Lucas Grassano Lattari

GPU Multilevel Narrow Banded Graph Cuts for Image and Video Segmentation

Niterói - RJ, Brazil

XX de setembro de 2010

Lucas Grassano Lattari

GPU Multilevel Narrow Banded Graph Cuts for Image and Video Segmentation

A dissertation presented to the Programa de Pós-Graduação em Computação from Universidade Federal Fluminense in partial fulfillment of the requirements for the Master degree. Research Area: Visual Computing and Interfaces.

Orientador:
Anselmo Antunes Montenegro

Co-orientador:
Marcelo Bernardes Vieira

Instituto de Computação
Universidade Federal Fluminense

Niterói - RJ, Brazil

XX de setembro de 2010

Dissertação de Mestrado submetida ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para obtenção do título de Mestre. Área de concentração: Computação Visual e Interfaces. Aprovada por:

Prof. Dr. Anselmo Antunes Montenegro
Orientador

Prof. Dr. Marcelo Bernardes Vieira
Co-Orientador

Prof. Dr. Marcelo Gattass
Pontifícia Universidade Católica do Rio de Janeiro

Prof. Dr. Paulo Cezar Pinto Carvalho
Instituto de Matemática Pura e Aplicada

Prof. Dra. Cristina Nader Vasconcelos
Universidade Federal Fluminense

Prof. Dr. Esteban Walter Gonzales Clua
Universidade Federal Fluminense

Abstract

This work focuses on providing efficient solutions for the Image and Video Segmentation problems using the Graph Cuts method in many-cores architectures. It is based on an extensive investigation of the state-of-art of the techniques and strategies proposed for the implementation of Graph Cuts in GPUs. The result of such investigation has led us to develop our own implementation and uncover the details of the algorithms and heuristics used to improve it which are sometimes obscure in some works found in the literature. The main contribution is an original technique called Parallel Multilevel Narrow Banded Graph Cuts. The fundamental idea is to compute the segmentation in a coarse version of the original image, and project the obtained result in a finer level of resolution where the segmentation needs to be recomputed only at a buffer region between the background and object partitions. The process is repeated successively until the final solution is obtained for a given level of refinement. This strategy yields visual results equivalent to those obtained by the application of the traditional method to the original image in the highest resolution. Furthermore, it leads to a reduction in the total number of processed nodes, improves the time complexity and decreases the memory usage. We performed tests comparing our method with other works, and the results obtained are very promising. Although we concentrated in the use of such method for the image and video segmentation problem, the results can be used to improve the solution of any problem formulated using the Graph Cuts methodology in GPUs.

Summary

List of Figures

List of Tables

1 Introduction p. 10

1.1 Objectives . . . . . p. 12

1.2 Structure . . . . . p. 13

2 Related Works p. 14

3 Energy Minimization via Graph Cuts p. 20

3.1 Pixel Labeling Problem . . . . . p. 21

3.2 Energy Minimization Models . . . . . p. 23

3.2.1 Gibbs Energy . . . . . p. 24

3.3 Background on Graphs . . . . . p. 25

3.3.1 Graphs Representation . . . . . p. 25

3.3.2 Graph Cuts . . . . . p. 31

3.4 Representing Energy Functions with Graphs . . . . . p. 33

3.4.1 Function Classes Optimally Minimized via Graph Cuts . . . . . p. 33

3.4.2 General Workflow . . . . . p. 34

3.5 Narrow Banded Strategy in 2D Images . . . . . p. 36

4 Graph Cuts Algorithms p. 40

4.1 Algorithms . . . . . p. 40

4.1.1 Algorithms based on augmenting paths . . . . . p. 40

4.1.2 Push-Relabel . . . . . p. 42

5 Graph Cuts in GPU architectures p. 47

5.1 GPU Computing . . . . . p. 47

5.2 GPU Graph Representation . . . . . p. 49

5.3 GPU Push-Relabel . . . . . p. 50

5.4 GPU Multilevel Narrow Banded Graph Cuts . . . . . p. 53

6 Experimental Results p. 59

7 Conclusion p. 71

References p. 73

List of Figures

1 Example of a binary segmentation. . . . . p. 10

2 Characteristic function describing the segmentation of a gray tone image. p. 22

3 Characteristic function separating an object from the background. . . p. 22

4 Graph neighborhood representation. . . . . p. 26

5 Example of a grid graph with different neighborhood systems for an 4x4 image. . . . . p. 27

6 Example of a node i its 4 edges highlighted. . . . . p. 28

7 Grid graph with detached nodes on the nodes without top and bottom neighbors (Figure 7(a)). Similarly, a grid graph without left and right neighbors are detached in Figure 7(b). . . . . p. 28

8 Example of a Breadth-First Search operation on a binary tree. . . . . p. 30

9 Example of a cut of a graph. . . . . p. 32

10 Example of a network flow with a maximum flow defined and a graph-cut of it. (CORMEN et al., 2001) . . . . . p. 32

11 Example of a graph based on an 3x3 image and the cut of it. . . . . p. 35

12 The multilevel banded Graph Cuts procedure. (LOMBAERT et al., 2005) p. 37

13 Example of an 8×8 image being projected to an upper level, becoming a 4×4 image. . . . . p. 38

14 An example of segmentation on different moments. . . . . p. 54

15 Downsampling from an image I_k to an image I_{k+1} with the highlighted nodes p_k and p_{k+1}. . . . . p. 55

16 Example of 4 iterations of the Push-Relabel method on a graph 4×4. p. 60

17 A graph comparison between the GPU Push-Relabel and CPU Boykov-Kolmogorov algorithms without Multilevel Narrow Banded method. . . p. 63

18 A graph comparison between the GPU Push-Relabel and CPU Boykov-Kolmogorov algorithms using the Multilevel Narrow Banded Graph Cuts with one level. . . . . p. 64

19 A graph comparison between the GPU Push-Relabel and CPU Boykov-Kolmogorov algorithms using the Multilevel Narrow Banded Graph Cuts with two levels. . . . . p. 64

20 Examples of image segmentation with and without Narrow Banded Graph Cuts. . . . . p. 66

21 Example of image segmentation of a 320 x 200 image. . . . . p. 67

22 Example of image segmentation of a 640 x 427 image. . . . . p. 68

23 Example of image segmentation of a 1024 x 683 image. . . . . p. 69

24 Example of image segmentation of a 1920 x 1281 image. . . . . p. 69

25 Example of three video frames doing human skin detection. . . . . p. 70

List of Tables

1 Definition of the weight costs of the edges based on the energy function terms. . . . . p. 36

2 Time comparison between Boykov-Kolmogorov algorithm and Push-Relabel without Narrow Banded technique. . . . . p. 63

3 Time comparison between Boykov-Kolmogorov algorithm and Push-Relabel at Level 1 of Narrow Banded technique. . . . . p. 64

4 Time comparison between Boykov-Kolmogorov algorithm and Push-Relabel at Level 2 of Narrow Banded technique. . . . . p. 65

5 Total errors of Narrow Banded Strategy compared to the original Push-Relabel. . . . . p. 65

6 Time analysis of video segmentation of human skin. . . . . p. 68

1 Introduction

In Computer Vision and related areas, it is very common to estimate a set of measurements in a discrete representation of the domain of a problem. For motion and stereo, these measurements can be disparities, while for image restoration or segmentation they can represent intensities. An example of an image segmentation is depicted in the Figure 1. Such measurements tend to vary smoothly almost everywhere, possibly changing significantly at the object boundaries. In the case of 2D images, the problem is to determine a mapping between a set of pixels $p \in P$ and a set of labels $l_p \in L$ where l_p is consistent and piecewise smooth. This is called the *Pixel Labelling Problem* (VEKSLER, 1999).

Figure 1: Example of a binary segmentation.

The Pixel Labeling Problem can be solved by using an Energy Minimization model. This provides an elegant framework in which the problem specification is posed independently from the method used to obtain it. It is possible to treat ambiguities in the early formulation of the problem to be solved in a manner that leads to a spatially smooth solution. Energy Minimization models are very popular in Computer Vision, being widely used in different problems that can be solved using a general framework. Some of these problems are: optical flow (SETIAWAN; LEE, 2007), image restoration (GREIG; PORTEOUS; SEHEULT, 1989), scene reconstruction (KOLMOGOROV, 2004), stereo matching (HONG, 2008), texture synthesis (KWATRA et al., 2003) among others.

The approach used in Energy Minimization is typically divided into two different steps. The first one is the formulation of an energy function which satisfies some prior conditions. This function measures the cost of each solution in the set of all feasible

solutions for the problem, that is, those that satisfy the set of the constraints defined previously. Usually, it is a summation of terms corresponding to soft or hard restrictions. It assigns high goodness to the solutions that satisfy the defined rules.

The second step is the minimization of the energy function which is rarely a trivial problem, because the most interesting functions are not convex, having many local minima and thousands of dimensions. The design of the energy function has an effect upon the optimization process. It is hard to determine an efficient and robust minimization method for the most cases and a few many produces only approximate solutions.

There are many advantages associated to the Energy Minimization approach. Besides creating a standard framework for different vision problems, it also allows abstraction from the details of each particular problem in a particular energy function. Once an energy function is defined for a different problem, the same optimization algorithm can be used to solve it.

In spite of its many advantages, its use is not so simple as it may suggest. The computational cost of the minimization step is usually high. For certain category of energy functions it is NP-Hard, which hinders the application of tractable methods to find the optimal solution. In order to deal with such problems, several approaches were proposed to obtain approximate solutions or even the exact global minimum of certain energy function classes. Many works have proposed different strategies to reduce the optimization complexity but several have yielded poor results in an attempt to minimize certain energy classes. In these cases such methods took an extremely long time to converge and in other situations they simply got stuck in the local minima, giving solutions with bad visual appearance.

Recently, a new method called Graph Cuts has emerged which can obtain the local minima solutions with strong properties or even the exact optimum. The Graph Cuts method has proven to be a useful multidimensional optimization tool enforcing piecewise smoothness while preserving relevant sharp discontinuities (BOYKOV; VESKLER, 2006). The execution can be done with polynomial complexity.

One of the disadvantages of the Graph Cuts is its applicability only for restricted classes of functions, limiting the possible problem scope. However, that does not impose critical limitations because these functions are very natural to Computer Vision. The two most used constraints in Computer Vision that are successfully incorporated into the Graph Cuts framework are based in the *local data* and in the *smoothness knowledge*. We can formulate a basic energy function as,

$$E(l) = E_{data}(l) + E_{smooth}(l) \tag{1.1}$$

where the data term restricts a desired solution to be very close to the local observed data and the smoothness term measures the extent to which l is not piecewise smooth. The former constraint favors solutions that explain well the observed data and the latter constraint tends to favor spatially smooth solutions. An ideal solution involves the balance of these parameters. Smaller values of the objective function means higher quality goodness and its global minimum is the optimal solution.

Note that Graph Cuts can be applied only to a limited set of energy functions. Hence, the choice of the minimization method can no longer be defined independently of the cost function. It is fundamental to design the energy function in a way that allows efficient minimization via Graph Cuts. This method constructs a specialized graph such that its *Minimum Cost Cut* exactly minimizes the energy function, providing a global optimal solution for N-dimensional spaces when the energy function is appropriately defined.

A very important property of the Minimum Cost Cut is its equivalency with the *Maximum Flow* problem of Network Flows Theory. This is established by the Ford and Fulkerson theorem (FORD; FULKERSON, 1962). The Maximum Flow is a very well known problem, being calculated with low-polynomial order complexity in many cases. Also, it is possible to solve the minimum cost cut using the Maximum Flow solution, obtaining the global optimum result in polynomial time.

1.1 Objectives

This work focuses on providing efficient solutions for the Image and Video segmentation problems using the Graph Cuts method in many-cores architectures. It is based in an extensive investigation of the state-of-art techniques and strategies proposed for the implementation of Graph Cuts in GPUs. The results of such investigation has led us to develop our own implementation and uncover the details of the algorithms and heuristics used to improve it which are sometimes obscure in some works found in the literature.

The main contribution is an original technique called *Parallel Multilevel Narrow Banded Graph Cuts* structure. It is based on the work from Lombaert et al (LOMBAERT et al., 2005). Its fundamental idea is to compute the segmentation in a coarse version of the original image, and project the obtained result in a finer level of resolution. The segmentation in this projection needs to be recomputed only at a buffer region between

the background and object partitions. The process is repeated successively until the final solution is obtained for a given level of refinement.

The Parallel Multilevel Narrow Banded strategy yields visual results equivalent to those obtained by the application of the traditional method to the original image in the highest resolution. Furthermore, it leads to a reduction in the total number of processed nodes, improving the time complexity and decreasing the memory usage. Potential errors may be inserted, but in a small factor. Our results were compared with other works, obtaining very promising results. Although we concentrated in the use of such method for the image and video segmentation problem, the results can be used to improve the solution of other problems formulated using the Graph Cuts methodology.

1.2 Structure

This work is organized as follows: the next chapter describes the related works that influenced our approach, like extensions and parallel implementations of the original Graph Cuts idea; Chapter 3 summarizes the work foundation, explaining how the image segmentation problem can be modeled using the Energy Minimization framework and solved via Graph Cuts, concluding with a formulation of the narrow banded heuristic to improve its execution; Chapter 4 presents the basic computational solution employed to implement Graph Cuts, describing how a graph is represented and the most common algorithms used to solve it, doing a brief comparison; Chapter 5 discusses how these structures can be implemented in the GPU architecture, including the entire implementation and heuristics used to improve the performance; in Chapter 6 the experimental results are evaluated for image and video segmentation of the referred method in GPU and how efficient it is under some circumstances; finally in Chapter 7 our conclusions and future works are featured.

2 Related Works

Many works have studied how to solve global minimization energies for Computer Vision, but the first were Greig et al (GREIG; PORTEOUS; SEHEULT, 1989) for image restoration. It was the first work to discover that powerful graph-based algorithms for combinatorial optimization can be used to minimize certain important energies in Computer Vision. They created an image-based two terminal s-t graph whose construction is based on the Gibbs Energy (GREIG; PORTEOUS; SEHEULT, 1989; VINEET; NARAYANAN, 2008; GARRET; SAITO, 2009; BOYKOV; VEKSLER; ZABIH, 1999; BOYKOV; JOLLY, 2001; BOYKOV; KOLMOGOROV, 2004; SA et al., 2006; KOLMOGOROV; ZABIH, 2002; LI et al., 2004; VEKSLER, 1999) such that the minimum cost graph cut gives the optimal binary labeling solution. Previously, such energies could not be solved by exact minimization methods. One alternative was to use metaheuristics like Simulated Annealing (KIRKPATRICK; VECCHI, 1983) just to obtain an approximate solution.

Greig's paper demonstrated that simulated annealing strategy reaches solutions very far from the global minimum even in simple examples of binary image restoration and that their graph-based method behaved better. However, when the method was presented at the first time it was not noticed because of its apparent limitation, being applicable only to image clustering techniques. It remained unnoticed for almost 10 years.

The Graph Cuts concept was preceded by a number of graph-based methods for image clustering that used combinatorial optimization algorithms or approximate spectral analysis techniques, like normalized cuts (SHI; MALIK, 2000). These works are mentioned in Boykov and Funka-Lea (BOYKOV; FUNKA-LEA, 2006). The goal of some of these approaches is to produce a completely automatic high-level grouping of image pixels. This means that they divide an image into blobs or clusters using only generic cues of coherence or a measure of affinity between pixels. Differently, Graph Cuts integrate more appropriately model-specific visual cues and contextual information in order to define more accurately particular objects of interest. This is also related to other categories of segmentation methods like Snakes (KASS; WITKIN; TERZOPOULOS, 1988), Active Contours

(XU; AHUJA; BANSAL, 2007), Intelligent Scissors (STALLING et al., 1996) and Level-Sets (SETHIAN, 1999).

In the late 90's, new Computer Vision applications have appeared and researchers figured out how to use Graph Cuts algorithms for more interesting binary and non-binary problems. Some of these works are the ones proposed by Roy and Cox (ROY; COX, 1998) for multi-camera stereo and Boykov and Jolly (BOYKOV; JOLLY, 2001) for image segmentation. Other works focused on proposing different ways to define edge weights for similar graphs and generalizes the method to handle distinct energy functions, as in Ishikawa and Geiger (ISHIKAWA; GEIGER, 1998), Ishikawa (ISHIKAWA, 2003) and Boykov et al (BOYKOV; VESKLER, 2006). After that, other papers have studied theoretical properties of graph constructions, as in Kolmogorov and Zabih (KOLMOGOROV; ZABIH, 2002), discussing what energy functions can be minimized via Graph Cuts with global optimality. However, their results are applied only to energy functions of binary variables with double and triple cliques. Nevertheless, nowadays the full potential of the Graph Cuts method in multi-label cases is still not entirely understood. Problems in three or more dimensions are still considered relevant challenges.

Researchers have been creating or comparing approaches for the minimization step. Well known examples are Snakes (KASS; WITKIN; TERZOPOULOS, 1988), Intelligent Scissors (STALLING et al., 1996) and Level-Sets (SETHIAN, 1999). Techniques like Gradient Descent (YILDIZ; AKGUL, 2009; BURGES et al., 2005) can be applied to any energy functions of continuous variables and others like Simulated Annealing can be used in any function of discrete variables. However, these generalities can imply in very poor results since they get stuck in the local minima or take an extremely long time to converge.

