 \cdot (4 + 2))(d + 2 \cdot (4 + 2))$ elements, where w, h, d are dimensions of the thread block.

Loop Unrolling

Profiling the kernel shows that lots of the time is spent in integer arithmetics caused mainly by looping through the patches, which happens for each voxel in search space. The range of the practical patch radius is small, so it is not a problem to change it from the kernel runtime parameter into compile time parameter (template parameter). This fixes the loop count in each kernel instance for the patch processing, and the inner loop can be completely unrolled by using `#pragma unroll`. This removes the integer arithmetics instructions from the list of the "hot spots", which leaves the memory access related instructions the biggest bottleneck of the implementation as expected.

After the shared memory preloading, this is the optimization with the biggest impact on the actual speed as can be seen in figure 2.4.

Block Size Tuning

The optimal size of thread block used for denoising kernel execution depends on several factors. On one side smaller blocks allows higher occupancy of streaming multi-processors, but on the other hand, large overlaps of the regions preloaded in neighboring blocks penalize small blocks – the ratio between the number of preloaded voxels and computed voxels should be as small as possible. This ratio grows for smaller blocks making poor utilization of the multiprocessor resources (shared memory, registers). Tests showed that the optimal thread block size is close to $16 \times 8 \times 8$. The 16 threads in x-dimension forms a half warp, which is very good for memory accesses. Full warp would be even better, but the size of the shared memory and suboptimal ratio between loaded and outputed elements per block penalize that setting.

Preventing Bank Conflicts

By removing all previously found hot spots the last one remaining is number of shared memory bank conflicts. As described in previous section the two halves of the warp will each access memory banks without conflict, but there is bank overlap between those halves of the warp. It cannot be prevented by introducing a padding between rows in shared memory as it would dramaticaly increase size of needed shared memory.

Simple observation can be made – both warp halves will access the exactly same memory possition except for one row, but these same accesses are offseted by single iteration (figure 2.3). If the warps will both start on the same row and process the overlapping ones synchronously, then the bank conflicts can be handled by multicasts (only available for compute capability 3.0 or better).

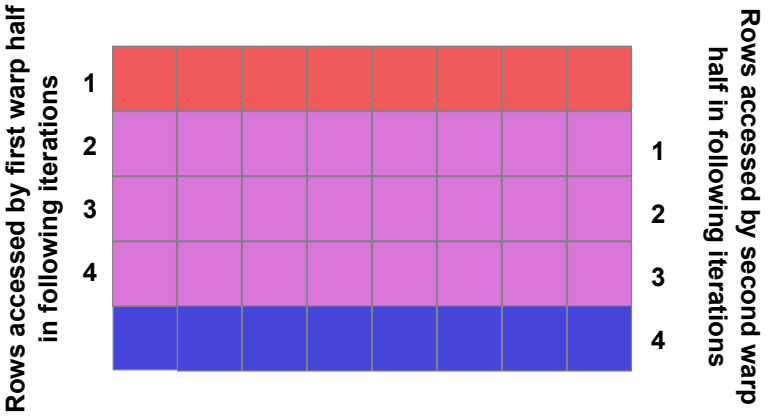


Figure 2.3: Scheme how the processing of the rows in each half of the warp is organized – the optimal approach would be to process purple rows synchronously and at the end do the red/blue rows, which are not shared.

Watchdog Timer

If some kernel is executed on graphics card which is not dedicated just for computation but is shared with a user interface (Windows, X Server), then the operating system uses watchdog timer with fixed time limit. If the kernel execution exceeds this time limit it is then terminated. This happens for the NLM denoising kernel when applied on bigger data.

There are several possible solutions. First are the obvious ones like installing dedicated GPU just for computation, but this limits the amount possible hardware configurations, which can run this implementation. On Windows system, the watchdog timer can be set to a bigger value, but that is problematic because it can cause the GUI to freeze during kernel execution. On Linux, the X Window server can be stopped and then the computation can be executed from a terminal, but this prevents its usage in interactive workflows.

The solution which is most general is splitting the task into several subtasks. For denoising, it means multiple kernel executions, where each kernel process only portion of the image. It is a similar strategy to the situation when input data are too large to fit into memory together with output data buffer. However, in the case of NLM denoising, the time limitation is usually hit before the memory limit.

Dividing the computation into subtask also means that each kernel execution uses the only fraction of the number of thread blocks, and it can easily happen that the block count does not saturate all SM units on GPU. To overcome this, the kernels for the subtasks can be executed in different streams. Concurrent kernel execution is supported on devices with compute capability 2.0 or higher (NVIDIA Corporation (2018a)), so if one kernel does not utilize the whole GPU other kernels can (and will) run at the same time. This strategy is compatible with the data splitting required for large data processing.

Benchmarks

Computation times for differently sized images and different settings were measured. The comparison between multithreaded CPU implementation and GPU

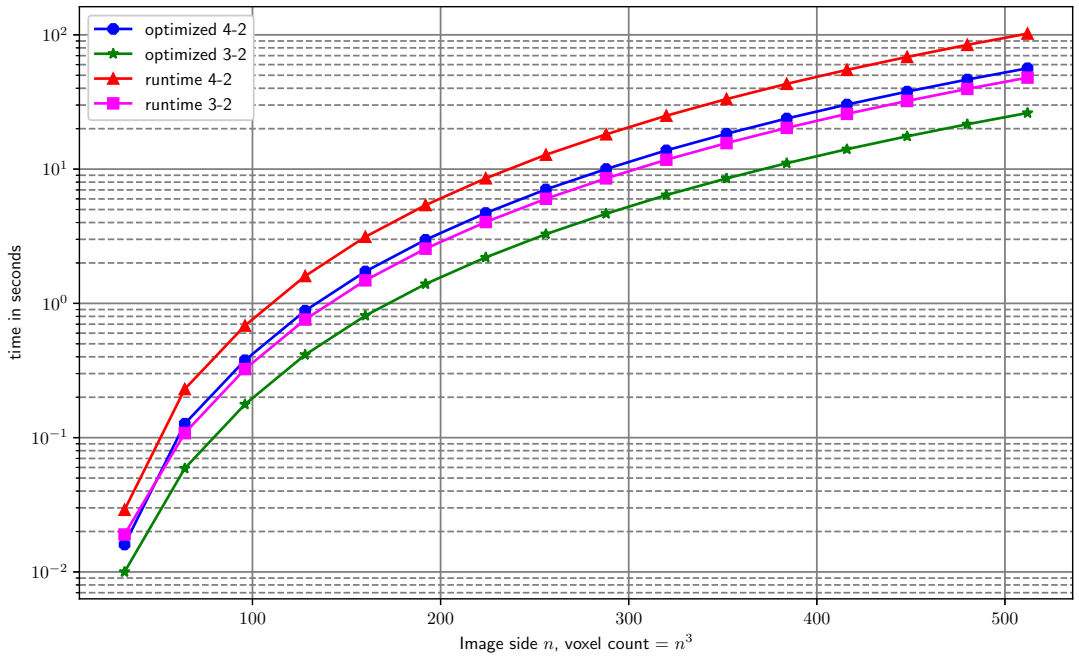


Figure 2.4: NLM denoising speed comparison for runtime specified parameters (patch and search space radii) and compile time specified parameters. Shown are results for search space radius equal to 3, 4 and patch radius equal to 2. Tested on NVIDIA GTX 980 4GB card.

implementation was presented in Horacek et al. (2011). As the NLM algorithm can effectively use the GPU resources the achieved speedup is close to two orders of magnitude.

As can be seen in figure 2.4 the final version of the algorithm produces results under a minute for sufficiently large image. These computation times are getting pretty close to actual loading times from hard drive.

2.2 Coherence Enhancing Diffusion

Contours of objects in an image can be detected by a large number of possible operators. The most common ones are based on gradient magnitude. The omnipresent noise, low contrast, and blur influence the quality of the computed contours.

Denoising of the input helps to improve the quality of the contours but does not help much in areas of low contrast, where the contours tend to be weak or even interrupted.

In Weickert (1999) was presented diffusion process which smooths lines and flow-like structures in the image. The process was then generalized for arbitrary dimension, so all coherent structures are smoothed.

In core of the method is computation of eigenvalues and eigenvectors of $n \times n$ matrices and that is done for each element of the processed image.

2.2.1 Diffusions

Mathematical description of diffusions is given by *Fick's laws*, where the first law relates the concentration to the flux and second law predicts how diffusion changes concentration over time. Basic version of both laws assumes homogeneous and isotropic medium and is parametrized by scalar *diffusion coefficient (diffusivity)*. Both laws can be generalized for *inhomogenous and anisotropic media*, so *diffusion tensor* dependent on position must be used instead of the diffusion coefficient.

The second law equation and boundary conditions for bounded function (image) $f : \Omega \to \mathbb{R}$:

$$\frac{\partial u(x,t)}{\partial t} = \nabla \cdot (D(x)\nabla u(x,t)) \quad \text{on} \quad \Omega \times (0,\infty) \tag{2.7}$$

$$u(x,0) = f(x) \quad \text{on} \quad \Omega \tag{2.8}$$

$$\langle \nabla \cdot (D(x)\nabla u(x,t)), n\rangle = 0 \quad \text{on} \quad \partial\Omega \times (0,\infty) \tag{2.9}$$

Where $D(x)$ is diffusion tensor describing the speed of diffusion in the anistropic environment. It must be smooth, symmetric and positive definite tensor.

The core of the coherence enhancing anisotropic diffusion lies in the way how is this diffusion tensor derived.

2.2.2 Structure Tensor

As one of the key structure descriptors is usually considered ∇u_σ the Gaussian smoothed gradient of an image u:

$$K_\sigma(x) = \frac{1}{(2\pi\sigma^2)^{m/2}} exp\left(-\frac{|x|^2}{2\sigma^2}\right) \tag{2.10}$$

$$u_\sigma(x,t) = (K_\sigma \star u(.,t))(x)(\sigma > 0) \tag{2.11}$$

Where the standard deviation σ denotes *noise scale*. The smoothing makes the operator ignorant to the smallest details, which are affected most by the noise.

Operator ∇u_σ works well as an edge detector but fails for enhancing parallel structures. It is caused by opposite directions of gradient vectors on opposite sides of the line and near parallel structures. These vectors then cancel out and changing σ does not help – smaller value does not remove fluctuations and bigger value erase important details. Gradient smoothing averages directions, not orientations.

To get descriptor which is invariant under the sign change Weickert (1999) used tensor product of gradient:

$$J_0(\nabla u_\sigma) = \nabla u_\sigma \otimes \nabla u_\sigma = \nabla u_\sigma^T \nabla u_\sigma \tag{2.12}$$

The matrix is symmetric and positive definite, and its eigenvectors are parallel and orthogonal to the gradient. The eigenvalues are $|\nabla u_\sigma|^2$ (for the parallel eigenvector) and 0 for the orthogonal ones.

Gaussian componentwise smoothing of the matrices averages orientations and not directions:

$$J_\rho(\nabla u_\sigma) = K_\rho * (\nabla u_\sigma \otimes \nabla u_\sigma) \qquad (\rho \geq 0) \tag{2.13}$$

J_ρ is called *structure tensor*. Smoothing does not affect that the matrix remains symmetric, positive and definite. So its eigenvalues are still in the domain of real numbers.

Eigenvalues can be ordered: $\mu_1 \geq \mu_2 \geq \ldots \mu_m$, and because J_ρ matrix is positive and definite, the eigenvalues are non-negative and also ordered by their magnitude.

Smoothing integrates the structure tensor over $O(\rho)$ neighborhood, and its eigenvalues represent the average contrast in the directions of its eigenvectors $w_1, \ldots, w_m$ over the region.

The eigenvector w_m corresponding to the smallest eigenvalue μ_m represents orientation with the lowest fluctuations, so called *coherence orientation*.

To describe degree of anisotropy original author defined coherence measure:

$$\kappa = \sum_{i=1}^{m-1} \sum_{j=i+1}^{m} (\mu_i - \mu_j)^2 \tag{2.14}$$

It is close to zero for isotropic structures and grows with increasing difference between eigenvalues.

The final form of the diffusion tensor for coherence enhancing diffusion is defined by its eigenvalues and eigenvectors.

Eigenvectors are chosen to be exactly the same as eigenvectors of the $J_\rho(\nabla u_\sigma)$ tensor and eigenvalues are computed as follows:

$$\lambda_i = \alpha \tag{2.15}$$

for $i = 1, \ldots, m-1$, and

$$\lambda_m = \begin{cases} \alpha & \text{if } \kappa = 0 \\ \alpha + (1-\alpha) exp(\frac{-C}{\kappa}) & \text{else} \end{cases} \tag{2.16}$$

where C is threshold parameter. $C \ll \kappa$ leads to $\lambda_m \approx \alpha$, and $C \gg \kappa$ makes $\lambda_m \approx 1$. Parameter $\alpha \in (0, 1)$ serves as a regularization parameter, which ensures that the smoothness of the structure tensor is carried to the diffusion tensor. Also, it forces the diffusion never to stop – even for isotropic structures there would be still slight linear diffusion.

By sticking the diffusion tensor back to the equation 2.7 the coherence enhancing diffusion is formaly defined:

$$\frac{u(x,t)}{\partial t} = \nabla \cdot (D(J_\rho(\nabla u_\sigma(x)))\nabla u(x,t)) \tag{2.17}$$

2.2.3 Algorithm

All the described computational steps translates easily into the algorithmic representation by replacing derivatives by finite differences.

Algorithm 3: Coherence enhancing diffusion

Data: Image u
smoothing factors σ and ρ
time step Δt
number of iterations n
Result: Smoothed image
for *fixed number of steps* **do**
    compute gradient ∇u_σ;
    compute structural tensor $J_0(\nabla u_\sigma)$;
    compute blured structural tensor $J_\rho(\nabla u_\sigma)$;
    compute diffusion tensor $D(J_\rho(\nabla u_\sigma))$;
    apply diffusion step;
end

Although this scheme requires rather small Δt timesteps to remain stable during iterations. Weickert (1999) proposes different scheme for solving the diffusion equation which allows bigger timesteps. Although this different scheme requires several data transformations, which are not easily covered by currently available computation patterns in CUGIP library. Thus the current implementation will stick with the presented simple algorithm, which is still fast enough even with the small time steps.

2.2.4 Implementation

The actual implementation of the algorithm 3 is straight forward as all the steps map well to the CUGIP meta-algorithms like `for_each()` and `transform()` as described in section 1.5.

The most expensive operation of the algorithm is the diagonalization of the smoothed structure tensor. The structure tensor is real symmetric and positive semidefinite matrix, so it has real-valued eigenvalues. Diffusions of 2D images will need diagonalization of 2×2 structure tensors, which is easy (it can be solved by solving the quadratic characteristic equation). Little more complicated is the 3D case where the diagonalization of 3×3 symmetric matrices is required. Kopp (2008) did a comparison of several methods tuned for diagonalization of 3×3 Hermitian matrices.

2.3 Conclusions

It was shown that GPU can be used for fast computation of the non-local means denoising algorithm and diffusion processes. Direct naive implementation is not always possible and GPU specific optimizations had to be employed to maximize data throughput.

As was shown in Horacek et al. (2011) and figure 2.6 both presented methods map well onto GPUs and bring significant speedup in comparison to CPU implementations.

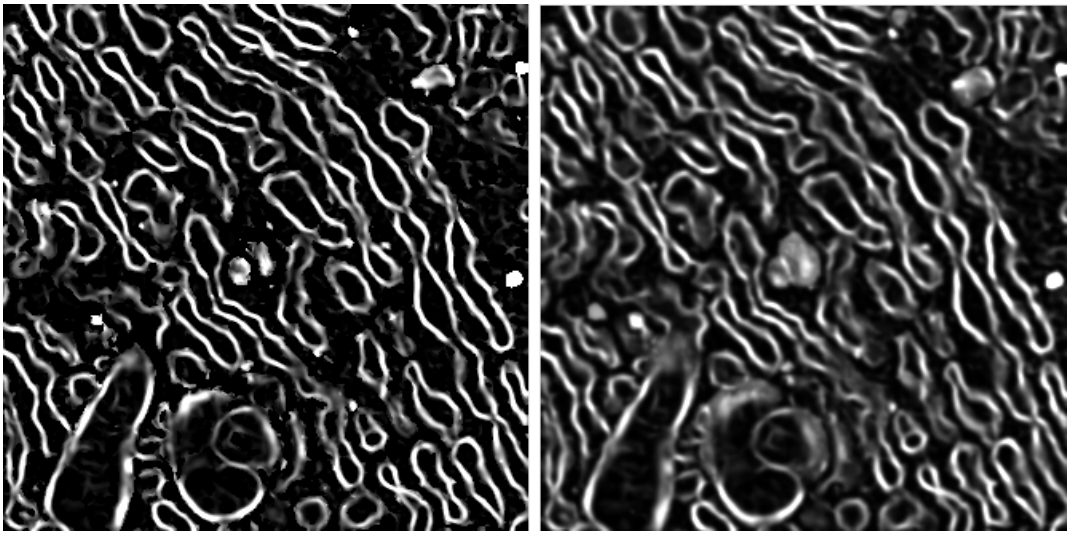


Figure 2.5: Example of smoothing coherent structures (masked endoplasmatic reticuli) in the microscopy image.

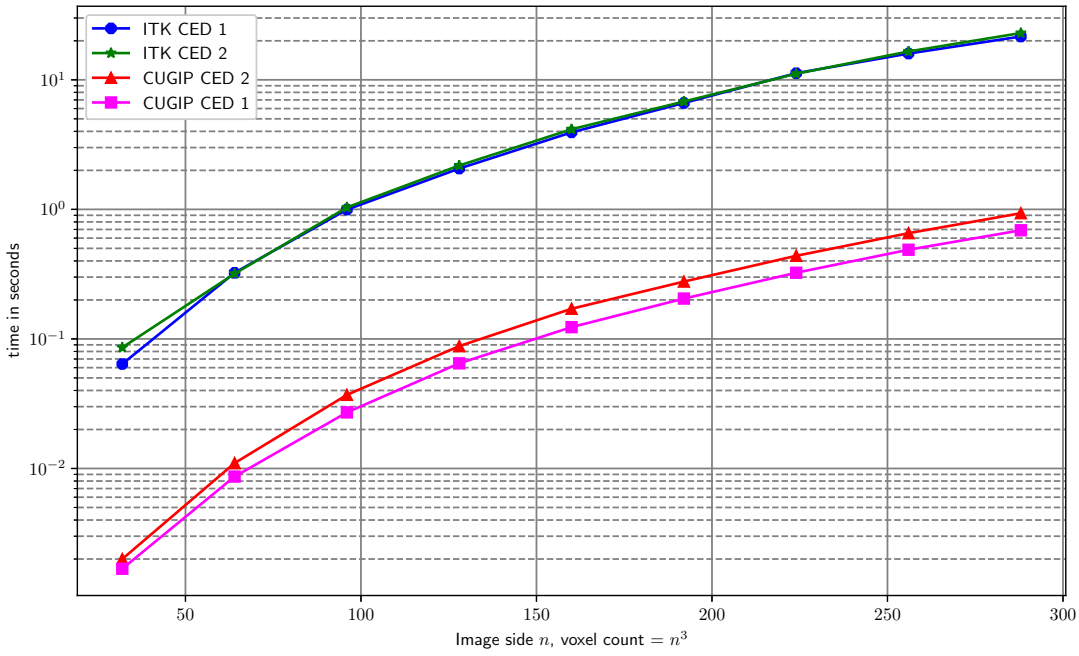


Figure 2.6: Comparison of ITK and CUGIP implementations of the coherence enhancing diffusion. Used two different settings of ρ parameters. Hardware used for measurement: NVIDIA GTX 1060 6GB, Intel i7-3770 CPU 3.40GHz 16GB, no-swap.

The accomplished computation times shows that it is possible to use these methods also in interactive image processing workflow, eventhough they are usually considered as too computationaly expensive.

Several further algorithmic optimizations can be found in literature (Coupé et al. (2008)), like block-wise computation, path-match and others, which should be also usable for GPU implementation and would push the computation time even lower.

Chapter 3

Transfer Functions

Modern GPUs and their computation power allow implementation of high-quality direct volume rendering methods (DVR) for real-time visualization.

A short overview of volume rendering methods will be given in this chapter, followed by a description of transfer function concept.

At the end of the chapter a novel extension to two-dimensional transfer functions will be introduced. The main idea is to bring some concepts normally used in offline segmentation methods and use them for interactive highlighting. The method which will be used is based on *objectness measure* (Frangi et al. (1998), Antiga (2007)) and allows shape analysis of image features in real-time.

3.1 Volume Data Visualization

A key part of interactive volume data processing is a visualization of the volume data itself. Showing input volume together with user defined data (masks, regions of interest, key points, etc.) and most importantly the results of processing influence highly the usability of the whole system.

Most common volume data visualization methods are based on 2D visualization when only single cut plane (slice) through the volume is shown.

Visualization systems usually support rendering of axis aligned cut planes know from medical imaging as *frontal, sagittal* and *transversal planes* (see figure 3.1). With some systems also allowing arbitrary positioning and orientation of the cut planes.

Slicing the volume has several advantages. It is fast approach both in the complexity of the actual implementation and in rendering times. Users are usually not limited by hardware demands for 2D rasterization. Another significant advantage is that users can easily look *inside* of the scanned objects without any occlusions.

Three-dimensional rendering of the volumetric data can be done by first extracting information from the image and constructing a geometrical representation of the image features. For the construction of meshes representing isosurfaces, several methods were published. For example *marching cubes* (Lorensen and Cline (1987)), *marching tetrahedra*, etc.

Generated mesh is then rendered by traditional methods for polygonal mesh visualization with usage of modern 3D accelerated graphics cards.

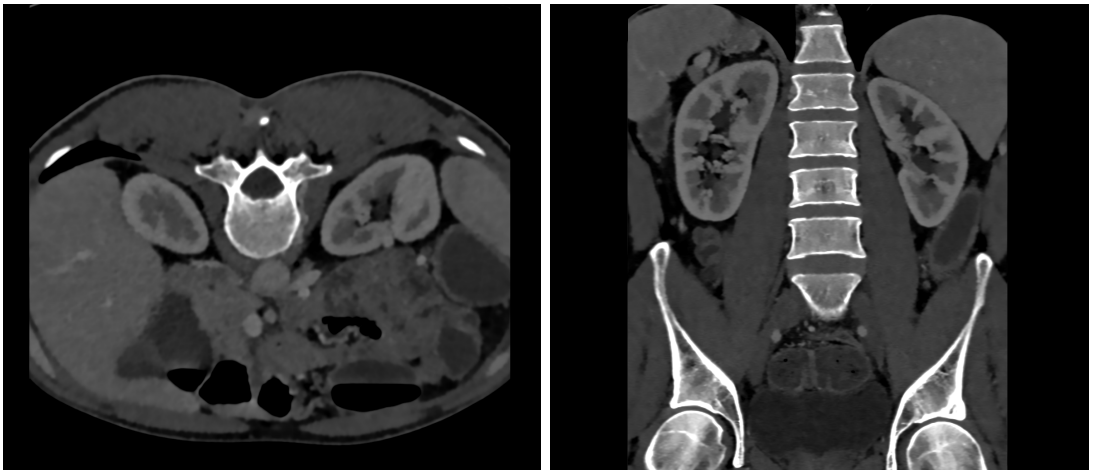


Figure 3.1: 2D rendering of CT transverse plane, frontal plane

Direct volume rendering (DVR) methods generate images of the 3D volumetric data without explicit extraction of any geometric primitives from the dataset. It uses optical model and assigns optical properties (color, opacity) to each voxel. During rendering these optical properties are accumulated according to the optical model to form final data view. Overview of real-time volume rendering methods can be found in the book Hadwiger et al. (2006).

3.2 Direct Volume Rendering

In the core of DVR methods is the *volume rendering integral* (Max (1995), Hadwiger et al. (2006)), which describes how the light is propagated through the medium. Most widely used variant is the emission-absorption optical model, which ignores light scattering and indirect illumination. It is a good compromise between generality and efficiency of computation.

$$I(D) = I_0 e^{-\int_{s_0}^{D} \kappa(t)dt} + \int_{s_0}^{D} q(s) e^{-\int_{s}^{D} \kappa(t)dt} ds \tag{3.1}$$

It describes how the light propagates through the volume from entry point s_0 to exit point D (camera position) along the ray (figure 3.2). q is source term describing emission at a certain point, I_0 is initial intensity at point s_0 (light emmitance of the background), κ is an absorption coefficient.

3.2.1 Discretization

Analytical evaluation of the volume rendering integral is usually not possible. Numerical methods are used for the computation of the approximate solution.

The basic approach is to split the integration interval into series of subintervals and estimate the integral in these subintervals. For each subinterval i color C_i and transparency T_i is computed, although transparency is usually replaced by opacity $\alpha_i = 1 - T_i$. These values are then combined into final value by a compositiong scheme.

Estimation of color and opacity in a segment is a key aspect to quality of the result. Basic approach is to assume constant value in the intervals, which

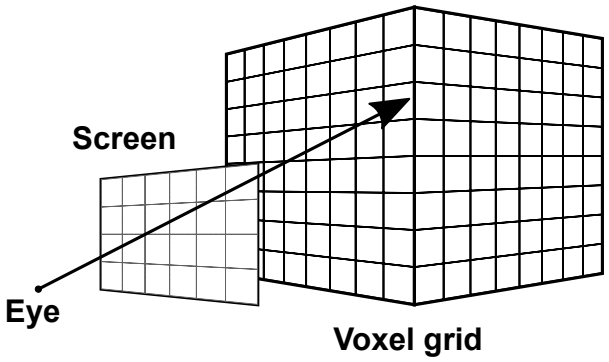


Figure 3.2: Ray casting scheme

also brings strong *aliasing* into the rendered image. More advanced methods for numerical integration (trapezoidal rule, higher order quadratures) can improve the quality together with methods designed directly for volume rendering like jittering of the ray start (replaces aliasing by noise), or pre-integrated transfer functions for incorporating higher frequencies from the color mapping scheme.

3.2.2 Compositing Schemes

Front-to-back Compositing

The front-to-back scheme is applied when the viewing rays are traversed in the direction from eye point to volume.

The iteration starts at the beginning of the volume and ends when the ray leaves the volume.

$$\begin{aligned} C_{dst} &\leftarrow C_{dst} + (1 - \alpha_{dst})C_{src} \\ \alpha_{dst} &\leftarrow \alpha_{dst} + (1 - \alpha_{dst})\alpha_{src} \end{aligned} \tag{3.2}$$

One of the useful features of this approach is possible *early termination*. If the accumulation process reaches a state when the overall opacity is close to 1 (traversed volume is almost opaque), then the rest of the ray can be ignored – it may bring a significant speedup, and the introduced error is negligible.

Back-to-front Compositing

By reversing the direction of ray traversal back-to-from scheme is obtained. The opacity accumulation is no longer needed, making the computation simpler, but in comparison to front-to-back scheme the early termination is not an option.

$$C_{dst} \leftarrow (1 - \alpha_{src})C_{dst} + C_{src} \tag{3.3}$$

Maximum Intensity Projection

A little special compositing scheme is maximum intensity projection (MIP) it does not accumulate color along the ray. It finds the maximal value instead. See figure 3.3a.

$$C_{dst} \leftarrow max(C_{dst}, C_{src}) \tag{3.4}$$

It is a simple and fast method. It is not suitable for studying low-intensity features, which can be suppressed by features of higher intensity. But small features of high intensity can be spotted more easily than in integral based compositioning.

The method ignores any space ordering along the ray which may cause difficulties with localization of the objects in the 3D space.

Isosurface Rendering

Another special compositing scheme can be defined for situations when volume isosurfaces must be rendered implicitly by ray traversal, instead of explicit construction in preprocessing step and rendering the surface directly.

The ray is traversed and changes in values are observed. If the searched isovalue falls into interval specified by the value in previous ray sample and current ray sample, it means that the ray crossed the isosurface between last two samples. Isosurface intersection position is then refined by interpolation or further sampling.

More samples from the neighborhood can be used to estimate image gradient. Moreover, because gradient is always perpendicular to the isosurface, it can be utilized as isosurface normal estimation, which is then used for color computation according to some shading model (see figure 3.3c).

If the isosurface is not rendered as completely opaque the front-to-back or back-to-front compositing schemes are used (based on ray traversal direction) for final color accumulation.

3.2.3 Renderer Implementation

Several approaches to GPU-based volume rendering exists. Abilities of each approach are usually closely related to the features (or lack of them) of the used hardware.

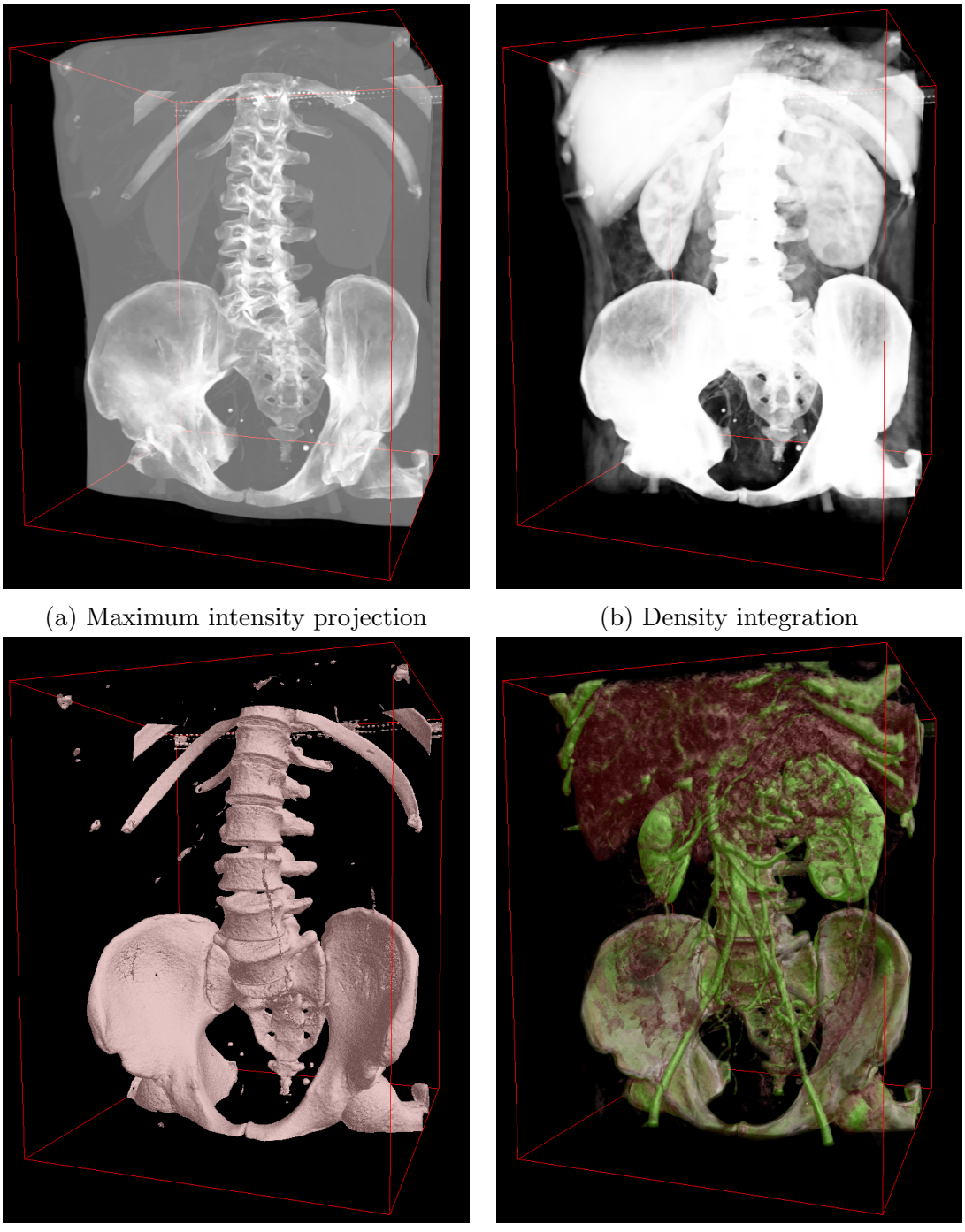
For example unsupported 3D texture objects limits the renderer, so it must store the volume in a set of 2D textures. That limits the rendering only to axis aligned object slices.

