# Speeding Up MRF Optimization Using Graph Cuts For Computer Vision

Thesis submitted in partial fulfillment
of the requirements for the degree of

*Master of Science (by Research)*
*in*
*Computer Science and Engineering*

by

Vibhav Vineet
200502028
vibhavvinet@research.iiit.ac.in

Center for Visual Information Technology
International Institute of Information Technology
Hyderabad - 500 032, INDIA
June 2010

International Institute of Information Technology
Hyderabad, India

# CERTIFICATE

It is certified that the work contained in this thesis, titled “Speeding Up MRF Optimization Using Graph Cuts For Computer Vision ” by Vibhav Vineet, has been carried out under my supervision and is not submitted elsewhere for a degree.

06.08.2010

Adviser: Prof. P. J. Narayanan

To that cosmos and the awesome universe
from which we have sprung and to which we belong.

# Acknowledgments

This dissertation would not have been possible without the help, continuous encouragement and support from many people. First, I would like to thank my elder brother, Hitesh, who has been a constant source of inspiration for me. Here I would like to mention the name of Prof. Richard P. Feynman whose interviews and his ways of thinking and looking at the world and problems with doubt and uncertainty helped me realize the importance of study and pleasure of finding things out. I would also like to thank my professor and guide Prof. P. J. Narayanan. During my stay at IIIT, I could learn a lot from him.

During my stay at IIIT, I was fortunate enough to work with many people. I would like to thank my seniors and lab mates at CVIT for their stimulating encouragement. I could learn many things from Prof. C. V. Jawahar, Jagmohan Singh, Jyotirmoy Banerjee, Pawan Harish, Visesh Chari and Himanshu Arora. I thank them for continuous enlightening discussions. I also enjoyed the company of Chand, Pagare, PK, Piyush, Chirag, Adi, Vipul, Himank, Shrikant, Raman and Sashidhar who made my stay at IIIT pleasurable. Finally, I thank my parents who have supported me in all my endeavors.

# Abstract

Many problems in computer vision are mapped to minimization of an energy or a cost function defined over a discrete Markov Random Field (MRF) or Conditional Random Field (CRF). The primary reason for the popularity has been the successes of the graph cuts based methods in solving various labeling problems, especially, image segmentation, stereo correspondence and image denoising. Subsequently, computer vision researchers focused on improving the computational efficiency of graph cuts based methods. In this dissertation, we present two methods to speed up the graph cuts to solve different MRF optimization problems optimally.

In the first half of the dissertation, we present efficient methods to speed up MRF optimization using graph cuts on the GPU. In particular, we present our optimal implementation of the push-relabel methods to solve various labeling problems and show how we have efficiently used different facilities available on the GPU. We also present the incremental $\alpha$-expansion algorithm to solve multilabel MRFs on the GPU and show how shifting the energy function computation and graph construction to the GPU speeds up the overall computation. We compare the efficiency of our approaches on the problems of image segmentation, image restoration, photomontage and disparity computation on the GPU.

The second half of the dissertation deals with the minimization of energy functions defined over large images involving millions of pixels and large number of labels. The computation and memory costs of the current vision algorithms increase at a very fast rate with the increase in the number of the variables in the MRF. This makes them inapplicable to be used for problems involving large images. We present *Pyramid Reparameterization* scheme to improve the computation efficiency without sacrificing global optimality for bilabel segmentation. We also formulate the multilabel problems like stereo correspondence and image denoising on a multiresolution framework using *Pyramid Reparameterization*. This formulation reduces both the computation and memory costs. We model the optimization problem on the pyramidal framework to dynamically update and initialize the solution at the current level with the solution from the previous level. The results of this method are compared with those obtained using conventional methods which solve graph cuts on the largest image.

# Contents

# List of Figures

# List of Tables

*Chapter 1*

# Introduction

## 1.1 Introduction

Many low-level computer vision problems like image segmentation, stereo correspondence, image denoising (Figure 1.1) etc. are usually modeled as label assignment problems. These problems target assigning a label from a given set to each pixel or block of image. They are generally modeled on a Markov Random Field (MRF) or Conditional Random Field (CRF) framework with each pixel or patch representing a variable in the random field which are solved using energy minimization methods.

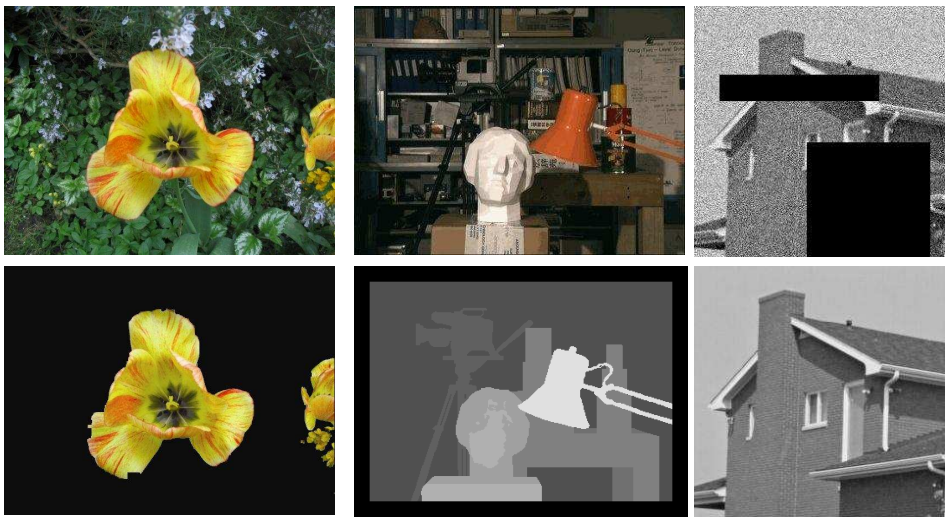


**Figure 1.1** Some of the instances of the labeling problems encountered in computer vision. First row: Input images for image segmentation, stereo correspondence and image denoising. Second row: Their corresponding output images.

Energy minimization methods try to achieve the global minima of an energy or a cost function defined over the image. Over the years, different optimization methods have been used to solve various problems in computer vision. But the popularity of energy minimization methods has risen since the

introduction of graph cuts based methods as a tool to solve them in polynomial time. Nonetheless, minimization of these energy functions is an NP-hard problem [49]. The complexity of the problem also increases as the number of variables and labels increases.

Fortunately, current advancement in the research have seen the emergence of many deterministic approximate algorithms which have improved the accuracy as well as the efficiency of the optimization methods. This saw new applications being built over these methods.

## 1.2 Markov Random Field

Random fields are of great importance to statistically model the problem on the data. They are generally used to model distributions which are multivariate and multidimensional and are not simple to model. One of such models is Markov Random Field which follows markovian properties.

Let us take a random field $X$ defined over $n$ variables $V = \{1, 2, 3, ..., n\}$. Each variable is associated with a neighborhood system $N$ defined as $N = \{N_i | i \in V\}$. Each variable in the random field receives a label from the label set $L = \{l_1, l_2, ..., l_m\}$. A clique $c$ is a set of random variables $X_c$ which are conditionally dependent on each other. Any possible assignment of labels to the random variables is called labeling or configuration. Let such assignment be denoted by $x$ which comes from a cartesian space of different configurations $L \times L \times ... \times L = L^n$.

A random field $X$ is said to be a Markov Random Field on $V$ with respect to a neighborhood system $N$ if it satisfies following two conditions:

$$P(x) > 0;\ \forall x \in X^n \qquad \text{(Positivity)} \tag{1.1}$$

$$P(x_v | \{x_u : u \in V - \{v\}\}) = P(x_v | \{x_u : u \in N_v\});\ \forall v \in V \qquad \text{(Markovian Property)} \tag{1.2}$$

Here we refer $P(X = x)$ by $P(x)$ and $P(X_i = x_i)$ by $P(x_i)$. A commonly used pairwise MRF is shown Figure 1.2.

Evaluating the joint probability is very hard. But the conditional distribution $P(x|D)$ over the labeling of MRF globally conditioned on data is equivalent to Gibbs distribution [18, 22] and can be written as:

$$P(x|D) = \frac{1}{Z} exp(-\sum_{c \in C} \psi(x_c)) \tag{1.3}$$

where $Z$ is a normalizing constant called the partition function and $C$ is set of all the cliques. $\psi(x_c)$ is potential function for $x_c = \{x_i, i \in c\}$. The corresponding Gibbs energy is given by:

$$E(x) = -logP(x|D) - log(Z) = \sum_{c \in C} \psi_c(x_c) \tag{1.4}$$

In other words, the energy function is the sum of the clique potentials of all possible cliques $C$. But an important special case of energy function is when only cliques of size upto two are considered. In this case, the energy function can be written as:

$$E(x) = \sum D_p(x_p) + \sum V_{(p,q)}(x_p, x_q) \tag{1.5}$$

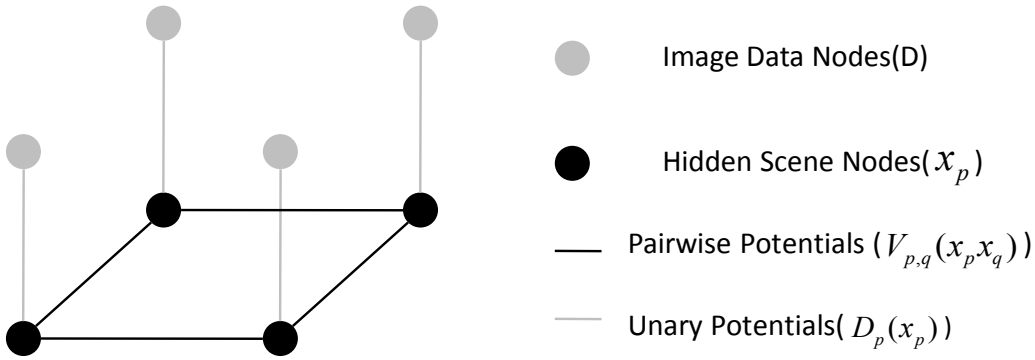



**Figure 1.2** The pairwise MRF commonly used to model labelling problems in computer vision. The random field contains the hidden nodes, the observed data nodes and the connectivity is limited to immediate neighborhood.

where $D_p(x_p)$ , the data term (unary potential), measures the cost of assigning a label to a pixel (or variable) $p$ and $V_{(p,q)}(x_p, x_q)$, the smoothness term (pairwise potential), measures the cost of assigning pair of labels to the adjacent pixels (or variables) $p$ and $q$.