Boykov and Jolly (BOYKOV; JOLLY, 2001) developed a technique to solve binary image segmentation by minimizing the Gibbs Energy using Graph Cuts. It was the first efficient work for N-dimensional applications, comprising region and boundary properties of elements. It is a general-purpose segmentation, needing initial user mark clues to characterize which elements probably belong to an object or a background set. Their solution allows new additional user marks even after the initial segmentation, without recalculating them from scratch. It was tested with 2D images and 3D volumes giving good and stable results even when the initial seeds were changed after the final result, performing with good speed. They motivated other works to approach different problems with similar methods.

Researchers have done other relevant contributions to extend Graph Cuts princi-

pally after Boykov and Jolly (BOYKOV; JOLLY, 2001). Some of them are geometric cues (BOYKOV; KOLMOGOROV, 2004; KOLMOGOROV; BOYKOV, 2005), regional cues based on Gaussian Mixture models for improved interactivity (ROTHER; KOLMOGOROV; BLAKE, 2004), super-pixels using watershed models (LI et al., 2004), integrating high-level contextual information (KUMAR; TORR; ZISSERMAN, 2005), segmentation using stereo cues (KOLMOGOROV et al., 2005), algorithms for dynamic applications (JUAN; BOYKOV, 2006), segmentation with 3D pose estimation (BRAY; KOHLI; TORR, 2006), segmentation uncertainty (KOHLI; TORR, 2006), solution of surface evolution PDEs (BOYKOV et al., 2006) and multilevel banded methods (LOMBAERT et al., 2005).

One of the main problems of the Graph Cuts approach is the computational complexity of its execution. Some of these minimization algorithms are quadratic or cubic, becoming very expensive for instances with a large number of pixels. Many works have studied the practical efficiency of it in Computer Vision and proposed major improvements (BOYKOV; KOLMOGOROV, 2004).

Boykov and Kolmogorov (BOYKOV; KOLMOGOROV, 2004) provided an experimental comparison between different Graph Cuts algorithms for Computer Vision applications. They analyzed the complexity of methods based in Goldberg and Tarjan (GOLDBERG; TARJAN, 1988), Ford and Fulkerson (FORD; FULKERSON, 1962) and a new method created by them. Their method was significantly faster than the previous ones. It is based on the original Ford and Fulkerson algorithm but it builds two simultaneous search trees for terminal nodes, reusing data at each iteration. Their implementation is the current sequential state-of-art method for Computer Vision implementations of Graph Cuts.

Other works have proposed heuristics to improve the original method. Juan and Boykov (JUAN; BOYKOV, 2006) proposed major improvements in the standard Graph Cuts algorithm. First of all, it uses the concept of initial cuts to start the method and enable a faster convergence. During the execution, their method returns some intermediate cuts, giving approximate solutions very rapidly. Their approach improved segmentation in static and dynamic images using the Pseudo-Flow algorithm (CHANDRAN; HOCHBAUM, 2009) instead of algorithms like Augmenting Paths or Push-Relabel. For videos, a single cut of the previous frame can be the initial cut of the next.

Lombaert et al (LOMBAERT et al., 2005) created a new heuristic to reduce the memory and time complexity of the standard Graph Cuts. It is a multilevel image strategy, where the original graph is lower scaled. The segmentation is done at the coarsest level and projected onto more refined levels. Potential errors can occur at the segmentation

boundaries and corrections are done only on these regions, defined as the *narrow band*. At each graph uncoarsening level, the segmentation is done only in the boundary regions, reducing the area to process and adjusting potential errors. As evaluated in their work, time and memory consumption are significantly reduced, principally for 3D input data and without major errors, never overcoming more than 3% in their experiments. The only problem of their approach is the loss of global optimal solution. But it remains very appropriate for huge data, like 3D volumes or videos.

Other works have proposed improvements on the definition of the cost function in the energy minimization framework. Li et al (LI et al., 2004) developed a method for interactive image cutout, focused on user usability without loss of performance. Their method consists of two steps: an object marking task, like presented in (BOYKOV; JOLLY, 2001) and a pre-segmentation computation, followed by a simple boundary editing process, creating the output segmentation. The nodes from the graph are not single pixels, but similar color regions generated by a Watershed algorithm (VINCENT; SOILLE, 1991), becoming superpixels. Their experiments yielded remarkable results in usability case studies, where users took overall less than 60% of the time using their software than traditional Magnetic Lasso feature present at common image processing software. The energy function used in our work is related to the one they proposed.

Vasconcelos et al (VASCONCELOS et al., 2007) developed a method to improve the efficiency of the Graph Cuts application reducing the total number of nodes at the energy computation task. Their method relies on a pre-processing step in the energy function computation that clusterizes similar pixels by using a quadtree data structure reducing significantly the total number of nodes of the graph to be segmented, accelerating the process. They showed how the quadtree structure can be constructed using graphics hardware via shaders. A reduction operator constructs an image pyramid which it marks for each texel when a similarity clustering was applied or not. All unnecessary information of non-leaf nodes is discarded. Their method was used to solve a foreground/background segmentation problem, achieving reasonably fast processing rates.

Lattari et al (LATTARI et al., 2010) proposed a new method to deal with the problem of automatic human skin segmentation on RGB color space model. An energy model is defined in terms of a database of skin and non-skin tones. Their work is based on the premise that skin colors form a small and unique subset of the RGB color space, which makes it easier to solve this specific case of segmentation. The proposed method has some advantages when compared with existing methods. Generally, Graph Cuts works

as (BOYKOV; JOLLY, 2001) are semiautomatic approaches, in which user interaction is required to determine seeds for the segmentation. Lattari's work proposed an automatic approach to determine an image segmentation without user interaction, just analyzing a database of people belonging to different ethnicities which is used to construct an energy function to be minimized using Graph Cuts. Also, the database can be used with different tones, not only skin segmentation. They obtained good visual results with this approach for human skin segmentation considering different ethnic groups and illumination conditions.

Some works attempted to improve the efficiency of the Graph Cuts method proposing versions running in parallel processors. Our work belongs to this category, running on Graphics Processor Units. Many parallel approaches were made using GPU Computing, after they became more popular and easy to use, mainly because of the development of programming libraries like CUDA (NVIDIA, 2009). Examples of works in this area are the ones proposed by Vinnet and Narayanan (VINEET; NARAYANAN, 2008), Hussein et al (HUSSEIN; DAVIS, 2007), Garret and Saito (GARRET; SAITO, 2009) and Yildiz and Akgul (YILDIZ; AKGUL, 2009).

Hussein et al (HUSSEIN; DAVIS, 2007) proposed the first implementation of the Graph Cuts algorithm in CUDA (NVIDIA, 2009) for general graphs. Hussein's paper proposed a CUDA version of the Push-Relabel algorithm, but it was not the first GPU implementation of it, made by Dixit et al (DIXIT; PARAGIOS, 2005). They presented some modifications on the original method: the only labeling scheme is the Global Relabeling heuristic (GOLDBERG; TARJAN, 1988) and sending flow from a node is an operation divided into two phases: *Push* and *Pull*. The first only sends flow, storing an amount on a temporary memory and the Pull updates all entire flow pushed before. This is necessary to avoid Read-After-Write hazard. They also introduced two optimizations. The first one is a lockstep Breadth-First Search which performs Prefix Sum Operations (BLELLOCH, 1990) when traversing each depth level. It can be considered a lockstep operation because only a single direction is traversed at a time, while the others are blocked. The second optimization is basically the emulation of a cache. Instead of loading the data of a single node in the global memory, it loads a 2D tile which is added into a lattice, a data structure created during the Breadth-First Search stage containing only visited nodes. Such strategy enables coalesced memory access, improving the algorithm speed. The speedup obtained is in the range of 1.7-4.5 over the CPU version proposed in (BOYKOV; KOLMOGOROV, 2004).

Vinnet and Narayanan (VINEET; NARAYANAN, 2008) proposed an implementation of Graph Cuts in CUDA for binary image segmentation using the Push-Relabel algorithm. They presented two versions of the same algorithm, using atomic and non-atomic operations. A new heuristic called Stochastic Cut is also proposed. This heuristic relies on the fact that after a few iterations, the processing done on the majority of the nodes is finished. When, at a given iteration, an entire thread block does not modify the residual graph by pushing flow, then the block is considered inactive and delayed for 10 iterations. Their implementation is very similar to the one described in (HUSSEIN; DAVIS, 2007), including the use of GPU shared memory and Push-Pull operations. One of the characteristics of their work is that it does not use Prefix Sum Operations. Finally, they defined a simple dynamic Graph Cuts implementation for video segmentation, always reusing solutions from the previous frames. They achieved a level of performance 10-12 times faster than (BOYKOV; KOLMOGOROV, 2004) and approximately 3 times faster than (HUSSEIN; DAVIS, 2007) for images. However, their approach is only applicable for grid graphs and not for general maximum flow calculations.

Garret and Saito (GARRET; SAITO, 2009) described a real-time silhouette extraction strategy for video segmentation, using GPU Graph Cuts. Their implementation uses a lock-free method for parallel Push-Relabel, always using atomic instructions. They developed a general structure for Graph Cuts, relying on additional null edges to overcome the problem of vertices with degrees different from the maximum degree, making the entire graph regular. They achieved very good speedups over the best sequential algorithms, obtaining 10 frames per second on full HD images.

Some GPU-based works do not use the Graph Cuts method directly. Yildiz and Akgul (YILDIZ; AKGUL, 2009) formulated the Graph Cuts optimization as a gradient descent solution on the GPU. Working differently from the previous Maximum Flow approaches, this solution is given by the Minimum Cut energy function formulation, solving the labeling problem directly without graph processing. It is based in Linear Programming and decreases spatial complexity. It is modeled by a Lagrange dual model and a modified aproximate objective function which is differentiable at any point. Their method needs less memory than the standard Maximum Flow methods, but gives some small errors at smooth regions because the used function is an approximation. For some examples, their method converges much faster then (BOYKOV; KOLMOGOROV, 2004).

3 Energy Minimization via Graph Cuts

The basic foundations of our solution are featured in this chapter. First, the Pixel Labeling Problem is presented. It is a generalization of some Computer Vision problems that can be formulated by using Graph Cuts framework in 2D, being also possible to extend it to 3D.

Energy Minimization Models which have a fundamental role in the Graph Cuts framework are also described. They define heuristic costs based on the pixels colors of the segmented image, determining the visual results of it. The processing time is an issue that also depends of the energy function design and must be taken into consideration. A very employed class of energy functions, called Gibbs Energy, is also investigated.

The theory necessary to understand the Graph Cuts method is featured later. We describe how the features of a graph can be represented in the context of Graph Cuts methods. The fundamental ideas used to solve the Graph Cuts method are also detailed in the corresponding section.

A connection between the concepts of the Energy Minimization models and the Graph Cuts method is addressed in Section 3.4. It summarizes the main results concerning the function classes that can be exactly optimized via Graph Cuts and describes a general workflow of how the energy function terms are incorporated into the Graph Cuts yielding the desired segmentation.

Finally, a heuristic called Multilevel Narrow Banded Graph Cuts is formally presented, showing how it works. Its use in Computer Vision problems, solved by Graph Cuts can be extremely advantageous, principally if we consider 3D volumes and 2D images with high resolution data.

3.1 Pixel Labeling Problem

Many Computer Vision problems are formulated using labeling approaches. Determining the skin pixels in an input image can be also modeled as a Pixel Labeling problem.

In order to specify a labeling problem, we need a set P of sites and a set L of labels. Sites represent arbitrary data set of image features such as pixels, voxels, edges, patches or segments in which we want to estimate some measurements. Labels attach some fundamental characteristic to a site $p \in P$ based on the quantity to be estimated. These sets are represented as:

$$P = \{p_1, p_2, ..., p_n\} \tag{3.1}$$

$$L = \{l_1, l_2, ..., l_k\} \tag{3.2}$$

The set P has frequently some natural structure, for instance, when representing a set of image pixels. Pixels in an image are arranged into a two dimensional matrix, which suggests some spatial relationship between elements.

It is necessary to model how these elements are connected and interact with each other. Each element q connected to p is considered a neighbor of p. The set of all elements nearby p is arranged on a set N_p and is called a *neighborhood* of p. The notion of neighborhood must satisfy the following properties:

1. $p \in N_p$
2. $p \in N_q \iff q \in N_p$

The connectivity and interaction among such elements is described by a *neighborhood system* N. N is the set of all neighboring pairs $p, q \in P$ and N_p composes a neighbor set of p. For matrix data, commonly used for representing rectangular images, N could represent a 4-neighborhood or a 8-neighborhood system. In the 4-neighborhood system, each element has a top, bottom, left and right neighbor.

The set L represents intensities, disparities or any other quantity to be estimated. Each label l represents some characteristic based on the elements $p \in P$. We want to classify the elements p based on the labels l. The labeling problem assigns a label l from the label set L to each site $p \in P$. Thus a labeling is a mapping from P to L. The solution of a labeling problem can be represented by a *characteristic function* X.

This function can be computed from some probability distribution that describes the likelihood of an element p belonging to a defined label l or not. The characteristic function X_A is a function defined for each p from the set P that determines whether p belongs to a subset A or not. It is represented as suggested in Equation 3.3.

$$X_A(p) = \begin{cases} 1, & p \in A \\ 0, & p \notin A \end{cases} \tag{3.3}$$

Equation 3.3 can be understood as follows. The label 1 assigns an element p to the set A. Elsewhere, the label 0 defines an element that does not belong to A. This is a discrete optimization problem, since the number of combinations is finite $|X| = l^n$ where $n = |P|$. As it can be seen, an exhaustive approach to find the optimal solution would have to search a huge number of possibilities, making such strategy extremely inefficient. Two examples of characteristic functions describing images are depicted in Figures 2 and 3.

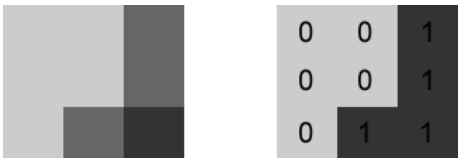


Figure 2: Characteristic function describing the segmentation of a gray tone image.

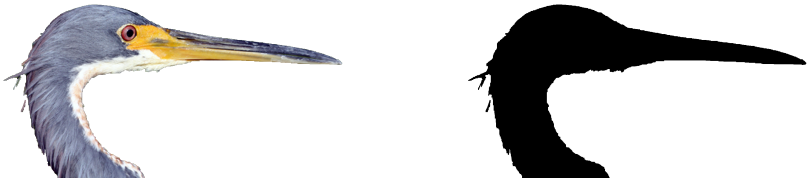


Figure 3: Characteristic function separating an object from the background.

In the context of our work, each element $p \in P$ represents a single image pixel. Thus, we will call P the *Pixel Set*. We will also use the 4-Neighborhood System. The set L has only two defined elements, one for the object region of interest and another for the background.

We will define these two labels as: O and B, respectively. Considering these statements, we will define our characteristic function for image segmentation as depicted in Equation 3.4 such that label 1 assigns an element p to the *Object Set* and the label 0 assigns an element p to the *Background Set*. The solution of the image segmentation

modeled by this problem is obtained by a characteristic function X of the input data. This function is treated as a matrix where the total number of elements is $|P|$ and one label from the set L has been assigned to each matrix element.

$$X(p) = \begin{cases} 1, & p \in O \\ 0, & p \in B \end{cases} \tag{3.4}$$

3.2 Energy Minimization Models

The design of an energy function is fundamental for computing the characteristic function. This calculation is done by a minimization process. More precisely, giving as input a 2D matrix data set M, the goal is to find a characteristic function X that is minimum among all possible arguments. Their general form can be described as (VEKSLER, 1999):

$$E(l_p) = E_1(x_p) + \lambda \cdot E_2(x_p) \tag{3.5}$$

The term $E_1(x_p)$ is called the *Data Energy* and assigns a cost for an individual p belonging to a set defined by a label $x_p \in X$. If p does not agree with the set, a heavy cost is obtained. However, if p is very similar to the set elements with a label l, then a small cost is generated. This term is typically straightforward to the energy design and its particular form depends on the level of noise of the imaging system. It measures how much the labeling to the pixel p disagrees with the observed data. This energy is adequate if each element information is considered independent. The only restriction is that it must not be negative.

The term $E_2(x_p)$ is called the *Smoothness Energy* and penalizes p giving high results when $p_q \in N_p$ have different labels, determining the smoothness solution. It assigns a heavy cost to the labeling which is not likely from the point of view of the prior knowledge. The design of such term is more tricky and depends on the problem at a hand. It is frequently hard to define this term even when the rules determining which labelings should have high or small cost are clearly stated.

One of the most discussed topics on Prior or Smoothness Energy is the concept of *discontinuity*. On many problems, the quantity to be estimated varies smoothly almost everywhere except at the object boundaries, where it may change significantly. The change across the boundary of the sets is considered a discontinuity. The term E_2 should

be designed appropriately considering the discontinuity property.

