Implementing Interactive 3D Segmentation on CUDA Using Graph-Cuts and Watershed Transformation

Jan Kolomazník
honza.kolomaznik@gmail.com

Jan Horáček
jan.horacek@gmail.com

Václav Krajíček
vajicek@cgg.mff.cuni.cz

Josef Pelikán
pepca@cgg.mff.cuni.cz

Department of Software and Computer Science Education
Faculty of Mathematics and Physics
Charles University in Prague, Czech Republic

ABSTRACT

In this paper we present a novel scheme for a very fast implementation of volumetric segmentation using graph cuts. The main benefit of this work is our approach to non-grid region adjacency processing on CUDA which to our knowledge has not been done yet in any efficient way. The watershed transform radically reduces the number of vertices for graph processing. Everything starting from watershed transformation and ending with graph cut was parallelized and is performed directly on the GPU.

Keywords: 3D segmentation, CUDA, watersheds, min-cut, push-relabel, max flow.

1 INTRODUCTION

Segmentation of 3D volumetric images brings a variety of problems. First of all, we are processing a large amount of data, rendering robust and precise algorithms prohibitively expensive.

In the case of 2D, the user can resort to manual segmentation if needed, while in 3D, manual segmentation becomes a very tedious task. To address this issue, an interactive (semi-automatic) method that would assist the user during the segmentation process is needed. In this paper, we propose an algorithm that segments the data based on initial user input (Figure 1), allowing the user to correct and improve the resulting segmentation by providing the algorithm with further hints.

We have chosen a segmentation approach based on the minimal graph cut algorithm. This is a well known image processing algorithm, however its computational complexity prevents it, at least without further improvements, from being practically useful for any kind of three dimensional data (CT, MRI). To alleviate these issues, we first transform the input data using the watershed transformation to an induced minor of the original graph, significantly decreasing the vertex count. We then process this data using the massively parallel architecture of programmable GPUs.

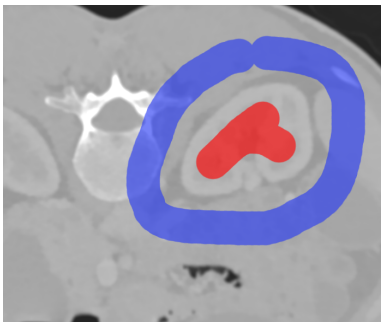

Figure 1: User input – CT image

2 RELATED WORK

Same overall (min-cut on watersheds) methods were chosen by authors in [8] for the interactive segmentation of liver and liver tumours, but they processed much smaller data in a longer time.

Several approaches can be used to compute the watershed transformation (there are also several equivalent definitions) [7] on CPU is a typical method based on flooding. The authors in [4] used Bellman-Ford shortest distance algorithm and cellular automata formulation on parallel architectures.

The min-cut problem is connected to finding the maximal flow through net. We needed an easily parallelizable algorithm. We therefore considered the push-relabel algorithm [1]. There are many papers on various implementations of the push-relabel algorithm. Most of them are focused on segmentation of 2D images, where the input graph is built as a neigborhood representation of image pixels – a grid where each vertex (except on borderlines) has a degree of 4 or 8 depending on pixel connectivity. Several CUDA implementations exist [3]

and all of them take the pixel/voxel based adjacency graph as their input, so they can easily optimize memory access of CUDA kernels. One of our goals is to implement on CUDA graph-cuts for general graphs.

3 OUR WORK

The reference single threaded CPU versions of the mentioned algorithms take minutes or at least tens of seconds to provide results on our data.

Because a CPU-only implementation of the algorithm is not feasible, a design decision has been made to make use of the computational power of the programmable GPUs. Furthermore, we observe that a lot time can be wasted by copying data between the host computer's main memory and the GPU's memory, making a GPU-only, in our case CUDA based, implementation the approach of choice. While some of the algorithms, e.g. image filtering, are naturally parallelizable and therefore suited for parallel processing, others must be completely redesigned, ideally into the form of cellular automata, which can be easily executed in CUDA.

3.1 Method Overview

Overview of method is listed in Algorithm 1.

Algorithm 1 Method overview

```
Denoising
Compute gradient magnitude of the image
Watershed transformation (Section 3.3)
Build region adjacency graph G (Section 3.4)
repeat
    Get user input
    Modify G according to user input
    Compute max-flow (Section 3.5)
    Find min-cut (Section 3.6)
    Present result to the user
until user satisfied
```

3.2 CUDA

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 the above-mentioned algorithms must 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. See [6] for details. We adressed these problems in our implementation.