The most probable or maximum a posterior (MAP) labeling $x^*$ of the random field is defined as

$$x^* = \arg\max_{x \in L} P(x|D) \tag{1.6}$$

which is equivalent to finding the global minimum of the energy function E [40].

The label sets can be continuous or discrete. In computer vision and in this dissertation, our focus is on the discrete label sets. Label set can be binary valued (as in case of image segmentation problem) or multivalued (as in case of stereo correspondence problem). We design the basic algorithm only for binary valued functions. Multivalued functions (or multilabeling problems) are projected as many binary valued functions which are solved iteratively. Generally, binary labeling problems are solved using graph cuts [1, 47, 8] and multilabeling problems are solved using $\alpha$-expansion [49] built on graph cuts.

## 1.3 Motivation

Energy minimization methods target to solve two associated issues - accuracy and efficiency. Graph cuts based methods have become very successful in providing accurate results. Graph cuts was first introduced in computer vision way back in 1980's [9] but it became famous after the work of Kolmogorov $et\ al.$ [44] who characterized the class of energy functions which can be accurately minimized via graph cuts. Since then, several efforts have been made to improve the computational efficiency together with the accuracy of the results using graph cuts.

In many real world applications, fast computation at realtime or near realtime rate is required. Examples include robot navigation, surveillance, video processing etc. In robot navigation, the robot

moves forward avoiding obstacles, which requires them to adaptively construct and understand the world around. Similarly, surveillance requires malicious activities to be recognized from the input images of the scene coming at realtime rate. Video processing is another area where realtime processing is required. With more and more videos being uploaded and watched on youtube and other web-servers, the users need the processed video at the client side at realtime rate. An example being extraction of an object of interest from the consecutive frames of a video or to get the denoised images at the client side.

The interactive tools have now become a part of each desktop computer as the users interact with the images to highlight or change some aspects of the images. With the ever increasing advancement in the camera culture, even our offshelf cameras are capturing large images containing millions of pixels. These large images are likely to contain multiple, distributed regions that constitute the foreground layers. Processing on them is challenging as the computation complexity increases proportionately with the increase in the number of pixels.

It is evident and has now become indispensable to process at a faster rate to achieve realtime performance on these applications. Computer vision researchers have over the years developed different methods to improve the computational efficiency but they still lack the speed to be used in these applications where realtime or near realtime performance is required. In this dissertation, we present different methods to improve the computational efficiency of the graph cuts method to solve different MRF optimization problems.

Solving computationally expensive problems on parallel architectures is a way to improve the speed. However, not all algorithms are scalable easily to parallel hardware models. The contemporary graphics processing units (GPUs) have emerged as popular high performance platforms because of their huge processing power at reasonable costs. The Compute Unified Device Architecture (CUDA) [31] from Nvidia provides a general-purpose parallel programming interface to the modern GPUs. The emerging standard of OpenCL [26] also promises to provide portable interfaces to the GPUs and other parallel processing hardware. The grid structure of the MRFs arising in vision problems makes the GPU an ideal platform for energy minimization problems. However, synchronization and memory bandwidth are bottlenecks for implementing them on the GPUs. In the first part of this dissertation, we explore MRF optimization on the GPU. In particular, we propose a method to efficiently use the GPU and their architecture for high performance.

Advancement in camera culture and ever increasing demand from entertainment industries have lead to the acquisition of images involving millions of pixels. Projecting these problems onto a multiresolution framework can solve them faster. In a mulitresolution framework, the problem is projected to and solved in a computationally inexpensive low resolution representation. The results are then transferred to higher resolution representation, providing a faster overall solution. In the second part of this dissertation, we present a novel method to perform graph cuts using a multiresolution MRF framework.

## 1.4 Contributions of the Thesis

The first part of this dissertation deals with improving the computational efficiency of graph cuts and graph cuts based methods on a parallel GPU hardware with applications to computer vision. The grid structure of the MRFs arising in vision problems makes the GPU an ideal platform for energy minimization problems. But synchronization and memory bandwidth are bottlenecks for implementing them on the GPUs. These are some of the issues we have addressed as well. We present a fast implementation of the push-relabel algorithm for mincut/maxflow algorithm for graph-cuts using CUDA, optimally utilizing the GPU architecture. We also propose stochastic cuts which adaptively manage the work load to improve the performance of push-relabel algorithm by factors for different problems on the GPU. We show the performance of our method on some real datasets. We also solve the problem of two label segmentation in consecutive frames of videos using dynamic graph cuts method [35]. This method is used to process problem instances similar in structure and the solution of one problem instance is used to initialize the next problem instance in a better way for faster convergence. Computations to evaluate the edge weights and to construct the graph can also exploit the GPU very well, providing a factor of speedup. We also present incremental $\alpha$-expansion, extending our basic and dynamic graph cuts implementation on the GPU to solve different multilabeling MRF optimization problems.

The second part of the dissertation deals with the processing of high resolution images involving large number of pixels and large number of labels using multiresolution MRF. We present Pyramid Cuts, a multiresolution approach to speed up graph cuts for energy minimization. We combine classical multiresolution or pyramid processing with graph cuts optimization. Our Pyramid Reparameterization exploits the optimal results at a resolution level of an image to initialize its graph at the next higher resolution level. Pyramid reparametrization involves an upsampling step to increase the resolution of graphs and a weight modification step to form a graph with the same mincut as the graph constructed at the higher resolution level. The mincut generated by pyramid cuts using multiple levels is identical to the mincut generated using graph cuts on the highest resolution image. We show the results on image segmentation, stereo correspondence and image denoising problems on higher resolution images with large number of labels.

## 1.5 Outline of the Thesis

- Chapter 2: This chapter reviews and studies a unified framework of energy minimization methods. We explain how a labeling problem in computer vision is solved using discrete optimization methods in a Markov Random Field (MRF) framework. We give general properties of MRF and discuss different maxflow/mincut algorithms. We also discuss graph cuts based approximate algorithms and some recent development towards improving the computational efficiency of these algorithms.

- Chapter 3: In this chapter, we present a fast implementation of the push-relabel algorithm for mincut/maxflow algorithm for graph-cuts using CUDA. We also propose stochastic cuts which adaptively improves the performance of push-relabel algorithm by factors for different problems on the GPU. We extend this work to explore other MRF optimizations on the GPU. We present the incremental $\alpha$-expansion algorithm for multilabel MRFs. We also propose a method to map dynamic energy minimization algorithms [35] to the GPU architecture which provides a magnitude of speedup compared to the CPU version.

- Chapter 4: In this chapter, we model the problem of energy minimization on the multiresolution MRF, where an optimization problem is solved at a coarser level and the solution is used to dynamically update the MRF instance at the next finer level. We prove that the mincut generated by pyramid cuts using multiple levels is identical to the mincut generated using graph cuts on the highest resolution image. We use this to solve many labeling problems in computer vision like image segmentation, stereo correspondence and denoising problems on images involving millions of pixels and large number of labels. We also use this method to focus on providing good interactive user experience to separate a distributed foreground layer in large images using a global optimization method, which provides a quick and approximate segmentation feedback like painting based methods.

- Chapter 5: In the last chapter of the dissertation, we conclude by giving the contribution of the thesis and promising future of the graph cuts based methods in the realm of the energy minimization methods.

*Chapter 2*

# Background and Related Work

## 2.1 Energy Minimization using Graph Cuts

Many problems in computer vision can be modeled as labeling problems. Examples include image segmentation, stereo correspondence, denoising, scene understanding problems etc. The labeling problems involve assigning the most probable label from a set to each pixel or the patch in the image. These problems are naturally formulated as energy minimization problems. The general form of these discontinuity preserving functions is:

$$E(x) = \sum_{p} D_p(x_p) + \sum_{(p,q)} V_{(p,q)}(x_p, x_q) \tag{2.1}$$

where $D_p(x_p)$ is the dataterm and $V_{(p,q)}(x_p, x_q)$ is the smoothness term. Over the years many methods have been proposed to improve the computational efficiency as well as the accuracy of the solutions. But graph cuts based methods are used extensively to find the MAP estimation of different discrete labeling problems in computer vision. These problems are formulated as st-mincut problems which are solved in dual space by solving corresponding max-flow problems.

### 2.1.1 The st-mincut/maxflow problem

Given a graph $G(V, E)$, the st-mincut algorithms target separating it into two disjoint sets of vertices. Due to the mincut-maxflow equivalence [41], the st-mincut is solved in polynomial time by maxflow algorithms [27, 20, 5]. First, we look at the the terminology related to the maxflow algorithms used in this thesis.

**Flow Network** is a directed graph $G(V, E)$ with two special vertices: source $s$ and sink $t$. Each edge $(u, v)$ is associated with a capacity value $c(u, v) \geq 0$ and a flow value $f(u, v)$. There are three constraints associated with the flow network:

$$\forall \{u, v\} \in V, f(u, v) \leq c(u, v) \qquad \text{(Capacity Constraint)} \tag{2.2}$$

$$\forall \{u, v\} \in V, f(u, v) = -f(v, u) \qquad \text{(Skew Symmetry)} \tag{2.3}$$

$$\forall u \in (V - \{s,t\}), \sum_{(u,v)\in V} f(u,v) = 0 \quad \text{(Flow Conservation)} \tag{2.4}$$

Maximum flow problems aim at finding the maximum flow that can be pushed from $s$ to $t$ in the flow network (or graph) without violating these constraints.

**Residual graph** $G_f$ of the graph $G$ has the same topology, but consists of the edges which can admit more flow. The residual capacity $c_f(u,v) = c(u,v) - f(u,v)$ is the amount of additional flow which can be sent from $u$ to $v$ after pushing $f(u,v)$.

Two algorithms are popular to compute the mincut/maxflow on graphs. The first one, due to Ford and Fulkerson [27] and modified by Edmond and Karp [20], repeatedly computes augmenting paths from $s$ to $t$ in the residual graph through which flow is pushed until no augmenting path can be found. The second algorithm, by Goldberg and Tarjan [5] , works by pushing flow from $s$ to $t$ without violating the edge capacity constraints. Rather than examining the entire residual network to find an augmenting path, the push-relabel algorithm works locally, looking at each vertex's neighbors in the residual network. Thus, it is more suitable for parallel implementation. We describe the push-relabel algorithm briefly.