Availability of the 3D texture objects allows usage of the generic slices through the rendered volume. Rendering slices parallel to the image plane can simulate ray sampling. Proper color blending rule for the framebuffer can do *back-to-front compositing* (equation 3.3).

Newest graphic cards support advanced shaders and that allows implementation of the proper raycasting engine, where each *fragment shader* call does single-ray traversal and integrates the information along the ray into the final color.

3.3 Transfer Functions

In the visualization, the scalar image contains abstract values, which represent some spatially varying physical property. In general, there often is no way to derive optical properties from such data. To alleviate this issue a user defined classification/mapping scheme can be used to assign emission and absorption coefficients. Such mapping is called transfer function.



(a) Maximum intensity projection

(b) Density integration

(c) Isosurfaces

(d) 1D transfer function

Figure 3.3: Volume data rendering

The basic type of transfer functions is one-dimensional transfer function, which assigns each scalar value its color and opacity. The user can draw this function with help of value histogram to identify important image features as can be seen in figure 3.4.

Overview of commonly used transfer functions is given in Arens and Domik (2010).

On the other end of transfer function design complexity lie interactive systems (Guo et al. (2011); Guo and Yuan (2013)), which first do analysis and classifica-

tion of the image and then let user paint on the features of their liking and erase unimportant image structures.

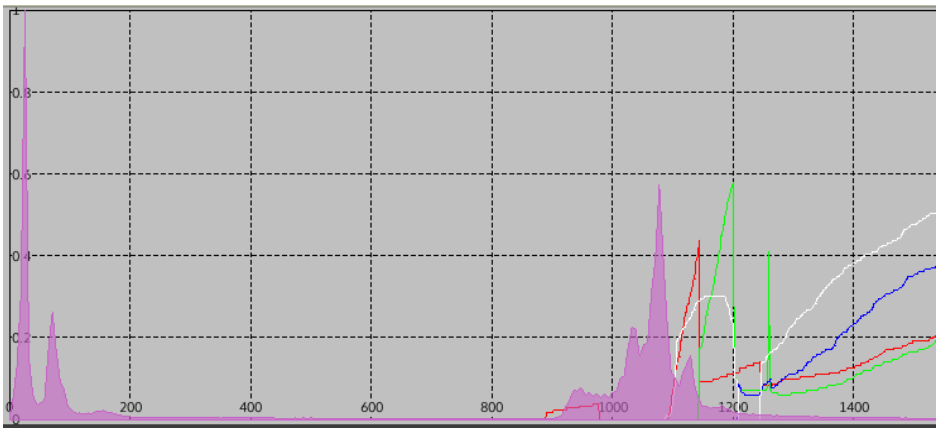


Figure 3.4: 1D transfer function design – density value histogram and color channels. Color curves are RGBA channels, which will be used to mix a color for each density value.

3.3.1 Transfer Functions of Higher Dimension

The concept of one-dimensional transfer functions can be extended into higher dimensions. The multi-dimensional transfer functions (MTF) will accept vectors instead of the scalar values.

There is a wide range of possibilities of how to obtain vector data required by multi-dimensional transfer functions. Information from two or more datasets can be fused by the MTF (CT+PET, dual source CTs), or the vector data can be computed by an analysis of the scalar data (Kniss et al. (2002)), commonly used approach would be to compute the gradient magnitude of the image as a second channel and apply 2D transfer functions.

This can allow differentiating contours (large gradient magnitude) from inner parts of the homogenous regions.

Example of how the 2D transfer function can be created by user can be seen in figure 3.5, where the scatterplot is used as a guidance how the gradient magnitude values are distributed in dependence on the scalar data.

From this can be seen that it is getting more complicated for users to design transfer functions for dimensions higher then two. In those cases, user usually works with some meta-attribute from which is then the multidimensional transfer function computed.

3.3.2 Size-Oriented Transfer Functions

Transfer functions which use just the intensity values at a point, and slightly more sophisticated 2D transfer functions using some simple local property like gradient magnitude, can be still quite limited. It may be hard to find anomalies in images containing structures of complex topology (vein system of the brain, cell organelles, etc.).

In the paper Correa and Ma (2008) authors designed transfer function, which can differentiate between objects of different size.

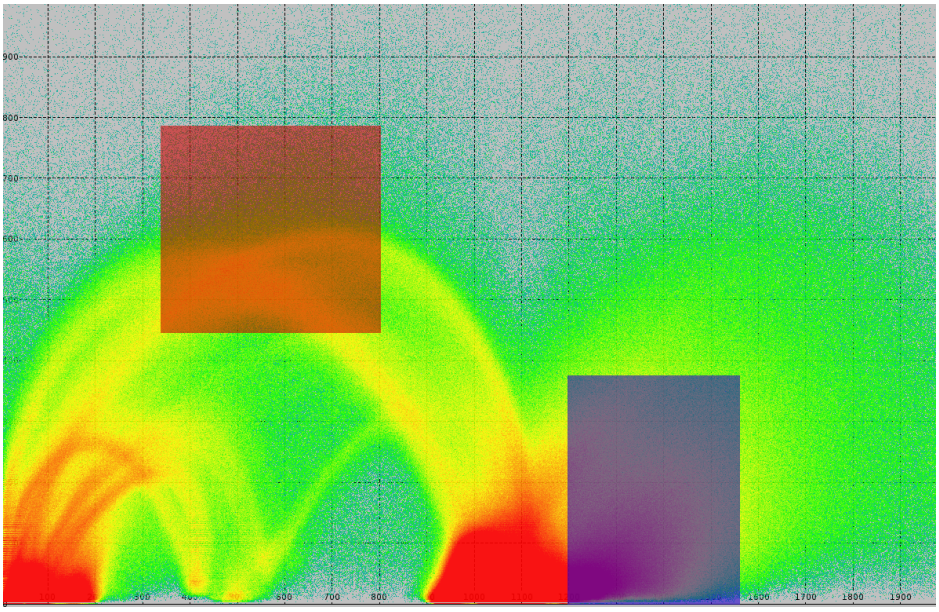


Figure 3.5: 2D transfer function design – scatterplot of intesity vs. gradient magnitude. Horizontal axis represent original scalar value, vertical is gradient magnitude. Coloring of the scatter plot reflects frequency of value/gradient magnitude combination in the image. The colored rectangles mark two 2D intervals in scatterplot – setting color and opacity for the value/gradient magnitude combinations inside those intervals.

They computed a *scale field* as a mapping $S : R^N \rightarrow R$, such that $S(\mathbf{x}) = \gamma$ represents scale of the local feature containing $\mathbf{x}$.

This scale field is computed in several steps. First, a multiresolution representation of the image is computed for given interval of "interesting scales" $[\gamma_{min}, \gamma_{max}]$. Authors computed the scale space iteratively by diffusion and they even experimented with nonlinear diffusions to generate scale levels.

Next step is construction of the *scale abstraction* of the image as a set of point-scale pairs $\{\mathbf{x}_i, \gamma_i\}$. It is done using a *blob detector*. Blob can be detected by searching for local maxima in both scale and space of normalized Laplacian:

$$\gamma \Delta L(\mathbf{x}) = \gamma(\sum_{i=1}^{n} \frac{\partial^2 f}{\partial x_i^2}) = \gamma tr(H(\mathbf{x})) \tag{3.5}$$

Where γ is the scale, $tr(H(\mathbf{x}))$ is a trace of Hessian matrix. *Matrix trace* is defined as sum of values on the square matrix diagonal.

The scale abstraction is then used to backproject the information into scale field. The problem is that blobs for different points can intersect and the scale information in those regions gets distorted. The scatter data interpolation is used to prevent this:

$$S(\mathbf{x}) = \sum_{\mathbf{x}_i \in N} \Theta(\|\mathbf{x} - \mathbf{x}_i\|)\gamma_i \tag{3.6}$$

where $\Theta(d)$ is the basis function like Gaussian kernel.

The scale field can be then employed in several ways. It can be either directly used for opacity modulation to hide features of undesired size. Or to highlight

features by size. In the original paper, this was applied to angiography images of the brain to highlight aneurism.

3.3.3 Size and Shape Oriented Transfer Functions

More information can be extracted from multiresolution representation of an image, not only size. Several filtering methods were published, which enhance structures of certain shape by analyzing the Hessian matrix.

The definition of *vesselness* measure was published in Frangi et al. (1998). It tries to enhance tubular structures in an image. First multiresolution representation of an image is created. Hessian matrix is computed for each point in each level of the multiresolution image. Eigenvalues of the Hessian matrix are then computed and sorted by their absolute value: $|\lambda_1| \leq |\lambda_2| \leq |\lambda_3|$.

The vesselness is computed under assumptions that for tubular structures following relations hold:

$$|\lambda_1| \approx 0 \tag{3.7}$$
$$|\lambda_2| \gg |\lambda_1| \tag{3.8}$$
$$|\lambda_3| \approx |\lambda_2| \tag{3.9}$$

Simply that the intensity is stable only in one direction. The following ratios will be composed into the vesselness equation.

$$R_A = \frac{|\lambda_2|}{|\lambda_3|} \tag{3.10}$$
$$R_B = \frac{|\lambda_1|}{|\lambda_2|\lambda_3|} \tag{3.11}$$
$$S = \sqrt{\sum_i \lambda_i^2} \tag{3.12}$$

The definition of vesselness for multiresolution scale σ:

$$V(\lambda)_\sigma = \begin{cases} (1 - exp(-\frac{R_A^2}{2\alpha^2}))exp(-\frac{R_B^2}{2\beta^2})(1 - exp(-\frac{S^2}{2\gamma^2})) & \text{if } \lambda_2 < 0 \text{ and } \lambda_3 < 0 \\ 0 & \text{else} \end{cases} \tag{3.13}$$

where parameters α, β, γ are user defined. And the constraint on the eigenvalue sign assumes selection of a bright object on a dark background.

The final vesselness value is found by evaluating the function over a scale range and selecting a maximum value.

The vesselness measure was generalized for other shapes in Antiga (2007), by defining the *objectness measure*. This works by extending R_A and R_B to support M-dimensional structures in N-dimensional image.

$$R_A = \frac{|\lambda_{M+1}|}{\prod_{i=M+2}^{N} |\lambda_i|^{\frac{1}{N-M-1}}} \tag{3.14}$$
$$R_B = \frac{|\lambda_M|}{\prod_{i=M+1}^{N} |\lambda_i|^{\frac{1}{N-M}}} \tag{3.15}$$

R_A and R_B are then analogically used for the objectness measure, with similar constraint on the eigenvalue sign:

$$O(\lambda)_\sigma = \begin{cases} (1 - exp(-\frac{R_A^2}{2\alpha^2}))exp(-\frac{R_B^2}{2\beta^2})(1 - exp(-\frac{S^2}{2\gamma^2})) & \text{if } \lambda_j < 0 \text{for } M < j \leq N \\ 0 & \text{else} \end{cases} \tag{3.16}$$

This shape analysis framework can be then used to define transfer functions. The simple approach would be precomputing the objectness measure for some scale range and use it directly as a second dimension in 2-dimensional transfer functions.

But more interesting usage would be by precomputing only the eigenvalues for Hessian matrices in the multiresolution image. And compute the objectness measure on the fly based on the shape selection and tuning of the α, β, γ parameters interactively.

This framework can be also connected to the previously referenced size oriented transfer functions.

Eigenvalues of the Hessian matrices can be used to generate the scale field.

One of basic properties of the matrix trace in relation to matrix product is:

$$tr(AB) = tr(BA) \tag{3.17}$$

From which arises the preservation of matrix trace for *similar matrices*:

$$tr(P^{-1}AP) = tr(P^{-1}(AP)) = tr((AP)P^{-1}) = tr(A) \tag{3.18}$$

This equality can be used to derive equality between *laplacian* of the function at a point and sum of eigenvalues of its *Hessian matrix* in the same point.

$$\Delta f = \sum_{i=1}^{n} \frac{\partial^2 f}{\partial x_i^2} = tr \begin{pmatrix} \frac{\partial^2 f}{\partial x_1^2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{pmatrix} = tr(H(f)) \tag{3.19}$$

$$tr(H(f)) = tr(H_\lambda(f)) = \sum_{i=1}^{n} \lambda_i \tag{3.20}$$

$$\Delta f = \sum_{i=1}^{n} \lambda_i \tag{3.21}$$

This equation ties the hessian multiresolution pyramid together with *scale field* from eq. 3.5.

It means that the hessian multiresolution pyramid can be used for both the *scale oriented* and *scale and size oriented* transfer functions.

Implementation

The most expensive operation of the preprocessing step is computation of the Hessian pyramid and diagonalization of the Hessians in each voxel. All this can be done on GPU by using same algorithm which was used for diagonalization of

the structure tensor (section 2.2, 3×3 symmetric matrix diagonalization from Kopp (2008)).

Only eigenvalues of the Hessian matrices are needed from the Hessian multi-resolution pyramid, that means that for each voxel three values must be stored. That suits well for the texture storage – each eigenvalue will be stored as a color channel in an RGB texture. The initial idea to store the pyramid in the mip-map texture have shown to be unfeasible as it puts strict limitations on the sizes of the pyramid levels. Thus a set of 3D textures is used and a subset is passed to the ray marching shader based on user specified size interval. Rest of user specified parameters (feature dimension, α, β, γ), which are need for computation of the *objectness measure* are supplied as shader arguments. Shader then computes the R_A, R_B and uses them to compute $O(\lambda)_\sigma$ for all textures in the provided texture subset and for each ray marching step. From this, it can be seen that the rendering speed can quickly drop down if a large size intervals covering a large number of pyramid levels are used.

The pyramid containing the Hessian eigenvalues can be also used to generate the scale field (equation 3.6) using the equation 3.21.

3.4 Conclusions

The objectness measure is usually used as a part of segmentation algorithms. Direct volume visualization can be considered as a fuzzy segmentation/classification process, so employing ideas originally used only for offline segmentation algorithms may lead to new ways for volume visualization.

An example of this approach was presented in this chapter. Shape and size oriented transfer functions based on the objectness measure have shown to be a useful tool for differentiation between objects of similar density, as can be seen in figure 3.6. The interactive nature of the transfer function settings allows fine-tuning the objectness measure. This is again an advantage in comparison to the usage in automatic/semi-automatic segmentation.

Although the user must be aware that the objectness measure is based only on local low-level data analysis and as such, it may produce false positives in several situations – long straight edges give a similar response as tubular structures, end points of tubular structures are similar to small balls. In practice, these false positives can be often easily distinguished by the first channel in 2D transfer function mapping.

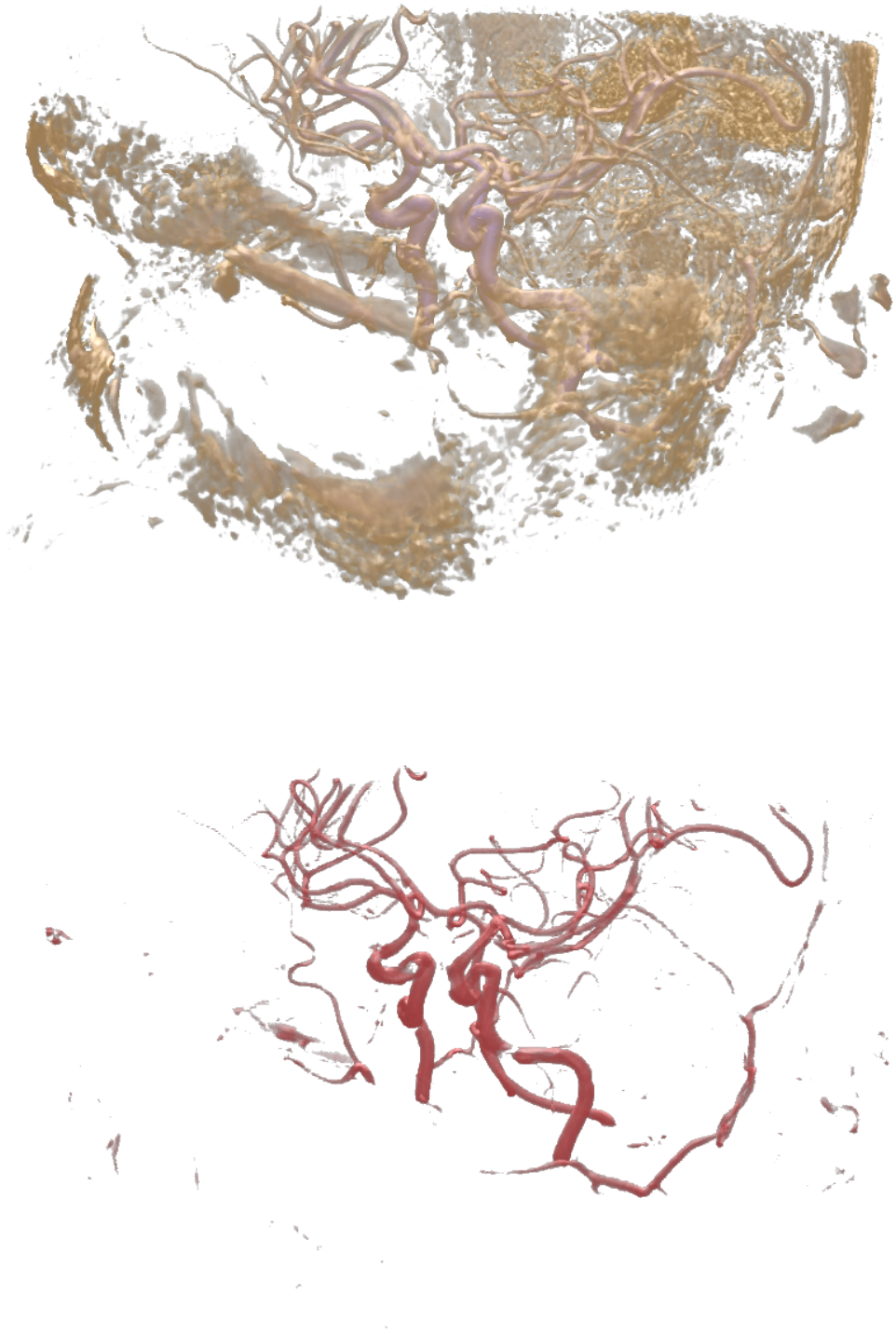


Figure 3.6: Top: 1D transfer function applied to angiography acan of humand brain. Contrast agent was used, so the veins seem to have same density as bones. Bottom: Shape transfer function used to highlight the tubular structures in brain. It removes clutter present in the previous image.

Chapter 4

Cellular Automata and Watershed Transformation

In this chapter an overview of cellular automata theory will be given and several additional extensions to the original concept will be introduced to allow efficient simulation of the cellular automata on GPUs. The presented theory will be then used to reformulate watershed transformation to the language of cellular automata, which will allow simple and efficient implementation on CUDA enabled devices.

4.1 Cellular Automata

4.1.1 Cellular Automata Introduction

A cellular automaton (CA) is a computational model. It is defined as a collection of cells of defined state, organized in a grid of specified shape. The grid has a property which defines a set of neighbors for each cell.

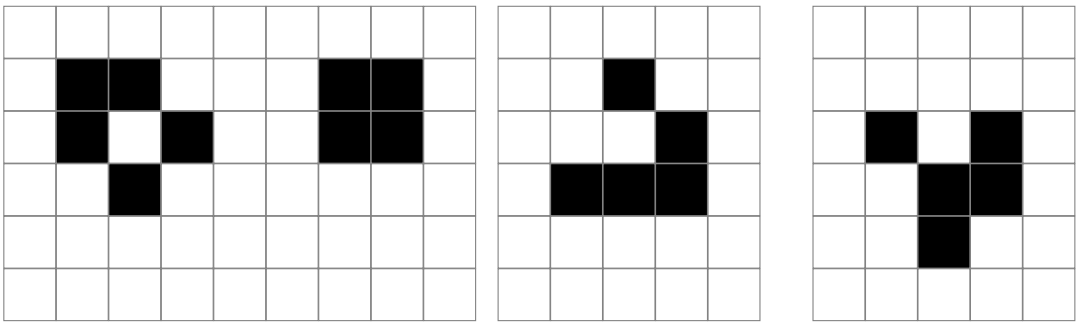
Cells evolve in discrete timesteps starting at initial time $t = 0$. New generation is created (advancing t by 1) according to a fixed rule which assigns each cell a new state based on the state in the previous generation together with the state of its neighbors. Formal definitions will follow in section 4.1.2.

Cellular automata can be viewed as dynamic systems for information distribution. Each cell can be interpreted as an agent who is gathering information from its neighbors and using it to replace or update its current state. The information flow is from neighbors into the cell, so information cannot be pushed directly to a neighbor. Although it can be simulated by special state set and automata rules.

Cellular automata were introduced by Stanislaw Ulam (lattice network for crystal growth simulation) and John Von Neumann when he was working on the problem of self-replicating systems (von Neumann (1951); Pickover (2009)).

But it was not until 1970s, when the publication of *Conway's Game of Life* (Gardner (1970)) led to wide spread study of two-dimensional cellular automata.

The Conway's Game of Life is simple two-dimensional automaton which simulates a life of cells. Each cell can be alive or dead. In each generation a new state for each cell is decided:



(a) Stable structures (b) Two iteration of a "glider" pattern

Figure 4.1: Example of Game of Life patterns

- Living cell surrounded by more than three living cells will die by over-population.
- Living cell with less than two living neighbors will die of under-population.
- Dead cell with exactly three alive neighbors becomes a live cell – reproduction.
- In other situations cell state does not change.

Experimental runs of the Game of Life cellular automaton show complex behavior from these simple rules. Several types of stable patterns emerging during simulations can be observed – static structures which do not change between iterations, oscillators cycling through set of shapes, and structures moving through the grid called *gliders* (see figure 4.1 for examples).

It was proven that Conway's Game of Life is *Turing complete* by showing the construction of counters and logical gates from the stable cellular patterns. Thus, it is possible to build finite state machine operating on two counters, which has the same computational power as *universal Turing machine* (Berlekamp et al. (1985)).

Main motivation for studying cellular automata in this thesis comes from the need for formal system which can be used to describe algorithms for the massively parallel hardware architectures like CUDA.

Cellular automata are by definition parallel – all cells are updated in a single moment when creating a new generation. This idea can be implemented in a software framework which efficiently emulates cellular automaton evolution.

By reformulation of computational problems into the language of cellular automata, a powerful tool can be created for proving algorithm correctness in a massively parallel environment.

Although cellular automata can be a useful tool for the theoretical study of algorithms the main focus of this chapter will be a study of problems *easily solvable* by cellular automata. "Easily solvable" is a vague description of a set of problems. As stated in previous paragraphs the CA has the same computational strength as a Turing machine, so any Turing-solvable problem will be solvable by CA, but for lots of problems it would require complicated mapping of inputs and outputs, constructing complex structures simulating processes from other computational models, etc. In this context "easily solvable by CA" will mean that

the algorithm will directly exploit inherent properties of cellular automata, like neighborhood processing and parallel computation, without strong adverse effects on computational complexity. This usually means algorithms, which follows some local or semi-local condition such as numerical solving of partial differential equation (PDE) or spatial image filters.

4.1.2 Definitions

Several different variants of cellular automata definitions can be found in the literature. Most of them are compatible/equivalent with each other. Some constrain the definition, for example by allowing only regular lattices of cells, thus preventing usage in unstructured domains.

In this thesis the definition 4.1 will be used as a formal description of cellular automaton and further extensions to the cellular automaton concept will build upon this definition.

Definition 4.1 (Cellular automaton - CA)**.**
Cellular automaton is defined as a tuple (S, G, N, f, s_0)*, where*

- *Set of possible cell states* $S \neq \emptyset$
- *Grid* G *– set of discrete points.*
- *Neighborhood* N *is a set of offsets:* $N = d_1, d_2, \ldots, d_n$*, such that* $\forall x \in G, d_i \in N : (x + d_i) \in G$*. Handling of the grid boundary problem will be described later.*
- *Rule* $f : S^{|N|+1} \rightarrow S$
- *Initial automaton state* $s_0 : \forall x \in G, s_0[x] \in S$

Time step:

$$\forall x \in G : s_{i+1}[x] = f(s_i[x], s_i[x + d_1], s_i[x + d_2], \ldots, s_i[x + d_n])$$

Although the topology of grid G is not constrained by some additional conditions (except the definition of valid neighborhood N), regular lattice in 2 or 3 dimensions and finite in size will be considered in the rest of the thesis. The reason is, of course, the focus on image processing tasks and one-to-one mapping between the automaton grid and input image. That also means that the neighborhood N relates directly to the pixel/voxel connectivity.

Typical types of the cell neighborhoods are *Moore neighborhood* (8-connectivity in two dimensions, 27-connectivity in three dimensions) and *von Neumann neighborhood* (4-connectivity in two dimensions, 6-connectivity in three dimensions).

The cell state set S is not limited much. It can be set as small as the *alive/dead* two state set of Conway's Game of Life, or theoretically infinite if using a set of natural or real numbers.

By taking a software engineering point of view, the cell state is the state of a "program" running in each cell and the update rule being the "program", so the cell state must contain all the variables, temporary values, and results needed by this "program". That is the reason why the cell states of the most

practical applications of cellular automata in the following sections can be seen as composite states (Cartesian products) of substate sets (of code, variables, temporary values and results).

To ease the discussion about the cell states, it is useful to define few special kinds of these substates.

Definition 4.2 (Catalyst substate)**.**
Let set of cell states be $S = C \times U$ (U being arbitrary substate). The substate C from set S is called catalyst state, if for each generation t and for each cell x: $s_t[x] = (c[x], u[x])$ apply that $s_{t+1} = (c[x], u[x]')$.

Catalyst state does not change through the computation. It is often the input which drives the actual computation. It is important to mark this kind of substate because it can be isolated in actual implementation and optimized away from the state update step, thus lowering the amount of memory needed and speeding up the computation of the update rule. From programmers point of view, it is a constant assigned at the initialization of the program (not necessarily compile time constant).

Definition 4.3 (Result substate)**.**
Let set of cell states be $S = U \times R$ (U being arbitrary substate). The substate from set R is called a result substate, if it is the output of the algorithm, currently executed by the cellular automaton.

This is a definition highly dependent on what is the actual output of the algorithm. If several CA instances can be used for solving the same problem, this substate will be common to all of them and rest of the substates can be ignored as different internal temporary substates.

Boundary Handling

The automaton grid is usually finite in practical cases. That brings the classical problem typical for majority of the spatial image filters – how to handle access to cells outside the valid grid region. Clearly, there is no universal solution, and the approach must be tailored to the actual cellular automaton instance.

Few of possible options:

- Boundary conditions based on PDE boundary conditions such as Dirichlet and Neumann conditions. The grid is embedded into an infinite set, where each cell has the cell value assigned by a condition. Commonly used special cases are:
  - All outside cells are zero.
  - Derivation in the direction of the boundary normal is zero – grid boundary values are repeated in the normal directions.
- Periodic extension of the grid or wrapping the coordinates around – mapping the grid on closed hypersurface like a torus.
- Using special *NULL* value to mark access outside the grid – the CA rules need to handle this special value.

More complex cellular automata often have composite cell states and as such it is possible that different approach is used for each substate, making also the boundary condition composite.

Convergence

To use cellular automata for solving practical problems, there must be defined a clear condition when to stop the automaton from producing new generations. This again depends highly on the solved task.

Common approaches used for example by iterative numerical algorithms are either fixing the maximal number of iterations, or by using measure of a difference between two grid states and stopping the automaton, when it drops under certain ϵ-threshold.

For future references it is useful to define an *equilibrium state*:

Definition 4.4 (Equilibrium state of cellular automaton)**.**
Cellular automaton (S, G, N, f, s_0) *has an equilibrium state* s_{eq} *when* $\exists n \geq 0, \forall i \geq n : s_i = s_{eq}$

In plain words, it means that state of the automaton no longer changes between iterations.

Automata Equivalence

As will be shown in following sections, some problems can be solved by several different cellular automata. To discuss properties of these automata in relation to the problem solved, there must be a definition of what it means that two automata solve the same problem. Different CA will probably have different domains of cell states, and as such, their final grid state after convergence cannot be used for direct equality comparison.

Definition 4.5 (Cellular automata equivalence)**.**
Let there be two cellular automata Φ *and* Ψ*. Their respective cell state sets are* $S_\Phi = C_\Phi \times T_\Phi \times R_\Phi$ *and* $S_\Psi = C_\Psi \times T_\Psi \times R_\Psi$*, where substates* C *are catalyst states,* T *are "temporary" states, which are not relevant for the solved problem* $\mathbb{B}$*, and* R *are results states. The results states are the solutions for the problem* $\mathbb{B}$*.*

Cellular automata Φ *and* Ψ *are considered equivalent in relation to problem* $\mathbb{B}$*, if for each final state exists bijective mapping* $f : R_\Phi \to R_\Psi$*, which maps the final result states between grids of automata* Φ *and* Ψ*.*

This definition of equivalence will be used later to compare results of different cellular automata computing watershed transformation – the bijective relation will map between different region labelings.

4.1.3 Cellular Automata on GPU

As was said earlier cellular automata were chosen as suitable formalization of computations on massively parallel architectures (especially CUDA).

The simplest mapping between these two domains is by letting each cell state update to be computed by a single thread. The global synchronization will end the

iteration when all new states are computed. Global synchronization is important – otherwise, the key property of all cells being updated at the same time would be broken. Discussion about asynchronous cellular automata will be opened in section 4.1.4.

Double Buffering

The standard approach to global synchronization of CUDA computation is separate kernel executions. This means that the cellular automaton implementation cannot do an in-place updates. All cell states must be updated together before any of the new states can influence other cells. The only exception is cellular automata with an empty neighborhood, which update the state of each cell only by using its previous state.

There is also an approach used in CPU multithreaded spatial filters. Computation proceeds on only part of the image and the modified overlap is stored in helper buffer to prevent modification of unprocessed image elements. This can be used to implement in-place synchronous automaton, but it requires global synchronization inside kernel which brings other complications as discussed in section 1.1.4. Also, the computation is limited only to a portion of the image. It means that only small number of the thread blocks will be running at any moment and that limits the GPU block scheduler in memory latency hiding as described in section 1.1.1.

Approach, which is usually chosen for such tasks, when the in-place updates are problematic, is to use two buffers. One serving as input and the other one as output. The input buffer is read only for cell update, so it prevents problems with concurrent updates in neighbors.

In following iteration these two buffers switch roles as the output buffer serves as input for next iteration.

Main problem with the double buffering approach is the increased memory requirement, which can be sometimes reduced by storing catalyst substates separately as those do not change during computation.

Preloading to Shared Memory

If the cellular automaton has a non-trivial cell neighborhood, then the important optimization would be preloading cell states into the shared memory in each thread block (discussed in sections 1.1.2 and 1.5.3) used for CA computation. This optimization is feasible only in scenarios, where each cell needs to be read multiple times from the same thread block – it is part of multiple neighborhoods.

The shared memory preloading has even bigger inpact on the compuation time especially for the asynchronous cellular automata as will be described in the following section.

4.1.4 Asynchronous Cellular Automata

Synchronized cell update can be limiting. When simulating living systems the existence of global synchronization step is an unnatural phenomenon as organisms tend to behave independently.