3.2.1 Gibbs Energy

It is possible to find a characteristic function of an object defined in a given domain by minimizing an objective function, i.e., given a set M, we have to find the characteristic function X which is the minimum argument of a function (BOYKOV; VEKSLER; ZABIH, 1999) and the partition sets. A widely used objective function in image segmentation is the Gibbs Energy (BOYKOV; VEKSLER; ZABIH, 1999; LI et al., 2004; GREIG; PORTEOUS; SEHEULT, 1989) defined as

$$E(X) = \sum_{p \in P} E_1(X(p)) + \lambda \sum_{\{p,q\} \in N} E_2(X(p), X(q)), \tag{3.6}$$

where $p, q \in P$, E_1 is the Data Energy and E_2 is the Smoothness Energy. The elements $p, q \in P$ belong to the set of elements P to be segmented, N is the set of connected elements and λ is a weight. Aiming to minimize the objective function, the term E_1 should be inversely proportional to the probability of p belonging to the set. It is usually given as

$$\begin{array}{lll} E_1(X(p)=1)=0 & E_1(X(p)=0)=\infty & \forall p \in O \\ E_1(X(p)=1)=\infty & E_1(X(p)=0)=0 & \forall p \in B \\ E_1(X(p)=1)=\phi(\rho_o) & E_1(X(p)=0)=\phi(\rho_b) & \forall p \in U \end{array} \tag{3.7}$$

where O is the set of object elements, B is the set of the background elements, U is the set of pixels whose labels are unknown and ϕ is a function inversely proportional to its parameters terms ρ_o and ρ_b, probabilites of p belonging to the set O and B, respectively. The first two equations define the sets of seed elements and the third defines the nonseed elements.

The minimization of some classes of energy functions can be considered a NP-Hard problem, requiring special methods to efficiently solve them. In our work we will minimize the Gibbs Energy using the Graph Cuts method. Chapter 3.4.1 describes how energy functions can be minimized in the context of Graph Cuts theory. For more details, see Boykov et al (BOYKOV; VEKSLER; ZABIH, 1999) and Kolmogorov and Zabih (KOLMOGOROV; ZABIH, 2002).

3.3 Background on Graphs

This section describes all concepts necessary to understand the minimization method related to Graph Cuts. The graph $G = (V, E)$ used in this work is a directed graph in which each edge $(u, v) \in E$ has a non-negative *capacity* $c(u, v) \geq 0$. If $(u, v) \notin E$, we assume $c(u, v) = 0$.

3.3.1 Graphs Representation

Graphs are an abstract notion used to represent some kind of connection between pairs of objects (AHO; HOPCROFT; ULLMAN, 1983). Many schemes are adopted to represent efficiently such connections as, for instance, *adjacency lists* and *adjacency matrices*. Other mechanisms can also be created to manage very specific graph types. It is important to be aware that none of these representations are better than the other. Their choices are dependent on the graph characteristics and the algorithm operations used.

Suppose a directed graph $G = (V, E)$ whose neighborhood is represented by an adjacency matrix, like in Figure 4(c). We can define the set $V = \{1, 2, 3..., n\}$, where $n \in \mathbb{N}$. The adjacency matrix A of G is an $n \times n$ matrix of booleans, where a_{ij} is true if and only if exists an edge from vertex i to j. Otherwise, the element indexed by (i, j) is assigned to zero.

$$a_{ij} = \begin{cases} 1, & (i, j) \in E \\ 0, & (i, j) \notin E. \end{cases} \tag{3.8}$$

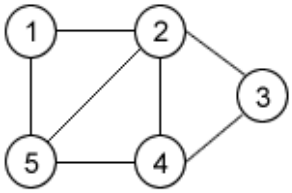
The main advantage of the adjacency matrix representation is that it enables very quickly accesses, independently of the size $|V|$ or $|E|$. This approach is very useful for the graph algorithms where it is important to check in a constant time if a given arc is present or not. This concept can also be extended to graphs with weighted values associated to each edge. Instead of storing a binary value indicating the presence of an edge (i, j) in the graph, it is possible to store a weight $c(i, j)$ according to a cost function c.

However, adjacency matrices have some disadvantages, according to the graph type used. It is not an appropriate choice to use if the graph is sparse, because the space complexity of this data structure is $\Omega(|V|^2)$, even if the graph has a number of arcs much less than $|V|^2$ arcs. An operation supposed to check all the positions of this matrix would require $O(|V|^2)$, increasing significantly the complexity of graph algorithms that traverse

the entire graph.

These problems are avoided with other representations which are more appropriate. One of these is the *adjacency list*. It stores in a list, in some order, all the vertices adjacent to a node i. The graph G can be represented by an array *HEAD*, where *HEAD[i]* is a pointer to the adjacency list of the node i. This representation requires storage proportional to the sum of the number of vertices plus the number of edges. It is adequate when $|E|$ is much less than $|V^2|$. An example of such structure is shown in Figure 4(b).

On the other hand, the adjacency list approach has also some problems. It is may take $O(n)$ time, where $n = |V|$, to determine whether there is an edge from i to j, since there can be $O(n)$ vertices on the adjacency list of i.



(a) Graph G

1 = {2, 5}
2 = {1, 5, 3, 4}
3 = {2, 4}
4 = {2, 3, 5}
5 = {1, 2, 4}

(b) Adjacency List of G

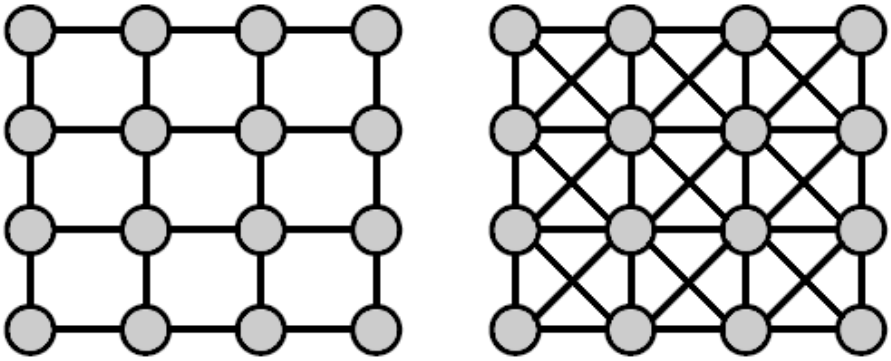
	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

(c) Adjacency Matrix of G

Figure 4: Graph neighborhood representation.

Depending on the problem domain, it is possible to specialize the adjacency list approach if the graph has some standard characteristics. For instance, the grid graph in Figure 5. Every vertex will have four edges, except the border nodes. It is a completely waste of resources if an adjacency matrix is used, because the total number of edges $|E|$ of G is small. Even an adjacency list defined with a dynamic pointer list is unnecessary, because the total number of adjacent nodes is fixed and the indices of the neighbors of a given node can be easily determined.

The grid graphs shown in Figure 5, exemplifies neighborhood structures based on 4 or 8 neighbors. Only the border nodes will have fewer neighbors than the center nodes. In order to define a grid graph we need two parameters: the *grid width* G_w which is the number of total nodes that compose a single graph row and the height of the graph G_h which is equal to number of all nodes that define a single column. All rows have the same number of nodes as the columns in a grid graph. It is important to be aware that the width of the graph does not have necessarily the same size of the height as they are two independent values.



(a) An 4-Neighborhood System (b) An 8-Neighborhood System

Figure 5: Example of a grid graph with different neighborhood systems for an 4x4 image.

Let $G(V, E)$ be a grid graph with a 4-Neighborhood System and $n \in V$ a node of G. In order to represent the edges using a neighborhood formalism, it is useful to define four functions that return the edges incident to n. However, these functions are only valid if the nodes are organized in a linearly indexed fashion. These functions represent 4 distinct edges outgoing a node in the top, bottom, left and right directions. The north edge function of a node n is defined by a function $e_{north}(n)$ which returns the edge $(n, n - G_w)$, outgoing to the top if it exists. The south edge function of n is represented by $e_{south}(n)$ which returns an edge $(n, n + G_w)$ if it exists. The west edge function $e_{west}(n)$ defines an edge $(n, n - 1)$ if it exists. Finally, the east edge function $e_{east}(n)$ returns an edge $(n, n + 1)$ if it exists. These functions are respectively formalized in Equations 3.9, 3.10, 3.11 and 3.12. Figure 6 shows an example of a highlighted node i where the four edge functions are represented for it, considering a graph G where $G_w = 4$ and $G_h = 5$.

$$e_{north}(n) = \begin{cases} (n, n - G_w) \in E, & n > (G_w - 1) \\ null, & n \leq (G_w - 1) \end{cases} \tag{3.9}$$

$$e_{south}(n) = \begin{cases} (n, n + G_w) \in E, & n \geq (|V| - G_w) \\ null, & n < (|V| - G_w) \end{cases} \tag{3.10}$$

$$e_{west}(n) = \begin{cases} (n, n - 1) \in E, & n \bmod G_w \neq 0 \\ null, & n \bmod G_w = 0 \end{cases} \tag{3.11}$$

$$e_{east}(n) = \begin{cases} (n, n + 1) \in E, & (n + 1) \bmod G_w \neq 0 \\ null, & (n + 1) \bmod G_w = 0 \end{cases} \tag{3.12}$$

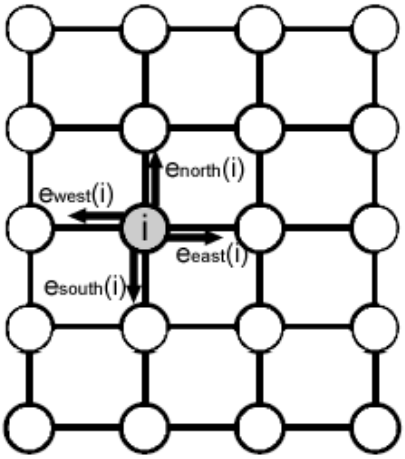
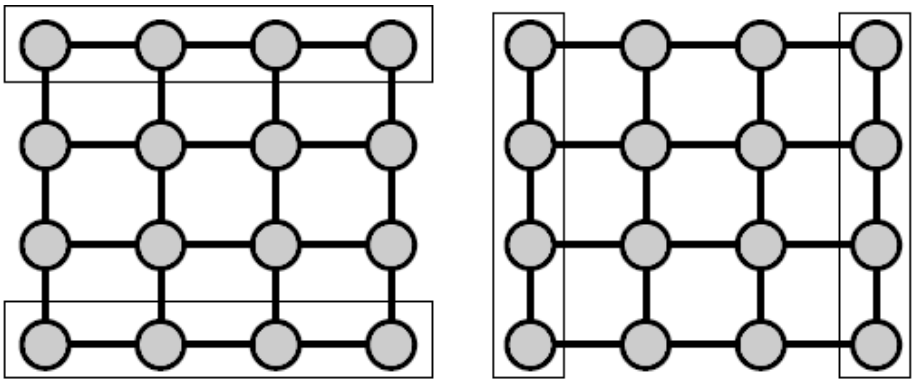


Figure 6: Example of a node i its 4 edges highlighted.

Algorithm 1 describes a routine to obtain the four neighbors of a grid graph G. The top neighbor of n exists if and only if n is not in the first row of the graph. Similarly, the bottom adjacent node does not exist if and only if n is not in the last row. Finally, n will have a left neighbor if it is in the first column and will have a right neighbor if it is in the last column. Figure 7 shows these nodes.



(a) Graph with the top and bottom nodes selected.

(b) Graph with the left and right nodes selected.

Figure 7: Grid graph with detached nodes on the nodes without top and bottom neighbors (Figure 7(a)). Similarly, a grid graph without left and right neighbors are detached in Figure 7(b).

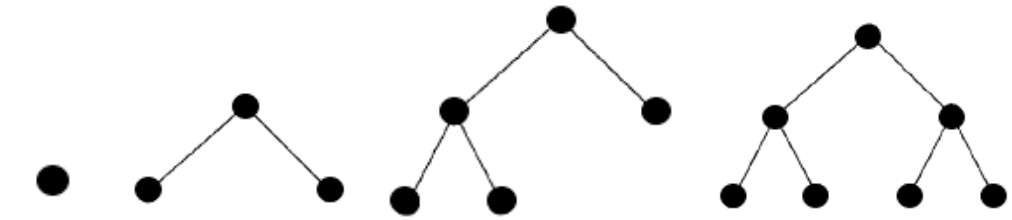
A very common operation on graphs is the search operation. One of the simplest algorithms for searching a graph and the archetype for many important graph algorithms is called the *Breadth-First Search* (CORMEN et al., 2001). Given a graph $G = (V, E)$ and a distinguished vertex s, the breadth-first search systematically explores the edges of G, discovering every vertex that is reachable from s. It starts from a s node at Level 0. Then s is expanded and their neighbors are obtained. These neighbors belong to Level 1. After this, each adjacent node of s is expanded. These expanded nodes will belong to Level 2,

Algorithm 1 Code snippet that determines the 4 neighbors of a node n in a grid graph similar to the Figure 5.

{Obtaining top neighbor of n}
if $i > G_w - 1$ **then**
  $n_{top} = n - G_w$
else
  $n_{top} = \emptyset$
end if
{Obtaining bottom neighbor of n}
if $i > |V| - G_w$ **then**
  $n_{bottom} = n + G_w$
else
  $n_{bottom} = \emptyset$
end if
{Obtaining left neighbor of n}
if $n \bmod G_w \neq 0$ **then**
  $n_{left} = n - 1$
else
  $n_{left} = \emptyset$
end if
{Obtaining right neighbor of n}
if $(n + 1) \bmod G_w \neq 0$ **then**
  $n_{right} = n + 1$
else
  $n_{right} = \emptyset$
end if

and so on. It is considered a very systematic strategy because it first considers all paths at the graph Level 1, then all those at Level 2, successively until finish.

A small example of such process is presented in Figure 8. It starts at Level 0 in the root node s (Figure (a)). The root node is expanded, discovering the neighbors of s in Level 1 (Figure (b)). The adjacent nodes of the first node of Level 1 are searched, starting Level 2 (Figure (c)). Finally, the neighbors of the second node of Level 1 are discovered, finishing the search (Figure (d)) (RUSSELL; NORVIG, 1995).



(a) Level 0 (b) Level 1 (c) Level 2 (d) Finished

Figure 8: Example of a Breadth-First Search operation on a binary tree.

The Breadth-First Search implementation is presented in Algorithm 2. This implementation builds a spanning forest, initially connecting only its root, represented in the implementation as node s. To implement it, a vertex queue and a binary array with size $|V|$ are necessary. The queue stores all visited nodes during the procedure and the binary array marks if a node was visited or not. It starts marking the node s as visited and stores it in the queue. An iteration is executed while the queue is not empty. All visited nodes will be enqueued during this stage. At each iteration, the node in the head of the queue is removed, and their adjacent nodes are marked as visited. This is repeated until the queue is empty.

Algorithm 2 Code snippet of the Breadth-First Search.

```
mark[s] ← visited
ENQUEUE(s, Q)
while |Q| ≠ 0 do
  u ← HEAD(Q)
  DEQUEUE(Q)
  for all v ∈ N[u] do
    if mark[v] = unvisited then
      mark[v] ← visited
      ENQUEUE(v, Q)
    end if
  end for
end while
```

Each visited vertex is placed in the queue once, so the body of the while loop is executed once for each vertex. Each edge (u, v) is examined twice, once for u and for v. The running time of this search is $O(MAX(|V|, |E|))$, when an adjacency list representation is used. Since $|E| \geq |V|$ is typical, usually this complexity is $O(|E|)$.

3.3.2 Graph Cuts

In order to understand the ideas behind the Graph Cuts method, it is necessary to define the concept of a *flow network* (CORMEN et al., 2001). A flow network is a graph $G = (V, E)$ where two vertices are distinguished: a *source* s and a *sink* t. We assume that every node lies on some path from s to t. Finally, $|E| > |V| - 1$. The *flow* in G is a real-valued function $f : V \times V \rightarrow \mathbb{R}$ that satisfies the following three properties:

1. $\forall(u, v) \in V \rightarrow f(u, v) \leq c(u, v)$.
2. $\forall(u, v) \in V \rightarrow f(u, v) = -f(v, u)$.
3. $\forall(u, v) \in V - \{s, t\} \rightarrow \sum_{u,v \in V} f(u, v) = 0$.

The defined quantity $f(u, v)$, which can be positive or negative, is called the *net flow* from vertex u to v and $c(u, v)$ is the capacity of an edge (u, v) of accepting flow. The value of a flow f is defined as,

$$|f| = \sum_{u \in V} f(s, u). \tag{3.13}$$

That is, the total net flow out of the source s. In the maximum flow problem, given a flow network G with source s and sink t, the goal is to find a flow of maximum value possible from s to t.

Intuitively, given a flow network G and a flow f, the *residual network* G_f consists of a graph $G(V, E_f)$ induced by the vertices V of G and the edges E_f that can admit more flow. Formally, the amount of additional net flow we can push from u to v before exceeding the capacity $c(u, v)$ is the residual capacity of (u, v), given by:

$$c_f(u, v) = c(u, v) - f(u, v). \tag{3.14}$$

In order to explain the concept of a cut of a graph it is necessary to define the concept of augmenting path. An *augmenting path* is a simple path from s to t in G_f. Each edge

of G_f admits some additional positive net flow from u to v without violating the capacity constraint $c(u, v)$ of the edge.

The concept of a cut of a graph is defined as follows. A cut $C(S, T)$ of a flow network G is a partition of V into two subsets S and $T = V - S$ such that $s \in S$ and $t \in T$. If f is a flow, then the new flow across the cut $C(S, T)$ is defined as $f(S, T)$. The capacity of the cut $C(S, T)$ is $c(S, T)$. An example of a graph-cut is presented in Figure 9.

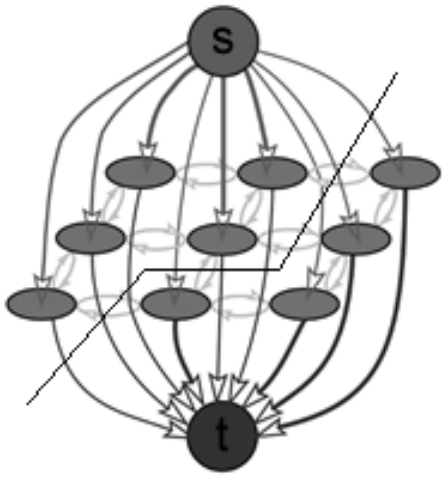


Figure 9: Example of a cut of a graph.