#### 2.1.1.1 Goldberg's Generic Push-Relabel Algorithm

The push-relabel algorithm constructs and maintains a residual graph at all times. The algorithm maintains two quantities: the excess flow $e(u)$ and the height $h(u)$ for all vertices $V' = V \cup \{s,t\}$ with $h(s) = |V|$ and $h(t) = 0$. Unlike ford-fulkerson's method, push-relabel methods do not maintain flow conservation property throughout their execution. But they however maintain preflow condition. The excess flow $e(u) \geq 0$ is the difference between the total incoming and outgoing flows at a node $u$ through its edges. The height $h(u)$ is a conservative estimate of the distance of vertex $u$ from the target $t$. Initially all the vertices have a height of 0 except for the source $s$ which has a height $|V|$, the number of nodes in the graph.

There are two basic operations in a push-relabel algorithm: *Push operation* (algorithm 1) pushes excess flow from a vertex to one of its neighbors and *relabel operation* (algorithm 2) relabels a vertex to its actual distance to the sink. The algorithm is sped up in practice by periodically relabelling the vertices using a global relabelling procedure or a gap relabelling procedure [6].

1: *Applies when:* $u$ is overflowing, $c_f(u,v) > 0$, and $h[u] = h[v] + 1$.
2: Push $d_f(u,v)$ = min$(e[u], c_f(u,v))$ units of flow from $u$ to $v$.
3: $d_f(u,v) \leftarrow$ min$(e[u], c_f(u,v))$
4: $f[u,v] \leftarrow f[u,v] + d_f(u,v)$.
5: $f[v,u] \leftarrow -f[u,v]$.
6: $e[u] \leftarrow e[u] - d_f(u,v)$.
7: $e[v] \leftarrow e[u] + d_f(u,v)$.

**Algorithm 1**: PUSH$(u,v)$ pushes excess flow from a vertex to one of its neighbours.

1: *Applies when:* $u$ is overflowing, and for all $v \in V$ such that $(u, v) \in E_f$, we have $h[u] \leq h[v]$.
2: *Action:* Increase the height of $u$.
3: $h[u] \leftarrow 1 + min(h[v]; (u, v) \in E_f)$

**Algorithm 2**: Relabel($u$) relabels a vertex to its actual distance to the sink.

Graph cuts separate the vertices of the graph into two disjoint sets $S$ and $T$, which is equivalent to performing bilabel segmentation of separating the foreground layer from the background layer in an image. But, to solve the multilabeling problems, graph cuts based $\alpha$-expansion [49] was proposed.

### 2.1.2 $\alpha$-expansion

The $\alpha$-expansion [49] is a popular iterative energy minimization algorithm (Algorithm 3) which solves multilabeling problems using multiple intermediate graph cuts steps. Let steps 2 to 7 form a cycle and the steps 3 to 6 form an iteration within a cycle.

The algorithm starts from an initial random labeling with each pixel in the image getting a random label from the label set. In each iteration, a graph is constructed based on the current labeling and the label $\alpha$. Vertices with label $\alpha$ do not take part in the graph construction but all others attempt to relabel themselves with $\alpha$. After each iteration of $\alpha$-expansion [49], the variable in the MRF retains either its current label or takes a new label $\alpha$. At each such step, there are exponential number of labeling possible, but graph cuts ensure the best possible configuration. Subsequently each step of graph cuts results in a decrease in the energy values. The algorithm terminates when there is no further decrease in the energy values.

1: Initialize the MRF with an arbitrary labeling $x$.
2: **for** each label $\alpha \in$ L **do**
3: Construct the graph on the current labeling and the label $\alpha$.
4: Perform graph cuts to separate vertices of the graph into two disjoint sets.
5: Assign new label to each pixel in the image and find the current configuration $x'$.
6: Calculate the energy value $E(x')$ of the current configuration $x'$.
7: **end for**
8: **if** $E(x') < E(x)$ then **then**
9: goto step 2
10: **end if**
11: return $x$.

**Algorithm 3**: $\alpha$-expansion($alpha$) method to solve multilabeling problems.

$\alpha$-expansion was proposed by Boykov $et\ al.$ [49]. They theoretically prove that the minimum energy achieved by $\alpha$-expansion will be at max twice the global minimum in the worst case. However, practically the final minimum energy values achieved are almost close to the global minimum which they empirically observe on image segmentation, stereo correspondence, motion and image restoration problems [49].

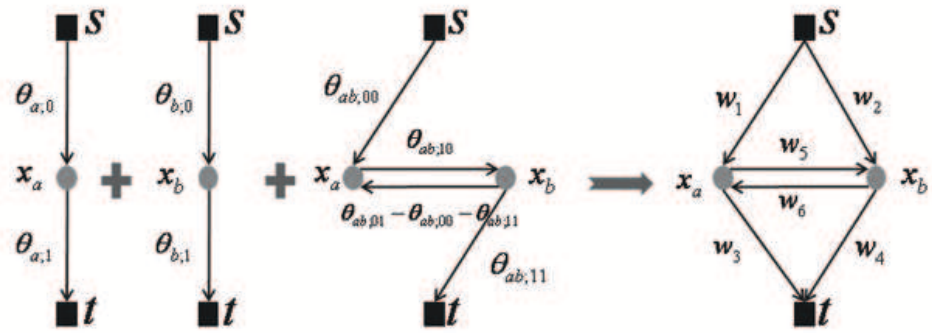



**Figure 2.1** The figure shows how individual unary and binary potentials are represented and combined to form an edgecapacitated graph. Whenever there are multiple edges between a pair of nodes, the edgeweights are merged together.

One of the key intermediate steps involved in the graph cuts based methods is the graph construction. Algorithms for finding the st-mincut requires the edgeweights in the graph to be non-negative. This restricts the class of the energy functions that can be solved using graph cuts based algorithms. Kolmogorov $et\ al.$ [44] characterize the class of energy functions which can be minimized via graph cuts and show a method to construct graph from an energy function without violating any constraints discussed in the section 2.1.1. We now explain the graph construction for energies involving binary random variables. The notations used are same as that used by Kolmogorov [43] to write a second order potential function which is:

$$E(x|\theta) = \theta_{\text{const}} + \sum_{v \in V, i \in L} \theta_{v;i}\delta_i(x_v) + \sum_{(u,v)\in E,(i,j)\in L^2} \theta_{uv;jk}\delta_j(x_u)\delta_k(x_v) \tag{2.5}$$

where $\theta_{v;i}$ is the penalty for assigning label $i$ to the latent variable $x_v$, $\theta_{uv;ij}$ is the penalty for assigning labels $i$ and $j$ to the latent variables $x_u$ and $x_v$ respectively. Further, the function of binary valued functions can be written as:

$$E(x|\theta) = \theta_{\text{const}} + \sum_{v\in V}(\theta_{v;1}x_v + \theta_{v_0}\bar{x}_u) + \sum_{(u,v)\in E}(\theta_{st;11}x_u x_v + \theta_{st;01}\bar{x}_u x_v + + \theta_{st;10}x_u\bar{x}_v + \theta_{st;00}\bar{x}_u\bar{x}_v) \tag{2.6}$$

The individual unary and pairwise terms of the energy functions are represented by weighted edges on the graph. Multiple edges between the same nodes are merged into one single edge. The constant term $\theta_{const}$ does not depend on the configuration $x$ and thus is not considered for graph construction. Our graph construction is explained in Figure 2.1.

Graph cuts based methods have been very successful in providing accurate results at a moderate time. Unfortunately, however these methods were not fast enough to be used for realtime applications which led to the development of new sophisticated methods which provide factors of speedup without sacrificing the accuracy. And, dynamic graph cuts [35] is one of the many effort in this direction.

### 2.1.3 Dynamic GrahCuts

There are many instances in computer vision where two problem instances are very similar in structure and topology. One being segmentation of consecutive frames of a video. In such cases, current problem instance can be initialized in a better way using the solution of previous problem instance to get a better starting point. Kohli and Torr [35] describe a reparameterization of the graph that maintains the flow properties even after updating the weights of a few edges. The resulting graph is close to the final residual graph and its mincut can be computed in a small number of iterations. There are two basic operations involved in their approach: updation and reparameterization.

#### 2.1.3.1 Reparameterization

We now explain the concept of reparameterization which is the central idea in the dynamic graph cuts. The general form of binary valued energy functions as discussed in section 2.1.2 in terms of energy parameter $\theta$ after expansion is:

$$E(x|\theta) = \theta_{\text{const}} + \sum_{v \in V} (\theta_{v;1} x_v + \theta_{v_0} \bar{x}_u) + \sum_{(u,v) \in E} (\theta_{st;11} x_u x_v + \theta_{st;01} \bar{x}_u x_v + + \theta_{st;10} x_u \bar{x}_v + \theta_{st;00} \bar{x}_u \bar{x}_v) \tag{2.7}$$

Two functions $E_1(x|\theta_1)$ and $E_2(x|\theta_2)$ are called reparameterization of each other if $E_1 = E_2$ for all $x$. There are a number of transformations which can be applied to an energy parameter vector $\theta$ to obtain its reparameterization $\hat{\theta}$. For instance the transformations given as:

$$\forall i, \quad \hat{\theta}_{v;i} = \theta_{v;i} + \alpha, \quad \hat{\theta}_{const} = \theta_{const} - \alpha. \tag{2.8}$$

$$\forall i, j \quad \hat{\theta}_{st;ij} = \theta_{st;ij} + \alpha, \quad \hat{\theta}_{const} = \theta_{const} - \alpha. \tag{2.9}$$

are valid energy parameter reparameterizations.

Since reparameterization of the energy functions define the same energy functions, the minimum energy labeling will be same for both the cases. This in turn means the graph constructed over the energy functions $E_1(x|\theta_1)$ and $E_2(x|\theta_2)$ will have the same st-mincut. This implies any reparameterization on an energy function results in a corresponding reparameterization of the graphs. Under these circumstances, the resulting graphs will be the reparameterization of original graph and will have the same st-mincut. The graph transformation corresponding to the energy function transformation given by equations 2.8 and 2.9 are shown in Figure 2.2. Kohli and Torr use the concepts of this reparameterization to recycle the flows from the previous min-cut solution to initialize the current problem instance [36].