For computations on GPU the global synchronization enforces the use of double buffering and limits the amount of computation that can be done by a single kernel execution (only one CA iteration).

The cell update process can be modified so it does not update all cells together:

Definition 4.6 (Standard asynchronous cellular automaton)**.**
It is defined as tuple (S, G, N, f, s_0) *(same as definition 4.1)*

Time step:

$$\forall x \in B_i \subseteq G : s_{i+1}[x] = f(s_i[x], s_i[x + d_1], s_i[x + d_2], \ldots, s_i[x + d_n])$$

Only subset of cells is updated in each step. This can be useful from several reasons. Its actual implementation is less constrained and the scheduling of the cell updates can be optimized for the computer architecture. Also the fact is that asynchronous cellular automata can emulate process of synchronous cellular automata as was proven in Nehaniv (2003). It broadens the set of possible approaches that can be used to solve a particular problem.

Even though this definition alleviates the constraint of global synchronization it still limits possible implementation optimization. Especially exploiting of the shared memory preloading (section 4.1.3). Standard asynchronous cellular automaton must make the updated state visible after each cell iteration, so it can be used by another update rule instance, which is applied on neighbors. This limits the efficiency of parallel implementation. By removal of this constraint the following variant of cellular automaton can be defined.

Definition 4.7 (Asynchronous cellular automaton with hidden updates)**.**
It is defined as tuple (S, G, N, f, s_0) *(same as definitions 4.1 and 4.6).*

For some region $R \subset G$*, the* $\overline{R}$ *is the closure of the region by using the neighborhood* N *– union of the cells in the* R *region and all their direct neighbors.*

In each iteration t *the disjoint set of regions* R_i *is updated in such way, that synchronous cellular automata* $(S, \overline{R_i}, N, f, s_t)$ *are applied separately and their final state is assigned back to the* R_i *region in the original automatons grid.*

It means that the CA in question will compute several cell updates without storing the result into the grid to be seen by others. Also, it does not read new updates for the neighbor states immediately.

Futher in the text, if not mentioned otherwise, by *asynchronous cellular automaton* will be meant *asynchronous cellular automaton with hidden updates.*

This definition directly maps to CUDA device execution primitives. By using thread blocks as the disjoint regions and doing as much iterations as possible just inside the thread block ($\overline{R_i}$ regions are stored in shared memory) without signaling the results outside until the kernel finishes.

This cellular automata extension can be also utilized for distributed computation. When processing extremely large datasets in computation cluster or cloud it is possible to simulate the cellular automaton separately on distributed portions of the grid without the need to explicitely synchronize after each iteration. Synchronization will happen only for the region overlaps less frequently than for the synchronnous cellular automaton.

Algorithm 4: Kernel for asynchronous cellular automaton with hidden updates

Data: In - global memory buffer
Result: Out - global memory buffer
load part of In buffer into shared memory buffer S;
syncthreads();
while *local updates possible* **do**
    $newState \leftarrow ApplyRule(threadIdx, S)$;
    syncthreads();
    $S[threadIdx] \leftarrow newState$;
end
$Out[offset + threadIdx] \leftarrow S[threadIdx]$

4.1.5 Speed of Convergence

Even though that a reformulation of a particular problem into the language of cellular automata (with proof of its correctness) is possible. It does not mean that the final algorithm implementation using the GPU cellular automata will be efficient. Trying to reformulate problem which is not suitable for massive parallelization can cause too complex cell update rules and thus too slow iteration computations.

But inefficient implementation can be obtained even for algorithms which have simple and fast rules. The inefficiency comes from large number of iterations needed before the convergence criterion is fulfilled.

The problem will be illustrated on the *connected component labeling (CCL)* - one of the basic image processing algorithms. Connected component labeling takes binary mask as input and assigns unique label to each masked region. Each element of the output image is either labeled as background or it has the exact same label as all of its non-background neighbors.

This simple property can be converted into cellular automaton rule 4.1, which keeps relabeling the cell until all foreground neighbors have same label as the cell in question. To ensure convergence the new label is always chosen to be the minimum from possible candidates.

Cellular Automaton 1: Connected component labeling
State: Label $l \in \mathbb{N}_0$

Grid: Regular lattice – each cell represents image element.

Neighborhood: Reflect desired connectivity of the labeled regions. $|N| = n$

Initial grid state: Unique label for each non-background element (assigned via scanline traversal, spacefilling curve, etc.)

Cellular automaton rule 4.1 (Connected component labeling)**.**

$$f_{ccl}(l_0, l_1 \dots l_n) = \begin{cases} 0 & \text{if } l_0 = 0 \\ min(\{l_i | i \in \{0, \dots, n\} \wedge l_i \neq 0\}) & \text{else} \end{cases}$$

Rest of the CA properties are simple. The grid and neighborhood are enforced by the input image topology. Cell state domain is the label set with special label for background. So the remaining property is the initial state of the automaton. By assigning unique label to each foreground cell an overlabeled image is obtained. Each iteration of the cellular automaton CA_{CCL} decreases number of the active labels and the process ends when the cellular automata rule 4.1 does not update any cell. The minimum of the neighboring labels is used to ensure that the relabeling is acyclical, thus preventing infinite relabeling cycles.

Large iteration count of the CA_{CCL} is caused by slow propagation of the label into the whole connected component. Cellular automaton can propagate information to distance of neighborhood radius (1 in most cases) in single iteration. So the number of iterations depends on size of the regions and on the positions of minimal labels in each connected component.

To tackle the problem of *excessive iteration count*, cellular automata extensions which focus on faster information propagation will be discussed in following sections.

4.1.6 Global State

One way to speed up the information propagation is by broadcasting it. Cellular automaton can be extended by special global state accessible from all cells during application of the update rule.

CA rules and iteration number can be considered as global state even in the standard CA definition, but the global state must be modifiable by the update rules to make a difference and possibly allow for the faster converging automaton.

The simplest global state possible is a two state flag. Such flag is practical for detection of the automaton convergence – stabilization. The rule will signal the flag everytime it assigns a new state to a cell. The flag is accessed from the rule only when it is signaled, also assignment of the *int* is atomic on CUDA devices, so there are no race conditions even without explicit usage of atomic instructions, which would otherwise cause serialization of the flag signaling. If the automaton did not converge (flag was signaled), then the flag need to be reseted before execution of the next iteration.

Formally the global state extension can be defined as follows:

Definition 4.8 (Cellular automaton with global state)**.**
It is defined as a tuple $(S, G, N, f, s_0, g, U, u_0)$*. Where* U *is the set of possible values for global state update,* u_0 *is the initial global state and* $g(\ldots)$ *is a function generating information for global state update when applied to all cells.*

$$\begin{aligned} f &: S^{|N|+1} \times u_t \rightarrow S \\ g &: S^{|N|+1} \times u_t \rightarrow U \end{aligned}$$

The global state update for generation $t+1$*:*

$$u_{t+1} = u_t \oplus \{g(s_t[x], s_t[x+d_1], s_t[x+d_2], \ldots, s_t[x+d_n], u_t) | \forall x \in G\}$$

where the $\oplus$ *is the update operation of the global state.*

To cover the global states in a generic way, the actual cellular automaton implementation looks like the algorithm 5. The *preprocess* and *postprocess* steps serve as synchronisation points for the global state. Consolidation of the global state (convergence flag reset, relabeling inputs, etc.) can be done in those steps.

Algorithm 5: Cellular automaton with global state

```
Data: CA grid in initial state
Result: Grid of converged CA
Init global state;
while not converged do
    preprocess global state and grid;
    run iteration;
    postprocess global state and grid;
end
```

Race Conditions

Having all update rules access the global state brings the question of the race conditions. Few scenarios should be considered:

- The global state modifications are synchronized, which brings possible bottleneck to the whole computation by limiting possible concurrency.
- Write operations does not colide and the global state serves only as a "bulletin-board".
- Updates are compatible – multiple cell updates will do the exact same change. Continuation flag serves as good example.
- Allow benign race conditions – makes the computation non-deterministic, but if the final state after the convergence is always correct, such race conditions can be allowed.

The correct scenario must be assesed for each case and the decision depends on the correctness and required speed as the relaxed synchronization requirements reduce the possibility of serialization of the update operations.

Union-Find

CCL and other algorithms, which mark disjoint image regions by different labels, define equivalence relation (reflexive, symmetric, transitive), where each of the differently labeled regions represents different equivalence class.

Union-find datastructure (Fredman and Saks (1989)), also known as *merge-find set*, is usually used to construct equivalence classes. It provides two operations:

- *Find* – return a class for an element.
- *Union* – merge two equivalence classes into one by providing some element from both of them.

The cellular automaton 4.1 propagates the information about equivalent classes too slowly. If the union-find datastructure is used as the global state, the information about equivalent labels can be propagated much faster and a root label for each class can be used for relabeling all elements in a connected component.

Traditional pointer based datastructure used in CPU implementations does not work well on GPUs, so a buffer for whole label range is created – the label serves as index into this buffer. This buffer defines the connection between labels from the same class – the *union* operation assigns the minimal label as a mapping for the bigger label if it improves current state (see algorithm 6).

This version differs from the optimal CPU version, which does the compaction of the tree representing the equivalence class. But this version is more suitable for GPUs as it prevents warp divergence, which would occur for the CPU version (Fredman and Saks (1989)).

The compaction step was moved into the postprocessing step of the cellular automaton with global state.

Algorithm 6: Union operation for two labels

```
Data: labels first and second, label buffer
minLabel = min(first, second);
maxLabel = max(first, second);
maxRoot = labelBuffer[maxLabel];
if maxRoot ¿ minLabel then
    labelBuffer[maxLabel] = minLabel;
end
return minLabel;
```

Cellular automaton rule 4.2 (CCL with global union-find datastructure)**.** *Let* $L = min(\{l_i | i \in \{0, \ldots, n\} \wedge l_i \neq 0\})$ *be the minimal valid label for updated cell neighborhood.*

$$[f, g]_{CCL\ global}(l_0, \ldots, l_n, u_t) = \\ = \begin{cases} [min(L, l_0), (l_0 \equiv L)] & \text{if } l_0 \neq 0 \wedge L < l_0 \\ [l_0, \emptyset] & \text{else} \end{cases}$$

Where $f_{CCL\ global}$ *is cell update rule,* $g_{CCL\ global}$ *is global state update rule and* u_t *is global state (union-find) in generation* t*.* $(l_0 \equiv L)$ *is the new equivalence relation, which will be registered in the global state.*

Uncertain Operations

If the union-find datastructure is implemented correctly, so it can be safely used in multithreaded environment, it may bring too much burden in form of atomic operations (assignment to the label buffer), uncoalesced memory accesses and thread divergence during the *union* step.

This can be alleviated by introduction of *benign race conditions*. By assuming that in one iteration of the cellular automaton two equivalence relations were discovered: $x \equiv y$ and $x \equiv z$, assuming that x, y, z were all from different classes.

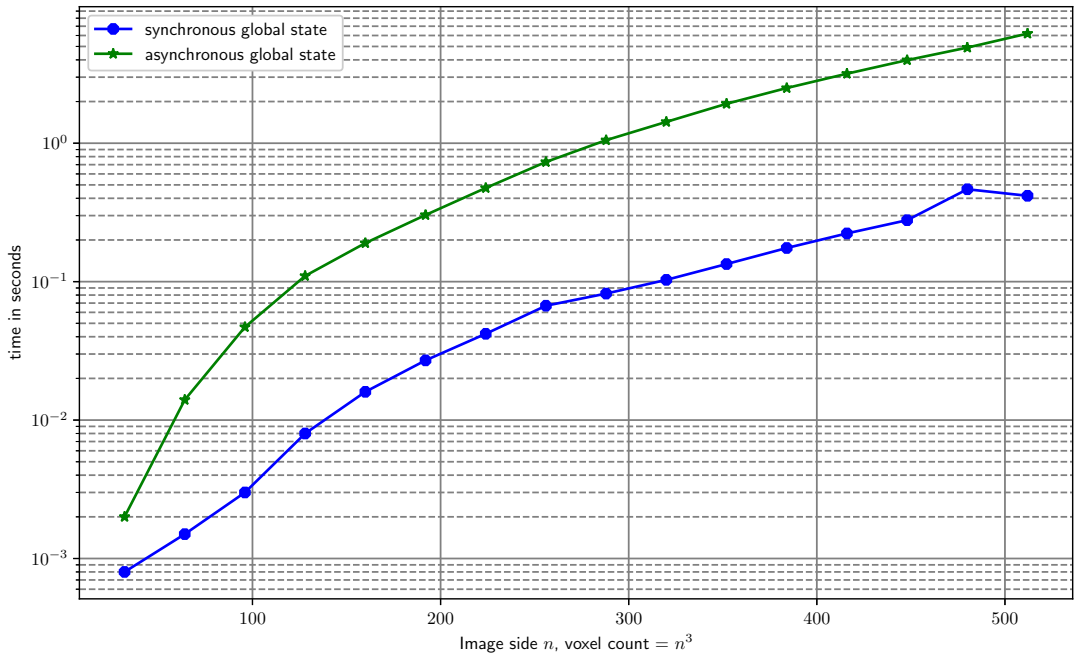


Figure 4.2: Connected component labeling.

Standard union-find algorithm would reflect both these relations after the CA iteration at the price of some possibly expensive operations.

If, on the other hand, merging is done in such way that traversal of the tree-like pointer structure is prevented in a price that some of the union operations are undone by concurrent ones, then a faster computation of the CA iteration can be implemented. Uncertainity of the union operation introduces race conditions, but these does not cause incorrect results. If addition of some of the detected equivalence relations are reverted, then these will be detected again in the next iteration and another attempt to union the relation can be made. In each iteration at least one of the finite set of conflicting unions will be handled, so the number of required iterations is finite. Even though intermediate grid states depend on the timing the final result is exactly the same, thus making the race conditions benign.

Asynchronous Automaton and Two Level Global State

Asynchronous cellular automaton with hidden updates (definition 4.7) can be used to futher optimize the usage of a global state. A congestion of the update operations can be lowered by two level global state. The first level serves as normal global state, but separate instances exist for each computation block. The second level is normal globally accessible state. Hidden updates happening inside the block use only the first level semi-global states and at the end of the series of hidden updates it will transfer the data from the first level to second level global state.

When all the modifications are applied, the automaton for connected component labeling will be doing similar operations as the algorithms in Hawick et al. (2010) Stava and Benes (2010). Single pass, two pass and multi pass algorithms were presented in these and similar papers. Which of these variant fit best to the

cellular automaton, depends on the global state used.

The comparison was done for synchronous CCL automaton with global state and asynchronous CCL automaton with two level global state (figure 4.2). The asynchronous automaton has the expected effect – the computation is done in single iteration. The computation of the synchronous automaton needs 2-4 iterations. But as can be seen from the graph, the management of the two level global state is too expensive in case of union-find.

4.1.7 Cells with Pointers

Analysis of the cellular automaton computation for the connected component labeling and similar tasks shows that the progress towards final results can be divided into two phases. First is a search phase, where the root label is located and the second phase is a propagation of this information back.

This observation can serve as a base for another cellular automaton extension. Introducing a pointer to extend the cell state, which will be used for accessing arbitrary other cell from the update rule (not only neighbors). It allows the propagation phase to be shortened if the search phase also updates these pointers, so they lead to the root labels. The propagation phase can be then done in a single step.

Definition 4.9 (Cellular automaton with pointers)**.**
Cell state se is in form $S_p \subseteq \{G, Null\} \times S$*, where* S *is state set by standard CA definition and* G *is set of grid coordinates. Rule is extended to* $f_p : \{S, Null\} \times S_p^{|N|+1} \to S_p$*, where* N *is size of the cell neighborhood, and time step between generation* i *and* $i+1$ *is in form:*

$$\forall x \in G : s_{i+1}[x] = f_p(s_i[s_i[x]^{(0)}], s_i[x + d_1], s_i[x + d_2], \ldots, s_i[x + d_n]) \tag{4.1}$$

Special state by definition: $s_i[Null] = Null$

Stable Pointer Paths

Experimentation with the cellular automata rules led to a definition of special case, which will be usefull in following sections:

Definition 4.10 (Search with stable pointer paths)**.**
Cellular automaton with pointers does a search with stable pointer paths if the following conditions hold:

1. *In the first iteration all cell pointers leads back to the cell itself.*
2. *Each cell with final result value assigned have pointer leading to itself.*
3. *If a cell* $s_t[x]$ *has a pointer leading to some other cell* $s_t[y]$*, the pointer is updated either when the pointer in* $s_t[y]$ *leads to another cell* $s_t[z]$ *and new pointer will be* z*, or the* $s_t[y]$ *has its result value, which will be used to assign result value to* $s_t[x]$*.*

Important property of the cellular automaton matching the definition 4.10 is that the pointer values during iterations create a path for each cell, which leads to the cell with result state assigned.

The fact that pointer leads away only when the result value is not yet assigned to the cell may be used optimize memory requirements by storing pointer and the result value at the same location.

Automaton Decomposition

An observation can be made about an automaton with rule fulfilling the definition 4.10. The cells which are in the final state before the first iteration, will never change their state and their pointer will always lead to itself. If a cell is not in the final state after the first iteration it means that none of its neighbors was in the final state and thus it must search via pointer obtained from one of its neighbors. The final state will be assigned to that neighbor also via the same pointer, or its extension along the stable search path. So the direct access to the neighbor is needed only during first iteration to initialize the pointers.

This fact can be used for automaton decomposition into two instances, where the first automaton will use a rule which accesses only the neighboring cells and assigns a new pointer where needed. Second automaton on the other hand can ignore the neighborhood and use only the pointer access to do the search.

Such decomposition brings two kinds of optimizations. It lowers the amount of memory accesses in each iteration and the catalyst substate can be ignored for the second automaton and as such freed from memory.

Stream Computation

In situations when large datasets that does not fit into device memory (or even into host memory) need to be processed by cellular automaton, a new implementation strategy must be used. Grid update must happen in several steps, which makes synchronous cellular automata unfeasible. Theoretically it is possible, but it would mean that for each iteration all grid segments must be loaded and the result stored again, which makes it very expensive operation.

Asynchronous cellular automata on the other hand bring the advantage of doing as much iterations on the loaded segment as possible before the computation switches to another grid segment. But even with the advantages of ACA the process in general would be to repeatedly load, process and store all the grid segments until the convergence criterion is achieved.

But as usual, for special cases a better solutions exists. Let Φ be *asynchronous cellular automaton with cell pointers* (follows definitions 4.7 and 4.9) and its cell update rule complies the definition of the *search with stable pointer path rule* (definition 4.10). The automaton computation can be wrapped in two pass process, which goes forward and then backward through the set of grid slices groups (slabs). This algorithm is a generalization of the algorithm used in Hucko and Sramek (2012) and its application to cellular automata.

To prove the correctness of the algorithm 7 two lemmas will be introduced:

Algorithm 7: Two pass streamed computation of the CA with stable pointer path

```
Data: CA grid in in initial state
Result: Grid of converged CA
calculate max slab size and slabcount fitting in the memory;
/* First pass                                                    */
load first slab;
load overlap slice with the second slab;
run CA on the slab until convergence;
for i ← 2 to slabcount - 1 do
    load overlap slice with slab i − 1;
    load slab i;
    load overlap slice with slab i + 1;
    run CA on the slab i until convergence;
end
load overlap slice with previous slab;
load last slab;
run CA on the last slab until convergence;
/* Second pass                                                   */
for i ← slabcount − 1 to 2 do
    load overlap slice with slab i − 1;
    load slab i;
    load overlap slice with slab i + 1;
    run CA on the slab i until convergence;
end
load first slab;
load overlap slice with the second slab;
run CA on the slab until convergence;
```

Lemma 4.1.
During the first pass, after processing of slab i, each cell of slab i has either its final value assigned, or its pointer leads into the slab $i + 1$.

Proof. After processing of the first slab it is obvious that only pointers aiming outside leads to second slab (no previous slab). So it remains to prove that there will not remain any pointer leading to some element inside the first slab. Argumentation will be based on the properties of search rule with stable pointer paths.

Suppose that there is some cell u, which has pointer leading to the another cell v inside the first slab. If the cell v does not have its final value assigned it would mean that its pointer leads to some other cell, but that conflicts with the convergence criterion – only exception is if the cell v lies in the first slice of the next slab, so it is never updated during the slab processing and also means that pointer from cell u leads outside the first slab. Also if the cell v has its final value assigned it would again mean that convergence criterion was not met, otherwise the cell u would have also its final value assigned.

Proof by mathematical induction will be used to prove that the condition

holds for all slabs. If the condition holds for all slabs until slab i it will be shown that the condition holds even for the slab i. Same argumentation for missing innward pointers can be used as was used for first slab, but now the pointers leading outside of the slab can possibly go also to the slab $i-1$ and not only to the slab $i+1$.

If during processing of the slab i some cell p updates its pointer to some cell q from last slice of the slab $i-1$, two outcomes are possible. Either the cell q has its final value assigned and then after convergence also the cell p will have its final value assigned, or the cell q contains a pointer leading into slab i (there is no other possible direction possible for the slab $i-1$). This means that after convergence if p still does not have its final value its pointer will definitely not lead into slab $i-1$ and this proves that the condition holds even for slab i and thus for all slabs after the first pass. □

Lemma 4.2.
During second pass (which goes backwards from the last slab to the first slab), after processing of the slab i, each cell of slab i has its final value assigned.

Proof. This lemma is simpler to prove then the previous one. It will be proved also by applying mathematical induction now going from last slab to the first one.

According to the lemma 4.1 all cells in the last slab will have their final value assigned (no next slab for pointers to lead in).

If the condition holds for all slabs from n down to $i+1$ it can be shown that it also holds for slab i. When processing slab i all remaining pointers lead to the slab $i+1$, but these have all their final value assigned and because of that all cells with pointers will use these values to assign their own final values. □

Lemma 4.3.
During the second pass following statements hold:

1. *Only single iteration of the cellular automaton needs to be executed on each slab.*
2. *Automaton rule will not need the cell neighborhood – only the cells referenced by the pointer.*
3. *The catalyst state is not needed and can be ignored.*

Proof. Point 1 and 2 are implied by the lemma 4.1, lemma 4.2 and definition 4.10. Before processing of slab i all its cells without final state assigned have a pointer leading to the border of the slab $i+1$. But all the cells in the slab $i+1$ have their final state assigned, so it takes only single iteration to assign the final values to the remaining cells in the slab i through their pointers. So no access to cell neighborhood is needed.

Since only the final state is transfered via pointer access and no catalyst substate is accessed it can be ignored (point 3). □

It implies from lemmas 4.1 and 4.2 that after the second pass all the cells will have their final value assigned and that means that the asynchronous cellular automaton converged.

An important property is that during second pass the CA executed on the slab will need only a single iteration to converge and the catalyst substate is

not needed as stated in the lemma 4.3. It also connects to the observation about possible automaton decomposition described in previous section. By employment of the decomposed automata the loading of the next slab during first pass can be partially overlapped with the computation of pointer only automaton as the catalyst substate can be released and catalyst substates for next slab can be loaded instead.

This generic algorithm becomes important later for out-of-core watershed transformation implementation (section 4.2.5).

4.2 Watershed Transformation

Watershed transformation is a region-based segmentation method. It is inspired by geology terms of catchment basins and watershed lines which divides landscape into domains of attraction for rain falling over the area, thus dividing the landscape into disjoint regions.

Alternative approach to watershed transformation interpretation is to imagine the landscape being immersed into a large body of water with holes in landscape local minima. Water leaking through different holes is colored by different color and color mixing is prevented by building dams on top of the ridges. These dams are our watershed lines and each differently colored lake will be the catchment basin.

It is often used as a first step (pre-segmentation) as it generates large number of the small regions. To get final result, wide range of methods is used to merge the regions into the final ones, which mark the segmented objects. As one such method can be considered minimal graph cut search on region adjacency graph, which will be discussed in chapter 5 (Kolomaznik et al. (2012), Stawiaski and Decenciere (2008)). Another often used approach is to compute intesity statistics for the regions and process them by trained classifier to divide them into categories of interest as can be seen in figure 4.3 and published in Horacek et al. (2015).

Overview of formal definitions and algorithms based on those different definitions will be given in the following sections together with their suitability for parallelization and efficient implementation on GPU. Terminology and definitions from Roerdink and Meijster (2000) will be used in the watershed transformation overview.

Available parallelization patterns used for watershed transformation are mostly based on dividing the image into tiles, distributing them between small number of threads/processes and combining the intermediate results (Bieniek and Moga (2000), Świercz and Iwanowski (2010)). These approaches are not well suited for implementations on GPU.

Alternative approaches to watershed transformation using cellular automata extensions defined in previous section 4.1 will be presented together with their speed comparison.

At the end, there will be shown an application of the streamed asynchronous cellular automaton with pointers – streamed two pass watershed transformation for out-of-core processing.

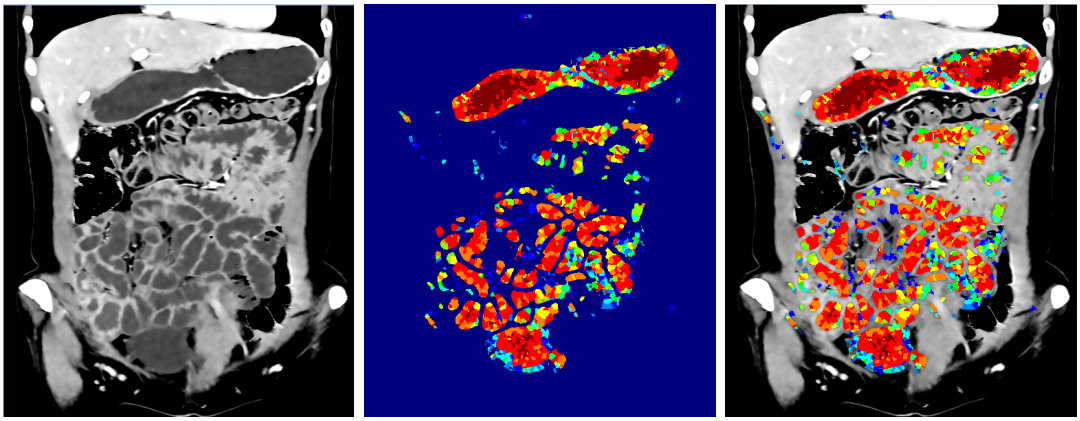


Figure 4.3: Application of the watershed transformation to segmentation of the small intestine in CT enterography – input data, probabilities of the regions to be inside the intestine (lumen), overlay of both

4.2.1 Terminology

Theory behind watershed transformation is based on extensions of basic geometry concepts, which starts with definition of *skeleton by influence zones* (Lantuejoul and Beucher (1981)), which is in essence an extension of the *n-dimensional Voronoi tesselation* (Roerdink and Meijster (2000)).

Voronoi tesselation (also Voronoi diagram or decomposition) is plane or space partitioning (depending on dimensionality of the problem) into disjoint regions. The partitioning is specified by positions of points in space called *seeds*. Each seed lies in a region formed by points whose distance to other seeds is greater than distance to the seed in the question.

If *Euclidean distance* is used for distance measurement the Voronoi regions (cells) will have shape of convex polygons (polyhedra in 3D) and the border between regions (points equidistant to two or more seeds) will consist of line segments (polygons in 3D).

The concept can be extended by letting the seeds to be point sets instead of isolated points and more general metric can be used as a distance measure.

Geodesic distance $d_A(a, b)$ between a and b within A, where $A \subseteq \mathbb{R}^d$ and $a, b \in A$ is an infimum of path length among all paths within A from a to b. Distance between $B \subseteq A$ and point $a \in A$ is defined as $d_A(a, B) = min_{b \in B}(d_A(a, b))$.

Definition 4.11 (Skeleton by influence zones - SKIZ)**.**
Influence zone of set B_i *(equivalent of Voronoi cell):*

$$iz_A(B_i) \quad = \quad \{p \in A | \forall j \in I - \{i\} : d_A(p, Bi) < d_A(p, B_j)\}$$

SKIZ as complement of all influence zones

$$SKIZ_A(B) \quad = \quad A - (\cup_{i \in I} iz(B_i))$$

Skeleton by influence zones (Lantuejoul and Beucher (1981)) is set of points which are equidistant to two or more sets B_i.

Definition 4.12 (Topographical distance).
Distance defined by length of shortest continuous curve across landscape between points p and q.

$$T_f(p,q) = \inf_{\gamma} \int_{\gamma} ||\nabla f(\gamma(s))|| ds$$

where the infimum is over all paths γ inside connected domain D and $\gamma(0) = p$ and $\gamma(1) = q$.

Using topographical distance in similar framework to the influence zones leads to the first definition of the watershed transformation (Meyer and Beucher (1990), Beucher and Meyer (1993)), Najman and Schmitt (1994), Meyer (1994)):

Definition 4.13 (Watershed transformation in continuous space).
Catchment basin for local minimum m_i:

$$CB(m_i) \quad = \quad \{x \in D | \forall j \in I \setminus \{i\} : f(m_i) + T_f(x, m_i) < f(m_j) + T_f(x, m_j)\}$$

where D is connected domain, f is image, I is index set (set of labels) and T_f is topographical distance defined for image f.
Complement of all catchment basins form the watershed lines:

$$Wshed(f) \quad = \quad D \cap (\cup_{i \in I} CB(m_i))^c$$

Let W be a special label, $W \notin I$. The watershed transform of f is mapping $\lambda : D \to I \cup \{W\}$, such that $\lambda(p) = i$ if $p \in CB(m_i)$, and $\lambda(p) = W$ if $p \in Wshed(f)$.

Continuous definition is useful to get a rough idea, how the watershed transformation should work.

Discrete case

In order to design an algorithms for watershed transformation the problem must be discretized first. Several discrete definitions of the watershed transformation exist, which deal differently with problems, which would come with a naive approach to discretization of the definition 4.13.

All the different formulations of watershed transformation would use the following definition of digital image:

Definition 4.14 (Digital image).
Digital image is triplet $G = P(D, E, f)$, where D is image domain and (D, E) is a graph representing relations between image elements (connectivity) and $f : D \to A$ where A is set with linear ordering.

One of the major problems for discrete watershed transformation are non-minima constant regions, the so called *plateau problem*. An image, which does not contain such regions will be called *lower complete*:

Definition 4.15 (Lower complete image).
Let $M = \bigcup_i m_i$, where $m_i \subseteq D$ are local minima of f. Image is lower complete if for each $p \in D \wedge p \notin M$ exist neighbor q, $(p, q) \in E$, such that $f(q) < f(p)$.

Images which are not lower complete can cause a trouble for some algorithms computing watershed transformation. Because according to the most of the definitions, watershed transformation is a morphological operation and as such it does not depend on the actual values of the image elements – it depends only on their ordering.

So one way to prevent *plateau problem* is *lower completion* of the input image by modifying image values in a way that non-minima plateaus are no longer present in the image and element ordering in the rest of the image remains the same. More about lower completion can be found later in this section.

Definition 4.16 (Discrete topographical distance)**.**
The lower slope $LS(p)$ of image f at an image element p, is defined as maximal slope leading to any of its neighbors of lower altitude:

$$LS(p) = \max_{q \in N(p) \cup \{p\}} \left(\frac{f(p) - f(q)}{d(p,q)} \right)$$

where $N(p)$ is set of neighbors of an element p and $d(p,q)$ is the distance between elements p, q (length of the edge). If $q = p$ the expression following the MAX-operator is defined to be zero. Thus for elements whose neighbors are all higher the lower slope is zero.