We use the newest CUDA framework's features (available in version 4.1) such as atomic operations, C++ templates and Thrust (*STL* like library of datastructures and algorithms executed on CUDA, see [5]). Templates are mostly used to allow process various input data formats, but we also pass all limits known during compilation as template parameters, so compiler can easily optimize cycles and local variables. The *Thrust* library is used to ease up data transfer between host and device code. We also use algorithms like *thrust::inclusive_scan* and *thrust::reduce*.

We developed a generic memory loading scheme for kernels executed on image data. Most of the memory reads are coalesced and warp divergence is prevented as much as possible. We load cube of 10x10x10 voxels into shared memory (internal 8x8x8 cube allows coalesced loading) for kernels working on 3x3x3 neighborhood of voxel.

3.3 Watershed Transformation

Most of the watershed transformation algorithms are serial in nature because of the internal usage of a priority queue or a similar structure:

1. Initialize a set of markers - image local minima. Each labeled by different ID.
2. Insert the neighbors of the marked regions into a priority queue (priority is the gray level of the image element).
3. Pop an element from the queue. If all its neighbors have the same label, label the element with same ID. Insert the neighbors that are not yet in the priority queue.
4. While the queue is not empty redo step 3.

Several parallelization aproaches for the serial versions are available (e.g. [7]), but none of them is suitable for GPUs. We decided to use a cellular automaton reformulation of the problem by [4].

First of all, we need to find the markers for the watershed transformation (local minima of gradient magnitude image). This is a fairly straighforward processing of each voxel in a 3x3x3 neighborhood. As a result we obtain an image with nonzero-labeled regions on zeroed background (each element with a different label). So as the next step we need to mark all compact regions with single label each. This is done by *connected component labeling (CCL)* [2].

We implement CCL as an iterative process which finds region equivalences (Algorithm 2). Two regions are considered equivalent (and should have the same label) when they are neighbors. So we are marking these equivalences in a lookup table and relabel the whole image in every iteration. At the end we do the final relabeling so that we produce a continuous sequence of labels.

Finally, in the cellular automata formulation of the watershed transformation, we try to optimize the distance of each voxel to the closest marker. All computations are based on local information only (in our case, 3^3 voxels) so it is tailored for CUDA (Algorithm 3).

Algorithm 2 Connected component labeling

```
Allocate ID equivalence lookupTable
Init lookupTable
Scan labelBuffer for equivalences (element neighbors with different nonzero label)
while lookupTable updated do
    Update lookupTable
    Relabel elements in labelBuffer by equivalences from lookupTable
    Scan labelBuffer for equivalences
end while
```

Algorithm 3 Watershed transformation

```
Initialize distance buffer (set local minima markers to zero and rest to infinity)
while distance buffer updated do
    for all voxels do
        Test neigbors for shorter path
    end for
end while
```

3.4 Construction of Adjacency Graph

The construction of the adjacency graph in CUDA is not straightforward, as dynamic data structures are difficult to implement effectively and using an adjacency matrix or similar datastructure is memory consuming.

We will construct a list of all edges and their weights. All algorithms are implemented for an undirected graph so each edge will be present in the list only once.

As a data structure we decided to implement a hash table using *open adress linear hashing*, where the edges are stored as a special 64-bit index (32-bit for each vertex). This way was chosen so we can atomically insert edge records into the hash table.

Because we detect edges multiple times (for every voxel on the common border), we have to check if an edge is already inserted into the table. We either accumulate information about the image gradient, or we insert a new edge record.

A typical situation is that we want to insert the same edge multiple times at one moment (threads in a block run locally) – so to prevent serialization of accesses in global memory, we have to use a two-level approach. The hash table is not created only in global memory, but also in shared memory for every block. Concurrent insertions are first handled in the context of one block and the accumulated data from these tables is then inserted into the global hash table at the end of kernel execution.

The hash table is then sorted so all records are in the begginning of the array. Because of the design of our graph algorithms we have not to create an adjacency list or other representation of the vertex neighborhood. All algorithms need only an edge soup and the vertex count.

3.5 Push-Relabel Algorithm

The basic version of the push-relabel algorithm [1] 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* assigned (flow yet to be distributed to its neighbors).

During the computation of the flow net, our algorithm constructs the so called pre-flow – defined as the flow, where some vertices are assigned some excess flow. 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:

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

Operation *relabel* handles 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 we assign minimal label to the processed vertex so it is higher than at least one its neighbors via unsaturated edge.

Both operations work on vertices and their neighbors. This is problematic for the CUDA implementation targeted on general graphs. CUDA kernel calls need to be as coherent as possible (same instructions, aligned memory accesses) to achieve maximal throughput. This is the reason why other CUDA graph-cut implementations (e.g. [3]) work only on grid graphs , where every vertex has the same number of neighbors (depending on connectivity).

We solved this issue by formulating *push* and *relabel* operations not over vertices but over edges. We process all edges and accumulate data for vertex update after the edge processing is finished. Atomic operations are used to update the vertex properties (excess, label) directly when it is needed.