#### 2.1.3.2 Recycling of Flows and Updation of Graphs

Consider $G_i$, $G_{i+1}$ as the edge-capacitated graphs and $G_i^r$, $G_{i+1}^r$ as the final residual graphs of two consecutive problem instances, which have the same topology and structure. We apply graph cuts on $G_i$

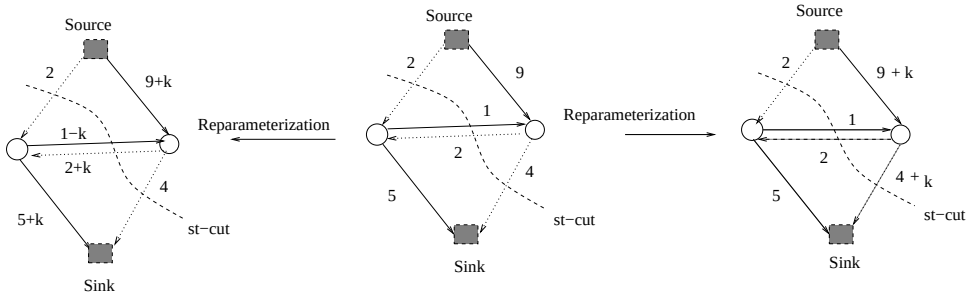



**Figure 2.2** Graph Reparameterization. The figure shows a graph $G$ (middle graph), its two reparameterizations $G_1$ (left graph) and $G_2$ (right graph) along with their respective st-mincuts as shown in [35].

to get $G_i^r$. The dynamic graph cuts update and reuse the flows from previous solution to get an approximate solution closer to $G_{i+1}^r$ using $G_i$, $G_{i+1}$ and $G_i^r$. There are two types of updation: $t$-edgeweights updation and $n$-edgeweights updation, without violating the capacity constraint. But, flow updation may result in negative edgeweights in the graph $G_{i+1}^r$, which is overcome by reparameterization step discussed in previous subsection 2.1.3.1. The max-flow is then computed on this reparameterized graph. The algorithm converges much faster due to the starting point being closer to the final residual graph. In the following sections, we use $c_{su}^{i+1}$, $c_{ut}^{i+1}$ to represent new edge-capacities from the vertex to the source $s$ and the sink $t$. Similarly, $r_{su}^{i+1}$ and $f_{su}^{i+1}$ represent the updated residual capacities and flows of the edge $(s, u)$.

**Modifying $t$-edgeweights:** The updated residual capacities can be computed as $r_{su}^{i+1} = r_{su}^{i} + c_{su}^{i+1} - c_{su}^{i}$, which in simpler form is $r_{su}^{i+1} = c_{su}^{i+1} - f_{su}^{i}$. The updation may result in negative edge-weight if $c_{su}^{i+1} < f_{su}^{i}$. A constant equal to $f_{su}^{i+1} - c_{su}^{i+1}$ is added to both the t-edges to overcome this issue which will not change the cut-edges but may change the cut values.

**Modifying $n$-edgeweights:** Similarly, the newly updated $n$-edge residual capacity in $G_{i+1}^r$ is: $r_{uv}^{i+1} = r_{uv}^{i} + (c_{uv}^{i+1} - c_{uv}^{i})$. The previous updation results into negative edgecapacities when the new edge capacity $c_{uv}^{i+1}$ is less than the flow $f_{uv}^{i}$, which violates the edge capacity constraint. The inconsistency arising from excess flow through edge $(u, v)$ is resolved by a single valid transformation of the residual graph as shown in Figure 2.2. The transformation involves adding a constant ($f_{uv}^{i}$ - $c_{uv}^{i+1}$) to the capacity of edges $(s, u)$, $(u, v)$ and $(v, t)$ and subtracting it from the residual capacity of edge $(v, u)$. The residual capacity $r_{vu}^{i}$ of edge $(v, u)$ is greater than the flow $f_{uv}^{i}$ passing through edge $(u, v)$. As added constant is always less than $f_{uv}^{i}$, the above transformation does not make the residual capacity of edge $(v, u)$ negative. The complete framework of updation and reparameterization is shown in Figure 2.3. This work was extended by Alahari $et\ al$. [25] to solve the multilabeling problems. They provide methods to dynamically update the flow across iterations and cycles of $\alpha$-expansion.

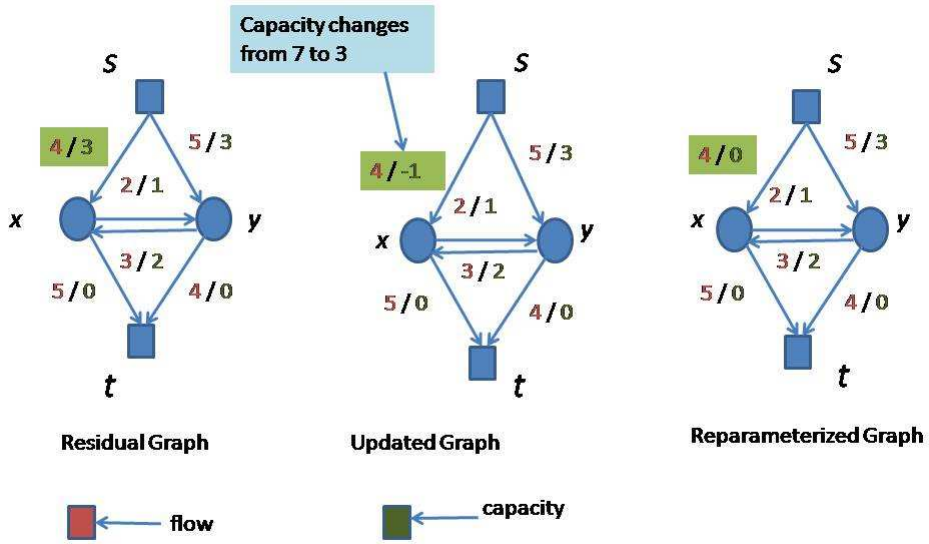



**Figure 2.3** Figure illustrates an example of updation and reparameterization for restoration of consistency as shown in [35]. It starts by showing a final residual graph consisting of two nodes x and y, obtained after a max-flow computation. For the second max-flow computation, the capacity of edge $(s, x)$ is reduced to three units resulting in the updated residual graph in which the residual capacity of edge $(s, x)$ is equal to $-1$. To make the residual capacities positive, the graph is reparameterized by adding 1 to the capacity of edges $(s, x)$, $(x, t)$. This gives the reparameterized residual graph in which the edge flows are consistent with the edge capacities.

### 2.1.4 Dynamic $\alpha$-expansion

This method is inspired by the dynamic computation discussed in previous subsection 2.1.3. The method uses the solutions and flows from a cycle to next to update and reparameterize the graphs corresponding to each label $\alpha$. When solving an expansion move in a cycle, the method reuses the flow from the corresponding move in the previous cycle to make the new computation faster. In the first cycle, it builds $k$ graphs $G_1^1$, $G_2^1$, ..., $G_k^1$ for each label. Mincut/maxflow is applied on each graph corresponding to the label to get the optimal minimum. Maxflow problems corresponding to all the labels are solved just as in standard $\alpha$-expansion method. For later cycles $l > 1$ of the algorithm, instead of creating a new graph $G_i^l$ for a label $i$, it dynamically update the corresponding graph $G_i^l$ from the previous iteration. This step involves updating the flows and the residual edge capacities. After these update operations, the maxflow algorithm is performed on the residual graph. As the number of changes in the graphs decrease in the latter iterations, the number of update and maxflow computations decrease. Hence, the optimal moves in these iterations are computed efficiently.

## 2.2 Other Related Works

Over the years, many new and sophisticated methods have been proposed to solve the energy minimization methods. But graph cuts based methods have been most successful in achieving the accurate results efficiently. It was used long back by Greig $et\ al.$ [9] to separate the foreground objects from the background objects. It was however popularized by Boykov $et\ al.$ who used it to solve many MRF optimization problems. The initial success attracted the attentions of many other researchers who tried to improve the efficiency and accuracy of the results.

Boykov $et\ al.$ [48] start two search trees from both $s$ and $t$. The algorithm reuses these search trees from iteration to iteration, alleviating the redundancy of starting the search trees again each time. Practically they achieve a factor of speed up compared to the conventional graph cuts especially on grid graphs ubiquitous in computer vision. Schimdt $et\ al.$ [14] presented an O($nlog(n)$) method for planar graphs. The concept is motivated by the graph theoretic work [15]. The main step involves a preprocessing step that consists of finding single-source shortest-path distances. The algorithm then repeatedly saturates the leftmost residual $s$-to-$t$ path in the graph till the convergence. Schimdt $et\ al.$ [14] used this method to match two different planar shapes in $O(n^2log(n))$ and segment a given image of $n$ pixels in $O(nlog(n))$. Kohli and Torr [35] proposed methods to reuse the flows from previous MRF instance to initialize the current MRF instance for faster convergence. Juan and Boykov [34] on the other hand presented active graph cuts which reuse the previous cuts by using the preflows and pseudoflows which they borrowed from the graph theoretic work [11] for better initialization and faster convergence. They showed the performance of their algorithms on segmenting consecutive frames of video.

Recently few effort have been made to solve the optimization algorithms on the parallel architecture. Dixit $et\ al.$ [32] implemented the Push-Relabel method on the GPU using the shaders. Their implementation was however slower than the CPU implementation. But, ever since CUDA came, new approaches have been followed to optimize the push-relabel method on the GPU archicture. Notable is the implementation of Hussain $et\ al.$ [29]. They first present the implementation of breadth first search (BFS) graph traversal on CUDA, which is extensively used in their push-relabel algorithm. They introduce two optimizations that utilize the special structure of grid graphs. The first one is lockstep BFS, which is used to reduce the overhead of BFS traversals. The second is cache emulation, which is a technique used by them to regularize memory access patterns. Later on many researchers devised new algorithms to harness the processing powers of the GPU and other multicore architectures. Delong and Boykov [**?**] gave a scalable graph-cut algorithm for N-D grids which attains non-linear speedup with respect to the number of processors on commodity multi-core platforms. Liang $et\ al.$ [7] designed a new parallel hardware to solve belief propagation [45] algorithm efficiently.