The cost for walking from pixel p to neighboring pixel q is defined as:

$$cost(p,q) = \begin{cases} LS(p) \cdot d(p,q) & iff(p) > f(q) \\ LS(q) \cdot d(p,q) & iff(p) < f(q) \\ \frac{1}{2}(LS(p) + LS(q)) \cdot d(p,q) & iff(p) = f(q) \end{cases}$$

The topographical distance along a path $\pi = (p_0, \ldots, p_l)$ between $p_0 = p$ and $p_l = q$ is defined as:

$$T_f^{\pi}(p,q) = \sum_{i=0}^{l-1} d(p_i, p_{i+1}) cost(p_i, p_{i+1})$$

The topographical distance $T_f(p,g)$ between p and q is minimum topographical distance along all possible paths between p and q.

This definition can be then used to define:

Definition 4.17 (Discrete watershed transformation by topographical distance)**.**
Let $(m_i)_{i \in I}$ be the collection of minima of lower complete image f. The basin $CB(m_i)$ of f is defined as:

$$CB(m_i) = \{p | \forall j \in I \backslash \{i\} : f(m_i) + T_f(p, m_i) < f(m_j) + T_f(p, m_j)\}.$$

Watershed lines can be found in the same manner as in definition 4.13 as a complement to the catchment basins. Algorithm 8 represents one approach to watershed transformation according the definition 4.17 (Meyer and Beucher (1990), Beucher and Meyer (1993), Najman and Schmitt (1994), Meyer (1994)), .

Another possible watershed transformation definition is based on the absolute ordering of the image elements. It is the formal description of region flooding through the local minima (Vincent and Soille (1991)):

Algorithm 8: Watersheds using topographical distance

Initialize *dist* for minimas by $im[m_i]$ and rest by ∞.
Each minima has different label.
while $V \neq \emptyset$ **do**
  $u = GetMinDistance(V)$
  $V = V - \{u\}$
  for all $v \in V$ with $(u, v) \in E$ **do**
    if $dist[u] + cost[u, v] < dist[v]$ **then**
      $Updatedist[v]$
      $lab[v] = lab[u]$
    else if $dist[u] + cost[u, v] = dist[v] and lab[v] \neq lab[u]$ **then**
      Mark v as WShed
    end if
  end for
end while

Definition 4.18 (Discrete watershed transformation by immersion)**.**

$$\begin{aligned} X_{h_{min}} &= \{p \in D | f(p) = h_{min}\} = T_{h_{min}} \\ X_{h+1} &= MIN_{h+1} \cup IZ_{T_{h+1}}(X_h) \end{aligned}$$

- X_i *is set of catchmen basins on level* i *and lower.*
- MIN_i *are local minimas on level* i*.*
- IZ *are influence zones of sets.*
- T_i *are connected threshold components on level* i *and lower.*

The definition 4.18 is the usual choice for the CPU implementations (Meyer and Beucher (1990), Beucher and Meyer (1993)). Parallelization of these algorithms is problematic as the procedure is usually driven by priority queue. The flooding based watershed transformation defined as cellular automaton was published by Noguet (1997) and used on specialized hardware (VLSI). The flooding option does not scale well for large number threads.

Better option for future parallelization offers watershed transformation based on local condition:

Definition 4.19 (Discrete watershed transformation based on local condition)**.** *For any image without plateaus, a function* L *assigning a label to each image element is called a watershed transformation if following conditions hold:*

1. $L(m_i) \neq L(m_j) \forall i \neq j$*, with* $\{m_k\}_{k \in I}$ *the set of minima of* f
2. *for each pixel* p *with* $\Gamma(p) \neq \emptyset, \exists p' \in \Gamma(p)$ *with* $L(p) = L(p')$

where $\Gamma(p)$ *is the set of lower neighbors of the element* p*.*

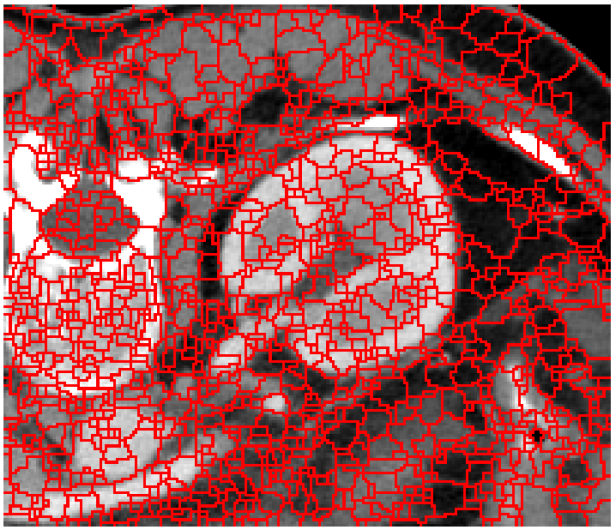


Figure 4.4: Watershed transfromation applied on image without strong denoising – lots of regions generated

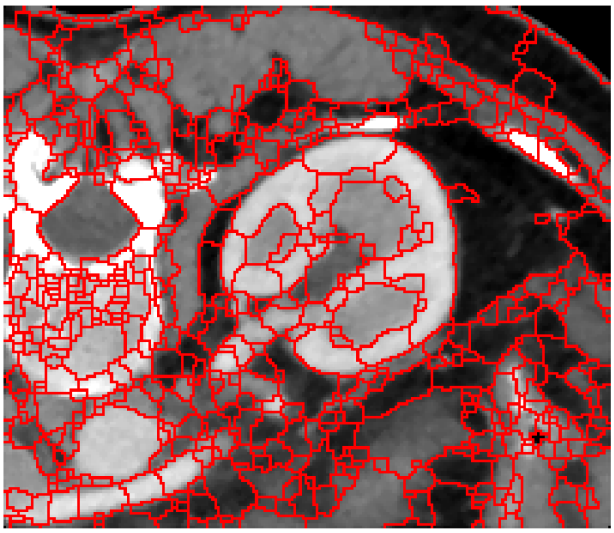


Figure 4.5: Watershed transformation applied on denoised image reduces number of generated regions

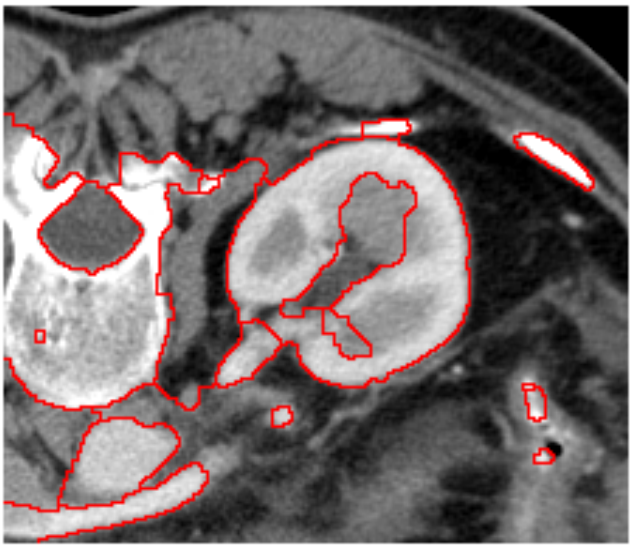


Figure 4.6: Watershed transformation from markers. Markers were generated by thresholding and cleaned up by morphological operations. It significantly reduces the number of generated regions.

Explicit and Implicit Watershed Lines

The removal of *SKIZ* lines and watershed lines in continuous case is a little problematic since there is no criterion for choosing between adjacent influence zones (catchment basins). But in practice it does not matter and even random assignment will work because the *measure* of these point sets is zero.

The watershed definitions, which do not use explicit watershed lines will be used in the rest of the chapter. Watershed lines can possibly occupy large portion of the processed image and thus distorting any statistics used on the regions. Also, it is more convenient to use implicit watershed lines for parallel algorithms as the local rules cannot decide easily if the image element should be marked as watershed line.

Watersheds from Markers

Important variant of watershed transformation is watershed transformation from markers (Meyer and Beucher (1990), Beucher (1994)). Instead of searching for local minima the algorithm accepts set of region masks which specify *watershed markers*. Flooding or searching for shortest topographical distances then uses these marker regions.

Marker regions represent one approach to limit number of the regions in final transformation and thus preventing oversegmentation. Regions with weak borders and with different labels in standard watershed transformation are merged

when not separated by different markers. In the figures 4.5 and 4.6 can be seen the difference between watershed from local minima and from markers generated by thresholding to find distinct regions.

In Roerdink and Meijster (2000) is shown that standard watershed transformation and watershed transformation from markers can be converted between each other (using local minima as markers and by forcing local minima in marker positions). So in the rest of the chapter only standard watershed transformation using local minima will be considered.

Algorithm 9: Watershed from markers segmentation algorithm

```
Initialize set of marker positions
Neighbors of marker pixels are put to priority queue (gray-level is the
priority)
while priority queue not empty do
  Pop pixel P with the highest priority.
  if already labeled neighbors of P have same label then
    Use same label for P.
  end if
  All non-labeled neighbors of P are put to priority queue.
end while
```

Lower Complete Image

As stated in the previous sections some of the watershed transformation formulations require lower complete image (only allowed plateaus are local minima) as its input (definition 4.15).

But in practice that requirement is often not met, so the input must be preprocessed and all non-minima plateaus eliminated.

Disadvantage of this strategy is that it is sensitive not only on the size of the input, but also on the values in the image and radii of the non-minima regions. Elements from these plateaus cannot be processed all in parallel, because each element is updated when its distance to the plateau border is known.

It also makes a worthwhile to observe what actually causes the presence of the non-minima plateaus. When the watershed transformation is applied to the gradient magnitude image, then the largest plateuas would be formed from the zero regions – homogeneous regions in the original. These zero regions would form the largest portion of the local minima.

The non-zero plateaus are then in the areas of the constant slope angle (in the original image). These non-trivial regions of constant slopes are often formed by less pronounced or blurred edges.

Here comes the questions whether the standard approach to lower completion of an image is *always* the desired one, because in the standard lower completion algorithm these regions will be equaly divided between neighboring watershed regions instead of having them in one region. If this behavior is not desired or not required, different approach can be used. Solution which builds the region border on the boundary of these non-minima plateaus is introduced in the section 4.2.3.

One small note must be made about images containing elements represented by floating point numbers. In such situation the chance of creating large non-minima plateaus is much lower than for the gradient image with integer elements due to missing rounding of the values and slight fluctuations, which are not present in the integer images.

4.2.2 Distance Based Watershed Transformation

Implementation of the watershed transformation based on topographical distance (definition 4.16) is better suited for parallelization than the watershed transformation by immersion, which uses priority queue in its core thus requiring serious modifications to be parallelizable. Topographical distance is defined on nodes (image elements – pixels, voxels), so there is no explicit global condition and distances can be computed for each node separately. Another advantage of this definition is that instead of computing distance to local minima region, any other region set can be used instead of local minima, thus making it *watershed transformation from markers*.

Distance based watershed transformation algorithms need to compute shortest distance to local minimum (or marker) for each image element. Several algorithms for shortest distance between nodes in graph can be found in literature. The requirement for the cellular automata based computation (and massively parallel computation in general) is that it is based on local operations only.

This applies for Bellman-Ford algorithm (Bellman (1958)), which keeps updating the distances between graph nodes as long as it can locally decrease the distance by chosing different path.

Authors of Kauffmann and Piche (2008) used this idea to implement synchronous a cellular automaton computing watershed transformation on GPU.

Cellular Automaton 2: Watershed based on distance
State: (v, l, d) – value as catalyst state, label (id) of the closest local minimum and the distance to that minimum.

Grid: Regular lattice – each cell represents image element.

Neighborhood: Reflect desired connectivity of the labeled regions. $|N| = n$

Initial grid state: cells from local minima have their labels assigned with distance set to zero. Rest of the cells have id set to *unknown* and distance set to infinite.

Cellular automaton rule 4.3 (Watershed based on distance.)**.**

$$f_{wd}((v_0, l_0, d_0) \dots (v_n, l_n, d_n)) = \begin{cases} (v_0, l_i, d_i + v_0) & \text{if } \exists i : d_i < d_0 \wedge \nexists j : d_j < d_i \\ (v_0, l_0, d_0) & \text{else} \end{cases}$$

The automaton keeps updating the shortest distance and because all distances are zero or positive, the process is finite (proof of Bellman-Ford algorithm correctness).

The original automaton rule definition from Kauffmann and Piche (2008) is for synchronous cellular automaton.

It can be easily shown that the rule can be applied even for *asynchronous cellular automaton with hidden updates*. Bellman-Ford algorithm does not require order in which the distances to nodes are updated. It does not have an influence on the correctness of the algorithm, it influences only the speed of convergence. It means that asynchronous computation is not a problem.

The rule never creates invalid state (wrong distance or label), so even the hidden updates operate on the valid input, although more optimal path could be already found in different block, making some of the hidden updates unnecessary, but never incorrect, thus making the rule feasible for asynchronous automaton with hidden updates.

But as mentioned the order of updates matter when considering speed. Too many iterations for hidden updates may cause slowdown, because too much time may be spend on updates using old data, as can be seen in method comparison (figure 4.12).

4.2.3 Watershed Transformation by Steepest Descent

Watershed transformation based on the steepest descent (definition 4.19) is defined only for the lower complete image (definition 4.15).

Cellular automaton solving the steepest descent based watershed transformation requires only the values from the input image as a catalyst substate and label as another part of the cell state.

Cellular Automaton 3: Watershed by steepest descent

State: each cell state (v, l) consist of two substates. Value $v \in \mathbb{R}$ and label $l \in \mathbb{Z}$. The value v comes from the input image on which the watershed transformation is applied and as such it is a *catalyst substate*. Label l is the temporary label of the cell (image element) and final region label after convergence.

Grid: Regular lattice – each cell represents image element.

Neighborhood: Reflect desired connectivity of the labeled regions. $|N| = n$

Initial grid state: Unique label for each element (assigned via scanline traversal, spacefilling curve, etc.).

Cellular automaton rule 4.4 (Watershed by steepest descent)**.**

$$f_{descent}((v_0, l_0) \dots (v_n, l_n)) = \\ = \begin{cases} (v_0, l_i) & \text{if } \exists i : v_i < v_j, \forall j \neq i \in (0, \dots, n) \\ (v_0, l_u) & \text{if } \min_i(v_i) = v_0 \wedge \exists u : l_u < l_v, \forall u \neq v \in (0, \dots, n) \\ (v_0, l_0) & \text{else} \end{cases}$$

The rule assigns the label of the lowest neighbor or the lowest label of the neigbor on the same level (in case of local minimum plateau). As for all methods, which divide the image into disjoint regions, equivalence relation is defined by this rule. The label reassignment can be accompanied by the merge step of the *union find* global state (as was used for connected component labeling). That also

means that asynchronous cellular automaton can and will be used in combination with rules 4.4, 4.5.

Cellular automaton rule 4.5 (Watershed by steepest descent with global state.)**.**

$$[f,g]_{descent\ global}((v_0,l_0)\dots(v_n,l_n),u_t) =$$
$$= \begin{cases} [(v_0,l_i),(l_0 \equiv l_i)] & if\ \exists i : v_i \leq v_j, \forall j \neq i \in (0,\dots,n) \\ [(v_0,l_u),(l_0 \equiv l_u)] & if\ \min_i(v_i) = v_0 \wedge \exists u : l_u < l_v, \forall u \neq v \in (0,\dots,n) \\ [(v_0,l_0),\emptyset] & else \end{cases}$$

Where $f_{descent\ global}$ is cell update rule, $g_{descent\ global}$ is global state update rule and u_t is global state (union-find) in generation t.

The steepest descent direction can be also used for cellular automaton with pointers, where the pointer is updated until it leads to the local minimum.

It is also possible to define such rule that it match the definition 4.10 about stable pointer paths:

Cellular Automaton 4: Watershed by steepest descent

State: substates (v,l,p) . Value $v \in \mathbb{R}$, label $l \in \mathbb{Z}$, pointer $p \in \mathbb{Z}$. v and l are interpreted in the same way as in previous cellular automata.

Grid: Regular lattice – each cell represents image element.

Neighborhood: Reflect desired connectivity of the labeled regions. $|N| = n$

Initial grid state: Each local minimum region has unique label. Rest of the cells have label unassigned. Pointer in each cell is a self-reference.

Cellular automaton rule 4.6 (Watershed by steepest descent with pointers.)**.**

$$f_{descent\ pointers}((v_0,l_0,p_0)\dots(v_n,l_n,p_n).(v_{p_0},l_{p_0},p_{p_0}) =$$
$$= \begin{cases} (v_0,l_0,p_{p_0}) & if\ p_0 \neq this \wedge l_{p_0}\ is\ not\ valid \\ (v_0,l_{p_0},this) & if\ p_0 \neq this \wedge l_{p_0}\ is\ valid \\ (v_0,l_0,p_i)) & else,\ so\ that\ v_i \leq v_j, \forall j \neq i \in (0,\dots,n) \end{cases}$$

The first two options follow the pointer path until valid label is found. The third option is either a no-operation in case there is no lower neighbor (local minimum with valid label), or it assigns pointer along the way to the minimum by following steepest descent.

Rule Decomposition

As was stated in the section 4.1.7, rules with stable pointer search path can split into two separate rules and two different automata will in sequence process the input with the same result.

If this decomposition is applied on rule 4.6, the two complementary rules 4.7 are created – one for the first iteration using only the neigboring cells and second for the rest of the iterations using only pointer accesses. The advantage of the separated rules is that during computation, each rule needs to do lower number of memory accesses and thus increases the speed of the computation.

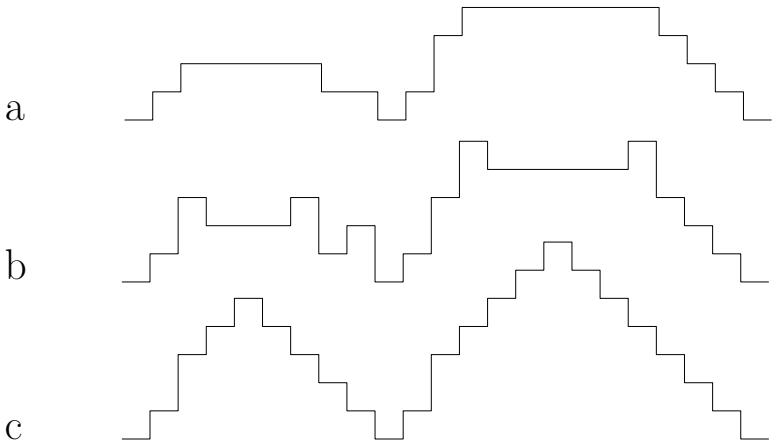


Figure 4.7: Traditional lower completion and alternative lower completion. a) the input data with non-minima plateaus, b) lower completion by border value increment, c) traditional approach, which continualy rises inner regions of the plateaus

Cellular automaton rule 4.7 (Watershed by steepest descent with pointers – decomposed.)**.**

$$f_{descent\ pointers}((v_0,l_0,p_0)\dots(v_n,l_n,p_n).(v_{p_0},l_{p_0},p_{p_0}) = \\ = \quad (v_0,l_0,p_i)) \quad v_i \leq v_j, \forall j \neq i \in (0,\dots,n) \quad .$$

$$f_{descent\ pointers}((l_0,p_0)\dots(l_n,p_n).(l_{p_0},p_{p_0}) = \\ = \begin{cases} (l_0,p_{p_0}) & if\ p_0 \neq this \wedge l_{p_0}\ is\ not\ valid \\ (l_{p_0},this) & if\ p_0 \neq this \wedge l_{p_0}\ is\ valid \end{cases}$$

Alternative Lower Completion

The standard algorithm for lower completion (Meyer (1994), Roerdink and Meijster (2000)) of the input image removes non-minima plateaus by introducing gradient into those areas.

A different approach to non-minima plateau handling was chosen. As shown in section 4.2.3 the area occupied by non-minima plateaus is usually small, so by introducing new watershed region for each non-minima plateau would not do a big harm. It is done by changing non-minima plateaus into local minima plateaus by increasing value of plateau border elements by some ϵ value thus preventing the cellular automaton to follow the down-slopes from inner area of the plateaus. Similar strategy was used in Nikodem (2009) for 2D watershed transformation. The idea is illustrated by figure 4.7.

To make this approach fit into the presented framework, it can be computed in single iteration of cellular automaton with rule:

Cellular automaton rule 4.8 (Lower completion rule.)**.**

$$f_{plateau}(v_0,v_1\dots v_n) = \begin{cases} v_0+\epsilon & if\ (\forall i : v_i \leq v_0) \wedge (\exists j,k : v_j < v_0 \wedge v_k = v_0) \\ v_0 & else \end{cases}$$

The rule 4.8 works as expected in 1D case. But in higher dimensions the rule modify values of image elements, which are not considered as plateaus. Curves

(in 2D and 3D) and surfaces (in 3D) of constant values and without neighboring image elements of higher values will also have their values updated.

It is possible to detect these pixels/voxels and prevent their update (it will be called the *ideal lower completion*), but it would unnecessarilly complicate the rule. Instead it will be shown that even if the rule remain as it is right now, the changes in no-plateau regions will not affect the result of the watershed transformation.

By studiing the influence of the lower completion on the watershed transformation based on steepest descent, it can be seen that label updates of the unnecessarilly modified pixels/voxels after the non-ideal lower completion will not change, because it depend solely on the direction to the smallest neighbor and not on value of the modified pixel/voxel (smaller must neigbor exist from 4.8.

So the only negative influence would be when some neighbor v of affected pixel/voxel u would have its label updated differently than in ideal lower completion without changes in non-plateau elements. This can happen in two ways - either u and v would share the same label after ideal lower completion and don't share it after non-ideal completion or vise versa.

Two neighboring elements u and v share their labels in two situations:

1. u is the smallest neighbor of v
2. u has same value as v and both are part of the same local minimum region.

If 1 applies to u, v after ideal lower completion it would also apply after non-ideal lower completion, because it means that u has bigger neigbor and because of that it will not be updated by rule 4.8.

Similar result is obtained by analysis of the situation 2 – u is local minimum after ideal lower completion, so it means it does not have smaller neighbor, and as such it will not be updated by rule 4.8.

So that means that the simpler rule 4.8 will be sufficient for lower completion of the input image and does not influence the result of the watershed transformation in comparison to the ideal algorithm for lower completion.

4.2.4 Steepest Descent with Plateau Handling

Similar idea as is behind the alternative lower completion (section 4.2.3) can be used to design rule for watershed transformation which does not need lower completion of the input image. This can be important factor when modification of the input image is not an option (ie. not enough GPU memory for the input copy).

As described in section 4.2.3 a precaution must be made to prevent merging of different watershed regions through non-minima plateaus. Instead of modifying input image an indicator image can be created. Each indicator image element is either *true* if a neighbor of smaller value exits or *false* otherwise. Advantage over copying of the input image would be in the size of the indicator image which can use 1 bit per image element instead of 32 bits for float input image. That can be an important reason to choose this approach instead of the lower completion, even when the automaton 4.9 is slower than 4.5.

The new watershed rule will work similarly to the steepest descent rule. But behaves differently when the processed element is part of the plateau region. If the element does not have a smaller neighbor, it prevents merging with a neighbor of the same value marked by the indicator image. This condition creates a non-minima plateau border of image elements which will not be merged with the inner parts of the plateau. These border voxels all have at least one smaller neighbor (because they were marked in the indicator image) so they will merge with their smallest neighbor. And because of that it can be seen that it did not change any relations between image elements and the same result is obtained from this watershed transformation rule as was obtained from steepest descent one with alternative lower completion (rule 4.8).

Cellular automaton rule 4.9 (Non-minima plateau handling.)**.**

$$f_{plateau}((v_0,l_0,b_0),(v_1,l_1,b_1)\dots(v_n,l_n,b_n)) =$$
$$= \begin{cases} (v_0,l_i,b_0) & \textit{if } \exists i : v_i < v_j, \forall j \neq i \in (0,\dots,n) \\ (v_0,l_u,b_0) & \textit{if } \min_i(v_i) = v_0 \wedge \exists j : l_j < l_0, b_j = \textit{false}, \forall j \in (1,\dots,n) \\ (v_0,l_0,b_0) & \textit{else} \end{cases}$$

The state (v,l,b) is value, label and border indicator (true/false).

4.2.5 Automaton with Pointers and Stream Processing

Watersheds based on steepest descent can be also computed by cellular automaton with pointers. Instead of merging labels in the CA rule the pointer is updated so it leads to the target of the smallest neighbor.

Pointer manipulation is based only on the state of the smallest neigbor, because it is comparison between values of catalyst substate this relation is stable through all the iterations. It means that the pointer is updated only when the pointer in that spacific smallest neigbor is updated. These properties match with the definition of the CA rule generating *stable pointer paths* and as such it can be used for computation by the streaming algorithm presented in section 4.1.7.

The final algorithm can be viewed as generalization of the algorithm presented in Hucko and Sramek (2012). Authors there used the algorithm for an out-of-core computation when they loaded slices from hard drive and at the end of each step stored the computed slice back to the hard drive.

Amount of GPU memory is usually smaller than amount of available RAM, so the streamed computation is also viable option for out-of-core computation in a sence that the whole dataset resides in RAM and only the thick slices are sent to GPU for calculation and then back to main memory. This approach is suitable even for extremely large datasets (larger than available RAM). The largest possible slabs would be loaded from hard drive to memory and these in turn again sliced and then sent to GPUs. Making it two level "slicing" algorithm.

When comparing the bandwith of the presented algorithms which is counted in gigabytes per second, the expected bottlenecks are reads and writes from hard drives.

4.2.6 Benchmarks

Set of artificial datasets was created. Images of increasing size were generated by randomly placing seeds inside the volume and computing distance field for those seeds (watershed transformation is equivalent to rasterized Voronoi diagram of the seed set). Several variants were generated for each image size, differing in number of the seeds and consequently in average size of watershed regions.

The artificial data were generated in the same way in the Chabardès et al. (2016), where authors presented a parallel O(n) algorithm for watershed transfromation and tested it on computing cluster with 8 core Intel(R) Xeon(R) CPU E5-2650 v2 processor. The computation times they achived were used as ground truth for speed comparison with the algorithms presented in this thesis. Second reference is the GPU oriented algorithm presented in Kauffmann and Piche (2008), which is formally a synchronnous cellular automaton.

All cellular automata variants were tested on two configurations:

- NVIDIA GTX 1060 6GB, Intel i7-3770 CPU 3.40GHz 16GB, no-swap
- Amazon Cloud – p3.2x instance with NVIDIA Tesla V100, 16GB

The tested algorithms were:

- Synchronnous automaton (definition 4.1), distance based rule 4.3 (Kauffmann and Piche (2008))
- Asynchronnous automaton (definition 4.7), distance based rule
- Asynchronnous automaton, steepest descent with global state – rule 4.5
- Asynchronnous automaton, steepest descent with pointers – rule 4.6
- Asynchronnous automaton, decomposed steepest descent with pointers – rule 4.7

As was discussed in previous sections, the major problem of the distance based watershed transformation is that number of cellular automata iterations depends on the size of the watershed regions. This fact is illustrated in figure 4.8, where it can be seen that the computation time increases quicker for the distance based methods in comparison to the steepest descent methods.

As can be seen in figures 4.9 and 4.11, all the cellular automata extensions brought a significant improvement in comparison to basic synchronous distance based cellular automaton.

As can be seen from the graphs, all the algorithms scale well with the increasing input, except for the big sizes, where the computational speed is hindered by continuous swaping between device and host memory (unified memory), because the amount of memory needed exceeds the amount of memory available on GPU.

Different algorithms have different demands on the amount of required memory. The worst one is the steepest descent with global state, which needs additional memory for the union-find global state, so it suffers most from the insufficient memory. Another reason why the algorithms behave differently is their different memory access patterns and number of memory accesses in global.

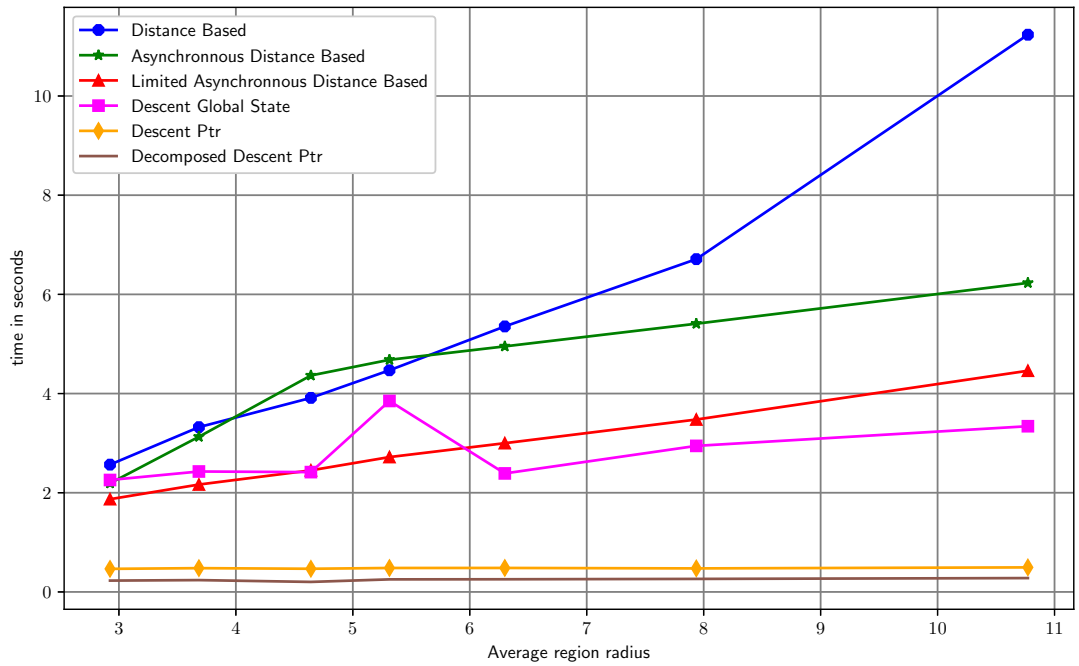


Figure 4.8: Processing of the 500^3 images with increasing size of the watershed regions.

The ratio between speeds of certain operations is slightly different on the Tesla card (memory access vs. numerical expression evaluation, etc.). As a result the distance based algorithms can compete with the steepest descent based for small region sizes (figure 4.10), but for the larger regions (figure 4.11) the linear dependence on the region radius manifest itself in longer computation times.

Second set of testing datasets is group of real medical CT images and one artifitial dataset containing a single big region. As a ground truth was taken watershed transformation from software package ITK (Kitware (2016)). Again best computation times (figure 4.12) were achieved byt the decomposed cellular automaton with pointers which brought $100x$ speedup in comparison to ITK version and $4 - 5x$ speedup in comparison to synchronnous distance based automaton (Kauffmann and Piche (2008)).

Hardware used for measurement was again: NVIDIA GTX 1060 6GB, Intel i7-3770 CPU 3.40GHz 16GB, no-swap.

4.3 Conclusions

The presented cellular automata formalism with all its extensions proved to be very useful tool for massively parallel algorithm design. It allowed implementation of a generic framework (part of CUGIP library), which can be used for fast prototyping of algorithms defined in terms of cellular automata.

Application of presented concepts to watershed transformation, together with alternative non-minima plateau handling led to creation of several fast algorithms, which surpasses CPU implementations and even the previously published massively parallel methods. An important feature of the final algorithm is that it is not so sensitive to data content as the other methods.