An example of a graph with a network flow describing a maximum flow is depicted in Figure 10. The theorem of Ford and Fulkerson (FORD; FULKERSON, 1962), also defined as a Maxflow-Mincut theorem, states that the minimum cost cut problem is equivalent to the maximum flow problem. In fact, the maximum flow value is equal to the minimum cost cut. If f is a flow in G, then the following conditions are stated:

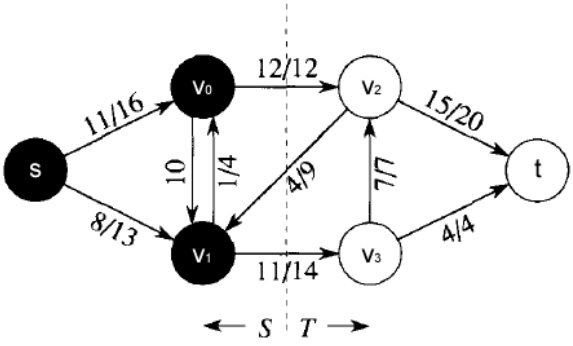


Figure 10: Example of a network flow with a maximum flow defined and a graph-cut of it. (CORMEN et al., 2001)

1. f is a maximum flow in G.

2. The residual network G_f contains no augmenting paths.

3. $|f| = c(S,T)$ for some $C(S,T)$ of G.

It is very simple to demonstrate a high-level proof of this statement. Let us consider the set of all possible cuts that separate s from t. Those cuts can be defined as a set of barriers that the flow has to cross to reach t. As the flow has to cross every barrier, then the maximum flow is bounded by the minimum capacity of the barriers. Since the barrier of the minimum capacity will be saturated first and separates s from t, the barrier of maximum flow is exactly the minimum cost cut.

3.4 Representing Energy Functions with Graphs

In this section the concepts necessary to represent Energy Function Minimizations via Graph Cuts are summarized. Let us consider the s-t graph $G = (V, E)$ with terminals s and t (KOLMOGOROV; ZABIH, 2002). Each cut on G has some cost given by an energy function mapping all cuts on G to the set of nonnegative real numbers. Any cut can be described by n binary variables $\{x_1, x_2, .., x_n\}$ where $n = |V - \{s,t\}|$. We define $x_i = 1$ when $v_i \in S$ and $x_i = 0$ when $v_i \in T$. Therefore, the energy F that represents G can be viewed as a function of n binary variables: $F(x_1, x_2, ..., x_n)$ and is equal to the cost of the cut defined by the configuration $x_1, x_2, ..., x_n$.

3.4.1 Function Classes Optimally Minimized via Graph Cuts

In order to execute the Graph Cuts framework with global optimality, the Energy Function needs to satisfy some conditions. A definition of a graph-representable energy function is presented in (KOLMOGOROV; ZABIH, 2002):

Definition 1. *A function F of n binary variables is called graph-representable if there exists a graph $G = (V, E)$ with terminals s and t and a subset of vertices $V_o = \{v_1, v_2, ..., v_n\} \subset V - \{s,t\}$ such that, for any configuration $x_1, x_2, ..., x_n$, the value of the energy $F(x_1, x_2, ..., x_n)$ is equal to a constant plus the cost of the minimum st-cut among all cuts $C = (S,T)$ in which $v_i \in S$ if $x_1 = 1$, and $v_i \in T$ if $x_1 = 0 (1 \leq i \leq n)$. We say that F is exactly represented by G and V_o, if this constant is zero.*

This definition formalizes the reducibility of the solution of F to a result of the minimum cut, meaning that these two problems are equivalent. But this is correct only if

the value of the energy F is identical to a minimum st-cut of a graph G, with the same configuration of the binary variables $\{x_0, x_1, ..., x_n\}$. This set represents our characteristic function X.

Only a class of Energy Functions can satisfy the Definition 1. A theorem was proposed by (KOLMOGOROV; ZABIH, 2002) to elucidate it:

Theorem 1. *Let F be a function of n binary variables described as following:*

$$F(x_1, x_2, ..., x_n) = \sum_i E_1^i(x_i) + \sum_{i<j} E_2^{i,j}(x_i, x_j).$$

Such that i and j are element sets. Then, F is graph-representable if and only if each term $E_2^{i,j}$ satisfies the inequality

$$E_2^{i,j}(0,0) + E_2^{i,j}(1,1) \leq E_2^{i,j}(1,0) + E_2^{i,j}(0,1),$$

such that 0 and 1 are labels that assigns the elements i and j to the respective sets. Functions that satisfy the condition from the Theorem 1 are considered *regular*. This theorem states that regularity is the only necessary and sufficient property to allow a graph representability by an energy function. However, this statement is only valid when F is defined on binary variables. A generalization is necessary to encompass more than two labels. A demonstration of this generalized theorem and its proof is presented in Kolmogorov and Zabin (KOLMOGOROV; ZABIH, 2002).

3.4.2 General Workflow

This section describes in more details the execution of the Graph Cuts algorithms in graphs based on images. We consider here that the subset of pixels marked as object is given by a set O_{seeds} and the subset marked as background is given by a set B_{seeds}. If a pixel p is marked as an object seed, then $p \in O_{seeds}$. If it is marked as a background seed, then $p \in B_{seeds}$. Also, we have $O_{seeds} \cap B_{seeds} = \emptyset$.

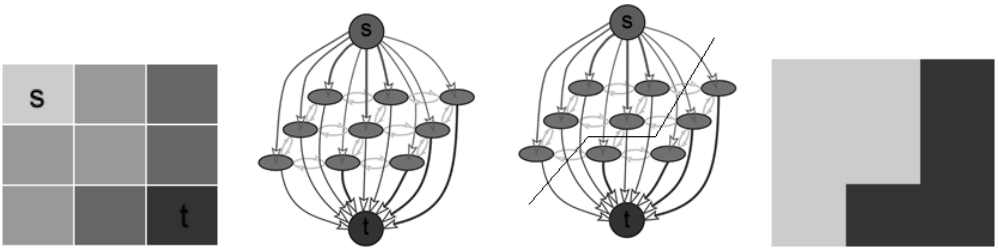
There are two methods for constructing the graphs based on Energy Minimization models defined on images. The graph proposed by Boykov et al (BOYKOV; VEKSLER; ZABIH, 1999) introduces auxiliary vertices and edges to solve graphs with arbitrary finite set of labels L. They use two methods called *swap move* and *expansion move* to obtain the

solutions. Both execute the same operation, where the expansion move can be considered an extension of the swap move.

Kolmogorov and Zabih (KOLMOGOROV; ZABIH, 2002) defined a graph construction which maintains a grid structure very similar to the one depicted in Figure 11. It is applicable to binary labeling and is very well suitable to SIMD architectures, as for instance, the GPUs. It is the strategy used in our work. The directed graph $G = (V, E)$ is constructed in a way where each pixel p of the original image is represented by a node $n_p \in V$. If two pixels p and q are neigbours, then an edge $e_{p,q} \in E$ is represented connecting them.

The general workflow to create a graph based on an image is described in Figure 11. Each pixel $p \in P$ is mapped to a node $n \in V$ of G. Given an image (Figure 11(a)), its graph representation is depicted in Figure 11(b). The edge weights reflect the parameters in the terms of Equation 3.7.

The next step is to compute the globally optimal minimum cut of the graph (Figure (c)). This is done by obtaining the maximum flow of the graph constructed in the previous stage. This cut gives a segmentation of the original image (Figure (d)).



(a) The original image I with two pixel seeds.

(b) The graph G_i based on I.

(c) The graph-cut of G_i.

(d) The output segmentation I.

Figure 11: Example of a graph based on an 3x3 image and the cut of it.

To segment a given image we create a graph $G = (V, E)$ with nodes $n \in V$ corresponding to the pixels $p \in P$ of the image. There are two additional nodes: a terminal s corresponding to the object region and a terminal t corresponding to the background region such that $V = V \cup \{s, t\}$.

The set of edges E consists of two types of undirected edges: *t-links* (terminal links) and *n-links* (neighborhood links). Each node n has two t-links (n, s) and (n, t) connecting n to each terminal. Each pair of neighboring pixels (p, q) where $q \in N_p$ is connected by a n-link. We can define the set E as $E = N_p \cup \{(p, s), (p, t)\}$. Table 1 defines the weight of the edges.

Table 1: Definition of the weight costs of the edges based on the energy function terms.

Edge	Weight Cost	If
(p, q)	$E_2(X(p), X(q))$	$(p, q) \in N_p$
(p, s)	$\lambda\phi(p_o)$	$p \in P, p \notin O_{seeds} \cup B_{seeds}$
	∞	$p \in O_{seeds}$
	0	$p \in B_{seeds}$
(p, t)	$\lambda\phi(p_b)$	$p \in P, p \notin O_{seeds} \cup B_{seeds}$
	0	$p \in O_{seeds}$
	∞	$p \in B_{seeds}$

The minimum cut of the graph so defined can be calculated by the maximum flow solution given by the Push-Relabel algorithm (GOLDBERG; TARJAN, 1988), which will be described in the next chapter. The sets O and B determine the object region and the background region, such that $O \cap B = \emptyset$.

3.5 Narrow Banded Strategy in 2D Images

The multilevel strategy is a heuristic based in the level sets literature to produce high quality solutions based in Graph Cuts, reducing the computational burden as said in Lombaert et al (LOMBAERT et al., 2005). In this technique, Graph Cuts are performed on a low-resolution graph image and the solution is projected to the next level by only computing the graph of the boundary surrounding the interface. Because the execution is only in the subgraph that comprises a narrow band, the time and space complexity is significantly reduced.

As mentioned in the last section, Graph Cuts are a powerful tool for Computer Vision problems, but the speed and memory consumption constrains are challenging problems to be solved. For example, the memory allocation of the method proposed by (BOYKOV; KOLMOGOROV, 2004) needs $24|V| + 14|E|$ bytes. For an example, a typical CT volume of size 512^3 can consume more than 8GB (LOMBAERT et al., 2005). Moreover, finding the minimum cut of a graph like this is impractical due to the polynomial worst case complexity.

This approach is very inspired by the multilevel graph partition methods as well as the well-known narrow band algorithm of the Level Sets. First, the minimum graph cut of a reduced version of the original graph is computed. After this, the minimum graph cut at successive higher levels of resolution are computed but only on a narrow band surrounding the boundary regions of object and background segments. This is necessary because at

higher resolutions errors may appear at these regions. By doing this, it is possible to achieve high quality segmentation results on large data sets with faster speed and less memory consumption allowing it to be used in a wider range of medical applications and where high performance of large data sets with real-time execution is crucial.

The multilevel banded Graph Cuts from (LOMBAERT et al., 2005) consists of three steps: coarsening, initial segmentation and uncoarsening. The coarsening is done directly on the image. This can be done with any standard multiresolution image technique. An example of simple procedure that can be used is downsampling. The original image is not considered part of the memory consumption overhead of this step. All these steps are depicted in Figure 12.

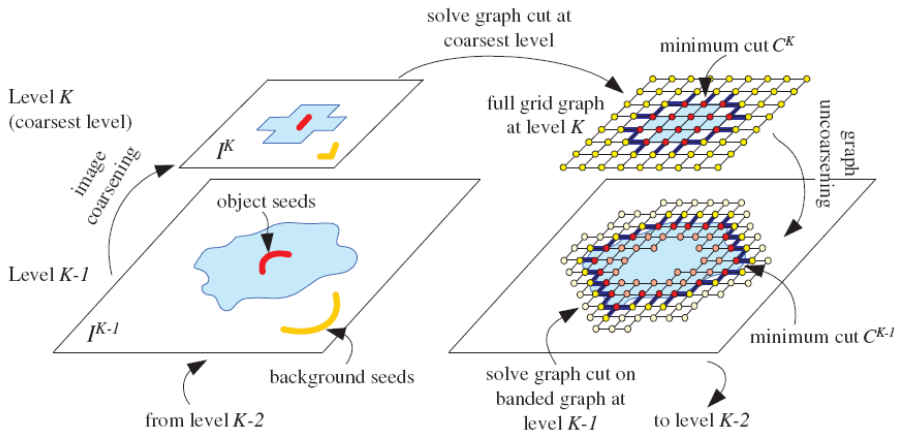


Figure 12: The multilevel banded Graph Cuts procedure. (LOMBAERT et al., 2005)

During the first stage, several many smaller images $\{I_1, I_2, ..., I_k\}$ are constructed based in the original image I_0 such that the size constraint $M_k < M_{k-1}$ is satisfied for each image dimension $n = \{1, 2, ..., k\}$ and each image level $k = \{1, 2, ..., k\}$. Obviously, the image seeds are also reduced and the multilevel graphs are constructed directly based upon these low resolution images.

The second stage is the segmentation of the coarsest image I_k where k is the largest level defined in this instance problem. A graph $G_k = (V_k, E_k)$ is defined for I_k and their minimum cut is obtained. This minimum cut yields a segmentation of the image I_k.

During the final step, a binary boundary image J_k is constructed, representing all these image elements that are identified by the nodes of the cut C_k, $k \in \{1, 2, ..., k\}$. The boundary image is projected onto a higher resolution boundary image J_{k-1} at level $k-1$. The resulting boundary image J_{k-1} contains a narrow band that limits the candidate boundaries of the object elements to be extracted from I_{k-1}. The band width can be controlled by an optional dilation parameter $d > 0$. If d is small, the method may not be

able to recover the full details of the objects with high shape complexity or large curvature. Moreover, if d is large, the computational benefits of banded graph cuts are reduced and the wider band may also introduce potential outliers far away from the desired object boundaries.

The graph G_{k-1} is constructed as follows. It is defined by the nodes inside the band from the boundary binary image J_{k-1}. The band outer layer region may introduce seeds for the background and the inner layer does the same for the object seeds. The seeds from J_{k-1} are used if no inner layer is produced due to segmenting tiny objects. Finally, new edges are assigned based in the cost function and afterwards, the minimum cut of G_{k-1} is calculated. The same procedure is repeated until minimum cut C_0 is solved, yielding the final result. A simple example of this process is pictured in Figure 13.

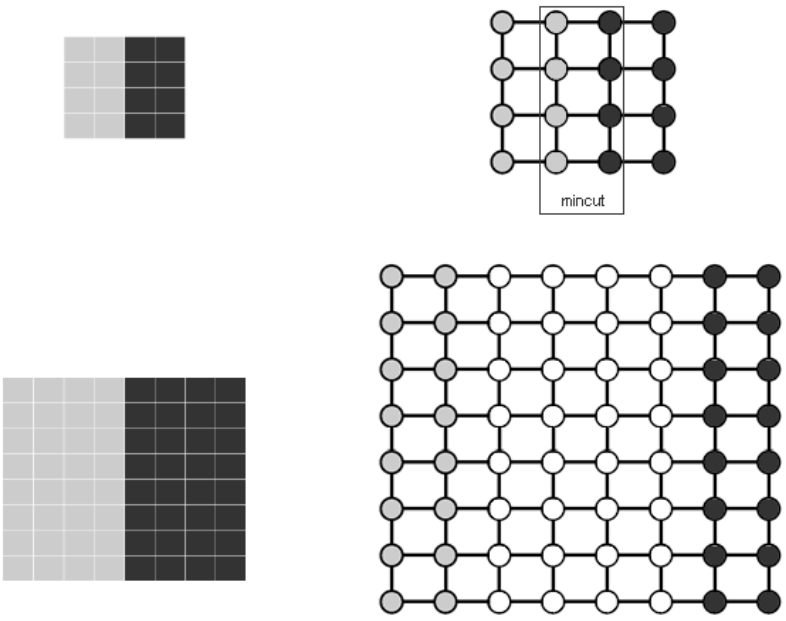


Figure 13: Example of an 8×8 image being projected to an upper level, becoming a 4×4 image.

A simple example of downsampling and upsampling is featured in Figure 13. During the coarsening step, the reduced image graph is constructed and after that the minimum cut nodes are obtained. The nodes that belong to the object set are gray and to the background set are black. When the uncoarsening operation is done and the graph is projected to a lower level, each node projects directly onto 4 nodes below. The projected nodes could be assigned into three different sets: to the Object Set, to the Background Set or to an unknown set. The unknown set nodes on the higher plane are part of the narrow band. In this level, only narrow band nodes form the graph that will be segmented

using the Graph Cuts method.

It is very important to observe that all multilevel graphs have band structure, except the one that has the highest level. This reduces significantly the amount of nodes to be processed compared to the image at the same level. It is unnecessary to process these nodes outside the band because they are the projected nodes of the upper level, which have well defined labels. Because a much smaller graph is processed in all resolutions, both the run-time and the memory consumption are reduced compared to the standard Graph Cuts. As evaluated in tests of (LOMBAERT et al., 2005), the complexity reduction for time and memory is on the order of magnitude of ten to the one.

4 Graph Cuts Algorithms

This chapter describes the details necessary to understand the computational implementation of the Graph Cuts method. The minimization algorithms that can be used to solve Graph Cuts with their implementation details and complexity study are featured in the following sections.

4.1 Algorithms

Two main classes of algorithms were proposed to solve the maximum flow problem for Graph Cuts minimization. One category is based on the Ford and Fulkerson original idea (FORD; FULKERSON, 1962) which enforces flow conservation during the whole process. Another category formulated by Goldberg and Tarjan (GOLDBERG; TARJAN, 1988), breaks the flow conservation rule until convergence. Examples of these will be described below.