The success of graph cuts algorithms saw the emergence of new applications being built over them. Graph cuts have been used to interactively separate the foreground and the background objects. Interactive image segmentation is a very subjective problem catering to the users' perspective. All the interactive techniques can be categorized in global and local methods. In the former, users specify a few seed-pixels for the foreground and background objects, from which gaussian mixture models (GMM) models

are built. A graph with pixels as nodes and with source and sink nodes for foreground/background is constructed over the image. An energy function defined on the graph is minimized globally using graph-cuts to extract the foreground layer [47]. Easy extraction of seeds is performed by the GrabCut system [8]. The graph could be defined over oversegmented regions for computational efficiency [28] or geodesic distances used for the segmentation [46].

Local interaction based methods require the user to trace out the boundary of the foreground or to move the brush along the interior. Intelligent Scissors exploit the local gradients [12] while Soft Scissor performs local energy minimization [23]. Painting over the region with local optimization is performed by Edge Preserving Brushes [10], Bilateral Grid [19], and Paint Selection [21]. Adobe PhotoShops Quick Selection is a painting tool within everyones reach [2]. These methods give instant feedback on the segmentation process, guiding the user interaction. Paint Selection provides easy interaction even on large images by building and optimizing the energy function in the neighbourhood of the brush. The interaction required by local methods is proportional to the area of foreground layer or the length of its boundary.

In computer vision, many problems are mapped as multilabeling problems. And over the years many new sophisticated approximate algorithms have been proposed to solve them. Some of the key algorithms which are used to minimize energy functions defined over MRF include $\alpha$-expansion and $\alpha\beta$-swap [49], max-product loopy belief propagation [45] and tree-reweighted message passing [43]. Alahari $et\ al.$ [25] and Komodakis $et\ al.$ [33] extended these ideas to multilabeling problems that use $\alpha$-expansion.

Multiresolution or pyramid processing of images has been used for many applications [16, 4, 21, 24, 42] for fast processing. Restricting the processing in a higher resolution image to a small band around the solution in a lower resolution produced considerable speedups on graph cuts as in the Banded Graph Cuts(BGC) method [16]. The size of the band is selected heuristically and the method is not guaranteed to give the global minimum of the energy function. This method drastically increases the computational speed of graph cuts, but is limited to the segmentation of large, roundish objects. Sinop and Grady [4] propose a modification of BGC that uses the information from a Laplacian pyramid to include thin structures into the band. Therefore, they retain the computational efficiency of BGC while providing quality segmentations on thin structures. They make quantitative and qualitative comparisons with BGC on images containing thin objects. Juan and Boykov [34] also use an approximate solution from a coarser level to initialize the graphs for faster convergence using active graph cuts, without sacrificing global optimality.

Ever since the energy minimization methods were introduced in computer vision to solve different labeling problems, many sophisticated algorithms have been proposed to solve these problems efficiently. Graph cuts based methods have proved to be the most successful methods. But the area is still very active, especially graph cuts have a promising future in the realm of the energy minimization methods.

*Chapter 3*

# Fast Graph Cuts on the GPU

Many realworld problems require computation to be done at realtime rate. Examples include robot navigation, surveillance, video processing etc. Ever since energy minimization methods were used as tools to solve labeling problems in computer vision, researchers have tried to improve the computation efficiency of the methods (algorithmically as well as on specialized hardware). Graph cuts based methods have proved to be iconic in providing accurate results at moderate times. In this chapter, we exploit the computation power of a parallel accelerator like the GPU to improve the speed of graph cuts based methods to solve different MRF optimization problems.

We present a fast implementation of the push-relabel algorithm for mincut/maxflow algorithm for graph-cuts using CUDA. We use the global memory and the shared memory on the GPU effectively. We use the atomic functions operating on the global memory whenever available. We also propose stochastic cuts which adaptively improves the performance of push-relabel algorithm by factors for different problems on the GPU. We also propose a method to map dynamic energy minimization algorithms [35] to the GPU architecture. We additionally explore multilabel MRF optimization problems on the GPU. We present the incremental $\alpha$-expansion algorithm for them. We recycle and reuse the flows from the previous MRF instances. Reuse of the flows from one cycle to the next as well as from one iteration to the next in the first cycle, and shifting the graph constructions to the parallel GPU hardware are the innovative ideas that produce high performance.

We tested our algorithm on image segmentation, stereo correspondence, image restoration, and photomontage problems. All the datasets (shown in Figure 3.1) are taken from the Middlebury MRF page [39, 30]. The energy functions used are the same as used by them. Our approach takes 950 milliseconds on the GPU for stereo correspondence on Tsukuba image of size $384 \times 288$ with 16 labels compared to 5.4 seconds on the CPU.

## 3.1 MRF Optimization on the GPU

Many vision problems are naturally formulated as energy minimization problems. These discontinuity preserving functions have two terms, data term and smoothness term. The general form of the


**Figure 3.1** Some of the images used in our experiments. Images 1-3 for color based image segmentation with 2 labels each. Images 4-6 for stereo correspondence problems. Images 7-8 for image denoising problems with 256 gray labels values each. Finally image 9 for photomontag case.

function is:

$$E(x) = \sum D_p(x_p) + \sum V_{(p,q)}(x_p, x_q) \tag{3.1}$$

where $D_p(x_p)$ , the data term, measures the cost of assigning a label to a pixel $p$ and $V_{(p,q)}(x_p, x_q)$, the smoothness term, measures the cost of assigning pair of labels to the adjacent pixels $p$ and $q$. The MRF is modeled as a graph with a grid structure with fixed connectivity. Our optimized push-relabel implementation does the bilabel segmentation for foreground/background separation in the image. But, for multilabel problems we use the $\alpha$-expansion algorithm to minimize the energy, which is posed as a series of two-label graph cuts. We also reuse the flows to initialize the current MRF instance from the previous iterations and cycles. Two basic steps of updation and reparameterization are also parallelized on the GPUs. Given an energy function, graph construction is the first step which is followed by the graph cuts applied to separate the graph into two disjoint sets of vertices.

### 3.1.1 Graph Construction on CUDA Architecture

Our graph-construction exploits the grid-structure that arises for MRFs defined over images. We adapt the graph construction of Kolmogorov $et\ al.$ [44] as explained in the section 2.1.2, which maintains the the grid structure, suitable for the GPU/CUDA architecture. The graph cuts algorithms require edge weights of the edges in the graph to be non-negative. They gave the condition of regularity which characterizes the class of energy functions which can be minimized via graph cuts. We construct the grid-graph such that each pixel represents a non-terminal vertex in the graph. We assume fixed connectivity which could be 4 or 8 neighbours for each vertex. Consequently, 4 or 8 two-dimensional arrays store the weights along the $n$-edges. Two other arrays hold the excess flow and the edge capacity to the target vertex for each vertex. An array to hold the heights and a mask array to hold the status of each node complete the representation. This representation can easily be extended to 3D grids for 3D graph cuts and other fixed connectivity patterns. Different strategies will have to be adopted for general graphs represented using adjacency list or adjacency matrix. Table 3.1 compares the graph construction timings on the GPU and the CPU for image segmentation problems. Porting this step to the GPU provides a factor of speed up.

| Image | Size | GPU Time(in msec) | CPU Time(in msec) |
|---|---|---|---|
| Sponge | 640X480 | 0.151 | 61 |
| Person | 600X450 | 0.15 | 60 |
| Flower | 600X450 | 0.15 | 60 |

**Table 3.1** The table compares the graph construction timings on the GPU and the CPU.

### 3.1.2 Push-Relabel Algorithm on CUDA

The CUDA environment exposes the SIMD architecture of the GPUs by enabling the operation of program *kernels* on data *grids*, divided into multiple *blocks* consisting of several *threads*. The highest performance is achieved when the threads avoid divergence and perform the same operation on their data elements. The GPU has high computation power but relatively low memory bandwidth. The GPU architecture cannot lock memory; synchronization is limited to the threads of a block. This places restrictions on how modifications by one thread can be seen by other threads. However, later versions of CUDA provide the facility of atomic functions. An atomic function performs a read-modify-write atomic operation on one 32-bit or 64-bit word residing in global or shared memory. For example, atomicAdd() reads a 32-bit word at some address in global or shared memory, adds an integer to it, and writes the result back to the same address. The operation is atomic in the sense that it is guaranteed to be performed without interference from other threads. In other words, no other thread can access this address until the operation is complete. Atomic functions can only be used in device functions and are only available for devices of compute capability 1.1 and above. Atomic functions operating on shared memory and atomic functions operating on 64-bit words are only available for devices of compute capability 1.2 and above.

The basic implementation of the push-relabel algorithm on a hardware without atomic capability requires three phases. Push is a local operation with each node sending flow to its neighbours and reducing own excess flow. A node can receive flow from its neighbours also. Thus, the net excess flow cannot be updated in one step due to the read-after-write data consistency issues. To maintain the preflow conditions without the read-after-write data consistency, we divide the operation into two kernels: Push Kernel and Pull kernel. However, atomic functions can perform read-modify-write atomic operations on 32-bit or 64-bit word residing in global or shared memory. So, we combine push phase and pull phase without any inconsistency. Section 3.1.2.1 describes the implementation on hardware with atomic capabilities. The *Push* phase pushes excess flow at each node to its neighbours and the excess flows of the neighbours and the flows in the corresponding edges are updated. The *Local Relabel* phase applies a local relabelling operation to adjust the heights as stipulated by the algorithm. The heights of the nodes can also be adjusted by applying breadth first search starting from the sink. The breadth first search step is very slow and slows the computation overall.