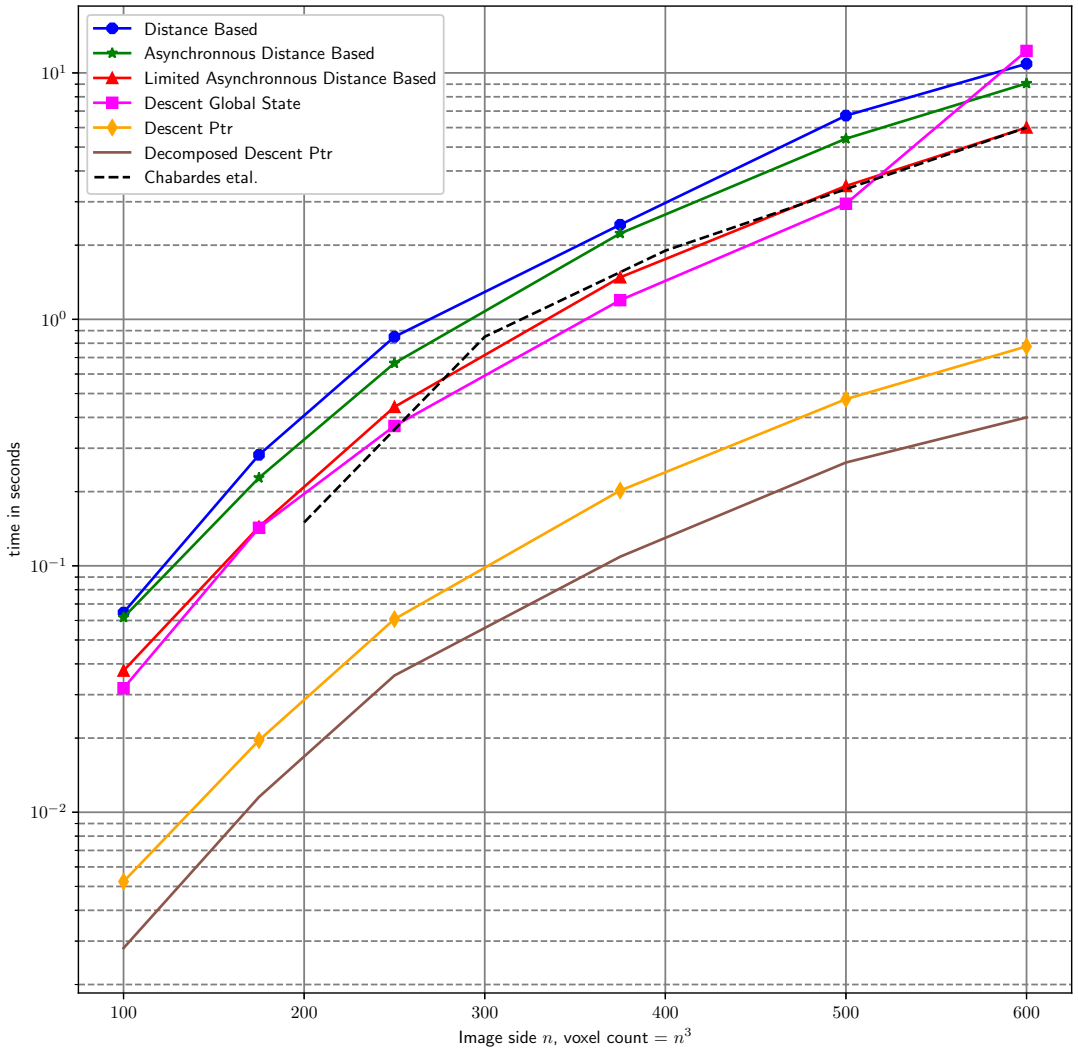


Figure 4.9: Test on the artificial data. Average region size is 2000 voxels. NVIDIA GTX 1060 6GB

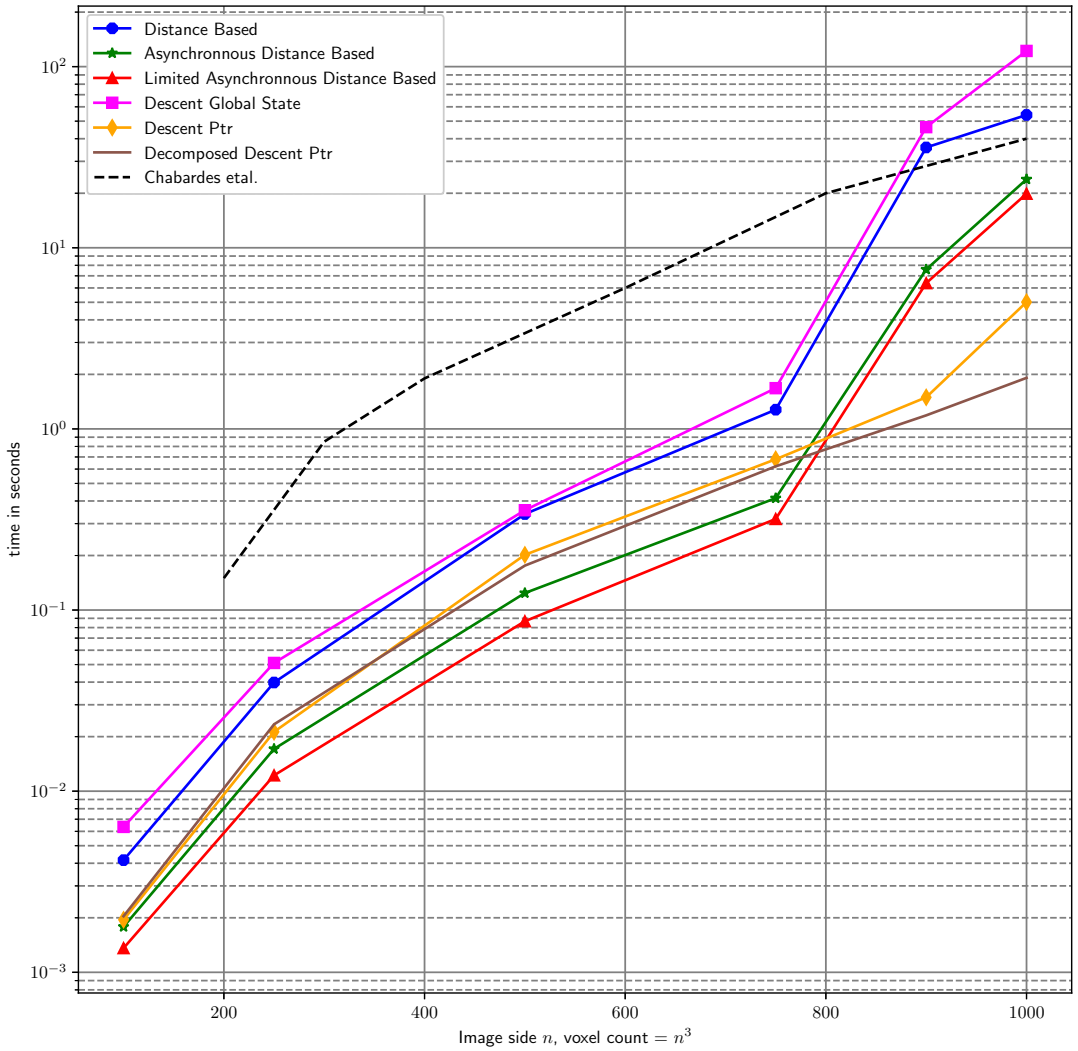


Figure 4.10: Test on the artificial data. Average region size is 500 voxels. NVIDIA Tesla V100, 16GB.

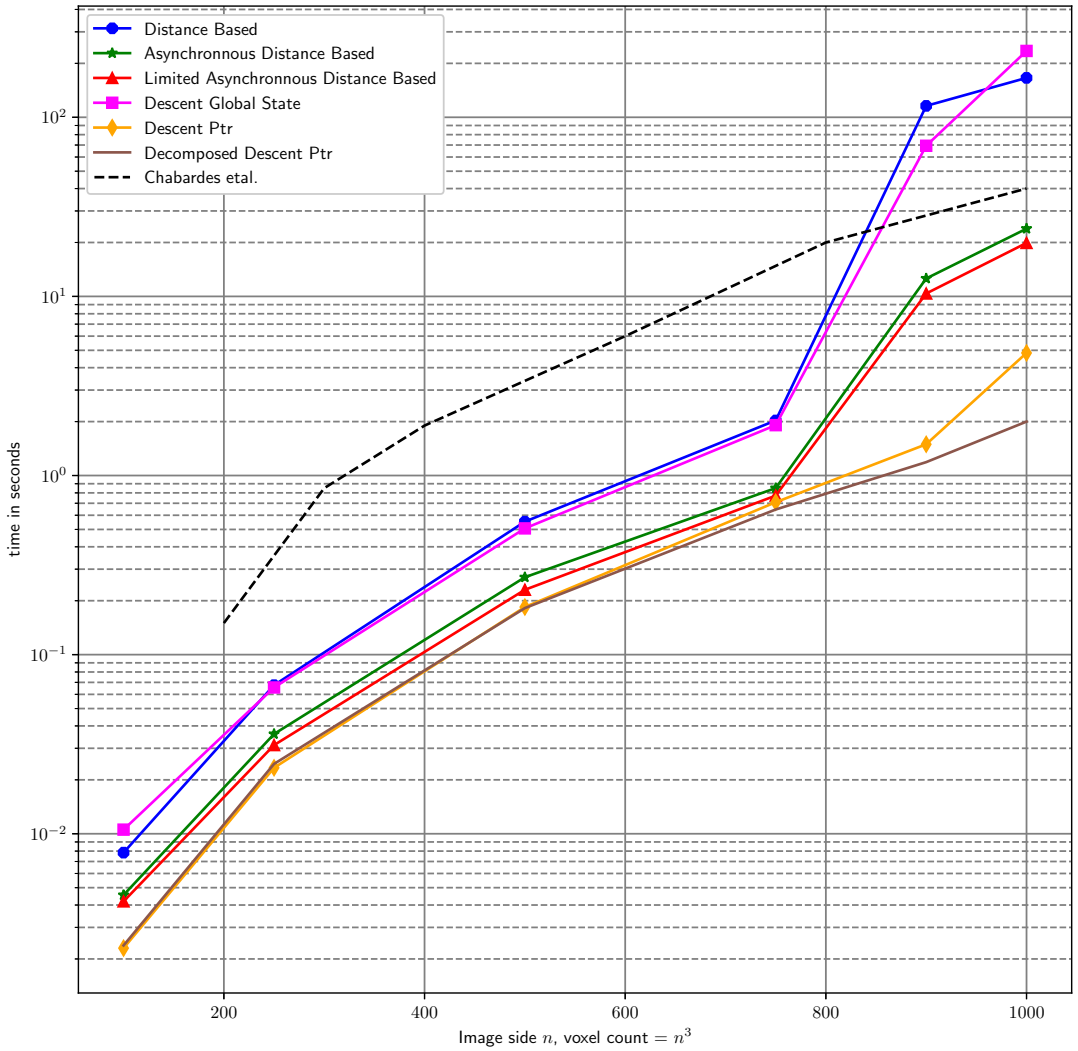


Figure 4.11: Test on the artificial data. Average region size is 20000 voxels. NVIDIA Tesla V100, 16GB

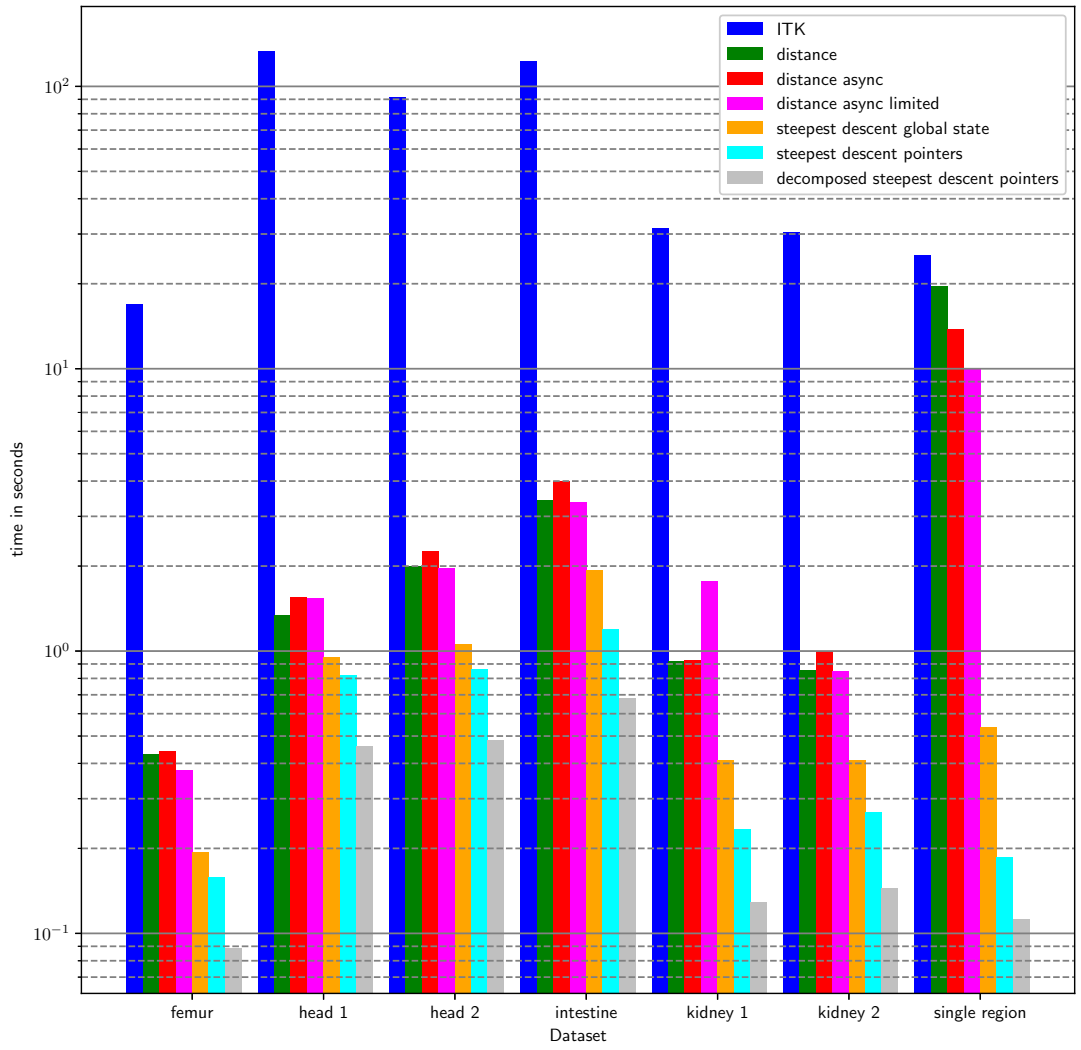


Figure 4.12: Speed comparison of different watershed transformations. ITK is CPU single threaded implementation. Synchronous distance based and its asynchrnous variants with unlimited hidden updates per iteration and limited (2) hidden updates per iteration. Synchronous steepest descent with pointers and synchronous steepest descent with union-find global state.

Table 4.1: Speed comparison of different watershed transformations, together with required cellular automata generations until convergence. ITK is CPU single threaded implementation. Synchronous distance based and its asynchrnous variants with unlimited hidden updates per iteration and limited (2) hidden updates per iteration. Synchronous steepest descent with pointers and synchronous steepest descent with union-find global state.

	ITK	distance		distance async		distance async limited		steepest descent pointers		steepest descent pointers2		steepest descent global state	
	time	time	iter.	time	iter.	time	iter.	time	iter.	time	iter.	time	iter.
femur	17.003	0.4316	17	0.4406	12	0.3775	12	0.1578	11	0.0883	11	0.1944	9
head 1	133.069	1.3363	14	1.5527	12	1.5361	12	0.8178	11	0.4579	11	0.9496	10
head 2	91.83	2.0064	18	2.2576	14	1.9597	14	0.8635	12	0.4835	12	1.0595	10
intestine	122.5	3.4392	20	4.0142	16	3.3541	16	1.1916	12	0.6792	12	1.9217	12
kidney 1	31.5196	0.9255	20	0.9276	12	1.7667	13	0.2334	11	0.1283	11	0.4094	10
kidney 2	30.45	0.8535	20	0.9964	15	0.8486	15	0.2675	12	0.1444	12	0.4105	10
single region	25.28	19.5585	202	13.7924	110	9.9625	111	0.1861	12	0.1116	12	0.5387	8

In the listing 4.1 can be seen an example how automaton for synchronnous steepest descent algorithm can be implemented in the framework.

Listing 4.1: Example of cellular automaton implementation.

```
struct WatershedSteepestDescentRule :
    WatershedSteepestDescentRuleBase
{
  using remove_reference = typename std::remove_reference<
      T>::type;

  // Generic rule implementation   works for different
      neighborhoods
  template<
       typename TNeighborhood,
       typename TConvergenceFlag>
  CUGIP_DECL_DEVICE
  auto operator()(
       int aIteration,
       TNeighborhood aNeighborhood,
       TConvergenceFlag aConvergenceState)
                  > remove_reference<decltype(aNeighborhood
                      [0])> const
  {
    //input, label of the current cell
    auto value = aNeighborhood[0];
    // locate lowest neighbor index
    int index = findNeighborForMerge(aNeighborhood);
    if (index !=  1 &&
        get<1>(value) != get<1>(aNeighborhood[index]))
    {
      // assign label of the lower neighbor
      get<1>(value) = get<1>(aNeighborhood[index]);
      // notify that algorithm is not yet finished
      aConvergenceState.signal();
    }
    return value;
  }
}

using Value = Tuple<float, int32_t>;

CellularAutomatonWithGlobalState<
          Grid<Value, tDimension>,
          MooreNeighborhood<tDimension>,
          WatershedSteepestDescentRule,
          ConvergenceFlag> automaton;
```

Chapter 5

Minimal Graph-cut in CUDA

Interactive 2D segmentation methods are available in image editing applications like Photoshop or Gimp for quite some time. On the other hand, interactive 3D segmentation approaches are not widely available (reasons discussed in the introduction). Applications providing 3D interactive or semi-automatic methods requiring a user input often fall back to 2D segmentation executed separately on slices and then combining the slice results back to 3D interpretation.

After publication of Boykov and Jolly (2000); Boykov and Kolmogorov (2004), minimum graph cut algorithms emerged as an increasingly useful tool which is currently used for solving wide range of computer vision problems.

One of the successful applications of the min-cut approach was image segmentation (Stawiaski and Decenciere (2008); Yi and Moon (2012); Kolomaznik et al. (2012)). Particularly important for this thesis was its application for interactive segmentation (Boykov and Jolly (2001)). Although the *real* interactivity (short computation times) was in past ensured only for 2-dimensional segmentation and/or relatively small images.

This chapter will focus on the design and implementation of the fast (parallelized) min-cut algorithm, which is suitable for 3D image segmentation. As voxel data tend to be large, the final algorithm should be able to compute the min-cut on the scaled down problem, where each graph node represents image subregion (computed by watershed transformation, or other superpixel generating algorithm) instead of single voxel. This means that the graph topology is not uniform and thus it limits the set of available approaches as lots of fast min-cut algorithms expect the graph to have grid-like topology (Jamriska et al. (2012)).

The presented algorithm is based on Goldberg's (push/relabel) max flow algorithm (Goldberg and Tarjan (1986); Cherkassky and Goldberg (1997)), which seemed to be the most easily parallelizable in comparison to other max-flow/min-cut algorithms. Also the parallelization scheme must be able to handle large number of threads, as the final implementation will be done for CUDA devices.

Modification of the classical relabeling scheme will be presented together with set of optimizations required by efficient implementation and an approximation of the search process, which lowers the computation time even more without serious sacrificing of the output precision.

5.1 Overview

Before dweling directly into the problem, a short terminology overview should be made.

Definition 5.1 (Graph cut)**.**
Given graph $G = (V, E, w)$ *where* V *is set of vertices and* E *set of edges and* $w : E \to \mathbb{R}$ *is a weight of an edge.*

Graph cut $C \subseteq E$ *is such set of edges that graph vertices* V *can be divided into two disjoint sets* $S, T \subseteq V$ *such that:*

- $\forall (u, v) \in C$: *either* $u \in S, v \in T$ *or* $u \in T, v \in S$
- $\forall (p, q) \in E \setminus C$: *either* $p, q \in S$ *or* $p, q \in T$.

Definition 5.2 (Minimal graph cut)**.**
Let weight of the graph cut C *be defined as* $w(C) = \sum_{e \in C} w(e)$*. Minimal graph cut* C_{min} *is such that for all valid graph cuts* C_i*:* $w(C_{min}) \leq W(C_i)$.

Definition of the graph cut can be extended by selection of two special vertices s, t:

Definition 5.3 ((s,t)-cut)**.**
For graph $G = (V, E, w)$ *and two vertices* $s, t \in V, s \neq t$*.* (s,t)*-cut* $C_{(s,t)} \subseteq E$ *is a graph cut that divides vertex set* V *into two disjoint sets* $S, T \subseteq V$*, such that* $s \in S$ *and* $t \in T$.

It is important to note that definitions 5.2 and 5.3 allow existence of multiple different minimal cuts in a graph. These cuts can have different sizes (edge count), but all must have equal sum of edge weights.

The algorithms presented in this chapter make use of *max-flow/min-cut theorem*, which was proven by Ford and Fulkerson in Ford and Fulkerson (1962). It states that minimal (s,t)-cut is equal to the maximal flow between s and t (source and sink). Also all edges of the cut are fully saturated in the flow network. This duality is often used to derive algorithms for the min-cut computation.

5.1.1 Segmentation

The segmentation based on min-cut search is done by building a graph for an input image. Where vertices represent either image elements (pixels, voxels), or disjoint subregions and graph edges model the positional relations between these nodes.

The algorithm is provided with information, which marks some of the graph vertices as foreground and some as background. Two special vertices – sink and source – are then added to the graph and all of the foreground marked vertices are connected to the sink and all background marked to the source. These newly added connections are usually called as *t-links* in literature. Original set of edges are called *n-links*. Schematic view can be seen in figure 5.1. Minimal graph-cut search is then computed on this extended graph. The two sets of vertices defined by the computed graph cut represent the foreground and background.

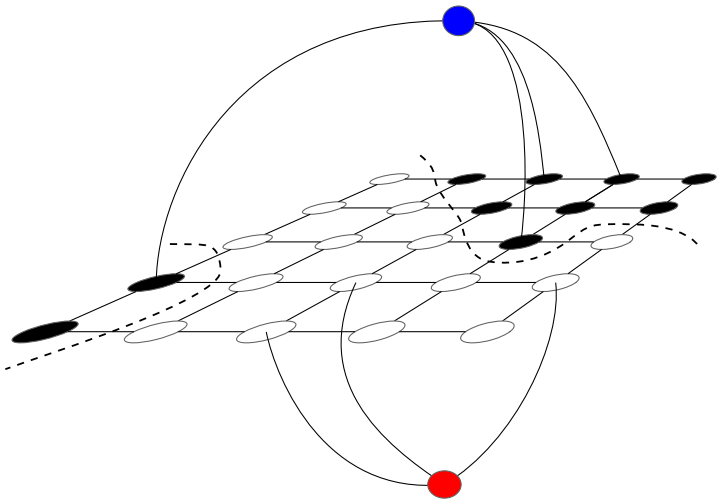


Figure 5.1: Graph cut scheme: Source (blue) is connected to subset of the background pixels. Sink (red) is connected to the subset of foreground pixels. Minimal cut (dashed lines) separates foreground and background regions.

5.1.2 CPU Graph-cut Implementations

Several algorithms for maxflow/min-cut algorithms can be found in literature on the graph theory and combinatorial optimization. But these algorithms were studied mostly outside of the scope of computer vision and image processing.

Most of the published algorithms use one of two major approaches. First is search for *augmenting paths* from source to sink presented by Ford and Fulkerson Ford and Fulkerson (1962). These algorithms usually differ in the means of finding the augmenting path which is then used to send flow to the sink. Dinic algorithm Dinic (1970) used breath-first search to find shortest possible augmenting paths in each step.

Boykov and Kolmogorov provided an experimental study (Boykov and Kolmogorov (2004)), where they compared available min-cut algorithms and developed a new one with improved performance for computer vision problems. Their algorithm is based on *augmenting paths* approach and it proved to be faster for large set of problems in comparison to Dinic algorithm and variants of Goldberg-Tarjan (Push-Relabel) algorithm.

The *Gridcut* algorithm published in Jamriska et al. (2012) is reimplementation of Boykov-Kolmogorov algorithm with a focus on cache-friendly memory layouts and compact datastructures. To achieve these goals the algorithm is specialized only for the grid graphs.

The grid-like organisation of vertices alleviate the need for explicit separate collection of edges. The additional edge information are stored together with each node in fixed-length array. Book keeping for augmenting paths is in datastructures, which are converted from *array of structs* into *struct of arrays* – typical optimization for cache friendly memory accesses.

Grid is also stored in blocked layout instead of scanline layout. Size of these blocks is same as cache line, so it increases the chance that by accessing a neighbor of some vertex data will be read from the same cache line.

The final algorithm outperforms the Boykov-Kolmogorov implementation both in speed and memory requirements, but price being paid is the limitation to grid-

like graphs.

In Xu et al. (2007) authors used graph-cuts as a tool to fix several flaws of the *active countours* segmentation, mainly the local minima problem. They alternate between active countours segmentation and graph-cut based segmentation. Results of each step is then used to initialize next iteration. Boundary generated by active contours is dilatated and two complementary regions are used as marker regions for the graph-cut segmentation. Minimal graph cut is then used directly as initialization of the active contours.

Parallel implementation of Goldberg's algorithm for multiple processors with shared memory was presented in Anderson and Setubal (1992). The main innovation of the presented approach was application of global relabeling heuristics, which were essential for good performance in practice. The heuristic is based on the BFS starting from sink and assigning distance to sink as a new label.

Another approach to parallelization and distributed computing was described in Strandmark and Kahl (2010). The graph-cut search problem was decomposed into smaller problems, which can be solved separately and their results then combined.

5.1.3 GPU Graph-cut Implementations

In Vineet and Narayanan (2008) authors presented several variants of parallel implementation of minimal graph-cut by push-relabel algorithm.

The paper was focused on segmentation in 2D images and graph-cut was computed on regular grid graph using pixel connectivity as graph edges.

First version they presented used atomic operations for all updates to prevent race conditions. The algorithm was divided into two kernels. First for push-pull step and second one for relabeling. Their algorithms used the same relabeling strategy as was used in the original Goldbergs version of the algorithm.

Second variant of the algorithm was devised to prevent the usage of atomic operations. The push-pull kernel was changed into push kernel, so that the flow is only sent from the excess vertices. The actual receive of the flow happens in the pull-relabel kernel, which also does the relabeling.

The grid graphs allowed the authors to exploit regular accesses to global memory by preloading almost all data needed in each thread block into shared memory, so neighborhood traversal of each vertex happens to be really fast.

They did not use any additional data structures to organize the progress of the algorithm because both kernels were always executed on the whole grid. This works well for moderately sized 2D images, because of the available computational power of modern GPUs, but causes worse computational complexity than standard version of the algorithm. This makes this approach unsuitable for large graph processing.

The last extension of their algorithm is based on observation that the flow changes mostly in the first few iterations and then number of "active" vertices drops rapidly. They extended the grid graph by *active bit* for each block and did computation only in active block for most of the iterations. The active bit during the step which happens only in every k-th iteration and processes all blocks.

More sophisticated approach was used by authors in Hussein et al. (2007). Their algorithm also exploits the topology of grid graphs to do the advanced

optimizations. Their implementation uses parallel BFS. They use implementation of BFS with $O(n^2)$ complexity, but present optimization which brings it down to $O(n)$ for grid graphs.

Parallel push was also divided into push and pull steps to prevent the need for atomic operations. Instead of using classical relabeling scheme they used a heuristic presented also in Goldberg and Tarjan (1986), which relabels the graph globaly by BFS starting from sink and assigning labels equal to distances to the sink. They use solely the BFS approach to set labels. By storing the visited vertices authors devise another heuristic for push step – the excess flow is pushed first from vertices with the highest label.

Hybrid approach to the parallel push-relabel algorithm was published in He and Hong (2010). They used again the BFS based global relabeling scheme and switched between CPU and GPU throughout the computation as the problem size changed.

5.1.4 Other Graph Algorithms for GPU

Key component of the graph-cut search algorithm, which will be presented later in this chapter, is the breath first search algorithm. Efficient implementations for GPU execution were published in several papers.

In Harish and Narayanan (2007) authors presented BFS and single source shortest path implementations. They did not use any queue of vertices to maintain the latest processed level. Instead they used a special flag for each vertex to mark it as visited, unvisited or frontier. In each step they executed the frontier processing on all vertices. This approach is efficient for smaller and medium sized graphs, but it worsens the theorethical computational complexity.

To achieve theoretical computational complexity $O(|V|+|E|)$ authors of the Luo et al. (2010) used implementation of hierarchical queue to manage active frontier. The hierarchical queue was used to prevent collisions when adding vertices in parallel from different threads.

The most extensive study on parallel BFS can be found in Merrill et al. (2012). Where they compared different approaches to warp divergence prevention during traversal of vertex neighbors.

5.2 Graph Topologies

When tuning the min-cut algorithm for special application (image segmentation in this case), an important factor is the actual topology of the constructed graph. The key decision is which image entities are represented by graph nodes and what relation defines the edges.

5.2.1 Grid Graph

The most direct mapping between an input image and graph constructed for min-cut search is by using image elements (pixels, voxels) as nodes of the graph and element connectivity is used for definition of the graph edges. The lattice of the image elements is thus transfered into the graph topology and grid-graph

is formed. Edge weights are usually chosen to reflect value difference between neighboring elements, controlled by the estimated noise deviation and weighted by actual distance between elements:

$$w_{p,q} = \exp\left(\frac{(I(p) - I(q))^2}{2\sigma^2}\right)\frac{1}{dist(p,q)} \tag{5.1}$$

If the graph cut search can be limited only to graphs with grid-like topology, the algorithm need to store only edge weights in the graph data structure, because edges are known implicitly from the grid topology description (grid size, type of neighborhood, dimensionality).

The fast implementation in Jamriska et al. (2012) uses the grid topology to optimize for caches. Same fact was also used in GPU minimal graph-cut search presented in (Hussein et al. (2007)).

5.2.2 Subregion Adjacency Graph

When using preprocessing step which divides input image into a set of regions $\{R_0, R_1, \ldots\}$, such that the segmented object (foreground) and background are unions of disjoint subsets of this region set.

Adjacency graph of these regions is then used as an input for the minimal graph-cut search. By definition, the adjacency graph can be constructed from the grid graph of image elements by repeated edge contraction, which merges all nodes from each subregion into one. Region adjacency graph is thus *induced minor* of the original grid graph.

In this case, the weight assigned to the edges must integrate information along common boundary of the two regions. One possible way (used in Stawiaski and Decenciere (2008)) is

$$w_{R_i,R_j} = \sum_{e_{p,q}, p\in R_i, q\in R_j} g(max(||\nabla I(p)||, ||\nabla I(q)||)) \tag{5.2}$$

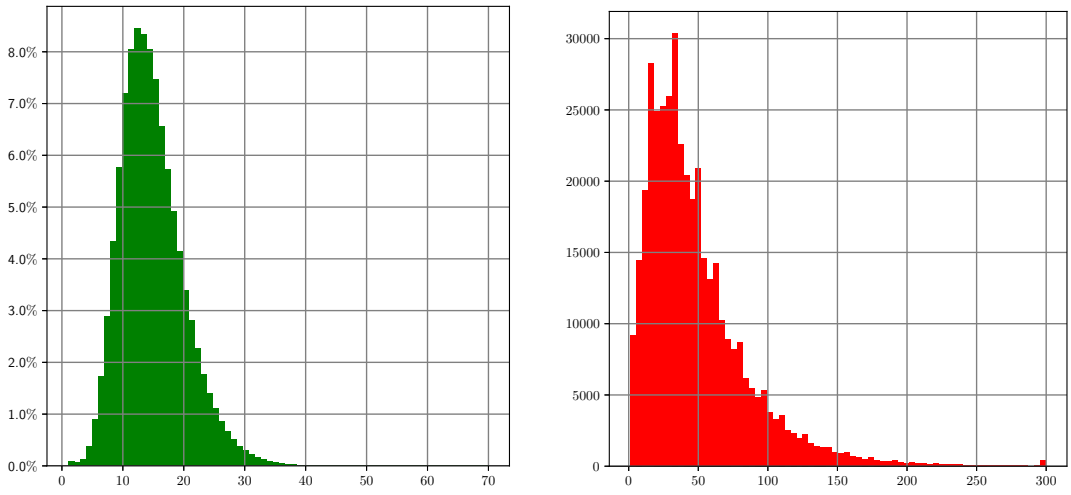
where R_i, R_j are regions in question, $e_{p,q}$ is an edge in original grid graph, $||\nabla I(p)||$ is image gradient magnitude at point p and $g()$ is strictly positive and decreasing function:

$$g(||\nabla I(p)||) = \left(\frac{1}{||\nabla I(p)||}\right)^k \tag{5.3}$$

where $k \in R^+$ is smoothing parameter, in Stawiaski and Decenciere (2008) authors used $k = 2$. Function $g()$ penalizes edges between regions which have boundary in are of big gradient (higher probability of cut).