4.1.1 Algorithms based on augmenting paths

The classical maximum flow algorithm based on the notion of augmenting paths was proposed by Ford and Fulkerson (FORD; FULKERSON, 1962). In order to understand the Ford-Fulkerson maximum-flow algorithm we must consider the concept of *augmenting paths*, which was defined in Section 3.3.2.

The Ford-Fulkerson Algorithm is based on the following statement: at each iteration, the algorithm tries to find an augmenting path p_a in $G_f = (V, E)$ capable of increasing the network flow by the largest residual capacity $c_f(p_a) = min\{c_f(u, v), \forall(u, v) \in p_a\}$ admissible along p_a. It updates continuously the net flow $f(u, v)$ between each pair (u, v) of vertices that are connected by an edge, if such edge exists (see Algorithm 3).

The first loop initializes the flow in the entire network to zero. The second loop repeatedly finds an augmenting path $p_a \in G_f$ and augments flow f along p_a by the

Algorithm 3 Ford-Fulkerson Maximum Flow calculation procedure.

```
for all (u, v) ∈ E do
    f(u, v) ← 0
    f(v, u) ← 0
end for
while exists an augmenting path p ∈ G_f do
    c_f(p) ← min{c_f(u, v), ∀(u, v) ∈ p}
    for all (u, v) ∈ p do
        f(u, v) ← f(u, v) + c_f(p)
        f(v, u) ← −f(u, v)
    end for
end while
```

residual capacity $c_f(p_a)$. When no augmenting paths are found, the algorithm is finished.

The running times of the algorithm depend on how the augmenting paths are chosen. If the augmenting paths are chosen by using a Breadth-First Search to find the shortest paths, it runs in polynomial time complexity $O(VE^2)$. This variation of the algorithm is called Edmonds-Karp algorithm (EDMONDS; KARP, 1972).

Boykov and Kolmogorov (BOYKOV; VEKSLER; ZABIH, 1999) proposed a new method to solve the maximum flow associated to Computer Vision problems using an approach based on Ford-Fulkerson algorithm. Instead of searching the shortest path, it searches for the maximal augmenting path. It uses two search trees on the residual graph, one with its root at the source s and another starting at the sink t. Each tree grows from its own terminal node.

When the two trees touch each other, an augmenting path is found. Flow is sent in this path as much as possible. After that, the residual graph is updated and new paths are searched, reusing the trees. The algorithm is finished when no other path can be found (Algorithm 4).

Algorithm 4 Boykov-Kolmogorov Maximum Flow calculation procedure.

```
while a tree can grow do
    Grow alternatively one tree.
    while There is a contact between the two trees do
        Detect and store the augmenting path.
        Send as much flow as possible through the path p_a.
        Update the residual graph G_f, until one edge e ∈ p_a is saturated.
        Fix the trees.
    end while
end while
```

The critical point of this algorithm lies on the management of the trees, for example

in growing or updating it to obtain a short augmenting path. It does not guarantee to find the shortest path as it uses some heuristics to search for the shortest one. The use of such heuristics produces good quality paths while keeping the efficiency of the overall process. In fact, the method has a compromise between choosing any path or the shortest one. The theoretical time bound complexity of this algorithm is $O(VE|C|)$, but in practice it is almost linear.

Boykov-Kolmogorov is considered the best method to compute Graph Cuts in Computer Vision for sequential machines. However, new parallel approaches using the Push-Relabel algorithm have given best times over it. This algorithm will be described in the next topic.

4.1.2 Push-Relabel

The Push-Relabel algorithm works in a more localized manner than the augmenting-paths methods. Instead of examining the entire residual network $G_f = (V, E)$ to search for an augmenting path, generic Push-Relabel algorithm works on one vertex at a time, analyzing and operating on its neighbors in the residual network. Furthermore, unlike the augmenting-paths based methods, it does not maintain the flow conservation property throughout the execution. The third rule in the flow definition (Eq. 3) is modified to:

$$\forall u \in V \quad \sum_{v \in V} f(u, v) \geq 0 \tag{4.1}$$

More precisely, during the method execution, the flow entering a node is not necessarily equal to the flow exiting it. This means that the original rule is relaxed.

The positive difference between a flow entering a node u and exiting it is called an *excess flow* e of u. A node with positive excess flow is called an excess node. An outgoing edge of u is any edge of the residual graph that leaves the node. Similarly, an incoming edge of u is an edge of residual graph that reaches node u. Based on these definitions, the excess e of a node u is a function given by:

$$e(u) = \sum_{|e_i|} f(e_i) - \sum_{|e_o|} f(e_o) \tag{4.2}$$

where e_i is an incoming edge of u and e_o is an outgoing edge of u. The excess flow e is the difference between the total incoming flow into the node minus the outgoing flow.

Using the previous concepts, it is possible to present a definition of a *preflow*. A *preflow* is a function $f : V \times V \rightarrow \mathbb{R}$ that satisfies:

1. $\forall(u, v) \in V \rightarrow f(u, v) \leq c(u, v)$.
2. $\forall(u, v) \in V \rightarrow f(u, v) = -f(v, u)$.
3. $\forall(u, v) \in V - \{s, t\} \rightarrow \sum_{u,v \in V} f(u, v) \geq 0$.

The first algorithm based on the idea of relaxing the flow conservation was introduced independently by Cherkassky (CHERKASSKY, 1979) and by Goldberg and Tarjan (GOLDBERG; TARJAN, 1988). The method presented in our work is based on algorithms of this category.

The basic intuition of this method is very different from those based on augmenting paths. Each node has two additional properties. First, to accommodate the excess flow, it is considered that each vertex has an outflow pipe leading to an arbitrary large reservoir that can accumulate fluid. Second, each vertex is on a platform whose height label increases as the algorithm progresses. The height label determines how the excess fluid of a node is pushed: only nodes with higher labels can push flow to the vertices at lower levels. All nodes start with a height and an excess flow equal to zero, except the height of the source s that is fixed at $|V|$ and the excess of s which is infinite. Naturally, the source s will push flow to all nodes connected to it.

Consider the nodes $\{u, v\} \in G_f$. The operation that pushes excess flow of a node u into a neighbor $v \in N_u$ is called *Push*. Eventually, the algorithm will try to push the excess flow of u into its neighbors, but none of these nodes $v \in N_u$ has height label below the height of u. To rid an overflowing vertex u of its excess flow, it is necessary to increase its height. Such operation is called *Local Relabel*. The height of u is increased by one unit above the height of the lowest neighbor that has an unsaturated edge connecting them. After the local operation, the excess of u can be pushed. When all paths to t are saturated, the algorithm has to send the remaining excess flow in the system back to the source s, by continuously increasing height of the vertices with excess flow until they achieve $|V|$. After this point, the preflow becomes a legal flow and also a maximum flow.

Formalizing, the Push-Relabel algorithm maintains the residual graph G_f of G, where the edges have costs equal to the amount of additional flow c_f we can push from node u to v, where $c_f(u, v) = c(u, v) - f(u, v)$. The value $c(u, v)$ is the capacity of the edge and $f(u, v)$ is the pushed flow necessary to saturate it. The value c_f is also called *residual*

capacity. Two variables are also defined for all vertices V: the excess flow stored into the node $e(u)$ and the height $h(u)$. The height h is an estimate of the vertex distance from u to the target node t.

Two node operations are defined: the Push operation is applied for $\{u, v\} \in V$ if $e(u) > 0$ and $h(u) = h(v) + 1$ for at least one $v \in N_u$ (Algorithm 5). The excess flow is pushed then from u to v until $e(u) = 0$ or $c_f(u, v) = 0$. The second operation is the Local Relabel, applied when $e(u) > 0$ and no Push operation is admissible to any neighbor vertex due to the height mismatch. The height of u is increased to the minimum neighbor height plus one (Algorithm 6).

Algorithm 5 Push on a node u.

$d_f(u, v) \leftarrow \min(e(u), c_f(u, v))$
$f(u, v) \leftarrow f(u, v) + d_f(u, v)$
$f(v, u) \leftarrow -f(u, v)$
$e(u) \leftarrow e(u) - d_f(u, v)$
$e(v) \leftarrow e(v) + d_f(u, v)$

Algorithm 6 Relabel on a node u.

$h(u) \leftarrow 1 + \min(h(v) \in N_u)$

Now we can describe the Push-Relabel algorithm. At first, a preflow-initialization is done. All vertices and edge data, like height, excess and residual capacities are initialized with zero.

Obviously, $h(s) = |V|$. For each node $u \in N_s$ where $c_f(s, u) > 0$ we define the flow values as:

$$f(u, v) = \begin{cases} c(u, v), & if\ u = s \\ -c(u, v), & if\ v = s \\ 0, & otherwise \end{cases} \tag{4.3}$$

After the preflow-initialization, we need to define a procedure called *Discharge*. The Discharge operation is done at a node $u \in V - \{s, t\}$ and executes a sequence of Push or Relabel operations continuously until a stop criterion is satisfied. The stop criterion is $e(u) = 0$, $\forall u \in V - \{s, t\}$. At each iteration, for a given node u, it checks a neighbor v and tries to do a Push into it. If there is an edge (u, v) such that $c_f(u, v) > 0$ and $h(u) = h(v) + 1$, then a push is done from u to v. If there is no admissible neighbor, a Relabel operation is done on u. While $e(u) > 0$, the Push and the Relabel operations are executed continuously. To determine what nodes belong to the minimum cut it is

necessary to detect which nodes were disconnected from t and returned excess flow to s. These nodes belong to S. Otherwise, these nodes belong to T. The pseudocode of such algorithm is presented in Algorithm 7.

Algorithm 7 Discharge operation on a node u.

```
while e(u) > 0 do
  v ← currentNeighbor(u).
  if v = NULL then
    relabel(u).
    currentNeighbor(u) ← head(N(u)).
  else
    if c_f(u, v) > 0 and h(u) = h(v) + 1 then
      push(u, v).
    else
      currentNeighbor(u) ← next(N(u)).
    end if
  end if
end while
```

We can now describe the generic algorithm pseudocode. It is very simple: while a given node has $e(u) > 0$ for a given node $u \in V - \{s, t\}$, discharge u using the Push and Relabel operations. This is presented in the Algorithm 8. It has $O(V^3)$ time complexity for any flow network $G = (V, E)$ running on sequential machines.

Algorithm 8 Push-Relabel on a residual network G.

```
preflow-initialization(G)
while e(u) > 0 and u ∈ V − {s, t} do
  discharge(u).
end while
```

The Local Relabel and the Push are the operations done in the basic Push-Relabel algorithm, but the procedure as described above has poor practical performance (GOLDBERG; TARJAN, 1988). Much unnecessary processing is done until convergence, because the heights are only updated locally, not considering the global picture of the distances. However, heuristics can be used to discharge the excess nodes to the sink s faster, once the paths to t are saturated. Here we describe two heuristics: the *Global Relabeling* and the *Gap Relabeling*. These heuristics check the entire residual graph and correct the heights globally.

The Global Relabeling operation updates the distance function defined on the residual graph by computing the shortest path distances in the residual graphs from all nodes to the sink. This can be done in linear time by using a Backwards Breadth-First Search

starting at the sink t node, adjusting exactly all heights. Eventually, some nodes that cannot be reached from t are considered disconnected and are removed from the list of processing nodes. Because it is very costly computationally, it is performed periodically, but it improves significantly the running time. The pseudocode of the Global Relabeling procedure is presented in Algorithm 9.

Algorithm 9 Global Relabeling on a residual network G.

Breadth-First-Search(t), where $\forall u$ discovered, $h(u)$ is updated with the correct traverse level.

The Gap Relabeling tries to find disconnected nodes from t in the graph G. It is based on the following statement. Suppose that $i \in \mathbb{N}$ and $0 < i < |V|$. At a certain stage of the algorithm there may be no nodes $n \in V$ with distance $h(n) = i$, but there are nodes u with $i < h(u) < |V|$, situation defined as a gap. These nodes u are converging to s, and the sink t is no more reachable from any of these vertices u. Therefore, the label of such nodes may be increased to $|V| + 1$ directly. If a linked list of nodes height is used then the overhead of detecting a gap is very small. It reduces significantly the number of operations executed on nodes that were detected to be disconnected from t. A high-level code snippet of it is shown in Algorithm 10.

Algorithm 10 Gap Relabeling on a residual network G.

```
if there is a gap height i ∈ nodes-height-list(G) then
  if h(u) = i, where u ∈ V then
    h(u) ← h(s) + 1.
  end if
end if
```

The Gap Relabeling significantly improves the practical performance of the Push-Relabel method, although usually not as much as the Global Relabeling. These heuristics are not independent, considering that the Global Relabeling discovers nodes disconnected from t and makes gaps less likely. However, the Gap Relabeling has small overhead compared to the Global Relabeling. Thus even if no gaps are discovered in a run of an implementation that uses both heuristics, the running time is almost the same as in the implementation that uses only Global Relabeling. In some cases, many gaps are found and the former implementation is faster than the latter.

5 Graph Cuts in GPU architectures

The first challenge to implement Graph Cuts in the GPU is to devise a way to implement the graph neighborhood. Considering that graphs based on images naturally have a grid structure, we can define a specific model to store their neighborhood structure in the GPU. This is important not only to facilitate the algorithm, but also to reduce the total use of global memory and the number of accesses, improving our solution.

It is recommendable to reduce the use of global memory as much as possible. Creating auxiliary data structures to represent neighborhood, like adjacency lists or adjacency matrices can impose excessive use of memory and increase the number of accesses and consequently the processing times.

In this chapter we describe in more detail our GPU Graph Cuts framework using the Push-Relabel algorithm. We also describe how the graph is constructed with a description of the algorithms in parallel. Our implementation is loosely based in the (VINEET; NARAYANAN, 2008; GARRET; SAITO, 2009; HUSSEIN; DAVIS, 2007) works and was developed with the CUDA library.

5.1 GPU Computing

Graphics Processing Units (GPUs) were initially developed as devices dedicated to graphics processing, improving the efficiency and the power of the graphics pipeline.

The first versions of GPUs were not programmable and all graphic instructions were implemented directly at hardware level. Later, with the evolution of hardware technology, a huge amount of memory and processing cores became available at small cost and started to be used even for non-graphical applications. The concept of using GPU for non-graphical programs is called General Purpose GPU Computing.

In the beginning, creating a non graphical GPU application was not an easy task because the solution had to be represented as a series of rendering passes following the graphics pipeline. Besides, any data had to be represented as texture data and obviously the size of texture memory available was a serious restriction factor.

With the advent of the new GPU models after the GeForce series 8 and the architectures like CUDA, the implementation of GPU computing applications became easier to be done.

This architecture consists on a unified arrangement of cores, which simplifies the GPU programming model by not treating the GPU as a pure graphics pipeline but as a typical multicore-processor. Further it improves the GPU model by removing memory restrictions or graphical idiosyncrasies for each processor. Data representation is also improved by providing friendly data structures to the programmer. All memory available in the CUDA device can be accessed by all processors with no restriction on its representation, though the access times may vary depending on the memory type used.

The CUDA environment is based on the SIMD parallel architecture, where program kernels process data grids, dividing multiple blocks in threads. It is important to obtain maximum performance by executing the same operation simultaneously on different data elements, avoiding code flow divergence. Divergent code produces poor performance because the CUDA model cannot deal efficiently with different instruction flows at a given moment. Another important fact is the lack of GPU memory lock. This brings restrictions on how threads can modify shared memory space.

At the hardware level, CUDA is a collection of multiprocessors consisting of a series of stream processors. Each multiprocessor contains a small shared memory, a set of 32-bit registers and finally texture and constant memory caches access which are common to all processors inside it.

Each processor in the multiprocessor executes the same instruction on different data, which makes it a SIMD model. Communication between multiprocessors can be done using the device global memory, which is accessible to all processors within a multiprocessor.

As a software interface, CUDA API is a set of library functions which can be coded as an extension of the C language. A compiler generates executable code for the CUDA device. For the programmer, a CUDA program is a collection of threads running in parallel.

Each thread can use a number of private registers for its computation. A collection of threads is called a block and runs on the same multiprocessor at a given time. The threads of each block have access to a small amount of common shared memory. Synchronization barriers are also available for all threads of a block. A group of blocks can be assigned to a single multiprocessor but their execution is time-shared. The available shared memory and registers are split equally among all blocks that timeshare a multiprocessor. Multiple groups of blocks are also time-shared on the multiprocessor for execution. The collection of all blocks in a single execution is called a grid.

Each thread executes a single instruction set called kernel. For each thread and block is given a unique ID that can be accessed within the thread during its execution. An algorithm may use multiple kernels, which share data through the global memory and synchronize their execution at the end of each kernel. Threads from multiple blocks can only synchronize at the end of the kernel execution by all threads.

5.2 GPU Graph Representation

The Maximum Flow/Minimum Cut approach for Computer Vision can be executed in grid graphs, where normally each node is associated to a single pixel of the original image.

This model is well suited to the GPU architectures, because each thread can operate on exactly one pixel in a SIMD fashion. Images have a grid format, making the mapping to this architecture very simple. Each vertex can have a 4 or 8-Neighborhood System connectivity. Two schemes are usually adopted in the GPU for neighborhood representation: adjacency lists and grid structures. In this work we use both structures.

The data associated to the vertices and edges is represented by arrays, where each array index i stores the data referent to a single node i. This data structure is appropriate to the SIMD model, which may be treated as an array of processors. Memory coalescence and data manipulation efficiency is improved considering that all threads will access and handle contiguous data. The array size of the non-terminal vertices and the edges' data is always $|V|$.