Our implementation exploits the structure of the grid-graph that arise for MRFs over images, where each pixel corresponds to a node and the connectivity is fixed to its 4-neighbours. The grid has the dimensions of the image. We organize them into a two-dimensional grid of geometry $B_x \times B_y$, where $B_x$ and $B_y$ are the number of thread blocks in $x$ and $y$ directions. Blocks are further divided into $D_x \times D_y$ threads. The maximum efficiency is achieved when $B_x$ and $B_y$ are multiple of $D_x$ and $D_y$ respectively. In our case, each thread block is of size 32×8. To achieve maximum efficiency, we pad the rows and columns to make them multiples of 32×8. Each thread handles a single node or pixel and a block handles $D_x \times D_y$ pixels. It needs to access data from a $(D_x + 2) \times (D_y + 2)$ section of the image. Each node has the following data: its excess flow $e(u)$, height $h(u)$, an active status $flag(u)$ and the residual edge capacities to its neighbours. These are stored as appropriate-sized arrays in the global (or device) memory of the GPU, which is accessible to all threads. This is shown in Figure 3.2.

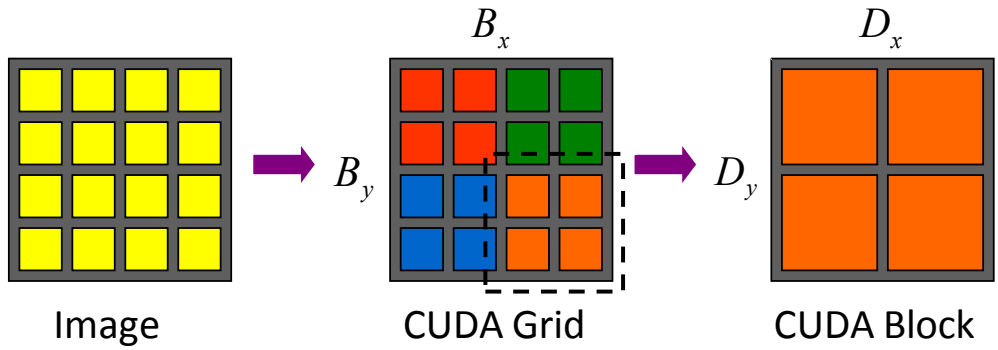



**Figure 3.2** Image is divided into $B_x \times B_y$ block. And each block is further divided into $D_x \times D_y$ threads where each thread corresponds to a pixel in the image.

A node can be active, passive, or inactive. Active nodes have their excess flow $e(u) > 0$ and $h(u) = h(v) + 1$ for at least one neighbour $v$. Passive nodes do not satisfy the height condition, but may do so after relabeling. If a node has no excess flow or has no neighbour in the residual graph $G_f$, it becomes inactive. The kernel first copies the $h(u)$ values of all nodes in a thread-block to the shared memory of the GPU's multiprocessor. Since these values are needed by all neighbour threads, storing them in the shared memory speeds up the operation overall.

#### 3.1.2.1 Graph cuts on hardware with atomic capabilities

**PushPull Kernel** The push operation can be applied at a vertex $u$ if $e(u) > 0$ and its height $h(u)$ is equal to $h(v) + 1$ for at least one neighbor $v \in G^r$, the residual graph. Algorithm 4 explains the implementation of the push operation on the GPU. Atomic writes to the global memory ensure syn-

**Input:** A residual graph, $G^r(V, E)$.

1: Load height $h(u)$ from the global memory to the shared memory of the block.
2: Synchronize the threads of the block to ensure the completion of load.
3: **if** $u$ is in the current graph **then**
4: Push excess flow to eligible neighbours atomically without violating constraints.
5: Update edge-weights of $(u, v)$ and $(v, u)$ atomically in the residual graph $G^r$.
6: Update excess flows $e(u)$ and $e(v)$ atomically in the residual graph $G^r$.
7: **end if**

**Algorithm 4**: Push Kernel: The excess flow $e(u)$ of each vertex is updated in this kernel.

chronization across the blocks of the grid.

**Local Relabel Kernel** Local relabel operation is applied at a vertex $u$ if it has positive excess flow but no push is possible to any neighbour due to height mismatch. The height of $u$ is increased in the relabeling step by setting it to one more than the minimum height of its neighboring nodes in the residual graph $G^r$. Algorithm 5 explains the implementation of local relabel performed on GPUs.

**Input:** A residual graph, $G^r(V, E)$.

1: Load height $h(u)$ from the global memory to the shared memory of the block.
2: Synchronize the threads of the block to ensure the completion of load.
3: **if** $u$ is in the current graph **then**
4: Update the activity bit of each vertex in the residual graph $G^r$.
5: Compute the minimum height of the neighbors of $u$ in the residual graph $G^r$.
6: Write the new height to the global memory $h(u)$.
7: **end if**

**Algorithm 5**: RelabelKernel: The height $h(u)$ of each vertex $u$ is updated if it requires the height adjustment and so relabeling step.

#### 3.1.2.2 Overall Graph Cuts Algorithm

The overall algorithm applies the above steps in sequence, as follows (Algorithm 6). The CUDA grid has the same dimensions as the image, say, $M \times N$. The CUDA block size is $B_1 \times B_2$ threads.

**Input:** A residual graph, $G^r(V, E)$.

1: Compute energies and edge weights from the underlying image.
2: On hardware with atomic capabilities: Perform *PushPullKernel* followed by *RelabelKernel()*; on the whole grid untill convergence.
3: On hardware with non-atomic capabilities: Perform *PushKernel*, *PullKernel* followed by *RelabelKernel()*; on the whole grid until convergence.

**Algorithm 6**: Algorithm explaining the complete push-relabel method on the GPU calling push kernel and relabel kernel iteratively till termination on the GPU.

Estimating the energies and weights can also be performed in parallel on the GPU. This can reduce the computation time on large image that use complex energy functions.

### 3.1.3 Stochastic Cut

We notice that most of the pixels get their actual label after a few iterations on all datasets. Figure 3.4 shows the labels after different numbers of iterations on sponge image. After 10 iterations only 643 pixels are labelled incorrectly for sponge image. Figure 3.3(a) and Figure 3.3(b) give an estimate of error (pixels getting incorrect lebels) and energy, respectively, as the computation of graph cuts progress. Only a few pixels exchange flow with their neighbours later. Processing nodes which are unlikely to exchange any flow with their neighbors results in inefficient utilization of the resources. We also explore the number of blocks that are active after each iteration. An active block has at-least one pixel which has exchanged flow in the previous iteration. The activity is determined based on the change in $n$ edge-weights and $t$ edge-weights in the previous iteration. The kernel marks whether each block is active. Based on the active bit, the kernel executes the other parts of the program. It is observed that after a few iterations, only 5-10% of the blocks are active, as shown in Figure 3.3(c). We delay the processing of a block based on its activity bit. A better model is to delay the processing of a block based on the likelihood and prior information, but we settle for a fixed delay for inactive blocks. We check the activity of a block after each 10 iterations. A block is processed for next 10 iterations if it is active otherwise the block is not processed. Algorithm 7 explains our modified push-relabel method.

**Input:** A residual graph, $G^r(V, E)$.

1: Check the active bit of the block.
2: Perform GPU Graph Cuts every iteration on all the above blocks and every 10th iteration on the inactive blocks.

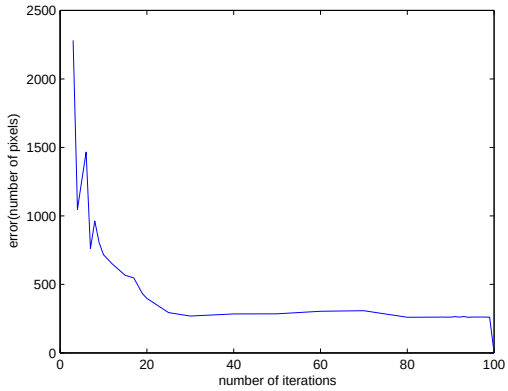
**Algorithm 7**: Kernel explains the steps involved in adaptively processing of blocks based on the activity of the block.

## 3.2 Levels of Optimizations on the GPU

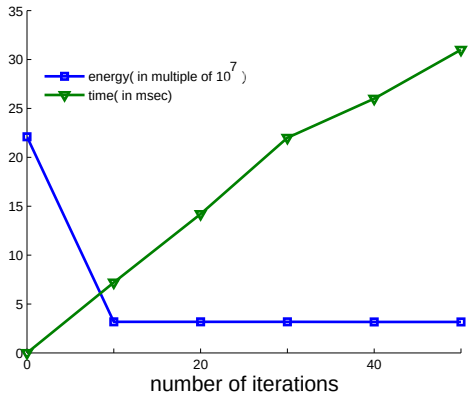
As the GPU has lower memory bandwidth, reducing global memory access is critical to performance. We explored the impact on the running time of different compact representation. The compact versions have to be split into constituent terms after reading. The active flag takes values: $0, 1$ or $2$ and $2$ bits are sufficient to store them. We can compact $16$ active bits in one word. In practice, heights can be represented using 16 bits or even 8 bits and edge weights using 32 bits or 16 bits. We also explored combining the flag bits along with the edge weights and heights.

Table 3.2 and Table 3.3 show the different parameters (incoherency, occupancy, shared memory uses, register counts) which effects the performance on the GPU. Table 3.4 shows the effect of compact representation. These tables show interesting results on GTX 8800 and GTX 280, as far as global reads

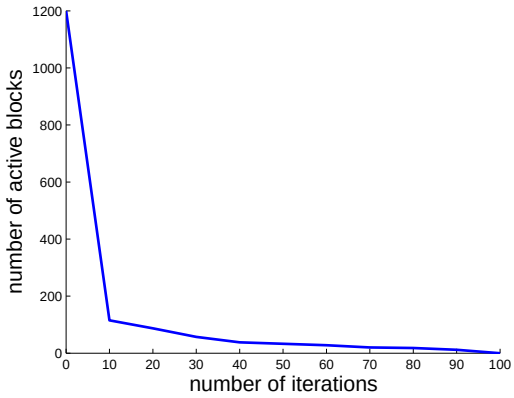
and global writes are considered. The Table 3.2 shows that there are almost 14% incoherent reads and 23% incoherent writes for *Non-Atomic CudaCuts* on GTX 8800. However, there is no incoherent reads and writes on GTX 280. There is always a tradeoff between the shared memory used per block and the register count per thread. These two factors decide the occupancy. As we try to compact more data



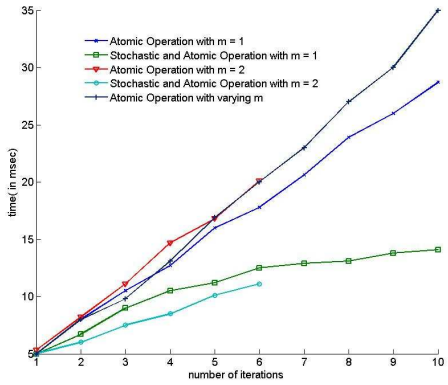

(a) The error (pixels) Vs. iterations graph for Sponge image.