As can be seen from figures 5.2 and 5.3, the graph build over the regions is significantly smaller than the original grid graph and as such does not require as much memory as the original grid graph.

By observation of subregions generated by oversegmentation methods (like watershed transformation), it can be noticed that the graph nodes of greatest degree are those corresponding to the biggest regions. Big regions have usually more neighbors than the small ones. These big regions are also those that will be most probably marked by the user if their position is incident to the area of interest. Because of this, the search for minimal graph cut happens in more or



(a) Vertex degrees

(b) Region sizes

Figure 5.2: Region statistics for femur dataset

less regular parts of the graph, with most of the high degree nodes handled by either user defined markers or their position is outside of the area of interest.

5.3 Push-Relabel

As mentioned in the previous sections – algorithms based on the search for augmenting paths are not suitable for massive parallelization. So the *push-relabel* (Goldberg and Tarjan (1986)) algorithm was chosen as a start point for parallel CUDA implementation.

The basic version of the push-relabel algorithm consists of the operations *push* and *relabel*, which are applied as long as the corresponding conditions are met. The algorithm ends, when neither of those operations can be used.

Each vertex has a *height* (label) and *excess* (flow yet to be distributed to its neighbors) assigned.

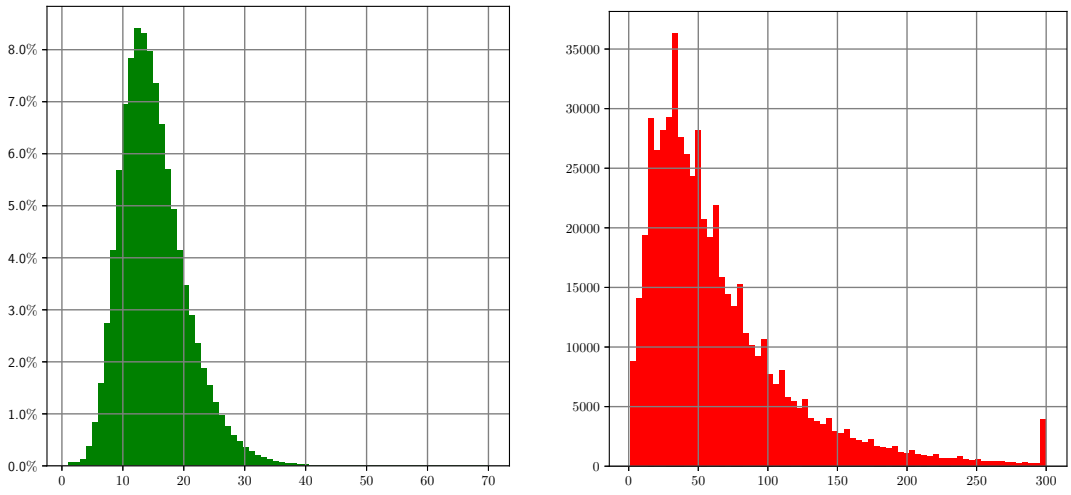
Algorithm 10: Push-relabel max-flow algorithm

```
Data: graph
Result: valid flow net
set excess to 0 for all vertices;
set label to 0 for all vertices except source which is ∞;
while push or relabel applicable do
    execute the operation;
end
```

During the computation of the flow net, the algorithm constructs the so called pre-flow net – defined as the flow net, where vertices can have some excess flow assigned. When the algorithm converges the only vertices with a nonzero excess are source and sink.

Operation *push* tries to decrease excess of some vertex u by sending maximal possible flow through unsaturated incident edges. Can be applied only when:

- $excess(u) > 0$



(a) Vertex degrees

(b) Region sizes

Figure 5.3: Region statistics for kidney dataset

- $capacity(u, v) - flow(u, v) > 0$
- $h(u) > h(v)$

Operation *relabel* handles the situations when some vertex has nonzero excess and cannot apply push:

- $excess(u) > 0$
- $h(u) <= h(v), \forall v, c(u, v) - f(u, v) > 0$

If these conditions hold, the minimal label can be assigned to the processed vertex, so it is higher than at least one its neighbors connected via unsaturated edge and so it can again distribute some of its excess.

Lemma 5.1.
If push from vertex v is not possible one of the following conditions must be true:

1. *v does not have valid label*
2. *It has zero excess.*
3. *All edges to the neighbors with valid lower label have been satured by push from v in current vertex labeling.*

Proof. It is direct consequence of the relabel definition. □

5.3.1 Min-cut

Minimal graph cut is not necessarily unique. Several different cuts can have the exact same minimal cost value.

Although an algorithm for finding maximal flow is used. The actual final flow-net is not needed to get the minimal cut.

Lemma 5.2.
The minimal cut is already determined in preflow net, when the removal of saturated edges creates a graph component T, such that $s \notin T$ and $t \in T$, where s is source and t is sink and all vertices $v \in T$, $v \neq t$ have zero excess.

Proof. Let S be the complement of the sink set T separated by saturated edges. S contains source s and possibly other vertices with positive excess. None of this excess can be propagated into the T through the saturated edges. So the algorithm will propagate the remaining excess back to source, either by decreasing flow through the edges or by creating circulation through the source. But none of this can influence the set T, so the edges connecting S and T form the minimal cut. □

Lemma 5.2 can be used to design an algorithm, which terminates the flow computation earlier and still computes correct min-cut. As was mentioned earlier, a heuristic published in Goldberg and Tarjan (1986) and also used in Hussein et al. (2007); He and Hong (2010) can be used for occasional relabeling. It is based on breath-first search starting from sink and assigning a new label to all visited vertices based on their distance from sink. The BFS algorithm can traverse only unsaturated edges in the direction from source. All vertices, which were not visited by the BFS will have an *invalid* label assigned and cannot participate in the push operation. By employing only this BFS-based relabeling scheme as a sole relabeling scheme an algorithm 11 is obtained.

Algorithm 11: Parallel Push-relabel min-cut algorithm

```
Data: graph G
Result: binary division of the vertex set
init vertex queue Q;
reset_labels(G);
reset_residuals(G);
preflow_initialization(G);
finished = false;
while !finished do
    bfs_relabel(G, Q);
    finished = !push(G, Q);
end
push_to_sink(G);
mark vertices with valid label as sink set;
mark vertices with invalid label as source set;
```

Algorithm 11 finishes exactly at the point specified by lemma 5.2. No push is possible and because all vertices in sink component have valid label, then according to 5.1, only condition number 2 remains, so it means that no vertex in the sink component have positive excess.

Another important fact is that the labeled vertices can be stored in a queue on their visitation. This layered vertex queue can be then used by the push operation. Visiting those vertices in reverse reduces number of necessary vertex-to-vertex push operations.

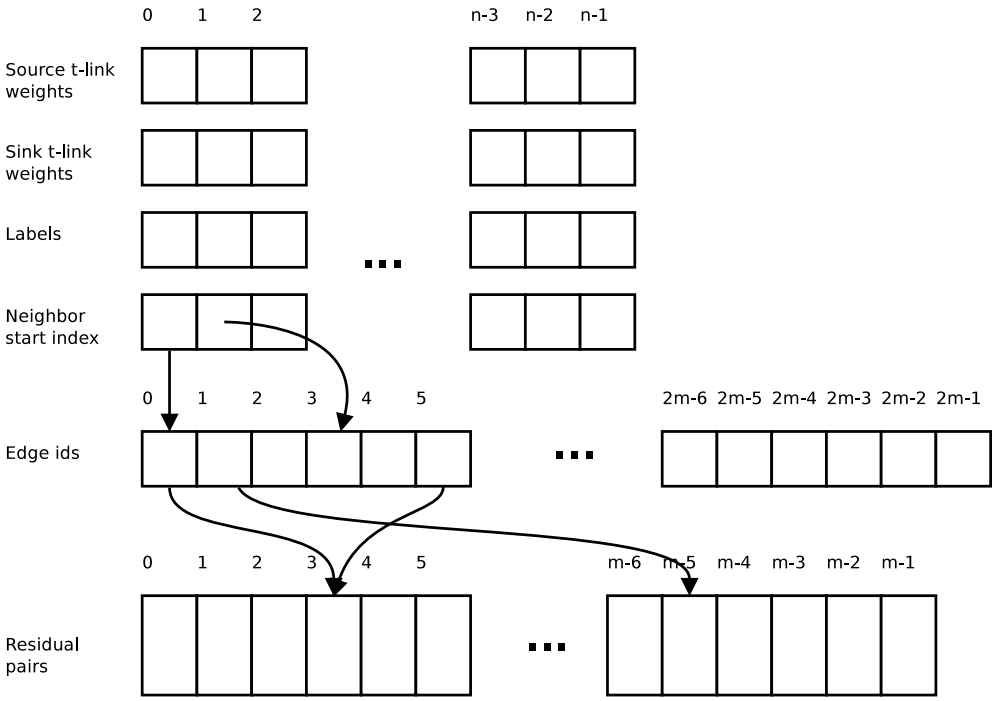


Figure 5.4: Graph representation scheme, where n is number of vertices, m is non-oriented edge count, so $2m$ is oriented edge count.

5.3.2 Graph Representation

Main graph operations needed during max-flow/min-cut computation are traversal of all vertex neighbors, accessing residual value of each connection and finaly need to update whole residual pair for each edge (forward and backward edge direction), because the residual decrease in one direction causes increase of residual in opposite direction by the same value.

Another requirement comes from the need for efficient graph update when user modifies the foreground and background markers. This can be done most easily by keeping t-links separate from rest of the graph edges, because n-links remains unchanged during user interactive session.

CUDA implemented algorithms are sensitive to memory access patterns. Most optimal are coalesced memory accesses, so wild pointer based datastructures are not an option.

As most suitable approach was chosen a set of linear buffers – each for separate graph attribute as can be seen in figure 5.4.

5.3.3 Relabel Step

The relabel step assigns a new label to all vertices in the graph. It starts from sink and assigns label 0 to all vertices connected to the sink through t-links. All these vertices are also added to the active vertex queue as level 0. Rest of the algorithm is based on BFS in the residual network. The main difference between standard BFS is that instead of following outgoing edges for each vertex, the algorithm follows incoming edges with non-zero residual (those that can be used by push operation to send some flow) leading to a vertex without valid label. All newly reached vertices are then added into active vertex queue as a new level

(algorithm 12).

Because the active vertex expansion is done in parallel with all other active vertices it can happen that the same vertex is reached from several different active vertices processed by different threads. Atomic *compare and swap* operation must be used for assignement of the new label to ensure that each vertex is added only once to the active vertex queue. Only the call which succeeded can add the vertex to the queue (algorithm 13).

Algorithm 12: Relabel - BFS from sink labeling

```
Data: graph G, empty vertex queue Q
init Q by vertices connected to sink;
while last level size in Q ≥ 0 do
    // Choose work distribution for kernel execution based on
       the current level size
    if last level ≤ single block limit then
        // Process multiple levels smaller than block size
        multiLevelSingleBlockRelabel(G, Q);
    else if last level ≤ multilevel global sync limit then
        // Process multiple levels using global synchronization
           barrier
        multiLevelGlobalSyncRelabel(G, Q);
    else
        // Process single level
        singleLevelRelabel(G, Q);
end
```

5.3.4 Push Step

The push step should redistribute the excess flow from vertices with valid label into their neighbors with lower label. All vertices with the valid label are stored in the layer ordered queue generated during relabel step.

By processing the queue in reverse a maximum possible excess flow is pushed towards the sink. As a result, after each push step a flow into the sink is increased, except the last iteration when there is no available excess flow to be pushed and whole algorithm ends (algorithm 14).

Atomic operations are used to update excess of the target vertex. Excess of the vertex the flow was pulled from, together with udate of edge residuals can be updated without atomics, because only single thread is accessing those values (algorithm 15).

5.4 Optimizations

The actual implementation of the algorithm can be optimized in several ways to gain maximum speed. There are possible algorithmic changes (like skipping certain steps) and several high level code optimizations in addition to the low level optimizations presented earlier.

Algorithm 13: Relabel - BFS kernel processing single label level

```
Data: L_i level from queue Q
Result: L_{i+1} in queue Q
init Q by vertices connected to sink;
index = threadIdx.x + blockIdx.x * blockDim.x;
while index < sizeof L_i do
    // expand vertex
    u = L_i[index];
    foreach neighbor v do
        if label_v is invalid and residual_{v,u} > 0 then
            // atomicCAS is used -- it may fail if another
               thread assigned the label first
            if atomic assignment of label i + 1 to vertex v was successful
             then
                append vertex v to L_i + 1;
    end
    index += gridDim.x * blockDim.x;
end
```

5.4.1 Algorithmic Optimizations

If the capacity of t-link leading to the sink is enough to accept any possible flow, two additional kernel executions can be prevented.

First operation in each relabel call is addition of the first level vertices which have connection to sink. Because the t-links to the sink will never be saturated the set of first level vertices will be exactly the same in each iteration. So if the deletion of the first level is skipped when the active vertex queue is cleared before each relabel step, then the BFS initialization of the first level can be called only once before the first iteration.

Second kernel call which can be skipped is the push of the excess through the t-links to sink at the and of each push step. Because the set of vertices connected through unsaturated t-links is stable through all iterations, all these vertices will have the lowest possible label for the whole computation. That means that excess accumulated in these vertices can only go to the sink. It means that the final push to sink in each iteration can be posponed until the end of the computation also if the only output expected from the algorithm is the graph partitioning, then the final push can be skipped completely.

"Hot Start"

If the graph-cut is used for interactive segmentation, an observation can be made. When user refines the input markers he can do it in two ways – either adds new markers into incorrectly classified regions, or changes already placed markers from previous iteration (fixing incorrect user input).

Addition of new markers either adds new t-links in the graph or increases capacity of already present t-links. None of these operations invalidates already computed flow (and preflow) in the graph. So instead of reseting the flow at the beginning of each user iteration, the previously computed preflow can be used as

Algorithm 14: Parallel push

```
Data: graph G, vertex queue Q ordered by label levels
Result: true if any of the push operations were successful
current level = last level in Q;
while current level ≥ 0 do
    if last level ≤ single block limit then
        // Process multiple levels smaller than block size
        multiLevelSingleBlockPush(G, Q);
    else if last level ≤ multilevel global sync limit then
        // Process multiple levels using global synchronization
           barrier
        multiLevelGlobalSyncPush(G, Q);
    else
        // Process single level
        singleLevelPush(G, Q);
end
```

initial state, but only if no t-link had its capacity decreased as a result of user input.

This modification can strongly decrease graph-cut computation time. The effect can be especially pronounced if the user updates are only small refinements.

5.4.2 Code Optimizations

Each kernel execution brings small overhead as described in chapter 1. Due to the need of global synchronization when doing BFS in residual network and push operation on BFS levels, number of kernel executions during graph-cut computation tend to be quite large, thus bringing nontrivial overhead.

By analysing the sizes of the BFS levels during relabel steps, a few observations can be made:

- Average size of the BFS levels decreases between relabel steps – as edges are saturated narrower bridges are formed.
- Small BFS levels tend to form small groups – small and large BFS levels do not usually alternate.
- In most cases the level sizes change slowly – from large to medium and then to small (with occasional fluctuations)

These observations use vague measurements like small, medium and large. More concrete meaning of those terms can be obtained by considering the tools at hand, specifically global synchronization primitives.

If *small* level is defined as level, which can be efficiently processed by kernel executed on grid with only single thread block. And if only single thread block executes the kernel then `__syncthreads()` serves as global synchronization primitive, allowing processing of more levels per single kernel execution. Number of levels processed per single kernel execution depends on size of the "small level

Algorithm 15: Push - kernel processing single label level, propagating excess to the lower levels.

```
Data: L_i level from queue Q
index = threadIdx.x + blockIdx.x * blockDim.x;
while index < Size(L_i) do
    // expand vertex
    u = L_i[index];
    if Excess(u) > 0 then
        foreach neighbor v do
            if Label(v) < Label(u) and Label(v) ≥ 0 then
                // Tries to atomically pull excess from u --
                   limited by residual of edge e_uv
                flow = TryPull(e_uv);
                if flow > 0 then
                    // Excess must be updated atomically
                    Excess(v) += flow;
                    Residual(e_uv) -= flow;
                    Residual(e_vu) += flow;
        end
        index += gridDim.x * blockDim.x;
end
```

groups" as mentioned in observation. Also the size limit for small level should not be strict, otherwise single "a little bit larger" level can break series of small iterations and kernel would need to be executed again and that can be slower than processing of the single little bit larger level.

In section 1.1.4, Xiao and c. Feng (2010) a possibility of global barrier implementation was described. The efficiency of this barrier descreases rapidly with number of thread blocks. And this factor can be used to define the size of *medium* BFS level. Medium sized level can be efficiently processed by multi-iteration kernel, which uses global barrier to sync between iterations. "Efficiently processed" means faster than other options – single block multi-iteration kernel, or single iteration kernel which will proces the *large* levels.

This strategy can be seen in algorithms 12 and 14, where the optimal BFS level processing is chosen based on its size.

There is another option how to limit the required control code executed on host for CUDA 5.0 and later and devices of Compute Capability 3.5. The support for *dynamic parallelism* allows kernels to start another child kernels. Reimplementation of the inner loop of the high level algorithms 12 and 14 was tested. The loop was wrapped into another kernel, which then executed kernels for push and relabel steps. Although this optimization did not bring any speedup, rather slowdown. It seemed that the parent kernel executed on single thread block, which has only single thread doing work is not and optimal pattern for dynamic parallelism in current CUDA version.

GPU profiler have shown that instruction level bottlenecks are uncoalesced memory accesses and warp divergence during graph traversal. Optimizations

presented in Merrill et al. (2012), where the traversal of the graph is organized into edge bundles, which are optimized to prevent warp divergence. Experiments have shown that the work reorganisation (which internally uses block-wise parallel reduce) is expensive and it does not pay of for graphs with vertices of low degree. As was shown in figures 5.2 and 5.3, the degree of vertices in adjacency graph is higher than in the original grid graph, but still quite low in comparison to the graphs tested in Merrill et al. (2012). At the end the direct approach to graph traversal was faster for the tested graphs.

5.4.3 Heuristics

Since the bottlenecks of the kernels are uncoalesced memory accesses, one way to improve the situation would be to reorganize the graph datastructure, so that graph traversal shows more locality.

Basic approach for grid graphs is vertex index assignment by different strategy. Instead of a scanline assignment, the indices are assigned in blocks, so graph traversal shows more memory access locality even for traversal in y and z axes and not only in x axis. This approach is not possible for generic graphs due to the non-uniform topology.

Experiments have shown that in most of the natural cases background markers are much bigger than foreground markers. It means that there is possible increase in task parallelism if the BFS start from source and not from sink. This oposite "polarisation" of the min-cut search (using foreground as source instead of sink) really leads to faster computation times, but when there are multiple minimal cuts possible, the algorithm does not find the minimal foreground set, because the presented min-cut search will always return minimal possible sink set – BFS will always find the graph-cut edge set closest to the sink, because it cannot traverse the flow net further to find different possibilities.

Situation is even more complicated, when an approximation from the following section is used. In this case the oposite polarization increses probability of the foreground spilage.

5.4.4 Approximations

Because targeted usage of presented version of graph-cut algorithm is interactive segmentation, it can be modified, so it returns only approximate solution, which can be computed faster. The solution should not deviate much from the precise solution and must be stable during user iterations (result should not change much in areas not edited by the user). Imprecisions caused by approximations can be easily fixed by the user by providing more information for critical parts.

Observation can be made if the vertex expansions during the relabel step ignore edges with residual smaller then some previously set constant ϵ, the algorithm tends to generate smaller sink-set by cutting off regions connected by weak bridges. That can be even beneficial for segmentation of objects with not well pronounced contours.

This behavior satisfy the requirement for the right approximation. It makes the final sink-set smaller, which is usually more desirable user experience (in

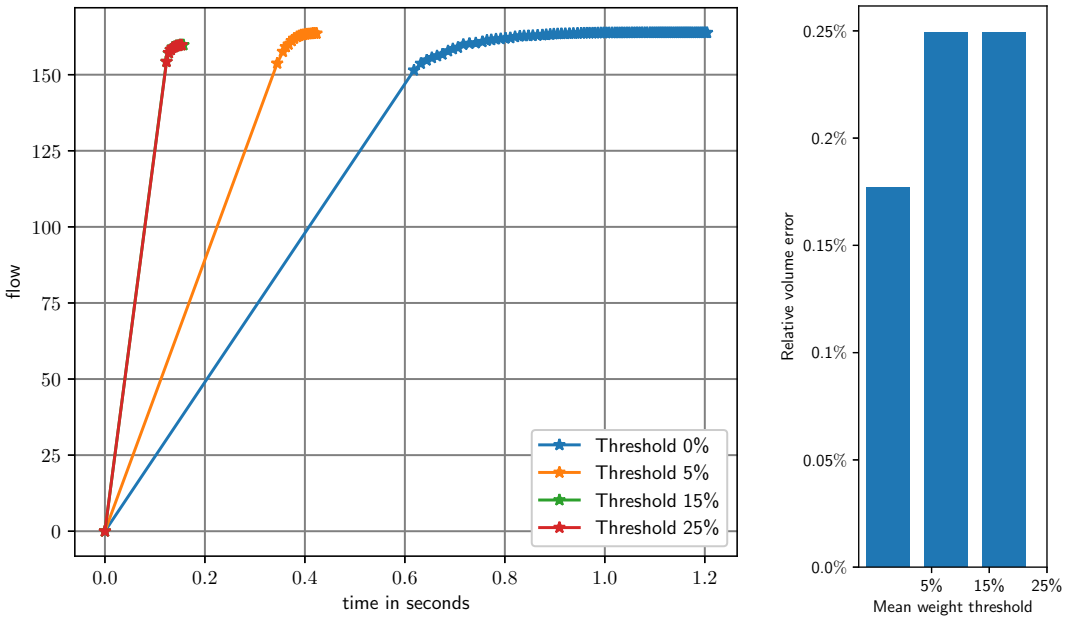


Figure 5.5: Number of required steps and speed of convergence: Femur 1

case that sink-set relates to the segmented object) instead of spillage into the background.

Experimental results (figures 5.5 and 5.6) have shown that using small threshold (fraction of the average edge weight) reduces the small BFS level problem at the end of computation, which limits possible parallelism. The measurements show massive speedup, with minimal effect on the final segmentation (minimal difference in produced binary masks).

An important note must be made that the pruning of the edges only at the beginning, but not during the algorithm does not have the same effect and as it does not prevent the small BFS level problem.

5.5 Benchmarking

The algorithm was tested on artificial data to check behavior in special cases and on a set of real medical computer tomography datasets. Grid-graphs were constructed over the image voxels and 6-connected neighborhood was used to define graph edges. Watershed transformations of the same images were used for definition of the region adjacency graphs.

A manually created markers for the foreground (segmented object) and background were created and two types of benchmarks were prepared. A grid-graph for the voxel grid was created and presented *cuda-cut* algorithm was compared with available implementations of *Boykov-Kolmogorov* and *Grid-cut* algorithms. The second set of tests was done on the graph constructed over the subregions generated by watershed transformation. The Cuda-cut algorithm is then compared only with Boykov-Kolmogorov, because Grid-cut cannot be used on non-grid graphs.

As can be seen in figure 5.7, the presented algorithm can compete with Boykov-Kolmogorov in a terms of speed, when applied on a grid graphs. Al-

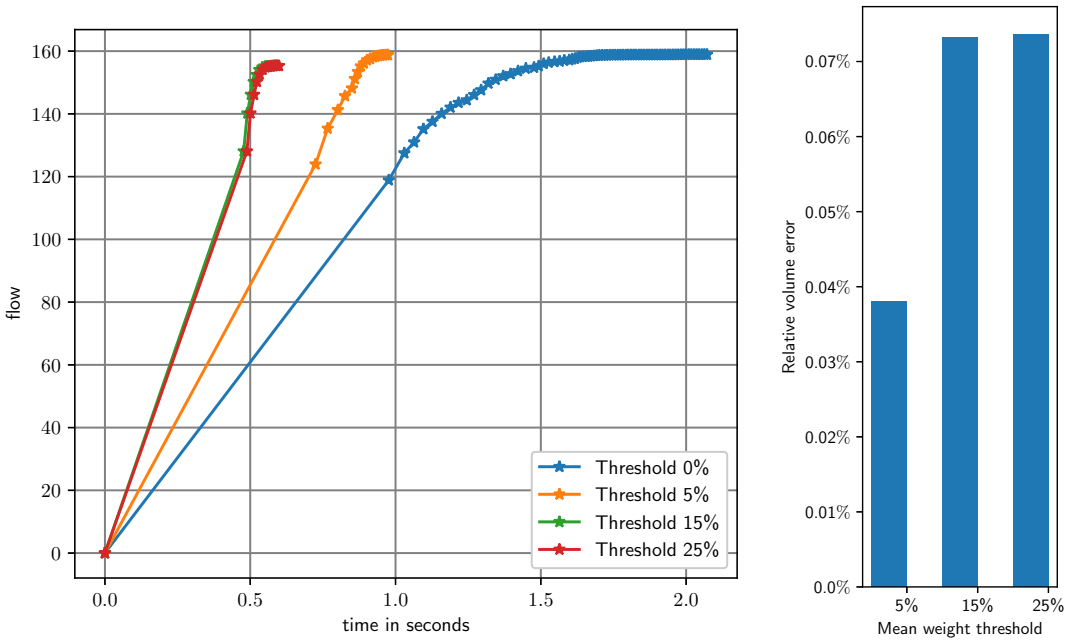


Figure 5.6: Number of required steps and speed of convergence: Kidney 2

though the Grid-cut algorithm is far ahead in comparison to Boykov-Kolmogorov and Cuda-cut. Main reason for this is their universality in respect to graph topology, which causes the speed penalization. By enabling the graph-cut approximation, the Cuda-cut algorithm can get close to (or even ahead) of the Grid-cut algorithm computational time.

The speed gain is even better for region adjacency graphs in comparison to the Boykov-Kolmogorov algorithm as can be seen in figure 5.8. This improvement is caused mainly by the increased average vertex degree, which leads to increase parallelism during relabel step and does not lead to collapsed search levels as often as for the grid-graphs.

5.6 Conclusions

It was shown that moving min-cut computation to GPU is a feasible strategy and does not create an obstacle in conversion of the interactive segmentation workflow to almost GPU only approach.

Although, the benchmarks have shown that it is hard to get same computation speed gain as was possible for image filters presented in previous chapters.

Profiling of the final implementation showed that the two major bottlenecks are uncoalesced memory accesses into the global memory during graph traversal and serialization of the computation caused by construction of narrow vertex levels during BFS traversal, which subsequently led also to the serialization of the following push operation.

The uncoalesced memory accesses problem can be possibly supressed by focusing only on grid graphs, but it is problematic for region adjacency graphs, which were target of the presented implementation.

The serialization of the relabel and subsequent push steps is unpredictable in

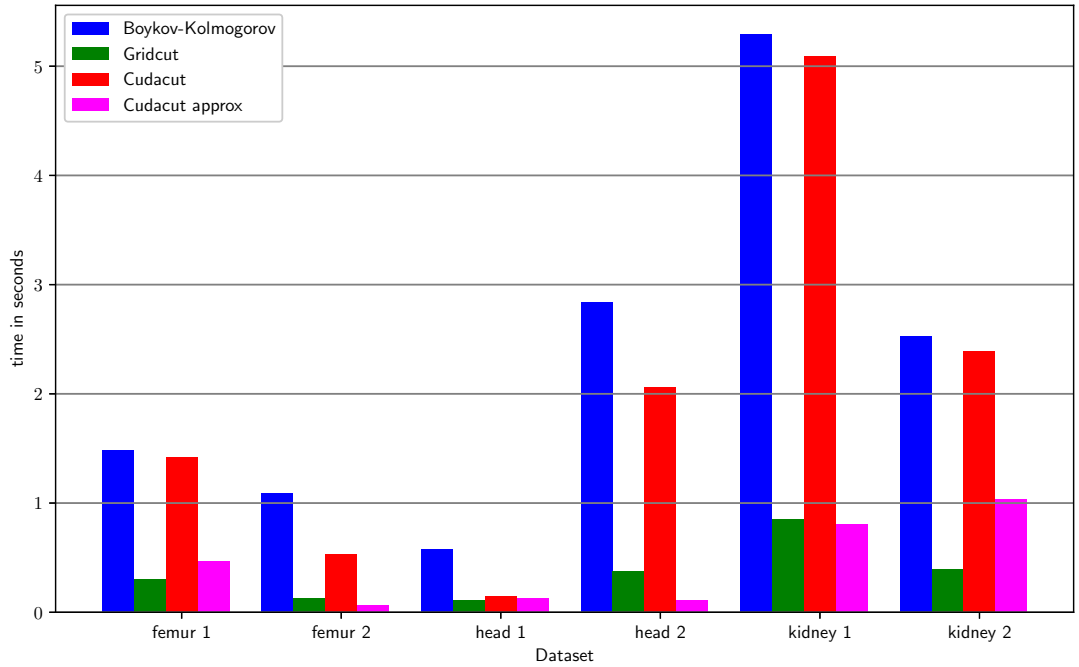


Figure 5.7: Speed comparison for grid graphs. Hardware used for measurement: NVIDIA GTX 1060 6GB, Intel i7-3770 CPU 3.40GHz 16GB, no-swap.

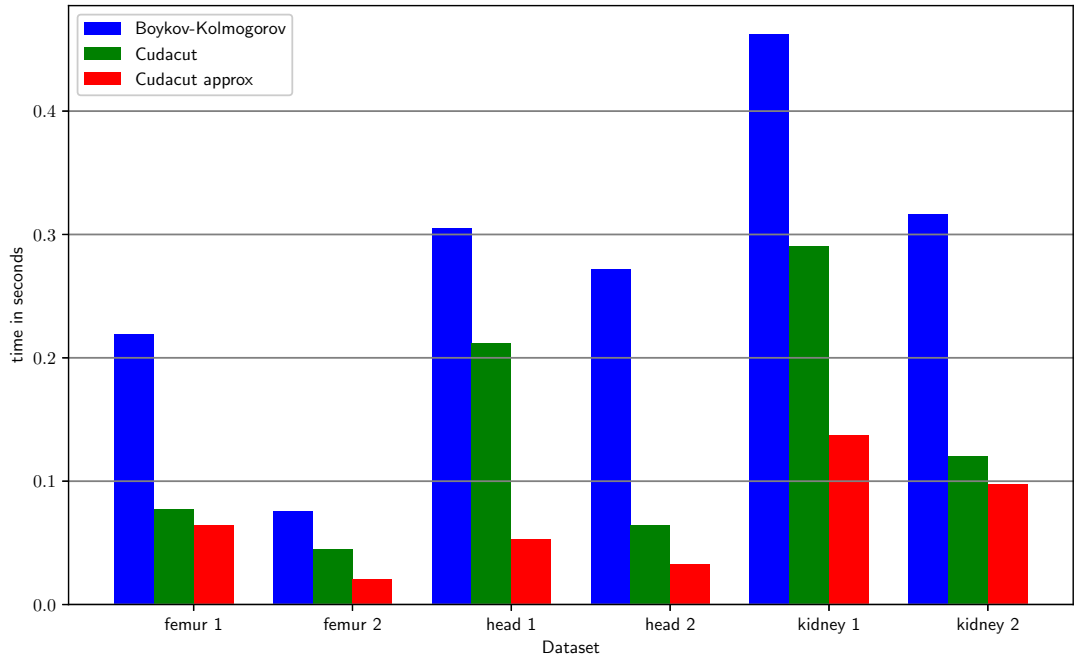


Figure 5.8: Speed comparison for region adjacency graphs. Hardware used for measurement: NVIDIA GTX 1060 6GB, Intel i7-3770 CPU 3.40GHz 16GB, no-swap.