In order to represent the vertices, two arrays are necessary: one to store the excess flow and another to represent the height labels that estimate the distances to the target node. To represent edges, six arrays storing residual capacities are sufficient, two for source and sink capacities, and four to the north, south, west and east directions. Even

nodes in the border will have a total of six edges represented in the arrays. However, the edges that do not exist for the border nodes will have a residual capacity equal to zero. This is done to guarantee memory coalescence.

The same format can be extended to represent non-grid graphs for SIMD architectures. This is done by modifying the original graph to become regular, inserting null edges until all nodes have the same number of neighbors as the node with the highest degree. This is necessary to represent the graph in a more friendly SIMD format (GARRET; SAITO, 2009), being able to perform the same operation across multiple data locations. These null edges inserted in the original system do not modify the graph solution, because their residual capacities will always be zero.

In order to represent adjacency lists in the GPU for non-grid graphs, more structures are necessary. Because the graph can have an arbitrary number of neighbors and its not simple to determine which nodes are adjacent like in the grid format, it is necessary to represent the neighborhood information with another structure. An array is used to represent the adjacency list in such a way that the array size is equal to the number of neighbors of the node with highest degree times the total nodes $|V|$. If a node $v \in V$ does not have a number of adjacent nodes equal to the node neighborhood with highest degree, then null edges are inserted.

5.3 GPU Push-Relabel

The implementation described in this section is based in the ideas presented in (VINEET; NARAYANAN, 2008) and (GARRET; SAITO, 2009) with a few modifications. The size of the CUDA grid is equal to the image dimensions, where each single thread is mapped to a pixel or a graph vertex.

Each node has the following data: excess flow and height, the same attributes defined in the original sequential implementation. These are stored as appropriate-sized arrays in the GPU global memory, becoming accessible to all threads. Auxiliary memories are also used, principally shared memory. It is a simple tiny memory available at each multiprocessor. Each multiprocessor handles a thread block with 512 threads. This value can vary with the specific hardware.

Two kernels are implemented: *Parallel Push* and *Parallel Local Relabel*. These two kernels do the same as the Push and Relabel operations of the sequential Push-Relabel algorithm. Respectively, one updates the excess and pushes flow to its neighbors and

the other applies a local relabeling operation to adjust height labels. For parallel correctness, the Parallel Push stage is divided into two phases, defined as *Push* and *Pull* (HUSSEIN; DAVIS, 2007). This is necessary because of the potential Read-After-Write hazard. The Parallel Push stage is implemented in a kernel in which each node sends flow to its neighbors but only modifies the edges residual capacities and not the excess of the neighbor nodes. The updated flow is stored temporarily in an auxiliary array. The Parallel Pull stage is executed in another kernel, usually implemented together with the relabel operation.

Algorithm 11 Push operation kernel on a node u.

Load $h(u)$ and $e(u)$ from the global memory to the shared memory of the block.
Synchronize threads to ensure completion load.
Push flow from $e(u)$ to the neighbors $v \in N_u$ that satisfies height properties and without violating $c_f(u, v)$ capacities.
Update the residual capacities of edges (u, v).
Update the excess flow $e(u)$.
Store the flow pushed to each edge in a temporary global array F.

Algorithm 12 PullRelabel operation kernel on a node u.

Load $h(u)$ and $e(u)$ from the global memory to the shared memory of the block.
Synchronize threads to ensure completion load.
Update excess flow $e(u)$ of each vertex and the residual capacities $c_f(u, v)$ with the flow from global array F.
Synchronize threads to ensure completion load.
Compute the minimum height of N_u.
Write the new height to global memory $h(u)$.

The Parallel Pull function updates the excess of the Parallel Push phase from the temporary array data. The Push and Pull stages implemented in two different kernels are unnecessary when Atomic Operations are employed, because this prevents the occurrence of the RAW inconsistency at the hardware level.

The first basic necessary optimization adds null edges in the image boundary nodes, similar to the idea in (GARRET; SAITO, 2009). This is necessary because boundary nodes have fewer neighbors than central nodes. Moreover, the CUDA Model is SIMD, restricting single instructions execution in a massive data grid. It is important to execute the same operations in each thread in a block, avoiding divergence. Divergence results in serialization of the instructions and a reduction in performance. It is appropriate for the Push and Relabel kernels to check six edges in each node, even if the boundary nodes do not contain six neighbors. Obviously, the null edges will make no difference in the final result.

A Parallel Global Relabel kernel based on the original work from (GOLDBERG; TARJAN, 1988; VINEET; NARAYANAN, 2008; HUSSEIN; DAVIS, 2007; ACCELERATING..., 2007) is also necessary to accelerate the algorithm. It is basically a Breadth-First Search starting from the sink t node and the only nodes that are visited are those with unsaturated neighbor edges adjacent to t and their neighbors. The first iteration checks which nodes have unsaturated sink edges. Such nodes are added to a list of nodes to be visited.

For each iteration, all the nodes in this list have their heights updated based on the distance from t. Then, they are removed from the list, and their neighbors are added. Each node is also marked as a visited node in another list. Only nodes never visited and added as a new node to be visited have their heights updated. This heuristic helps on quickly Push-Relabel convergence, but every execution of it is very slow. It is important to execute it only a few times during the execution of the algorithm.

Algorithm 13 Global Relabeling operation kernel starting on a node t on a graph G_f.

```
Initialize a frontier array F_a, a temporary frontier array F_ta and a visited nodes array
V_a with zeros.
F_a[t] ← true.
LevelBFS ← 0.
while ∃F_a[x] = true, such that x ∈ V do
  for all u ∈ V in parallel do
    if F_a[u] = true and V_a[u] = false then
      V_a[u] ← true.
      F_a[u] ← false.
    end if
    for all v ∈ N_u do
      if cf(v, u) > 0 then
        LevelBFS ← LevelBFS + 1.
        h(v) =← LevelBFS.
        F_ta[v] = true.
      end if
    end for
  end for
  F_a ← F_ta. {Copy all data from F_ta to F_a}
end while
```

Another heuristic implemented in the GPU Push-Relabel algorithm is a parallel version of the Gap Relabel, based on the original heuristic from (GOLDBERG; TARJAN, 1988). In this heuristic, we change the nodes height in parallel to the maximum limit if they are disconnected from the residual graph. We maintain a binary list with the size of all possible heights that the nodes could assume, storing binary values if there is at least one node with that height. All positions of this list are checked simultaneously by a Gap Relabel

kernel and if there is at least one position with 0 data, but the next position is 1, a gap is found and this index is the gap height. Another kernel is executed, where all nodes with height greater than this gap are changed to the maximum instantly. This heuristic improves the convergence of the method. The main advantage of this compared to the Global Relabel is the small overhead and speed of a single iteration.

Algorithm 14 Gap Relabeling operation on a graph G_f.

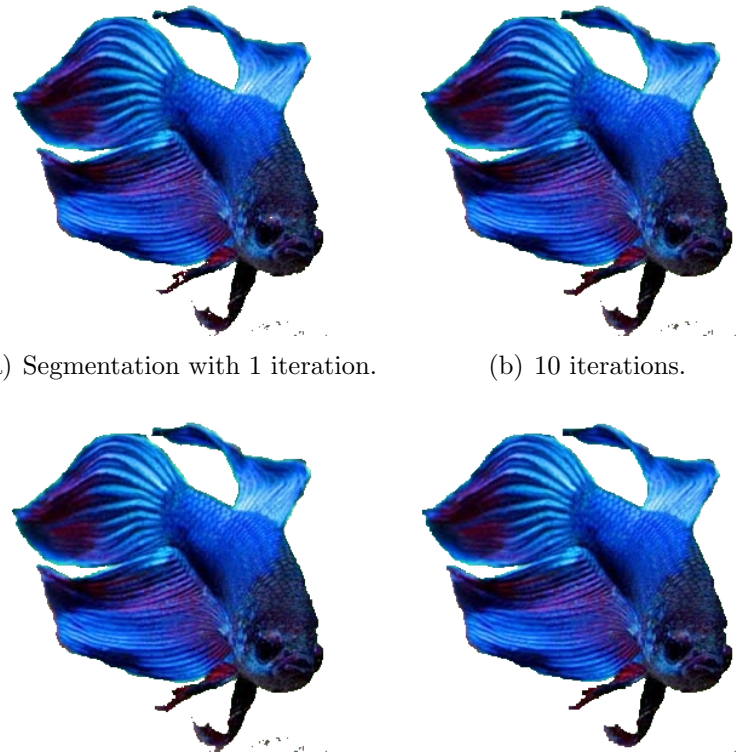
```
Initialize a binary gap array G_a and a gap variable with zeros.
Check all h(u), ∀u ∈ V. If exists at least one node h(u) = n, then G_a[n] = true.
Synchronize all threads.
Find the minimum n such as G_a[n] = true and G_a[n + 1] = false.
gap ← n.
if gap > 0 then
  for all u ∈ G_f in parallel do
    if h(v) > gap then
      h(v) ← h(s) + 1.
    end if
  end for
end if
```

Recently, Vineet and Narayanan (VINEET; NARAYANAN, 2008) have proposed a new heuristic for parallel many-cores architectures called *Stochastic Cuts*. The principle of their idea relies on the rapidly convergence of the most graph nodes at the first iterations. Figure 14 shows the final result after some number of iterations. Only a few nodes have excess flow that needs to be returned to the source. Processing nodes which are unlikely to exchange any flow with their neighbors results in inneficient utilization of the resources.

The Stochastic Cuts can be implemented as follows. At each iteration the total number of active blocks is verified, such that an active block has at least one node that exchanged flow in the previous iteration. It can be detected by checking if the excess flow of all nodes has changed. It is important to avoid the computation of inactive blocks to obtain performance. If an inactive block is detected, its execution is delayed for 10 iterations.

5.4 GPU Multilevel Narrow Banded Graph Cuts

The major contribution of this work is described in this chapter, where a new technique is presented based on the work of Lombaert et al (LOMBAERT et al., 2005). They originally implemented it for sequential machines, but here we extend it to other SIMD parallel architectures as, for instance, GPUs. In our work, the technique is combined with a GPU



(a) Segmentation with 1 iteration. (b) 10 iterations.

(c) 50 iterations. (d) Final result.

Figure 14: An example of segmentation on different moments.

version of the Push-Relabel method, but it can be used with any other GPU Graph Cuts algorithm.

The main motivation for using this heuristic with the GPU Push-Relabel implementation is the fact that this algorithm can be inefficient for certain instances of the problem, i.e., for 2D high resolution image segmentation or 3D huge volume data. Sometimes, the Global Relabel and Gap Relabel heuristics are not sufficient to produce efficient results. However, speedup is not the only desired improvement. The memory usage increases significantly for huge instances of Graph Cuts problems. Hence, it is interesting to reduce its usage when possible.

As mentioned before, the basic idea of this heuristic is to compute the segmentation at a lower resolution reduced version of the original image. After the computation of the segmentation for the coarse version of the image, we project the obtained solution onto a higher resolution image space. The segmentation is repeated again in this projected image, but only on a narrow band that separates the object and background partitions. The process is repeated until all reduced images are projected and the original image is segmented.

Our proposal to implement it in GPU is described for image segmentation, but we believe that the approach can be easily adapted to consider other Graph Cuts problems. The coarsening step is implemented in a way very similar to the original CPU implementation. All images are constructed and stored in CPU. Each image represents a different level of resolution. The object and background image seeds are also simultaneously reduced. This reduction is done by a downsampling operation executed directly in the image colors.

The original image is represented at Level 0. The downsampling operation yields an upper level image whose dimensions are equal to the dimensions of the lower level image divided by a factor of two. Formally, consider an image I_k represented in the Level k with width w_k and height h_k. If an image is created for the level k, then I_k has width $w_k = w_{k-1}/2$ and height $h_k = h_{k-1}/2$.

Let I_k be an image belonging to the level k. An image I_{k+1} in level $k+1$ is obtained by applying a downsampling operation on I_k. The downsampling operation preserves pixels with even integer coordinates and removes those with odd coordinates. Hence, each pixel $p_k \in I_k$ with coordinates (xp_k, yp_k) is mapped to a pixel $p_{k+1} \in I_{k+1}$ with coordinates $(xp_{k+1} = xp_k/2, yp_{k+1} = yp_k/2)$ if the coordinates of p_k are even. If xp_k and yp_k are odd then its not considered (Figure 15).

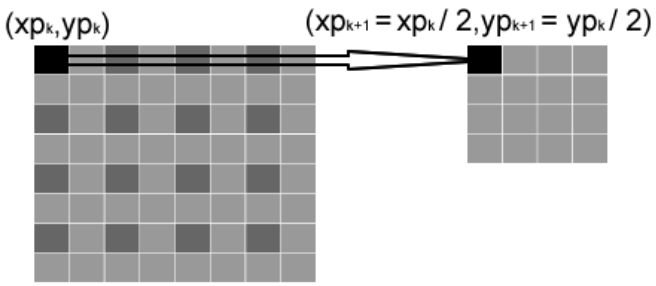


Figure 15: Downsampling from an image I_k to an image I_{k+1} with the highlighted nodes p_k and p_{k+1}.

The coarsening step is finished when all multilevel images are created and stored in the CPU memory. The GPU computation starts after this step. The next operation is the segmentation of the coarsest level image at the highest level. A cut of the graph representing this image is computed. This graph uses the grid format representation and is solved as in the original GPU Push-Relabel method.

The uncoarsening step starts when the segmentation of the coarsest level image is done. After the segmentation of the highest level image I_k, the boundary image result

of it is projected back to I_{k-1}. Differently from the coarsening stage where the projected data are pixel colors, in the uncoarsening stage they are binary labels. As defined in 3.5, the narrow band region of I_{k-1} will contain these projected nodes from the minimum cut of I_k and candidate nodes that are neighbor to the minimum cut region, defined by a dilation parameter d. These nodes will form the graph G_{k-1} that needs to be computed by Graph Cuts. We can conclude that G_{k-1} will have an arbitrary form, depending on the boundary in I_k that separates part of the object and background segments.

The arbitrary form of the graphs based in narrow band regions impose some limitations on GPU Graph Cuts implementation. Graphs based in images are usually represented as grid graphs in GPUs, such that every thread processes each graph node. This is very appropriate because it benefits from the SIMD structure of GPUs while easily models the graph. But the graph based on the narrow banded layer does not have a natural structure to be represented by a grid format.

It is possible to represent a graph of a narrow band using the grid neighborhood representation by adding null edges and null nodes until the graph is completed so that it can be easily represented in a SIMD fashion. But this is not the better choice because much empty data is necessary, considering that a narrow band is a very thin structure compared to the original graph. Much overhead would be introduced slowing down the whole process. The fundamental characteristic of the narrow banded technique is to reduce the original processed data and yield results equivalent to the original segmentation. Adding null information just to preserve this format removes the advantages of it.

Two others graph representations can be considered to the narrow banded graph-cuts solution. Adjacency matrices are not appropriate because the total number of edges is small compared to the number of nodes, requiring much memory to represent it. However, adjacency lists are useful because it can store easily a non-fixed number of edges for each node. In GPUs, adjacency lists are easy to handle and can be represented in a compact manner. All nodes that belong to the narrow banded graph are identified by an index, such that they are linearly arranged. This is clear if we consider the following example: if a graph G using adjacency lists have a 4-Neighborhood System, then the highest degree is four and four arrays are necessary to store each possible neighbor of each node. Each array will store the index of a neighbor of a node $n \in G$, such that the accesses for a neighbor of n are done acessing the index n of each adjacency array. In this instance, the total neighbors of n can vary between four and one. If n does not have four neighbors, then some of these arrays are filled with flags.

However, the introduction of an adjacency list for each node that belongs to the narrow banded graph implies in some problems. Differently from the grid representation, in which an access to a neighbor node is done directly without explicit indexes, it is necessary to access an adjacency list to search for a node. The addition of another data structure into the GPU memory increases the complexity of the process, requiring more global memory accesses, since each node needs to access the neighbor index in an adjacency array. It also harms the memory coalescence, because each neighbor can be stored in arbitrary memory positions.

In our implementation, we represented the narrow banded graphs by adjacencing lists because it is the approach that introduces less overhead, maintaning the simplicity of the solution. The slowdown caused by using the adjacency list is not critical because the total number of nodes that belong to the narrow band is generally small. For instance, an image with one million pixels will probably have a narrow band of only thousands of nodes. The reduction of the total number of nodes is impressive. Consequently, the negative effects caused by introducing the adjacency lists are small.

A snippet code description sufficient to implement the Narrow Banded Graph Cuts structure is presented as follows (Algorithm 15). Adjacency lists are used, but similar data structures of grid graphs are still employed: arrays to store excess flow and heights of nodes and residual capacities of edges. Differently from the grid scheme in which each node is acessed directly, new arrays are defined just to store neighbors that belong to the narrow banded of a given level. The construction of adjacency lists is done in this step. A search is performed for each node of the graph that belongs to the narrow band and if there is at least one adjacent node in it that also belongs to the narrow band, then these nodes will be neighbors and their adjacency lists will store the index of each other.

Algorithm 15 Narrow Banded Graph Cuts.

Load the original image I_0.
Create low resolution images $\{I_1, I_2, ..., I_k\}$ where k is the highest level degree.
Construct the grid graph G_k based on the image I_k.
Calculate the Push-Relabel(G_k) obtaining the minimum cut C_k.
{i receives the current narrow banded level to be processed}.
$i \leftarrow k - 1$;
while $i \geq 0$ **do**
  Project the binary image from I_{i+1} to I_i.
  Create a Narrow Banded Graph based on the projected nodes from $u \in C_{i+1}$.
  Calculate the Push-Relabel(G_i) obtaining the minimum cut C_i.
  $i \leftarrow i - 1$;
end while
Create the final result based on the characteristic function binary image from I_0.