(b) The energy vs. iterations and time vs. iterations plot for Sponge image. It gives an estimate of energy drop as the graph cuts computation progresses.




(c) The number of active blocks with the number of iterations of the graph cuts for sponge image.




(d) The plot compares the performance of different methods for Sponge image as graph cuts computation progresses on GTX280.

**Figure 3.3** Figures $a, b, c$ illustrate the need for stochastic cuts to adaptively manage the work load on the GPUs. Figure $d$ compares the performance of various methods on the GPU.

in a 32 bit word, the efficiency decreases. When heights are in 8 bits and edgeweights are in 16 bits, we get the worst performance. Compacting the data reduces the global memory accesses at the cost of higher number of computations due to the shifting of the data. The register count per thread is also

increased, which reduces the occupancy and so the efficiency. When height, edge-weights and mask all use 32 bits word, we get the best performance on GTX 280. Table 3.5 shows timings of two different implementations discussed in the previous section on the standard image of different sizes on GTX 280.

**Figure 3.4** First two images are the input sponge image and the segmented output image. The other images shows the errors after 10, 20, 40, 80, 90, 100 iterations of graph cuts for Sponge Image.

| Kernel | Occ | Incoh(%) Load | Coh(%) Load | Incoh(%) Store | Coh(%) Store | Shared(Bytes) MemUsed | Reg Used |
|---|---|---|---|---|---|---|---|
| 8800_Push | 1 | 23.3 | 76.7 | 72.5 | 27.53 | 1448 | 9 |
| 8800_Relabel | 0.67 | 0 | 100 | 0 | 100 | 1532 | 16 |

**Table 3.2** Non-atomic: heights, edgeweights, masks are stored in a word. Push and Pull operations are in separate Kernel. (Occ - Occupancy and Reg - Registers used.)

| Kernel | Occ | Incoh(%) Load | Coh(%) Load | Incoh(%) Store | Coh(%) Store | Shared MemUsed | Reg Used |
|---|---|---|---|---|---|---|---|
| 280_Push | 1 | 0 | 100 | 0 | 100 | 1532 | 10 |
| 280_Relabel | 1 | 0 | 100 | 0 | 100 | 1532 | 9 |

**Table 3.3** Atomic: heights, edgeweights, masks are stored in a word. Push and Pull Kernels are combined.(Occ - Occupancy and Reg - Registers used.)

The regular connectivity of the grid graphs results in efficient memory access patterns from the global memory as well as from the shared memory. The use of shared memory in different kernels speeds up the operations by over 20%.

Heights can be stored in one-dimensional or two-dimensional shared memory block. Storing heights in one-dimensional block is efficient. We use a logical OR of the active bit of each node to check the termination condition. Logical OR is evaluated by all active nodes writing a $1$ to a common global memory location. Though CUDA model doesn't guarantee an order of execution, OR can be computed quickly.

The push-relabel algorithm can be modified to perform $m$ push operations before each relabel operations. The experimental results show if relabeling is done every other iteration, the speed increases. However, that does not extend to the higher values of $m$. The multiple push operations before each relabel operation exhausts excess flow quickly. In this way, the algorithm converges in fewer number of iterations. When there is bias towards data term, higher values of $m$ will get efficient performance.

| No. Of bit (ht/edgeweights/mask) | Occupancy | Shared Memory Used | Registers Used | Time(in ms) $(m = 1)$ | Time(in ms) $(m = 2)$ |
|---|---|---|---|---|---|
| 32/32/32 | 1/1 | 1360/1360 | 13/10 | 18.06 | 11.06 |
| 32/32/2 | 1/1 | 2384/1360 | 13/10 | 19.62 | 11.34 |
| 30/32/2 | 1/1 | 2384/1360 | 13/10 | 20.24 | 12 |
| 30/16/2 | 0.75/1 | 2384/1360 | 20/10 | 20.89 | 12.43 |
| 8/32/32 | 1/1 | 1360/1360 | 13/10 | 21.56 | 13.46 |
| 32/16/32 | 0.75/1 | 1360/1360 | 20/11 | 21.5 | 12.6 |
| 16/16/32 | 0.75/1 | 1360/1360 | 20/11 | 21.9 | 13.1 |
| 8/16/32 | 0.75/1 | 1360/1360 | 20/11 | 24.78 | 15.6 |
| 8/16/2 | 0.5/1 | 2384/1360 | 23/11 | 25.82 | 16.7 |

**Table 3.4** The table evaluates different parameters which determine the efficiency of implementation on the sponge image on GTX 280. First column gives the different possible combinations of heights, edgeweights and masks in one word. The second, third and forth columns give the occupancy, shared memory used and register used per thread respectively, for PushPull kernel and Relabel kernel as in Atomic case.

| Image | Size | Non-Atomic | Atomic |
|---|---|---|---|
| Sponge Image | $640 \times 480$ | 28 | 18 |
| Flower Image | $608 \times 456$ | 33 | 32 |
| Person Image | $608 \times 456$ | 31 | 31 |

**Table 3.5** The timings on standard images on GTX 280.

Otherwise, value of $m$ should be kept lower. The cuda cuts on flower image converges in 90 iterations when $m = 1$ and in 65 iterations when $m = 2$. Table 3.6 shows the effect of varying $m$ on timings and number of iteration for convergence of the algorithm on flower image.

When using atomic CUDA Cuts, performing $2$ pushes before each relabel performs the best. However, starting with $m = 1$ and increasing it to 2 after about $40$ iterations performs the best on non-atomic CUDA Cuts.

## 3.3 Dynamic Graph Cuts on the GPU

Repeated application of graph cuts on graphs for which only a few edges change weights is common in applications like segmenting frames of a video. Kohli and Torr describe a reparametrization of the graph that maintains the flow properties even after updating the weights of a few edges [36] as discussed in section 2.1.3. The resulting graph is close to the final residual graph and its mincut can be computed in a small number of iterations.

| m | Number of iteration | Time (ms) |
|---|---|---|
| 1 | 90 | 33 |
| 2 | 65 | 26 |
| 3 | 60 | 39 |
| 4 | 56 | 49 |

**Table 3.6** Comparison of running times of CUDA Atomic implementation without stochastic operations on GTX 280 when value of $m$ is changed on flower image.

The final graph of the push-relabel method and the final residual graph of the Ford-Fulkerson's method are same. So, we adapt the reparametrization scheme to the leftover flow that remains after the push-relabel algorithm. Updation and reparameterization are two basic operations involved in the dynamic graph cuts as discussed in the section 2.1.3. These operations assign new weights/capacities as a modification of the final graph without violating any constraints. The frame-to-frame change in weights is computed for each edge first and the final graph from the previous iteration is reparametrized using the changes. It finds the pixels which change their labels with respect to the previous frame. This operation is performed in kernels in parallel. The two basic operations, updation and reparameterizations, are performed by these kernel. So, the maxflow algorithm terminates quickly on them, giving a running time of few milliseconds per frame. The running time depends on the percentage of weights that changed.

## 3.4 Incremental $\alpha$-expansion on the GPU

Energy minimization algorithms try to reach the global minima of the energy functions. They will converge faster if initialized close to optimum point. Initialization can have a huge impact on the computation time. Alahari et al. [25] and Komodakis et al. [33] extend the concept of dynamic graph cuts to solve multilabeling problems. In this section, we present our incremental $\alpha$-expansion method motivated by the Alahari's method discussed in section 2.1.4. There are three basic steps involved:

- First, we adapt the re-parametrization given by Kohli and Torr to the push-relabel algorithm. The final graph of the push-relabel method and the final residual graph of the Ford-Fulkersons method are the same. We can apply similar reparametrization steps to the leftover flow for the push-relabel algorithm. The graph is updated using reparameterization from one step to another instead of being constructed from scratch.

- Second, we adapt the cycle-to-cycle relation used by Komodakis et al. and Alahari et al. to $\alpha$-expansion. For this, we store the graph at the start of each iteration for future use. The final excess flows at the end of each iteration of a cycle is also stored for use with the same iteration of the next cycle. The edge weights for an iteration in the next cycle are compared with the stored edge

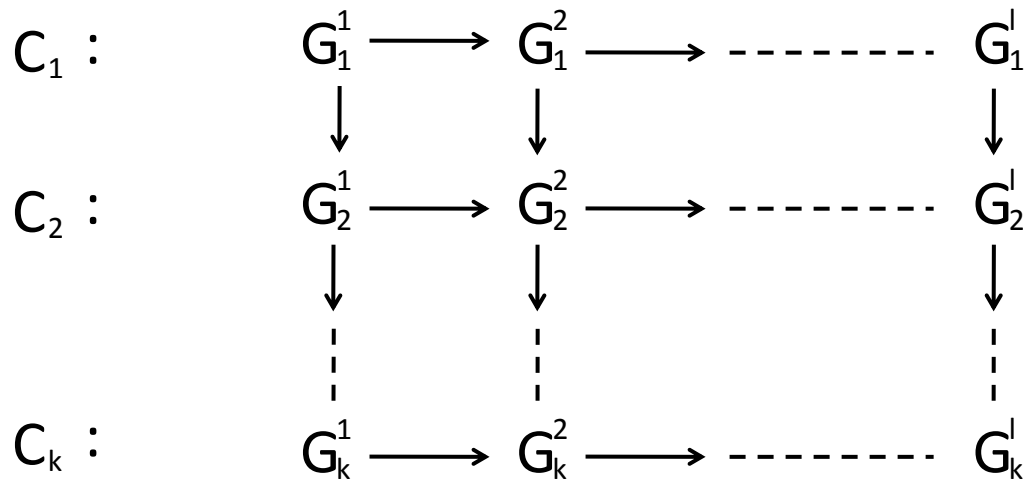



**Figure 3.5** Incremental $\alpha$-expansion. Arrows indicate reparameterization based on the differences in graph constructed. $G_i^j$ is the graph for iteration $j$ of cycle $i$.

weights from the previous cycle. Reparametrization is applied to the stored excess flow from the previous iteration, based on their difference. The reparametrized graph is used for the graph cuts leading to faster convergence. Cycle-to-cycle reuse of flow typically results in a speed up of 3 to 4 times in practice.