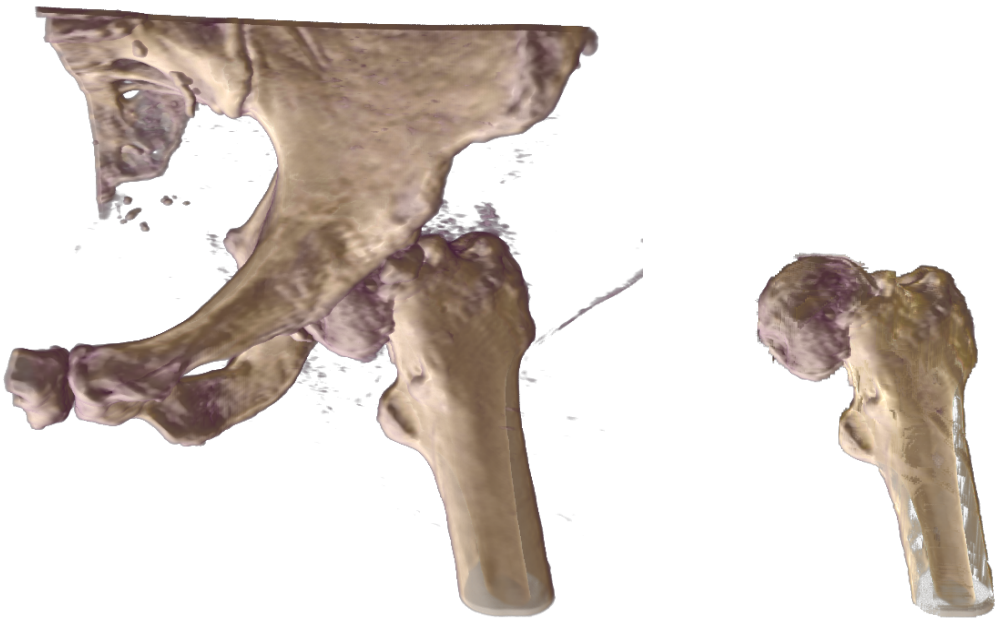


Figure 5.9: Segmentation of the human hip

sense that the achieved speedup depends on actual graph weights. This issue can be handled partially by employment of the presented approximation strategy at the cost of algorithm precision.

One of the limitations of the GPU implemented min-cut search is the limited amount of the GPU memory, which currently doe not support swapping and out-of-core implementation would require non-trivial changes.

The optimized version of the algorithm is now part of the *CUDA Generic Image Processing* library (Kolomaznik (2016)).

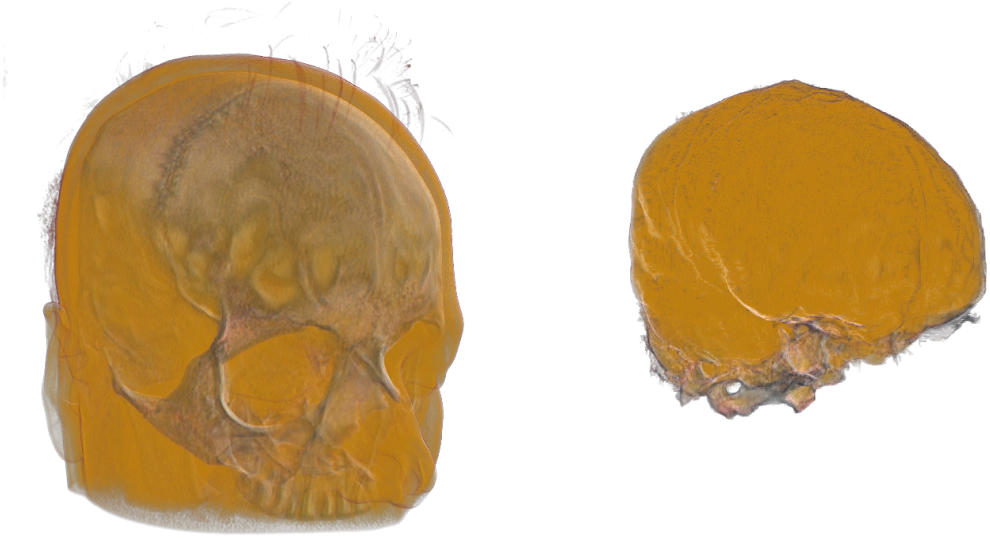


Figure 5.10: Segmentation of the human brain

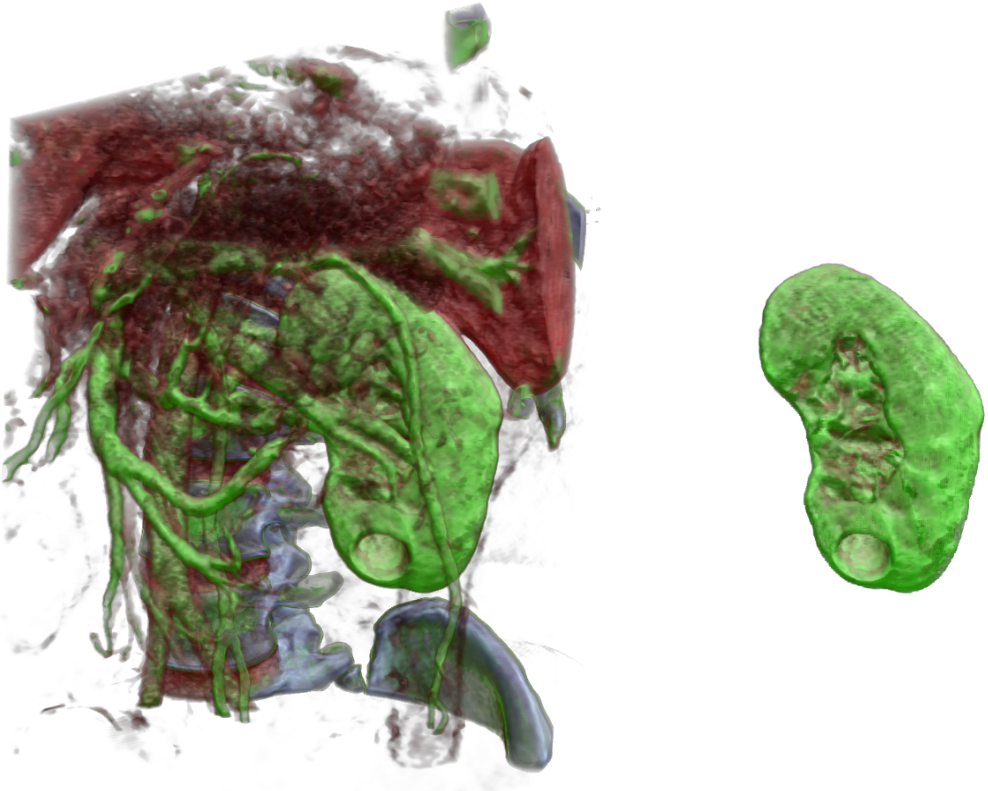


Figure 5.11: Segmentation of the human kidney.

Chapter 6

Statistical models

Image segmentation is one of the most essential problems in the field of computer vision. Although this topic has been extensively studied, common segmentation algorithms often serve as a preprocessing method of other algorithms. Automatic segmentation can hardly obtain satisfied results without high level knowledge of interest object Heimann and Meinzer (2009); Cremers et al. (2007); Leventon et al. (2002); Yue and Tagare (2009).

In medical imaging is often the situation complicated by the fact that the segmented organ is affected by certain pathologies (tumors, deformations, scar tissue), which are hard to model due to their unpredictability.

Also, methods suitable for automatic segmentation tend to be time consuming especially when dealing with low quality input (low resolution, noise, scanning artifacts, etc.).

Besides automatic and semi-automatic methods variety of fast interactive methods Zhao and Xie (2013) were published. User provides information about segmented object in form of constrains, seeds or similar mechanism. By evaluation of the segmentation output user can improve the result by updating input interactively. Session ends when object is segmented with desired precision.

Main disadvantage of the interactive segmentation algorithm is the amount of work user must do to get satisfying results, particularly for 3D volume segmentation.

Low level statistical models were designed to address these issues and were published in Kolomazník et al. (2015). These models can provide most of the required input information and user can focus on the most problematic (blurred, damaged) parts of the segmented object.

These models are called low level, because they do not describe complex properties and relations to the surrounding objects like other sophisticated but slow methods Tsai et al. (2003); Okada et al. (2008).

Design of the presented models was focused mainly on these properties:

- Easily embedable into various segmentation algorithms.
- Low computational complexity.
- Applicable during preprocessing step.

The final solution is consisting of two models. One describing spatial distribution of intensity values and the other directions of possible contour normals.

These models were embedded and tested in two segmentation frameworks. As energy minimization problem in parametric snakes/surfaces Jacob et al. (2001) and in minimal graph-cut based segmentation Boykov and Jolly (2001); Yi and Moon (2012); Kolomaznik et al. (2012).

6.1 Intensity Distribution

6.1.1 Formulation

First modeled property is spatial intensity distribution of segmented object (foreground) and its surroundings (background). Model provides two probabilities for pixel/voxel on position $\mathbf{x}$ of intesity $I(\mathbf{x})$ being inside the segmented region (R_{in}) resp. outside the segmented region (R_{out}).

$$\begin{aligned} P_{in}(I(\mathbf{x}), \mathbf{x}) &= P(I(x) \wedge x \in R_{in}) \\ P_{out}(I(\mathbf{x}), \mathbf{x}) &= P(I(x) \wedge x \in R_{out}) \end{aligned} \tag{6.1}$$

In special case when foreground/background intensities are independent of position (homogeneous object on homogeneous background), basic properties of the conditional probability can be used and following equations would be obtained:

$$\begin{aligned} P(I(x) \wedge x \in R_{in}) &= P(I(x)|x \in R_{in}) \cdot P(x \in R_{in}) \\ P(I(x) \wedge x \in R_{out}) &= P(I(x)|x \in R_{out}) \cdot P(x \in R_{out}) \end{aligned} \tag{6.2}$$

In Jacob et al. (2001) authors used energy term $E_{original}$

$$E_{original} = \int_{S_{in}} -log\left(\frac{P(I(x)|x \in R_{in})}{P(I(x)|x \in R_{out})}\right) d\mathbf{x} \tag{6.3}$$

It relates probability of point being inside segmented region (R_{in}) having some intensity value and probability of being outside (R_{out}) with actual intensity value. This energy term reaches its minimum when regions S_{in} (intermediate segmentation result) and R_{in} are the same. It works well if it is used for selection of object of homogeneous intensity on homogeneous background. Segmentation driven by this energy often fail if there is an area with similar intensity as the segmented object, because there are no spatial constrains (if not introduced in another way).

Decision was made to extend this model by spatial information and use 3-dimensional (2D segmentation) or 4-dimensional (3D segmentation) probability distribution function instead of 1-dimensional from the original term. The proposed formulation then looks like this:

$$E_{region} = \int_{S_{in}} -log\left(\frac{P_{in}(I(\mathbf{x}), \mathbf{x})}{P_{out}(I(\mathbf{x}), \mathbf{x})}\right) d\mathbf{x} \tag{6.4}$$

$$E_{region} = \int_{S_{in}} -log\left(\frac{P(I(x) \wedge x \in R_{in})}{P(I(x) \wedge x \in R_{out})}\right) d\mathbf{x} \tag{6.5}$$

In cases when independence of intensity and position (homogeneous regions) can be assumed, the equations 6.2 can be used. Using this and the fact that logarithm of a product is sum of logarithms a simplified version of the energy term is obtained.

$$E_{region2} = -\int_{S_{in}} log\left(\frac{P(I(x)|x \in R_{in})}{P(I(x)|x \in R_{out})}\right) + \\ +log\left(\frac{P(x \in R_{in})}{P(x \in R_{out})}\right) d\mathbf{x} \tag{6.6}$$

When intesity and position are independent $E_{region2}$ equals E_{region}. First term is same as $E_{original}$ and the second term is based only on spatial distribution. Second term was used for example in Jacob et al. (2001) as constrain energy, defined by user or trained from samples. By introduction of another parameter α into this equation and by convex combination of these two models the behavior of segmentation process can be further influenced.

$$E_{region3} = -\int_{S_{in}} \alpha \cdot log\left(\frac{P(I(x)|x \in R_{in})}{P(I(x)|x \in R_{out})}\right) + \\ +(1-\alpha) \cdot log\left(\frac{P(x \in R_{in})}{P(x \in R_{out})}\right) d\mathbf{x} \tag{6.7}$$

This simplified model works well only for homogeneous regions on homogeneous background due to intensity-location independence assumption.

6.1.2 Training

First step is proper alignment of the images from the training set (section 6.3.1). For each aligned image a binary mask representing segmented object is needed.

Intensity of each image element is recorded to one of the histograms available for its spatial coordinates. One histogram is for P_{in} probability and the second is for the P_{out} probability. Decision about which should be used is made by the binary mask query.

These histograms tend to be sparse due to the low number of images from training set in comparison to the number of possible values in the intensity range. *Parzen window* can be used to get smooth density estimation. Also, the resolution of the model is lower than resolution of the input images to ensure spatial smoothness of the trained model.

6.2 Shape Model

Previous model works quite well but still it has some flaws. With increasing variability of data in training set there grows larger area around segmented region, where P_{in} and P_{out} are almost the same. Boundary tends to fluctuate or takes the shortest path (depending on internal and constrain energy) in these regions. From this comes need for some other energy, which forces boundary to have proper shape.

Figure 6.1: Two binary masks from training set and computed vector field.

6.2.1 Formulation

A vector field which tries to model behavior of the boundary normals, is used. The shape energy is based on dot product of boundary normal and vector in the trained field on the same position. The energy term for this model is formulated as curve integral in 2D and surface integral in 3D over object's boundary.

$$E_{shape2D} = -\oint_C \mathbf{u} \cdot \upsilon^+ dr \tag{6.8}$$

$$E_{shape3D} = -\oiint_S \mathbf{u} \cdot \upsilon^+ dS \tag{6.9}$$

Where $\mathbf{u}$ is trained vector field and υ^+ is vector field of curve/surface unit normals oriented outwards from segmented region.

As will be shown in section 6.3 it is useful to express shape energy as region integral (surface, volume) instead of integral over curve/surface. For that *Divergence theorem* (Green's theorem in 2D, Gauss–Ostrogradsky theorem in 3D) can be applied to previous equations.

$$E_{shape2D} = -\int_S \nabla \cdot \mathbf{u}(x,y) dxdy \tag{6.10}$$

$$E_{shape3D} = -\int_S \nabla \cdot \mathbf{u}(x,y,z) dxdydz \tag{6.11}$$

6.2.2 Training

Either binary masks, or contour representation of segmented objects are needed for each aligned image from training set. The vector field containing region normals is computed for each dataset, either by computing normals from boundary representation, or by computing normalized gradient of region binary mask. These vector fields are too sparse and rough (those computed as binary mask gradient). So some smoothing process must be applied to increase the area of influence.

For this purpose there exists suitable method in form of *Gradient Vector Flow* Xu and Prince (1997), Paragios et al. (2001), which is used for image gradient enhancement in snake based segmentation algorithms. It is form of diffusion process suitable for vector fields. Parameters for diffusion depend on size of training set and properties of segmented regions.

If the gradient vector flow is applied to the vector fields from the training set a set of vector fields with increased support (nonzero area) is obtained. The goal is to compute one final vector field with following properties:

- Vectors have same direction as segmented region boundary passing that point.
- Size of vector reflects certainty of the direction.
- Vector field should be smooth.

Energy minimization scheme (again inspired by GVF) is used to achieve these requirements.

$\mathbf{v^1}, \mathbf{v^2}, \mathbf{v^3}, \ldots$ are vector fields of dimension n from training set.

$$E(\mathbf{u}) = \int_\Omega - \sum_i \mathbf{v^i} \cdot \mathbf{u} + \lambda \sum_i \sum_j \left(\frac{\partial u_i}{\partial x_j} \right)^2 + \kappa \|\mathbf{u} - \sum_i \mathbf{v^i}\|^2 \tag{6.12}$$

The desired vector field $\mathbf{u}$ should minimize this energy formulation, which consist from three parts. First term ensures minimal direction deviation from each training vector field. Second forces the final field to be smooth by minimizing partial derivatives and the third term prevent divergence by trying to keep $\mathbf{u}$ close to training vector field. Regularization parameters λ and κ tune the tradeoff between the first, second and third term.

The equation can be simplified by using basic properties of dot product, and by introducing sum of training vector fields $\mathbf{w} = \sum_i \mathbf{v^i}$.

$$E(\mathbf{u}) = \int_\Omega -\mathbf{w} \cdot \mathbf{u} + \lambda \sum_i \sum_j \left(\frac{\partial u_i}{\partial x_j} \right)^2 + \kappa \|\mathbf{u} - \mathbf{w}\|^2 \tag{6.13}$$

$\mathbf{u} = [u_1, u_2, \ldots, u_n]$ where n is field dimension. The equation can be solved by using set of *Euler-Lagrange equations*. For i in $1, 2, \ldots, n$, where n is field dimension.

$$-w_i + 2\kappa(u_i - w_i) - 2\lambda \sum_j \frac{\partial}{\partial x_j} \frac{\partial u_i}{\partial x_j} = 0$$

$$2\kappa u_i - (2\kappa + 1) w_i - 2\lambda \Delta u_i = 0 \tag{6.14}$$

This can be used in *steepest descent* or similar optimization algorithm.

6.3 Model Usage

6.3.1 Alignment

Quality of the trained model and its usefulness depends on the proper alignment of the training images and alignment between model and the segmented object in the processed image.

Various approaches to this task exist Yao and Summers (2009), but most of them would again increase computational complexity of the whole pipeline. Since this method is designed for interactive segmentation it was decided to let the user select important features from which the aligning transformation can be computed.

For the test case (section 6.4) the user was let to specify extreme points (poles) of segmented kidney in transversal slices of CT scan.

6.3.2 Parametric contours/surfaces

Wide range of segmentation methods based on deformable models (snakes, levelsets, etc.) can be found in the literature. The presented models were used in parametric snakes/surfaces framework, where contours are defined as spline curve/patch and their shape is controlled by set of control points.

In comparison to shape representation in form of a levelset, the parametric boundary representation can be easily used only for segmentation of objects with simple topologies. It is quite problematic to introduce shape with holes.

The motivation for usage of the parametric boundary representation is again speed. Whole framework is easilly paralelizable. Also, curves and surfaces manipulated by set of control points is well known concept from other graphical software tools. These properties makes it again good candidate for interactive segmentation algorithm.

If the models from previous sections are to be used in parametric boundary segmentation it is useful to compute curve/surface integral instead of surface/volume integral (*divergence theorem*), which is more time consuming. Jacob et al. (2001) used this step to bind all energy terms into one *unified energy* term. The exactly same principle will be presented.

By assuming that the surface Φ is oriented, so normal vectors $(\Phi_u \times \Phi_v)/\|\Phi_u \times \Phi_v\|$ are oriented outwards. $F(x,y)$ and $F(x,y,z)$ are 2D/3D unified energy terms.

$$\int_S F(x,y)dxdy = \oint_C \left(\int_{-\infty}^{y} F(x,\tau)d\tau\right)dx \tag{6.15}$$

$$= -\oint_C \left(\int_{-\infty}^{x} F(\tau,y)d\tau\right)dy \tag{6.16}$$

$$\int_V F(x,y,z)dxdydz = \oiint_S \mathbf{G_x}\cdot d\mathbf{S} = \oiint_S \mathbf{G_y}\cdot d\mathbf{S} = \oiint_S \mathbf{G_z}\cdot d\mathbf{S} \tag{6.17}$$

$$\mathbf{G_x} = (\int_{-\infty}^{x} F(\tau, y, z)d\tau, 0, 0) \tag{6.18}$$

$$\mathbf{G_y} = (0, \int_{-\infty}^{y} F(x, \tau, z)d\tau, 0) \tag{6.19}$$

$$\mathbf{G_z} = (0, 0, \int_{-\infty}^{z} F(x, y, \tau)d\tau) \tag{6.20}$$

But for usage in optimization scheme, the partial derivatives with respect to control parameters (control points) are needed. Only 3D version is shown as 2D is special case and was presented in Jacob et al. (2004).

The definition of surface integral of second kind is used.

$$\oiint_S \mathbf{G_x} \cdot d\mathbf{S} = \int_u \int_v \mathbf{G_x}(\Phi) \cdot (\Phi_u \times \Phi_v) du dv \tag{6.21}$$

And now it will be shown how to compute partial derivative of the equation in respect to x-coordinate of i-th control point.

$$\frac{\partial}{\partial c_{i,x}} \int_u \int_v \mathbf{G_x}(\Phi) \cdot (\Phi_u \times \Phi_v) du dv \tag{6.22}$$

$$\begin{aligned}
&= \int_u \int_v \frac{\partial}{\partial c_{i,x}} [\mathbf{G_x}(\Phi) \cdot (\Phi_u \times \Phi_v)] du dv \\
&= \int_u \int_v \frac{\partial}{\partial x} [\mathbf{G_x}(\Phi) \cdot (\Phi_u \times \Phi_v)] \frac{\partial x}{\partial c_{i,x}} du dv \\
&= \int_u \int_v \Big[\frac{\partial \mathbf{G_x}(\Phi)}{\partial x} \cdot (\Phi_u \times \Phi_v) + \\
&\quad + \underbrace{\mathbf{G_x}(\Phi) \cdot \frac{\partial (\Phi_u \times \Phi_v)}{\partial x}}_{=0} \Big] \frac{\partial x}{\partial c_{i,x}} du dv \\
&= \int_u \int_v (F(\Phi), 0, 0) \cdot (\Phi_u \times \Phi_v) \frac{\partial x}{\partial c_{i,x}} du dv
\end{aligned}$$

$\frac{\partial x(u,v)}{\partial c_{i,x}}$ will be often in simple form. For splines the basis functions are obtained.

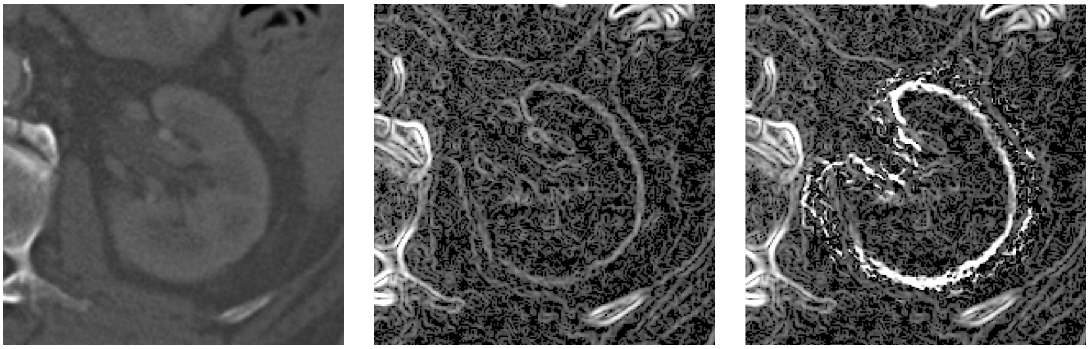
An equation is available for each partial derivative in respect to curve/surface control point, which is easily computable.

6.3.3 Minimal Graph-cut

The model can be used for segmentation based on computing minimal graph cut Boykov and Jolly (2001); Yi and Moon (2012); Kolomaznik et al. (2012) discussed in previous chapter.

User marks segmented object and background. This information is then incorporated into the graph in form of *t-link* weights.

Drawing foreground/background markers can be tedious especially in 3D. Two markers can be computed by input image thresholding using conditions 6.23. So



(a) (b) (c)

Figure 6.2: a) Human kidney image - part of abdominal CT scan b) Gradient magnitude c) Modulated gradient magnitude.

initial markers can be obtained automaticaly from the model. t_F and t_B are foreground/background probability thresholds.

$$M(\mathbf{x}) = \begin{cases} \text{F} & P_{in}(I(\mathbf{x}), \mathbf{x}) > t_F \\ \text{B} & P_{out}(I(\mathbf{x}), \mathbf{x}) > t_B \\ 0 & \text{Rest} \end{cases} \tag{6.23}$$

The second part of the presented model can be used to modulate *n-link* weights. Boundaries of objects tend to follow strong edges/ridges in the input image. So it means that contour goes through parts of the image with big gradient magnitude and its normals at those parts have same direction as the gradient.

Model contains a vector field which should represent normal directions of possible boundaries, so the influence of parts of image with properly oriented gradient can be boosted by this vector field.

$$\mathbf{G}_{new}(\mathbf{x}) = \mathbf{G}(\mathbf{x})\Big(1 + \alpha(max(0, \hat{\mathbf{G}}(\mathbf{x}) \cdot \mathbf{u}(\mathbf{x})))\Big) \tag{6.24}$$

Where $\mathbf{G}$ is original image gradient, $\hat{\mathbf{G}}$ normalized image gradient, $\mathbf{u}$ our shape model and α parameter controlling strength of the effect.

To prevent distortion of vector field the direction of the vector is not changed, only its length is modulated. Vectors ere not shortened, only those with proper direction are elongated. Example of gradient modulation in figure 6.2.

6.4 Results

Presented models are aimed for initialization of interactive segmentation algorithms. So the quality the result also depends on user input. To rule out influence of the user input two tests were implemented, which work without user input.

First one was modified thresholding, where aligned model serves as classifier. Properly tagged voxels were counted together with number of false positives and false negatives. These values are presented relatively to the object volume obtained from manual segmentation.

Second method used were parametric snakes based on presented statistical model and term which attracts contour to areas with bigger gradient magnitude. Error was again expressed as a percentage of whole object volume.

Table 6.1: Left kidney (post-contrast): *Correct* – part of organ volume correctly classified; E_{out} – false positive; E_{in} – false negative; E_{snakes} error of snake based segmentation (false positive and negative)

patientID	*Correct*	E_{out}	E_{in}	E_{snakes}
1	88,9%	8.6%	11.0%	7.1%
2	70.0%	6.0%	30.0%	26.3%
3	96.6%	13.0%	3.3%	6.2%
4	91.5%	4.5%	8.5%	1.6%
5	91.9%	3.5%	8.0%	1.9%
6	65.2%	9.5%	34.8%	4.2%
7	84.0%	2.4%	15.9%	10.3%
8	89.1%	2.2%	10.9%	5.4%

Table 6.2: Left kidney (without contrast agent): *Correct* – part of organ volume correctly classified; E_{out} – false positive; E_{in} – false negative; E_{snakes} error of snake based segmentation (false positive and negative)

patientID	*Correct*	E_{out}	E_{in}	E_{snakes}
9	84.5%	6.4%	15.4%	2.2%
10	91.5%	29.3%	8.5%	20.6%
11	89.5%	3.6%	10.5%	1.0%
12	92.8%	11.1%	7.1%	10.4%
13	93.4%	17.4%	6.6%	17.2%
14	71.7%	2.3%	28.3%	9.4%
15	91.4%	2.8%	8.6%	0.2%
16	61.8%	3.7%	38.2%	2.6%

In tables 6.1 and 6.2 are results of left kidney segmentation in CT images (with and without contrast agent – model trained for each case separatelly).

Measured errors show that the proposed models generate quite precise initialization in almost all cases – degenerated or damaged organs (patients 2 and 16) are not covered by models.

6.5 Conclusions

Low level statistical models were introduced, which successfully address initialization problem of interactive segmentation algorithms for class of compact objects.

Presented models describe only few certain properties of the segmented objects, but not their topology or shape variability. This leaves us with very rough initialization in case of complicated objects or objects with big shape variability. But even in this case the decrease in amount of user labor needed for initialization of the full-fledged segmentation algorithm is significant.

Chapter 7

Complete Workflow

By combining the algorithms presented in previous chapters into single workflow (figure 7.1), all the computation steps and visualization can be executed on GPU. This is a major advantage, since it eliminates the need for repeated data transfer between device memory and host memory. And alleviate the heavy work from CPU cores, which can be used for other tasks.

As discussed in the previous chapters most of the tasks in the workflow can be computed much faster on GPU. The slowest part remains the preprocessing with expensive NLM denoising (even though a massive speedup was achieved).

7.1 Preprocessing

As can be seen in figure 7.1 the first required user interaction is defining the foreground and background markers. And all steps before this can be executed automatically. As was discussed in chapter 2, the parameters for most of the filters can be deduced either from knowledge of the scanning method and its parameters or via estimation of parameters like standard deviation of the image noise directly from input data.

An important fact is that user not necessarily needs to work on preprocessed data. If the input image is not extremely noisy user can start to create markers even for the original data and the preprocessing happens in the background. The preprocessing need to be finished before graph construction and first execution of the graph-cut search.

7.2 Markers Initialization

In the two-dimensional case, it is easy for the user to specify background markers, which surrounds the segmented object. It is problematic to draw surrounding markers manually in three-dimensional case. Without surrounding markers, there is always possibility of spillage of the segmented region into the outer parts of the image.