6 Experimental Results

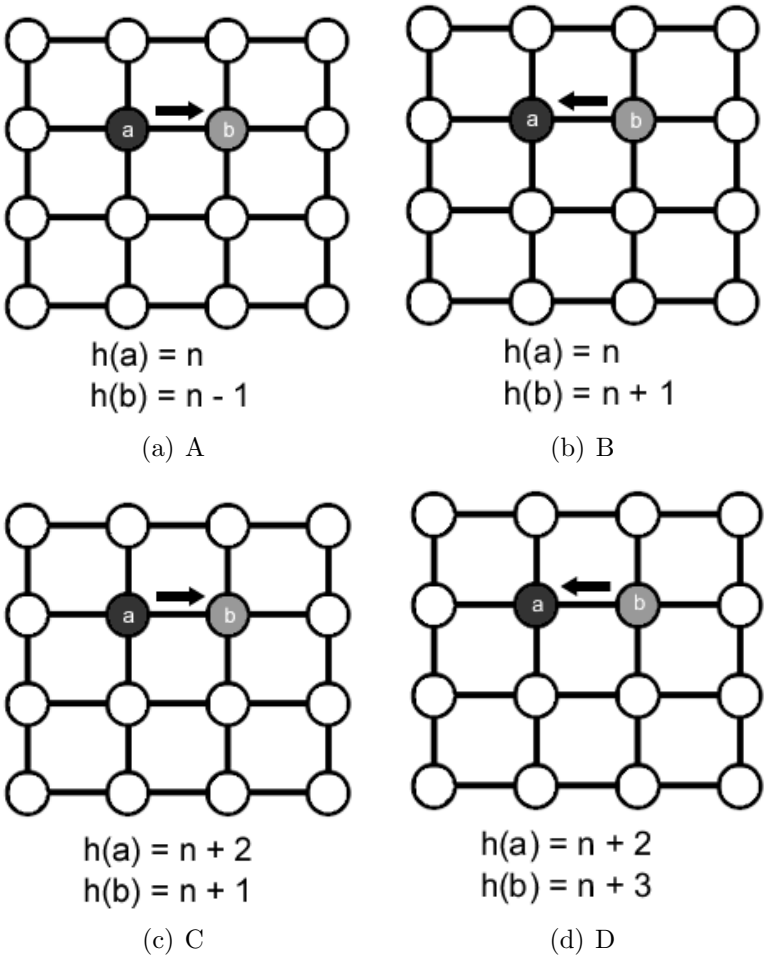
Many works have proposed different methods and techniques to improve and extend the standard Graph Cuts framework, but most of them just considered sequential machines to implement it. Only recently the GPU architecture is being exploited in Graph Cuts research. This work acts directly into this research area, studying many of these proposals. Based on the existing works, we developed our own implementation. We also propose an original contribution: a GPU version of a technique that was presented only for single core architectures. All results obtained by our Graph Cuts implementation with different parameters are described in this chapter, with a few comparisons with other works.

Our graph cut implementation is loosely based on the works in (VINEET; NARAYANAN, 2008; GARRET; SAITO, 2009; HUSSEIN; DAVIS, 2007). One of the minimization energies used in this work is the same as (LI et al., 2004) and its calculation was implemented on the GPU. The other is based on Lattari et al (LATTARI et al., 2010). Energy construction usually involves only arithmetic calculations, being very appropriated to GPU architectures. This is also important because it minimizes the bandwidth bottleneck of memory data transfering between GPU and CPU. The images used in the tests belong to the Berkeley Segmentation Dataset (MARTIN et al., 2001). The OpenCV library (BRADSKI, 2000) was used to handle images and videos.

The implementation was evaluated under several images and videos with different resolutions. Many parameters have influence in the final running time. One important parameter is the block thread size of each kernel. After some tests, we found out that the 32 x 16 block size gives the best results. The GPU used in our tests was a Nvidia Tesla C1060 with 240 cores. This GPU has the maximum number of threads per block limited to 512 threads. The CPU tests were done on a Core 2 Duo with 2.53 Ghz.

After our tests, we concluded that the Push-Relabel algorithm can be very slow due to the convergence of nodes that have positive excess flow and arcs disconnected from t.

These nodes need to be processed until their heights are increased to the maximum and the excess flow is returned to the source. Much processing time can be necessary until this occurs. Consider Figure 16 that specifies a situation that may slowdown the whole process. To simplify, it is a grid graph based on an 4×4 image. Consider two nodes a and b such that $e(a) > 0$, $e(b) = 0$, $h(a) = n$, $h(b) = n - 1$. In this example, $b \in N_a$ is the only unsaturated neighbor that admits a Push operation. A Push operation is done from a to b (Figure 16(a)). Similarly, the only admissible edge from a is (a, b) with $c_f(a, b) > 0$ and it can only send flow back to a (Figure 16(b)). Both nodes a and b change flow continuosly until their heights $h(a)$ and $h(b)$ are greater than $h(s)$. At this moment, they return the flow back to the source.



(a) A (b) B

(c) C (d) D

Figure 16: Example of 4 iterations of the Push-Relabel method on a graph 4×4.

The example of Figure 16 is the main reason of the slow convergence of the standard Push-Relabel algorithm. The problem occurs because only local relabel operations are used, not considering the global picture of the nodes heights. Even the GPUs are insufficient to execute rapidly in this case because a huge number of unnecessary iterations are executed. Heuristics to improve the computation of these nodes are necessary. The main

used heuristics to improve it are the Global Relabel and the Gap Relabel.

The Global and Gap heuristic improve the algorithm efficiency significantly, reducing running times from minutes to seconds or even milliseconds. Periodically, these heuristics can be executed to speedup the method. The Global Relabel adjusts globally the labels of every excess node in the residual graph, reducing many unsaturated push operations, but it is a costly computational operation because requires a breadth-first search operation. The Gap Relabel detects nodes that are disconnected from t and automatically changes their height to the maximum, avoiding unecessary node processing. A single iteration of it is much faster compared to the Global Relabel, but many executions are necessary until all gaps are found, principally in huge graphs.

As in Cherkassky and Goldberg (CHERKASSKY; GOLDBERG, 1994), the processing of the Push-Relabel algorithm with only the Global Relabel is much faster than with only the Gap Relabel. Even computing simultaneously the method with those heuristics has similar times with only the Global Relabel. Taking this into consideration, this implementation uses only the Global Relabel.

Another important heuristic used to improve this method's is convergence is based on the Stochastic Cuts (VINEET; NARAYANAN, 2008). It checks if a node exchanges flow with its neighbors based in the activity of the nodes with flow excesses. When an excess node is disconnected from t, it will attempt to exchange flow with their neighbors in a cycle increasing their heights until the maximum, as in Figure 16. It is important to observe that these disconnected nodes will push the same flow values to the same neighbors repeatedly. If a thread block has only disconnected excess nodes like this, it cannot be considered active, because it is not really modifying the residual graph. Consider an example of a node n in this situation. If it pushes flow to a neighbor at the iteration i, naturally it will receive again the same excess flow at the iteration $i + k$ because it is a cycle. At the iteration i the excesses of all nodes will be identical to the iteration $i + k$, characterizing that this block is not really exchanging flow between nodes and is inactive.

It is really fast to detect a block with only disconnected nodes. It is necessary to periodically store the excess of all nodes in a temporary array and compare it in a future iteration. If all nodes in a thread block have the same excess flow of a few iterations ago, then it is not exchanging flow and is considered inactive. We experimented to interrupt the algorithm when all blocks are considered inactive, obtaining very fast results with exact visual results. A similar strategy was implemented by (VINEET; NARAYANAN, 2008) and available for download into their webpage and was used in this work. This heuristic is

efficient because it detects very fast when active nodes are exchanging flow only among them in a cycle. This heuristic is very important to obtain performance.

By implementing the heuristics mentioned before, principally the Stochastic Cuts, our performance results were similar to the (VINEET; NARAYANAN, 2008). Based on their implementation, we develop an GPU extension of the Graph Cuts framework by implementing the Multilevel Narrow Banded Graph Cuts method from (LOMBAERT et al., 2005). Our results are very promising, improving the speed of the GPU Graph Cuts with pratically identical visual results, surpassing the CPU method for Graph Cuts presented in (BOYKOV; VEKSLER; ZABIH, 1999) using the same heuristic in CPU. We think that our heuristic is very important not only to speedup the method, but also to reduce the memory usage, fundamental for Graph Cuts problems that needs large memory resources, like 3D volume segmentation. A full description of our results is featured in this Chapter.

As presented in Boykov and Kolmogorov (BOYKOV; KOLMOGOROV, 2004), their approach was tested with different Computer Vision problems, being more efficient at the majority of tests when compared with the sequential Push-Relabel algorithm. They lost or had comparable results only in 3D volume segmentation and 2D graphs with highly connected data. Before the GPU Graph Cuts, the Push-Relabel approach was very ineficient to solve image segmentation or other Computer Vision problems. Two reused search trees are extremely appropriate to be used in 2D graphs, saturating the admissible paths much faster than the Push-Relabel method (see Boykov and Kolmogorov method in chapter 4.1.1). However, considering the exponential growth of the GPU memory and processor technology, the latter is becoming more competitive against it. Probably with 3D Graph Cuts problems the speedup will be more significative.

Another factor that counts against the CPU is the calculation of the energy function, necessary in the Graph Cuts framework. In most cases it is composed of many arithmetic operations, being more appropriate to be implemented in the GPU. It is possible to process energy functions in the GPU and transfer data do the CPU algorithm, but the bandwidth can be very restritive, limiting the solution principally for real-time solutions like video. It is more interesting to compute the energy function in the GPU and access directly the array data in the GPU memory using a GPU Graph Cuts algorithm.

Three different graphs compare the performance of our GPU Graph Cuts implementation with the state-of-art CPU Boykov-Kolmogorov. A comparison of the time processing of the algorithms without Multilevel Narrow Banded techniques is featured in Figure 17 and Table 2. The time processing of the methods implemented with a single level of

Multilevel Banded Graph Cuts is featured in Figure 18 and Table 3. Finally, an analysis of the times spent in the Graph Cuts with two levels of Multilevel Narrow Banded Graph Cuts is depicted in Figure 19 and Table 4. The total errors in our experiments is featured in Table 5.

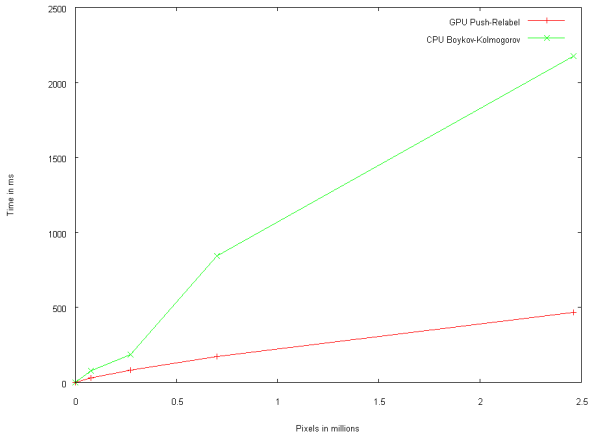


Figure 17: A graph comparison between the GPU Push-Relabel and CPU Boykov-Kolmogorov algorithms without Multilevel Narrow Banded method.

Table 2: Time comparison between Boykov-Kolmogorov algorithm and Push-Relabel without Narrow Banded technique.

Time comparison without narrow banded from Figure 17				
Algorithm	Image Dimensions	Contruction in ms	Cut in ms	Total time
BK	320 x 240	62	15	77
	640 x 427	154	32	186
	1024 x 683	501	343	844
	1920 x 1281	1067	453	2176
PR	320 x 240	5	26	31
	640 x 427	25	56	81
	1024 x 683	28	143	171
	1920 x 1281	36	431	467

After the tests of the GPU Multilevel Narrow Banded proposal, we concluded that our technique is very promissing. The preliminar implementation of this solution improves the GPU algorithm speedup in a two times factor in most cases, comparing the results of Tables 2 with 4. The total errors introduced by using this heuristic is very small, being imperceptible to the user, because the total nodes of the graph based in the narrow band region is tiny compared to the original area (Table 5).

The choice of the graph level depends on the image characteristics, like color distribution or the resolution. It is not a good decision to use many graph levels for very small images. The speedup is insignificant and the chance of lost details during segmentation

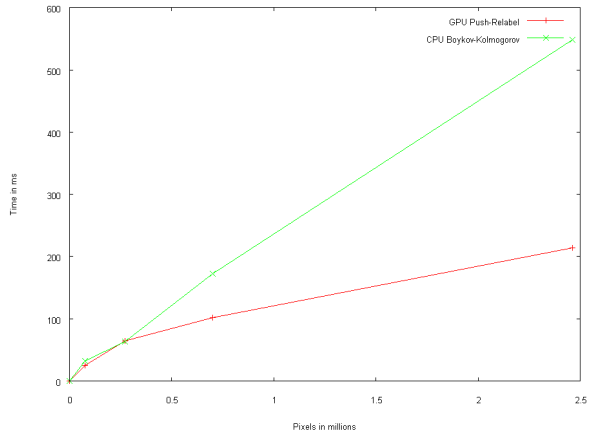


Figure 18: A graph comparison between the GPU Push-Relabel and CPU Boykov-Kolmogorov algorithms using the Multilevel Narrow Banded Graph Cuts with one level.

Table 3: Time comparison between Boykov-Kolmogorov algorithm and Push-Relabel at Level 1 of Narrow Banded technique.

Time comparison with narrow banded level 1 from Figure 18				
Algorithm	Image Dimensions	Contruction in ms	Cut in ms	Total time
BK	320 x 240	16	16	32
	640 x 427	48	15	63
	1024 x 683	109	62	172
	1920 x 1281	439	109	548
PR	320 x 240	4	21	25
	640 x 427	31	33	64
	1024 x 683	41	61	102
	1920 x 1281	82	132	214

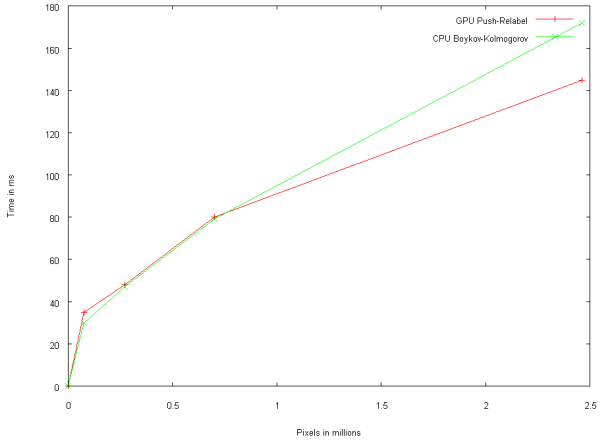


Figure 19: A graph comparison between the GPU Push-Relabel and CPU Boykov-Kolmogorov algorithms using the Multilevel Narrow Banded Graph Cuts with two levels.

Table 4: Time comparison between Boykov-Kolmogorov algorithm and Push-Relabel at Level 2 of Narrow Banded technique.

Time comparison with narrow banded level 2 from Figure 19				
Algorithm	Image Dimensions	Contruction in ms	Cut in ms	Total time
BK	320 x 240	30	0	30
	640 x 427	31	16	47
	1024 x 683	48	31	79
	1920 x 1281	140	32	172
PR	320 x 240	7	35	42
	640 x 427	26	34	60
	1024 x 683	44	54	98
	1920 x 1281	99	70	169

Table 5: Total errors of Narrow Banded Strategy compared to the original Push-Relabel.

Total Pixel Errors by Level		
Image Dimensions	Level	Pixels errors %
320 x 240	L1	0.381%
	L2	0.369%
	L3	0.407%
640 x 427	L1	0.155%
	L2	0.128%
	L3	0.147%
1024 x 683	L1	0.225%
	L2	0.215%
	L3	0.224%
1920 x 1281	L1	0.178%
	L2	0.175%
	L3	0.235%

is high, impacting on the number of pixels errors. Images with 320 x 240 had approximately 4 times more errors than 640 x 427, increasing when more multilevel graphs are employed. Also, errors can appear on images with very high dimensions and much small details. That is true in the bird of Figure 20.

(a) Original Image.

(b) Without Narrow Banded Graph Cuts.

(c) With three levels of Narrow Banded Graph Cuts.

(d) Original Image.

(e) Without Narrow Banded Graph Cuts.

(f) With three levels of Narrow Banded Graph Cuts.

Figure 20: Examples of image segmentation with and without Narrow Banded Graph Cuts.