CUDA implementation of *push* is quite straightforward (Algorithm 4), except we iterate over edges not over vertices (we do not have information about vertex neighbors).

Relabel must be implemented in three phases:

1. locating vertices with nonzero excess (parallel predicate check)
2. checking all edges to find new label to vertices marked in step 1 (Algorithm 5)
3. assigning new label to marked vertices

Algorithm 4 Parallel push

for all $Edge \in E$ **do**
  if $Label[Edge.v1] > Label[Edge.v2]$ **then**
    $pushFromTo(Edge.v1, Edge.v2, Edge)$
  else if $Label[Edge.v2] > Label[Edge.v1]$ **then**
    $pushFromTo(Edge.v2, Edge.v1, Edge)$
  end if
end for

Algorithm 5 Parallel relabel – phase 2

for all $Edge \in E$ **do**
  if $Edge.v1$ enabled and relabel conditions hold **then**
    Use *atomic min* (so more suitable value cannot be overriden) to assign new possible height for $Edge.v1$ in buffer
  else
    disable $Edge.v1$
  end if
  ... symetrically for opposite orientation
end for

Algorithm 6 Parallel BFS

Allocate CUDA arrays $F1$, $F2$ and W (visited)
$fill(F1, false)$
$F1[start] = true$; $W[start] = true$
repeat
  $fill(F2, false)$
  $bfsKernel(E, V, F1, F2, W)$ (Algorithm 7)
  $markFrontierAsVisited(F2, W)$
  $swap(F1, F2)$
until no vertex added to frontier

Algorithm 7 BFS kernel

for all $Edge \in E$ **do**
  if $Edge.v1 \in F1 and Edge.v2 \notin W$ **then**
    $F2[Edge.v2] = true$
    mark frontier as not empty
  end if
  ... symetrically for opposite orientation
end for

3.6 Consolidation of Results

Min-cut can be easily obtained from the flow net. Simply execute the breadth first search (Algorithm 6) either from the source or the sink and stop on saturated edges – these edges are our min-cut. All vertices visited during BFS form the first set of regions (either background or the segmented object) while the rest is the second set.

At the end we usually apply some morphological corrections (morphological opening), because the result is an union of several watershed regions, which have a typical property – jagged contours.

Algorithm	100^3	200^3	512x512x256
Markers	0.006s	0.035s	0.794s
Watersheds	0.049s	0.121s	2.648s
Min-cut	0.281s	0.436s	2.381s

Table 1: Speed of presented algorithms

4 RESULTS

We tested our implementation on NVIDIA Fermi GPU (GTX 560). We can compare our results (Table 1) with published results ([8], [4]). Our solution typicaly achieves a throughput increase by factor of 10x-20x.

In comparison to the CPU implementation, our CUDA algorithms demonstrate a 50x-70x speedup.

5 CONCLUSION

We have shown an effective implementation of a basic segmentation method based on the watershed transformation and a push-relabel max-flow algorithm.

The problem with large number of image voxels was partially solved by working on regions instead on image elements and to decrease the computation time even more we successfully used CUDA enabled GPUs.

The used algorithms were modified for massivelly parallel architectures. We removed the problematic push-relabel algorithm dependence on graph topology (limitation of the other implementations). Our modifications also made the requirement of vertex neighborhood in graph representation redundant.

ACKNOWLEDGEMENTS

This work was supported by the Grant Agency of Charles University, Prague (project number 355311).

REFERENCES

[1] A V Goldberg and R E Tarjan. 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, 1986. ACM.

[2] K. A. Hawick, A. Leist, and D. P. Playne. Parallel graph component labelling with gpus and cuda. *Parallel Comput.*, 36:655–678, 2010.

[3] Mohamed Hussein, Amitabh Varshney, and Larry Davis. On implementing graph cuts on cuda. *First Workshop on General Purpose Processing on Graphics Processing Units*, 2007.

[4] C. Kauffmann and N. Piche. Cellular automaton for ultra-fast watershed transform on gpu. In *Pattern Recognition, 2008. ICPR 2008. 19th International Conference on*, pages 1 –4, 2008.

[5] NVIDIA Corporation. CUDA Toolkit 4.0 Thrust Quick Start Guide, 2011.

[6] NVIDIA Corporation. NVIDIA CUDA C programming guide, 2011. Version 4.1.

[7] Jos B. T. M. Roerdink and Arnold Meijster. The watershed transform: definitions, algorithms and parallelization strategies. *Fundamenta Informaticae - Special issue on mathematical morphology*, 41(1-2):187–228, 2000.

[8] Jean Stawiaski and Etienne Decenci. Interactive liver tumor segmentation using graph-cuts and watershed. *Liver Tumor Segmentation - MIDAS - Grand Challange*, 2008.