Region of Interest

The simple approach in creating the surrounding markers is letting the user specify rough region of interest and apply the inverse mask as the background markers

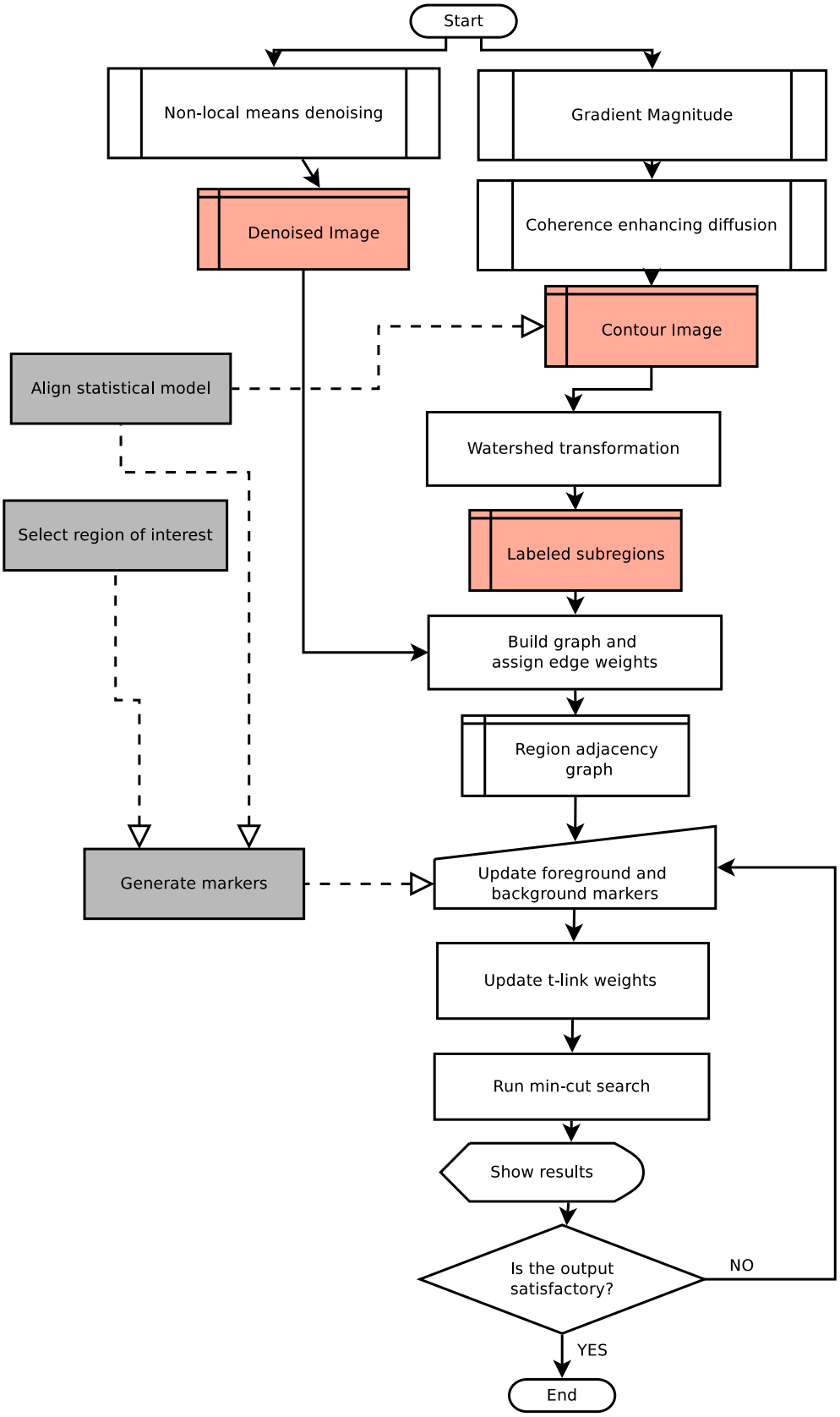


Figure 7.1: Segmentation workflow. Datsets required during the computation are marked red. Grey are the optional marker initialization steps.

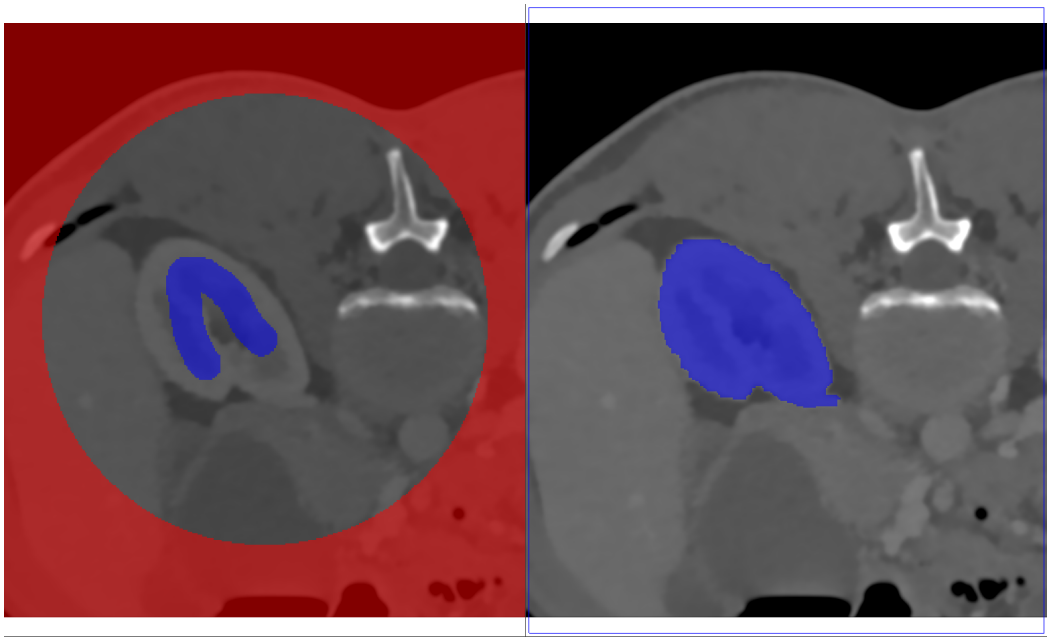


Figure 7.2: Segmentation of human kidney. On the left side is rough markers initialization (red) and user specified foreground marker (blue). On the right side is segmentation result

outside this region. The region of interest can be specified as simply as putting a big ball or box in the center of the image (figure 7.2).

Output of Simple Segmentation Algorithm

Outputs of fast and simple segmentation algorithms, such as thresholding, can be used as another useful approach to markers initialization. The important factor is again that thresholding can be computed quickly and quality of the result is not an issue, because it will be refined either by the user or the min-cut segmentation algorithm.

Statistical Models

One of the more robust solutions is the application of statistical models presented in chapter 6.

These models can provide precise markers, but on the other hand, the requirement for their training to the specific task may be limiting.

7.3 User Interaction

Drawing Markers

User specified markers (or corrections to previously initialized ones) are drawn like in any other raster image editor. Even though it is possible to draw directly in 3D views. It is not as intuitive for users as a classical 2D approach. Another reason is that it would require more tuning of the transfer functions to highlight the segmented object.

The 3D nature of the input data can be employed in a different way even for slice based marker specification. Instead of a flat brush (circle), which is typical for image editors, a three-dimensional brush (sphere) can be used, so even though the user does editing in single slice the markers are propagated into neighbor slices.

Result Visualization

The key advantage of running all steps on GPU is the prevention of unnecessary memory transfers between host and device. This fact can be used by a renderer, because data buffers can be shared between CUDA and OpenGL, so after each iteration the output of min-cut search can use for masking the volume for DVR, or as an overlay in slice based rendering.

7.4 Pipeline Recalculation

User input in form of the region markers can be considered as *stable input* – it means that even if other parameters of the workflow changes (strength of denoising, edge weight assignment) the markers should remain valid. By employing this fact, user can fine tune the results by assigning different control parameters in various parts of the workflow and obtain recalculated results without the need to specify the markers from scratch.

Conclusions

This thesis focused on the design of interactive segmentation workflow. So as the fundamental aspect of all interactive approaches is the response time for the user actions, the main objective was to lower the computation times as much as possible. Each step of the workflow was assessed and various possible optimizations to reduce the processing times were considered.

The core of the workflow is combination of the watershed transformation and segmentation based on the minimal graph-cut search. Both these methods are well covered in the literature, so the quality of their outputs was not measured in this thesis, as it can be found in the referenced papers and books. The same applies to the two preprocessing algorithms – non-local means denoising and coherence enhancing diffusion.

CUGIP Library. Tests have shown that most suitable approach to lower the computation time of a large set of image processing algorithms is to design parallel algorithms, which can be used on massively parallel architectures like modern GPUs.

NVIDIA CUDA framework was used for GPU-based computation. Also, as it is a low-level library the first major achievement presented in this thesis is a development of CUDA Generic Image Processing library (CUGIP), which was designed specifically for implementation of massively parallel algorithms without complexities brought into the code by CUDA framework.

CUGIP library proved itself to be very useful tool. Its qualities were also demonstrated commercially as its subset was already licensed by a big vendor of electron microscopes and micro-CTs.

Shape and Size Transfer Functions. A new kind of shape and size oriented transfer functions for real-time direct volume rendering was designed by extending the concept of 2D transfer functions, which use a multi-resolution version of the image containing the eigenvalues of Hessian matrices on each scale. Frangi's vesselness measure or its extended version in form of the *objectness measure* are then computed based on an interactive setting of the size and shape, which should be highlighted.

Preprocessing Filters. Two algorithms which produce high-quality outputs were implemented to deal with the high amount of noise typical for lots of volume data modalities. First is non-local means denoising, which in its standard form is extremely expensive when running on CPU. The highly optimized version was implemented on GPU. Even though it is still the

slowest part of the whole workflow, this algorithm was chosen, because of its ability to reduce noise without damaging important image features. This increases the robustness of the rest of the workflow. Without it there would be more user work needed, computation between the interactive steps would take longer as a larger number of watershed regions would be generated. So it seemed to be a better solution to try to move expensive computation into preprocessing phase, which could be more easily hidden from the user (part of the acquisition process, done offline on data server).

Straightforward parallel implementation of the coherence enhancing diffusion was also implemented to deal with low-quality gradient magnitude images which are used as input for the watershed transformation.

Cellular Automata and Watershed Transformation. A big part of this thesis was devoted to using cellular automata as a formalism for massively parallel computing.

Three main extensions of the cellular automata were presented, which address the issues of the standard cellular automata. The first is the introduction of asynchronnous cellular automata with hidden updates, which allows more efficient mapping between cellular automata definition and CUDA-enabled GPUs, also makes easier to execute such automaton on distributed computation systems (multi-GPU/multi-node computation).

The second definition covers cellular automata with a global state. This concept is used to deal with a large number of cellular automata generations, which are needed to communicate information from one part of the grid into another. The modifiable global state allows each cell to provide information instantly visible to the rest of the cells. In practice, this concept can lead to the race conditions or need of locking, when more cells update the global state. To address this issue, the global state with uncertain insert operation was introduced, which allows the existence of benign race conditions (do not prevent algorithm convergence).

The third cellular automaton extension introduces a notion of cell pointers as a part of the cell state, which provides access to any cell in the grid. This extension also deals with the problem of slow information propagation. It was shown that cellular automaton with cell pointers could be used to implement efficient two-pass out of core the algorithm to process large datasets, which does not fit into GPU memory.

These three definitions were then applied to cellular automaton formulation of the watershed transformation, together with an alternative approach to image lower completion, which is needed for watershed transformation based on steepest slope search.

The cellular automata extensions proved to be successful in reducing computation time needed for watershed transformation. The final times again highly surpassed the time required by CPU implementations.

Graph-cut Based Segmentation. The regions generated by watershed transformation were then used as an input to the graph-cut based segmentation.

A parallel version of the Goldberg's push-relabel algorithm was designed, which can be utilized for non-grid graphs.

The speed of the algorithm was compared with the publicly available implementations of the Boykov-Kolmogorov and Grid-cut algorithms.

It was shown that it is quite challenging to implement a GPU-based parallel algorithm, which would surpass the CPU implementations for non-grid graphs. For the tests on the grid graphs the grid-cut remained the fastest solution. Mainly because of the optimizations which can be done for grid-graphs (implicit edges, cache-friendly memory access patterns) cannot be used for non-grid graphs. The Boykov-Kolmogorov algorithm was slower in most of the cases and same applied to the tests on the graphs generated from watershed regions, where grid-cut cannot be used.

Even though that the designed parallel algorithms was the fastest in most of the cases, it is quite unpredictable in the actual computation time. It is more data sensitive than the sequential algorithms. In special cases, the BFS traversal through the residual network can generate a large number of the small levels. So this causes the drop in GPU utilization because the parallelization is done over these levels. This adverse effect can be suppressed by introduction of an approximation scheme, which significantly speeds up the computation and reduces the problematic slow convergence at the end. The price is a slightly lower precision of the algorithm, but it was shown that it does not have significant adverse effects on segmentation precision.

Statistical Models. A low-level statistical model was designed as an optional part of the workflow to lower the amount of user work required when drawing foreground and background markers. As the marking of the segmented objects can be a tedious work for objects in three dimensions.

The model is composed of two parts. First one covers the intensity distribution and provides probabilities whether a point is inside or outside, based on the position and the intensity at that position. The second part of the model provides an average normal vector for the contour of the segmented object. The size of the vector depends on a certainty of that information.

The model can be applied to modulate the gradient image and thus making the boundary edges more pronounced in areas of high certainty. And the probabilistic part of the model can be used to generated two sets of markers in the areas of high probabilities. The user can then focus mainly on tuning and fixing problematic areas.

It was also shown that these models can also be used in other segmentation frameworks as an energy formulation for the deformable models was also defined.

All of the presented algorithms can be executed on GPUs, thus freeing the CPU for other tasks and preventing the data "ping pong" between host and device memory. With continuous development and improvements of modern graphics cards, then more tasks can be moved away from CPU.

Right now the major obstacle, which complicates the GPU-based computation on the large data sets, is a limited amount of graphics memory. Manual swapping back to the host memory and partitioning of the computation tasks complicates otherwise clean solutions.

Future Work

All parts of the presented workflow can be most probably improved even further. Several ideas for additional development are to be listed.

CUGIP library is by its essence continuous work in progress. Algorithm implementations should employ newly available hardware features, which come with each new generation. Major CUDA extensions which are currently not yet supported by CUGIP library list memory allocations in kernels, dynamic parallelism and usage of multiple GPUs.

In-kernel memory allocations and deallocations combined with dynamic parallelism can be used to implement various GPU-only algorithms incorporating multiresolution analysis and adaptive refinement.

Dynamic parallelism and employment of multiple GPUs will lower the amount of the required CPU control code, as more complicated algorithms can be executed just on GPUs.

The optimizations presented in chapter 2 applied on the NLM denoising and coherence enhancing diffusions can be labeled mostly as implementation optimizations. Various algorithmic optimizations were published especially for the NLM denoising, which improve the algorithmic complexity and the number of memory acceses required by each image element computation. Several of these algorithmic optimizations should be portable to GPU, thus lowering the computation time even more.

The shape and size oriented transfer function is a point of interaction between user and the visualization system. Manual drawing and tuning of the two dimensional transfer function can be quite tedious especially for minor details. Employing strategies from WYSIWYG[1] transfer function UI systems may lead to more intuitive usage. User can scribble over the important features in the previsualization of the rendered volume and the transfer function mapping buffer would be computed from the scribbled areas.

Presented cellular automata extensions can be used for wider variety of problems then was presented, thus making it a large area for further development.

As was shown, asynchronnous cellular automata with hidden updates can be also used for distributed computing, so work distribution over several GPUs or computational nodes is another possible research direction.

As mentioned, the minimal graph-cut search on GPU brings lower speedup in comparison to other GPU implemented algorithms. Profiler shows that there are still places where the futher optimizations may be possible.

[1] What You See Is What You Get

Bibliography

Adobe (2009). Generic image library documentation. http://www.boost.org/doc/libs/1_61_0/libs/gil/doc/index.html.

Anderson, R. J. and Setubal, J. a. C. (1992). On the parallel implementation of goldberg's maximum flow algorithm. In *Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '92, pages 168–177, New York, NY, USA. ACM.

Antiga, L. (2007). Generalizing vesselness with respect to dimensionality and shape. In *The Insight Journal*. Kitware.

Arens, S. and Domik, G. (2010). A survey of transfer functions suitable for volume rendering. In *Proceedings of the Eurographics/IEEE VGTC Workshop on Volume Graphics*.

Bellman, R. (1958). On a Routing Problem. *Quarterly of Applied Mathematics*, 16:87–90.

Berlekamp, E. R., Guy, R. K., and Conway, J. H. (1985). *Winning ways for your mathematical plays, 1 : games in general*. Academic press, London, New York, Sydney.

Beucher, S. (1994). Watershed, hierarchical segmentation and waterfall algorithm. In Serra, J. and Soille, P., editors, *Mathematical Morphology and Its Applications to Image Processing*, pages 69–76. Springer Netherlands, Dordrecht.

Beucher, S. and Meyer, F. (1993). The morphological approach to segmentation: The watershed transformation. *Mathematical Morphology in Image Processing*, Vol. 34:433–481.

Bieniek, A. and Moga, A. (2000). An efficient watershed algorithm based on connected components. *Pattern Recognition*, 33(6):907 – 916.

Boykov, Y. and Jolly, M.-P. (2000). Interactive organ segmentation using graph cuts. In *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2000*, pages 276–286, Berlin, Heidelberg. Springer Berlin Heidelberg.

Boykov, Y. and Jolly, M.-P. (2001). Interactive graph cuts for optimal boundary & region segmentation of objects in n-d images. In *Computer Vision, 2001. ICCV 2001. Proceedings. Eighth IEEE International Conference on*, volume 1, pages 105–112 vol.1.

Boykov, Y. and Kolmogorov, V. (2004). An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26(9):1124–1137.

Buades, A., Coll, B., and Morel, J. M. (2004). On image denoising methods. Technical report, Technical Note, CMLA (Centre de Mathematiques et de Leurs Applications).

Buades, A., Coll, B., and Morel, J. M. (2005). A review of image denoising algorithms, with a new one. *Multiscale Modeling & Simulation*, 4:490–530.

Chabardès, T., Dokladal, P., Faessel, M., and Bilodeau, M. (2016). A parallel, o(n) algorithm for unbiased, thin watershed. In *Image Processing (ICIP)*, pages 2569–2573.

Cherkassky, V. B. and Goldberg, V. A. (1997). On implementing the push—relabel method for the maximum flow problem. *Algorithmica*, 19(4):390–410.

Coifman, R. R. and Donoho, D. L. (1995). Translation-invariant de-noising. In Antoniadis, A. and Oppenheim, G., editors, *Wavelets and Statistics*, pages 125–150. Springer New York, New York, NY.

Correa, C. and Ma, K.-L. (2008). Size-based transfer functions: A new volume exploration technique. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1380–1387.

Coupé, P., Yger, P., Prima, S., Hellier, P., Kervrann, C., and Barillot, C. (2008). An optimized blockwise non-local means denoising filter for 3d magnetic resonance images. *IEEE Trans. on Medical Imaging*, 27(4):325–441.

Cremers, D., Rousson, M., and Deriche, R. (2007). A review of statistical approaches to level set segmentation: Integrating color, texture, motion and shape. *International Journal of Computer Vision*, 72:215.

de Guzman, J. and Kaiser, H. (2016). Spirit documentation. http://www.boost.org/doc/libs/1_61_0/libs/spirit/doc/html/index.html.

Dinic, E. A. (1970). Algorithm for Solution of a Problem of Maximum Flow in a Network with Power Estimation. *Soviet Math Doklady*, 11:1277–1280.

Ford, L. R. and Fulkerson, D. R. (1962). *Flows in Networks*. Princeton University Press.

Frangi, A. F., Niessen, W. J., Vincken, K. L., and Viergever, M. A. (1998). Multiscale vessel enhancement filtering. In Wells, W. M., Colchester, A., and Delp, S., editors, *Medical Image Computing and Computer-Assisted Intervention — MICCAI'98*, pages 130–137, Berlin, Heidelberg. Springer Berlin Heidelberg.

Fredman, M. and Saks, M. (1989). The cell probe complexity of dynamic data structures. In *Proceedings of the Twenty-first Annual ACM Symposium on Theory of Computing*, STOC '89, pages 345–354, New York, NY, USA. ACM.

Gardner, M. (1970). Mathematical Games: The fantastic combinations of John Conway's new solitaire game "life". *Scientific American*, 223:120–123.

Goldberg, A. V. and Tarjan, R. E. (1986). A new approach to the maximum flow problem. In *Proceedings of the eighteenth annual ACM symposium on Theory of computing*, STOC '86, pages 136–146, New York, NY, USA. ACM.

Guo, H., Mao, N., and Yuan, X. (2011). Wysiwyg (what you see is what you get) volume visualization. *IEEE Trans. Vis. Comput. Graph.*, 17(12):2106–2114.

Guo, H. and Yuan, X. (2013). Local wysiwyg volume visualization. In *2013 IEEE Pacific Visualization Symposium (PacificVis)*, pages 65–72.

Hadwiger, M., Kniss, J. M., Rezk-salama, C., Weiskopf, D., and Engel, K. (2006). *Real-time Volume Graphics*. A. K. Peters, Ltd., Natick, MA, USA.

Harish, P. and Narayanan, P. (2007). Accelerating large graph algorithms on the GPU using CUDA. In Aluru, S., Parashar, M., Badrinath, R., and Prasanna, V., editors, *High Performance Computing – HiPC 2007*, volume 4873 of *Lecture Notes in Computer Science*, pages 197–208. Springer Berlin Heidelberg.

Harris, M. (2007). Optimizing parallel reduction in CUDA. "http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf".

Hawick, K. A., Leist, A., and Playne, D. P. (2010). Parallel graph component labelling with GPUs and CUDA. *Parallel Comput.*, 36:655–678.

He, Z. and Hong, B. (2010). Dynamically tuned push-relabel algorithm for the maximum flow problem on CPU-GPU-Hybrid platforms. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–10.

Heimann, T. and Meinzer, H.-P. (2009). Statistical shape models for 3d medical image segmentation: A review. *Medical Image Analysis*, 13(4):543 – 563.

Horacek, J., Horak, M., Kolomaznik, J., and Pelikan, J. (2015). An automatic algorithm for tracking small intestine in ct enterography. In *Digital Image Computing: Techniques and Applications (DICTA), 2015 International Conference on*, pages 1–8.

Horacek, J., Kolomaznik, J., Pelikan, J., and Horak, M. (2011). Denoising volumetric data on gpu. *International Joint Conference on Imaging Theory and Applications (IMAGAPP)*.

Hucko, M. and Sramek, M. (2012). Streamed watershed transform on gpu for processing of large volume data. In *Proceedings of the 28th Spring Conference on Computer Graphics*, SCCG '12, pages 137–141, New York, NY, USA. ACM.

Hussein, M., Varshney, A., and Davis, L. (2007). On implementing graph cuts on CUDA. *First Workshop on General Purpose Processing on Graphics Processing Units*.

Intel (2016). Threading building blocks. www.threadingbuildingblocks.org.

Jacob, M., Blu, T., and Unser, M. (2001). A unifying approach and interface for spline-based snakes. In *in Proc. SPIE Med. Imaging, l. 4322*, pages 340–347.

Jacob, M., Blu, T., and Unser, M. (2004). Efficient Energies and Algorithms for Parametric Snakes. *IEEE Transactions on Image Processing*, 13(9):1231–1244.

Jamriska, O., Sykora, D., and Hornung, A. (2012). Cache-efficient graph cuts on structured grids. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, pages 3673–3680.

Kauffmann, C. and Piche, N. (2008). Cellular automaton for ultra-fast watershed transform on gpu. In *Pattern Recognition, 2008. ICPR 2008. 19th International Conference on*, pages 1 –4.

Kitware (2016). Insight toolkit (ITK). itk.org.

Kniss, J., Kindlmann, G., and Hansen, C. (2002). Multi-dimensional transfer functions for interactive volume rendering. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 8(3):270 — 285.

Kolomaznik, J. (2016). Cuda generic image processing (CUGIP) repository and documentation. https://github.com/JanKolomaznik/cugip.

Kolomaznik, J., Horacek, J., Krajicek, V., and Pelikan, J. (2012). Implementing interactive 3d segmentation on CUDA using graph-cuts and watershed transformation. *International Conferences in Central Europe on Computer Graphics, Visualization and Computer Vision*.

Kolomazník, J., Horáček, J., and Pelikán, J. (2015). Low level statistical models for initialization of interactive 2d/3d segmentation algorithms. In *Proceedings of the 10th International Conference on Computer Vision Theory and Applications - Volume 1: VISAPP, (VISIGRAPP 2015)*, pages 686–692.

Kopp, J. (2008). Efficient Numerical Diagonalization of Hermitian 3
times 3 Matrices. *International Journal of Modern Physics C*, 19:523–548.

Lantuejoul, C. and Beucher, S. (1981). On the use of the geodesic metric in image analysis. *Journal of Microscopy*, 121(1):39–49.

Leventon, M., Grimson, W., and Faugeras, O. (2002). Statistical shape influence in geodesic active contours. In *Biomedical Imaging, 2002. 5th IEEE EMBS International Summer School on*, page 8 pp.

Lorensen, W. E. and Cline, H. E. (1987). Marching cubes: A high resolution 3d surface construction algorithm. *SIGGRAPH Comput. Graph.*, 21(4):163–169.

Luo, L., Wong, M., and Hwu, W.-m. (2010). An effective GPU implementation of breadth-first search. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 52–55, New York, NY, USA. ACM.

Max, N. (1995). Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99–108.

Merrill, D., Garland, M., and Grimshaw, A. (2012). Scalable GPU graph traversal. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 117–128, New York, NY, USA. ACM.

Meyer, F. (1994). Topographic distance and watershed lines. *Signal Processing*, 38(1):113–125.

Meyer, F. and Beucher, S. (1990). Morphological segmentation. In *Journal of Visual Communication and Image Representation - JVCIR*, volume 1, pages 21–46.

Najman, L. and Schmitt, M. (1994). Watershed of a continuous function. *Signal Processing*, 38(1):99 – 112. Mathematical Morphology and its Applications to Signal Processing.

Nehaniv, C. L. (2003). Asynchronous automata networks can emulate any synchronous automata network. *Journal of Algebra*, pages 1–21.

Nikodem, J. (2009). Plateau problem in the watershed transform. *Computing and Informatics*, 28.

Noguet, D. (1997). A massively parallel implementation of the watershed based on cellular automata. In *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors*, ASAP '97, pages 42–, Washington, DC, USA. IEEE Computer Society.

NVIDIA Corporation (2018a). CUDA FAQ. https://developer.nvidia.com/cuda-faq.

NVIDIA Corporation (2018b). CUDA Toolkit 9.1 Thrust. http://docs.nvidia.com/cuda/thrust.

NVIDIA Corporation (2018c). CUFFT. http://docs.nvidia.com/cuda/cufft.

NVIDIA Corporation (2018d). NVIDIA CUDA C programming guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide. Version 9.1.

Okada, T., Yokota, K., Hori, M., Nakamoto, M., Nakamura, H., and Sato, Y. (2008). Construction of hierarchical multi-organ statistical atlases and their application to multi-organ segmentation from ct images. In *Proceedings of the 11th international conference on Medical Image Computing and Computer-Assisted Intervention - Part I*, MICCAI '08, pages 502–509, Berlin, Heidelberg. Springer-Verlag.

Paragios, N., Mellina-Gottardo, O., and Ramesh, V. (2001). Gradient vector flow fast geodesic active contours. In *Computer Vision, 2001. ICCV 2001. Proceedings. Eighth IEEE International Conference on*, volume 1, pages 67 –73 vol.1.

Perona, P. and Malik, J. (1990). Scale-space and edge detection using anisotropic diffusion. *IEEE Trans. Pattern Anal. Mach. Intell.*, 12(7):629–639.

Pickover, C. C. A. (2009). *The math book : from Pythagoras to the 57th dimension, 250 milestones in the history of mathematics*. New York: Sterling,.

Pock, T., Unger, M., Cremers, D., and Bischof, H. (2008). Fast and exact solution of Total Variation models on the GPU. In *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW '08. IEEE Computer Society Conference on*, pages 1–8. IEEE.

Roerdink, J. B. T. M. and Meijster, A. (2000). The watershed transform: definitions, algorithms and parallelization strategies. *Fundamenta Informaticae - Special issue on mathematical morphology*, 41(1-2):187–228.

Rudin, L. I., Osher, S., and Fatemi, E. (1992). Nonlinear total variation based noise removal algorithms. *Phys. D*, 60(1-4):259–268.

Stava, O. and Benes, B. (2010). Connected component labeling in cuda. In *GPU computing gems emerald edition*, page 569.

Stawiaski, J. and Decenciere, E. (2008). Interactive liver tumor segmentation using graph-cuts and watershed. *Liver Tumor Segmentation - MIDAS - Grand Challange*.

Strandmark, P. and Kahl, F. (2010). Parallel and distributed graph cuts by dual decomposition. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference*, pages 2085–2092.

Stroustrup, B. (1994). *The Design and Evolution of C++*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA.

Świercz, M. and Iwanowski, M. (2010). *Fast, Parallel Watershed Algorithm Based on Path Tracing*, pages 317–324. Springer Berlin Heidelberg, Berlin, Heidelberg.

Tsai, A., Yezzi, A., Wells, W., Tempany, C., Tucker, D., Fan, A., Grimson, W. E., and Willsky, A. (2003). A shape-based approach to the segmentation of medical imagery using level sets. *IEEE transactions on medical imaging*, 22(2):137–154.

Vincent, L. and Soille, P. (1991). Watersheds in digital spaces: an efficient algorithm based on immersion simulations. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(6):583–598.

Vineet, V. and Narayanan, P. (2008). CUDA cuts: Fast graph cuts on the gpu. In *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW '08. IEEE Computer Society Conference on*, pages 1–8.

von Neumann, J. (1951). The general and logical theory of automata. In Jeffress, L. A., editor, *Cerebral Mechanisms in Behavior – The Hixon Symposium*, pages 1–31. John Wiley & Sons.

Weickert, J. (1999). Coherence-enhancing diffusion filtering. *International Journal of Computer Vision*, 31(2):111–127.

Xiao, S. and c. Feng, W. (2010). Inter-block GPU communication via fast barrier synchronization. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12.

Xu, C. and Prince, J. L. (1997). Gradient vector flow: A new external force for snakes. In *Computer Vision and Pattern Recognition*, pages 66–71.

Xu, N., Ahuja, N., and Bansal, R. (2007). Object segmentation using graph cuts based active contours. *Computer Vision and Image Understanding*, 107(3):210 – 224.

Yao, J. and Summers, R. (2009). Statistical location model for abdominal organ localization. In Yang, G.-Z., Hawkes, D., Rueckert, D., Noble, A., and Taylor, C., editors, *Medical Image Computing and Computer-Assisted Intervention - MICCAI 2009*, volume 5762 of *Lecture Notes in Computer Science*, pages 9–17. Springer Berlin, Heidelberg.

Yi, F. and Moon, I. (2012). Image segmentation: A survey of graph-cut methods. In *International Conference on Systems and Informatics*.

Yue, Y. and Tagare, H. (2009). Learning to segment using machine-learned penalized logistic models. *2012 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, 0:58–65.

Zhao, F. and Xie, X. (2013). An overview of interactive medical image segmentation. *Annals of the BMVA*, 2013(7):1–22.

List of Definitions

4.1 Definition (Cellular automaton - CA) . . . 52
4.2 Definition (Catalyst substate) . . . 53
4.3 Definition (Result substate) . . . 53
4.4 Definition (Equilibrium state of cellular automaton) . . . 54
4.5 Definition (Cellular automata equivalence) . . . 54
4.6 Definition (Standard asynchronous cellular automaton) . . . 56
4.7 Definition (Asynchronous cellular automaton with hidden updates) . . . 56
4.8 Definition (Cellular automaton with global state) . . . 58
4.9 Definition (Cellular automaton with pointers) . . . 62
4.10 Definition (Search with stable pointer paths) . . . 62
4.11 Definition (Skeleton by influence zones - SKIZ) . . . 67
4.12 Definition (Topographical distance) . . . 67
4.13 Definition (Watershed transformation in continuous space) . . . 68
4.14 Definition (Digital image) . . . 68
4.15 Definition (Lower complete image) . . . 68
4.16 Definition (Discrete topographical distance) . . . 69
4.17 Definition (Discrete watershed transformation by topographical distance) . . . 69
4.18 Definition (Discrete watershed transformation by immersion) . . . 70
4.19 Definition (Discrete watershed transformation based on local condition) . . . 70

5.1 Definition (Graph cut) . . . 89
5.2 Definition (Minimal graph cut) . . . 89
5.3 Definition ((s,t)-cut) . . . 89

List of Cellular Automata Rules

4.1 Cellular automaton rule (Connected component labeling) 57
4.2 Cellular automaton rule (CCL with global union-find datastructure) 60
4.3 Cellular automaton rule (Watershed based on distance.) 74
4.4 Cellular automaton rule (Watershed by steepest descent) 75
4.5 Cellular automaton rule (Watershed by steepest descent with global state.) . . . 76
4.6 Cellular automaton rule (Watershed by steepest descent with pointers.) 76
4.7 Cellular automaton rule (Watershed by steepest descent with pointers – decomposed.) 77
4.8 Cellular automaton rule (Lower completion rule.) 77
4.9 Cellular automaton rule (Non-minima plateau handling.) 79