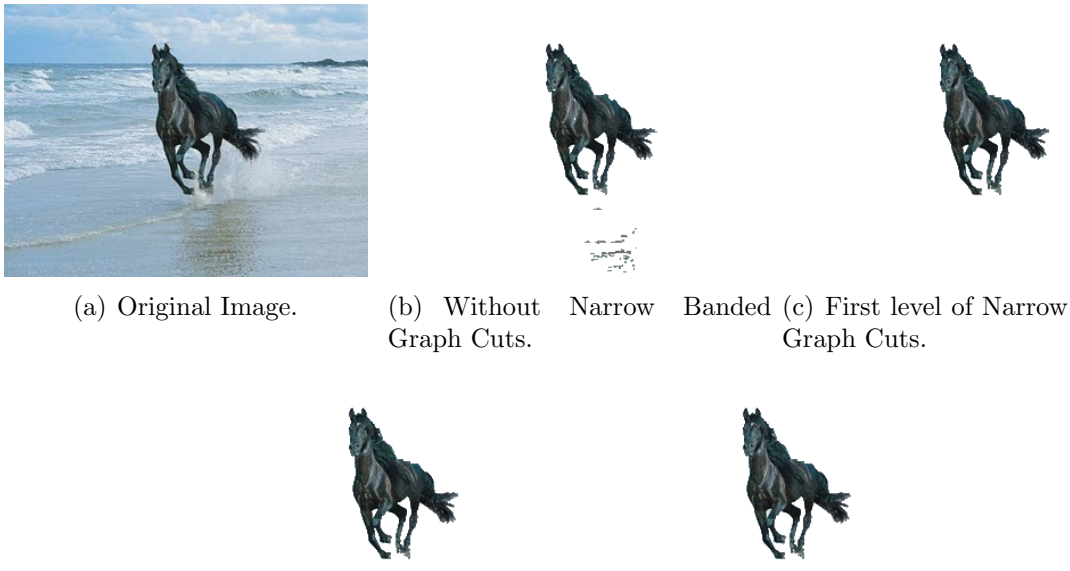
This work is also compared with other GPU approaches. This implementation is more efficient than (HUSSEIN; DAVIS, 2007). Even with their implementation using lockstep breadth-first search and cache emulation, they could do a segmentation of an image with 2 million pixels in 2 seconds in a GeForce 8800 GTX (128 stream processors). In our work, a segmented image with 2.5 million pixels could be computed in less than 500 ms. It is important to note that their method implements more complex structures like prefix sums that is more appropriated to general and not grid graphs. Also, they execute constantly the global relabeling, implying in a overall slow of their method.

The work from Vineet et al (VINEET; NARAYANAN, 2008) was the most important reference to this work, obtaining similar time results. They could do 90 Graph Cuts per second on an 640 x 480 image on a GeForce GTX 280, a GPU similar to ours. It is a very impressive time. This work was implemented based on theirs and we tried to compare it with ours. However, tests on their implementation from their source code did produced many incorrect visual results unless the amount of regularization on the image was low. Probably some inconsistency between the energy used by us and their graph construction produced the wrong result.

The work from Garret and Saito is based on a lock-free implementation of the Push-Relabel algorithm made by Hong (HONG, 2008) but implemented in the GPU. It is very efficient against Boykov and Kolmogorov approach for video obtaining 200 miliseconds of a cut in an image with 1920 x 1080 in a GeForce GTX 280 (240 cores). However, some points of their work were not clear.

As observed in their paper, it is not mentioned why the GPU implementation does not have the same visual results of the CPU. Generally, the minimum cut obtained with the GPU technique is the same as the Boykov-Kolmogorov (BOYKOV; KOLMOGOROV, 2004) or any CPU approach. Moreover, in the original paper in which they based their work (HONG, 2008), it is concluded that the detection of the algorithm termination is not very easy to be done without locks. They did not propose how it is done and states that a further study is necessary. Similarly, Garret's work does not explores exactly how the algorithm termination is done. This is important not only to have a good performance but also to obtain the correct results.

We also present many image examples of the technique, depicting the results at each graph level. The segmentations done in Figures 21, 22, 23 and 24 compare directly the images with zero and three different graph levels, showing that the errors are pratically imperceptible. Many examples are featured.



(a) Original Image.

(b) Without Narrow Banded Graph Cuts.

(c) First level of Narrow Banded Graph Cuts.

(d) Second level of Narrow Banded Graph Cuts.

(e) Third level of Narrow Banded Graph Cuts.

Figure 21: Example of image segmentation of a 320 x 200 image.

Experiments with video segmentation were also evaluated. These tests were done

(a) Original Image.

(b) Without Narrow Banded Graph Cuts.

(c) First level of Narrow Banded Graph Cuts.

(d) Second level of Narrow Banded Graph Cuts.

(e) Third level of Narrow Banded Graph Cuts.

Figure 22: Example of image segmentation of a 640 x 427 image.

Table 6: Time analysis of video segmentation of human skin.

Video Dimensions	Mean Time(ms)	Variance(ms)	FPS
320 x 240	24	0.0001	41
640 x 480	67	0.0001	15
1280 x 960	176	0.0008	6

using a Nvidia Tesla C1060 with 240 stream processors. We used a different energy function constructed by ourselves to detect human skin segmentation without explicit seeds given by a user (LATTARI et al., 2010). Those tests were done with video sequences of sizes equal to 320 x 240, 640 x 480 and 1280 x 720. Table 6 shows the results in terms of the number of frames per second for these videos. Good segmentation results were obtained for low resolution video sequences (320 x 240), achieving approximately 41 FPS. However, with the increase of the resolution of the video, the number of frames per second reduced significantly because the energy construction and the graph-cut became more complex to compute. For high resolution videos the computation is slow, obtaining 6 FPS. We also computed the variance of the data in order to show that the dispersion of the data is not significant. An example of the segmented video sequences used in these tests is featured in Figure 25.

(a) Original Image.

(b) Without Narrow Banded Graph Cuts.

(c) First level of Narrow Banded Graph Cuts.

(d) Second level of Narrow Banded Graph Cuts.

(e) Third level of Narrow Banded Graph Cuts.

Figure 23: Example of image segmentation of a 1024 x 683 image.

(a) Original Image.

(b) Without Narrow Banded Graph Cuts.

(c) First level of Narrow Banded Graph Cuts.

(d) Second level of Narrow Banded Graph Cuts.

(e) Third level of Narrow Banded Graph Cuts.

Figure 24: Example of image segmentation of a 1920 x 1281 image.

Figure 25: Example of three video frames doing human skin detection.

7 Conclusion

Graph Cuts is a method widely used to solve different Computer Vision problems. Their advantages are numerous: efficient to compute when compared to other methods, produces great visual results, introduces a workflow that separates the intrinsic details of the problem to be solved and the minimization algorithm, among others. However, some challenges remain in this area, principally the computational efficiency of the Graph Cuts minimization step.

With the development of the General Purpose GPUs and principally by programming libraries like CUDA, new approaches were proposed to improve the Graph Cuts minimization algorithm. This work presents the background of this area, the basic concepts necessary to understand the Graph Cuts solution, different algorithms to implement it in CPU and GPU and a new heuristic proposed to improve efficiency of the GPU Graph Cuts implementation.

The preliminar implementation of the proposed technique called GPU Multilevel Narrow Banded allowed a speedup of two times in the computation of graph-cuts for images with 1920×1281 (Tables 2 and 4). The total errors inserted by it was insignificant, as in Table 5. The implementation of the proposed heuristic is not completely optimized and improvements can still be achieved. The most advantageous cases of its use were not tested as, for instance, 3D volume segmentation. We believe that many instances that are not possible to be processed efficiently in GPU can be possible with our heuristic as typical CT volumes that have 8GB of data or more (LOMBAERT et al., 2005).

On future works we intend to improve the characteristics of our preliminar implementation of the Multilevel Banded Graph Cuts. The adjacency lists implementation in GPU is not well optimized considering the characteristics of GPUs' architectures, leaving considerable room for improvement. Also, we want to extend our implementation to efficiently compute 2D high resolution data and 3D volumes. Speedup on 2D videos is also possible reusing segmentation of previous frames and taking spatio-temporal coherence

into consideration.

References

ACCELERATING large graph algorithms on the GPU using CUDA. In: . Springer, 2007. v. 4873, n. LNCS, p. 197–208. Disponível em: <http://www.springerlink.com/index/y4816x2q7475v93n.pdf>.

AHO, A. V.; HOPCROFT, J. E.; ULLMAN, J. D. *Data Structures and Algorithms*. [S.l.]: Addison Wesley, 1983.

BLELLOCH, G. E. *Prefix Sums and Their Applications*. [S.l.], 1990.

BOYKOV, Y.; FUNKA-LEA, G. Graph cuts and efficient n-d image segmentation. *Int. J. Comput. Vision*, Kluwer Academic Publishers, Hingham, MA, USA, v. 70, n. 2, p. 109–131, 2006. ISSN 0920-5691.

BOYKOV, Y.; JOLLY, M. Interactive graph cuts for optimal boundary and region segmentation of objects in n-d images. *Proc. IEEE Int. Conf. on Computer Vision*, p. I: 105–112, 2001.

BOYKOV, Y.; KOLMOGOROV, V. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *IEEE Trans. Pattern Anal. Mach. Intell.*, IEEE Computer Society, Washington, DC, USA, v. 26, n. 9, p. 1124–1137, 2004. ISSN 0162-8828.

BOYKOV, Y. et al. An integral solution to surface evolution pdes via geo-cuts. In: *In ECCV*. [S.l.: s.n.], 2006. p. 409–422.

BOYKOV, Y.; VEKSLER, O.; ZABIH, R. Fast approximate energy minimization via graph cuts. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, v. 23, p. 2001, 1999.

BOYKOV, Y.; VESKLER, O. *Graph cuts in vision and graphics: Theories and applications*. [S.l.]: Springer-Verlag, 2006. 79-96 p.

BRADSKI, G. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

BRAY, M.; KOHLI, P.; TORR, P. H. S. Posecut: Simultaneous segmentation and 3d pose estimation of humans using dynamic graph-cuts. In: *In ECCV*. [S.l.: s.n.], 2006. p. 642–655.

BURGES, C. et al. Learning to rank using gradient descent. In: *ICML '05: Proceedings of the 22nd international conference on Machine learning*. New York, NY, USA: ACM, 2005. p. 89–96. ISBN 1-59593-180-5.

CHANDRAN, B. G.; HOCHBAUM, D. S. A computational study of the pseudoflow and push-relabel algorithms for the maximum flow problem. *Operations Research*, v. 57, n. 2, p. 358–376, 2009.

CHERKASSKY, B. V. A fast algorithm for computing maximum flow in a network. *Collected Papers: Combinatorial Methods for Flow Problems*, v. 3, p. 90–96, 1979.

CHERKASSKY, B. V.; GOLDBERG, A. V. On implementing push-relabel method for the maximum flow problem. *Algorithmica*, v. 19, p. 390–410, 1994.

CORMEN, T. H. et al. *Introduction to Algorithms*. [S.l.]: MIT Press, 2001.

DIXIT, R. K. N.; PARAGIOS, M. *GPU-cuts: Combinatorial optimisation, graphic processing units and adaptive object extraction*. [S.l.], 2005.

EDMONDS, J.; KARP, R. M. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, ACM, New York, NY, USA, v. 19, n. 2, p. 248–264, 1972. ISSN 0004-5411.

FORD, L.; FULKERSON, D. *Flows in Networks*. [S.l.]: Princeton University Press, 1962.

GARRET, Z. A.; SAITO, H. Real-time online video object silhouette extraction using graph cuts on the gpu. *Proceedings of the 15th International Conference on Image Analysis and Processing*, p. 985–994, 2009.

GOLDBERG, A. V.; TARJAN, R. E. A new approach to the maximum flow problem. *Journal of the ACM*, v. 35, p. 921–940, 1988.

GREIG, D. M.; PORTEOUS, B. T.; SEHEULT, A. H. Exact maximum a posteriori estimation for binary images. *Journal of the Royal Statistical Society*, 1989. Disponível em: <http://www.jstor.org/stable/2345609>.

HONG, B. A lock-free multi-threaded algorithm for the maximum flow problem. In: *IPDPS*. [S.l.: s.n.], 2008. p. 1–8.

HUSSEIN, A. V. M.; DAVIS, L. On implementing graph cuts on cuda. *First Workshop on General Purpose Processing on Graphics Processing Units*, 2007.

ISHIKAWA, H. Exact optimization for markov random fields with convex priors. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, v. 25, p. 1333–1336, 2003.

ISHIKAWA, H.; GEIGER, D. Segmentation by grouping junctions. *Computer Vision and Pattern Recognition, IEEE Computer Society Conference on*, IEEE Computer Society, Los Alamitos, CA, USA, v. 0, p. 125, 1998. ISSN 1063-6919.

JUAN, O.; BOYKOV, Y. Active graph cuts. In: *In CVPR*. [S.l.: s.n.], 2006. p. 1023–1029.

KASS, M.; WITKIN, A.; TERZOPOULOS, D. Snakes: Active contour models. *International Journal of Computer Vision*, v. 1, n. 4, p. 321–331, 1988.

KIRKPATRICK, C. D. G. J. S.; VECCHI, M. P. Optimization by simulated annealing. *Science*, v. 220, p. 671–680, 1983.

KOHLI, P.; TORR, P. H. S. Measuring uncertainty in graph cut solutions - efficiently computing min-marginal energies using dynamic graph cuts. In: *In ECCV*. [S.l.: s.n.], 2006. p. 30–43.

KOLMOGOROV, V. *Graph Based Algorithms For Scene Reconstruction From Two Or More Views*. Tese (Doutorado) — Cornell University, Ithaca, New York, USA, 2004.

KOLMOGOROV, V.; BOYKOV, Y. What metrics can be approximated by geo-cuts, or global optimization of length/area and flux. *Computer Vision, IEEE International Conference on*, IEEE Computer Society, Los Alamitos, CA, USA, v. 1, p. 564–571, 2005. ISSN 1550-5499.

KOLMOGOROV, V. et al. C.: Bilayer segmentation of binocular stereo video. In: *In: IEEE CVPR (2005)*. [S.l.: s.n.], 2005.

KOLMOGOROV, V.; ZABIH, R. What energy functions can be minimized via graph cuts? *IEEE Transactions on Pattern Analysis and Machine Intelligence*, v. 26, p. 65–81, 2002.

KUMAR, M. P.; TORR, P. H. S.; ZISSERMAN, A. Obj cut. In: *CVPR '05: Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05) - Volume 1*. Washington, DC, USA: IEEE Computer Society, 2005. p. 18–25. ISBN 0-7695-2372-2.

KWATRA, V. et al. Graphcut textures: Image and video synthesis using graph cuts. *Proceedings of ACM SIGGRAPH*, 2003.

LATTARI, L. et al. Colour based human skin segmentation using graph cuts. *17th International Conference on Systems, Signals and Image Processing*, p. 223–226, 2010.

LI, Y. et al. Lazy snapping. *ACM Trans. Graph.*, ACM, New York, NY, USA, v. 23, n. 3, p. 303–308, 2004. ISSN 0730-0301.

LOMBAERT, H. et al. A multilevel banded graph cuts method for fast image segmentation. In: *ICCV '05: Proceedings of the Tenth IEEE International Conference on Computer Vision (ICCV'05) Volume 1*. Washington, DC, USA: IEEE Computer Society, 2005. p. 259–265. ISBN 0-7695-2334-X-01.

MARTIN, D. et al. A database of human segmented natural images and its application to evaluating segmentation algorithms and measuring ecological statistics. In: *Proc. 8th Int'l Conf. Computer Vision*. [S.l.: s.n.], 2001. v. 2, p. 416–423.

NVIDIA. *NVIDIA CUDA Programming Guide*. 2009.

ROTHER, C.; KOLMOGOROV, V.; BLAKE, A. Grabcut": Interactive foreground extraction using iterated graph cuts. *ACM Transactions on Graphics*, v. 23, p. 309–314, 2004.

ROY, S.; COX, I. *A Maximum-Flow Formulation of the N-camera Stereo Correspondence Problem*. 1998.

RUSSELL, S.; NORVIG, P. *Artificial Intelligence: A Modern Approach*. [S.l.]: Prentice Hall, 1995.

SA, A. et al. Actively illuminated objects using graph-cuts. *Computer Graphics and Image Processing, Brazilian Symposium on*, IEEE Computer Society, Los Alamitos, CA, USA, v. 0, p. 45–52, 2006. ISSN 1530-1834.

SETHIAN, J. A. *Level Set Methods and Fast Marching Methods - Evolving Interfaces in Computational Geometry, Fluid Mechanics, Computer Vision, and Materials Science*. [S.l.]: Cambridge University Press, 1999.

SETIAWAN, D. A. N. A.; LEE, C. W. Optical flow in dynamic graph cuts. *IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, p. 288–291, 2007.

SHI, J.; MALIK, J. Normalized cuts and image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, p. 888–906, 2000.

STALLING, D. et al. *Intelligent Scissors For Medical Image Segmentation*. 1996.

VASCONCELOS, C. N. et al. Using quadtrees for energy minimization via graph cuts. In: *VMV*. [S.l.: s.n.], 2007. p. 71–80.

VEKSLER, O. *Efficient Graph-Based Energy Minimization Methods In Computer Vision*. Tese (Doutorado) — Cornell University, Ithaca, New York, USA, 1999.

VINCENT, L.; SOILLE, P. Watersheds in digital spaces: An efficient algorithm based on immersion simulations. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, IEEE Computer Society, Los Alamitos, CA, USA, v. 13, p. 583–598, 1991. ISSN 0162-8828.

VINEET, V.; NARAYANAN, P. Cuda cuts: Fast graph cuts on the gpu. *IEEE Conference on Computer Vision and Pattern Recognition Workshops*, p. 1–8, 2008.

XU, N.; AHUJA, N.; BANSAL, R. Object segmentation using graph cuts based active contours. *Comput. Vis. Image Underst.*, Elsevier Science Inc., New York, NY, USA, v. 107, n. 3, p. 210–224, 2007. ISSN 1077-3142.

YILDIZ, A.; AKGUL, Y. S. A gradient descent approximation for graph cuts. *DAGM Symposium*, p. 312–321, 2009.