- Third step is the incremental step for the later iterations of first cycle, which has no stored value for reparametrization. Nodes with label $i$ do not take part in the iteration $i$ of each cycle; all other nodes do. The graph remains nearly the same from iteration $i$ to iteration $(i+1)$, with a few nodes with label $(i+1)$ dropping out and those with label $i$ coming in. We reparametrize the final excess flows from iteration $i$ using the difference between the graphs at the start of iterations $i$ and $(i+1)$ for the first cycle. In our experience, the iteration-to-iteration reuse of flow for the first cycle reduces the running time of the first cycle by 10-20%.

Figure 3.4 and Algorithm 8 explain our approach for the incremental $\alpha$-expansion.

**Limitations:** Our incremental $\alpha$-expansion method provides an efficient method to solve multilabeling problems on the GPU. However, it needs to store $L$ graphs $G^j$, $1 \leq j \leq L$, one for each iteration. It also stores $L$ excess flows at the end of each iteration of a cycle. The first cycle needs one additional graph to be stored. The extra book-keeping leads to high memory requirements and usage.

## 3.5 Experimental Results

We test our implementations on several standard images (Figure 3.1) for image segmentation, stereo correspondence, image denoising problems and photomontage. The energy terms used are the same as

**Input:** A residual graph, $G^r(V, E)$.

1: Initialize the graph
2: for the first cycle:
3: Construct graph $G_1^1$for $\alpha = 1$, save in `prev`
4: Perform 1-expansion for label 1 using flagged graph cuts
5: Save final excess flow graph in `eflow`
6: **for** labels l from 2 to $L$ **do**
7: Construct graph $G_1^l$ for current label $l$
8: Reparametrize `eflow` based on difference with `prev`
9: Perform $l$-expansion for label $l$ using flagged graph cuts
10: Save final excess flow graph in `eflow`
11: **end for**
12: for latex cycles $i$, iterations $j$
13: Construct graph $G_i^j$
14: Reparameterize based on $G_i^j$ and $G_{(i-1)}^j$
15: Perform $l$-expansion for label $l$ using flagged graph cuts

**Algorithm 8**: KERNEL_INCREMENTAL: Explains the steps involved in incremental $\alpha$-expansion for solving multilabeling problems.

those given in the Middlebury MRF page [39]. The running time also depends on the number of threads per block as it determines the level of parallelism. We experimented with different numbers of threads per block. A block size of $32 \times 8$ threads gives the best results with 256 threads per block. The reported times of the GPU algorithm does not include the time to compute the edge weights. We compared our implementation with $\alpha$-expansion [49].

### 3.5.1 Bilabel MRF Optimization

For segmentation problems, we tested our implementations on various real datasets. Figure 3.6 shows the results of image segmentation on the Person image, Sponge image, Flower image and a synthetic noisy image. The running times for these are tabulated in Table 3.8 along with the time for Boykov's sequential implementation of graph cuts.

The figure 3.3(d) evaluates different optimization methods on GTX280 on sponge image. The stochastic cuts performs the best when $m = 2$. It converges in 70 iterations to the global minimum. However, when $m = 1$ the convergence of graph cuts takes 100 iterations.

Figure 3.7 shows the results of independent segmentation of the frames of a video using our implementation of dynamic graph cuts. The frame-to-frame change in weights is computed for each edge first and the final graph from the previous iteration is reparametrized using the changes. The CUDA implementation of the dynamic graph cuts is efficient and fast. It finds the pixels which change their labels with respect to the previous frame. This operation is performed in kernel in parallel. The two basic operations, updation and reparameterizations, are performed by this kernel. So, the maxflow algorithm terminates quickly on them, giving a running time of 4 milliseconds per frame. Table 3.7 compares

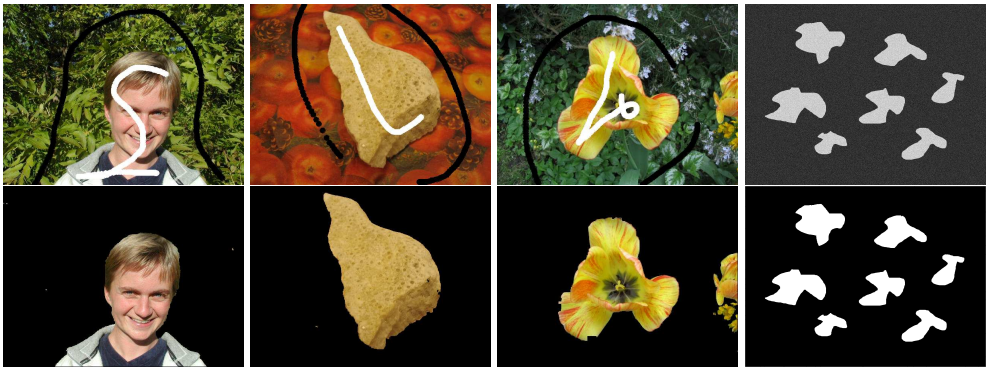


**Figure 3.6** Binary Image Segmentation: Person, Sponge, Flower, and Synthetic images

the time taken to dynamically segment the 30 frames averaging over 10 frames each. The running time depends on the percentage of weights that changed.

**Figure 3.7** Segmenting frames of a video using dynamic graphs

### 3.5.2 Multilabel MRF optimization

We tested our incremental $\alpha$-expansion algorithm on different standard problems such as stereo correspondence, image restoration, and photomontage on various images. Stereo correspondence results are shown on Tsukuba, Venus and Teddy images. Image restoration results are shown on Penguin image and photomontage on the Panorama image. All the datasets are taken from the Middlebury MRF page [30]. The energy functions used are the same as used by them.

Figures 3.8(a) and 3.8(b) show the results of our approach on Tsukuba, Teddy images respectively for stereo correspondence. The results of restoration problem on Penguin image is shown in Figure 3.8(c) and of photomontage problem on Panorama image in Figure 3.8(d). Timings are shown on Middlebury code on the CPU, Fast-PD and dynamic $\alpha$-expansion on the CPU, our basic implementation without flow reuse, and the complete incremental $\alpha$-expansion. Our incremental $\alpha$-expansion on the GPUs is 5-8 times faster than the $\alpha$-expansion on the CPU using Middlebury code [30]. Stereo correspondence on Tsukuba image with 16 labels takes 950 milliseconds on the GPU compared to 5.4 seconds on the CPU. Dynamic $\alpha$-expansion [25] and Fast-PD [33] takes 3.23 seconds for the same. Figure 3.8(e) compares the total times for convergence for different levels of optimization discussed above on various datasets.

| Frames | CPU time | GPU_1 time | GPU_2 time |
|---|---|---|---|
| 10-20 | 108 | 32 | 4 |
| 20-30 | 123 | 37 | 7 |
| 30-40 | 187 | 58 | 11 |

**Table 3.7** Compares the efficiency of our implementation of dynamic graph cuts on the GPU. It shows the average time for segmenting 10 frames of the cow video. CPU time and GPU_1 time shows the computation time taken for st-mincut/maxflow to segment each frame independently on CPU and GPU independently. GPU_2 computes the time taken to segment each frame by dynamically reusing and updating the flows from the st-mincut solution of the previous frame to initialize the solution for the current frame.

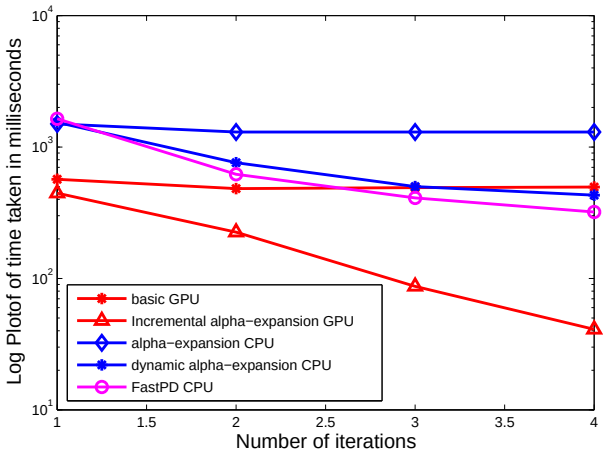
## 3.6 Conclusions

In this chapter, we presented an implementation of graph cuts on the GPU using CUDA architecture. We used the push-relabel algorithm for mincut/maxflow as it is more parallelizable. We carefully divided the task among the multiprocessors of the GPU and exploited its shared memory for high performance. We also present incremental $\alpha$-expansion algorithm for high performance multilabel MRF optimization on the GPU.

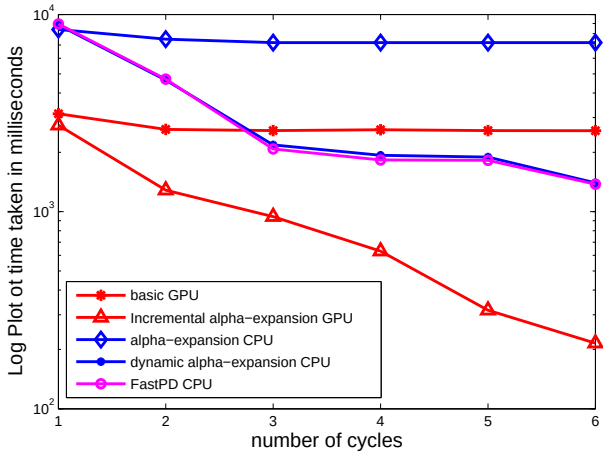
We are able to perform over $60$ graph cuts per second on $640 \times 480$ images. This is $6 - 10$ times faster than the best sequential algorithm reported. More importantly, since a graph cut takes only $30$ to $40$ milliseconds, it can be applied multiple times on each image if necessary, without violating real-time performance. We also presented the incremental $\alpha$-expansion algorithm for high performance multilabel MRF optimization on the GPU. We efficiently utilize the resources available on the current GPUs. We are able to get a speedup of $5 - 8$ times on standard datasets on various problems. Our system brings a near-real time processing of MRF to the reach of most users as the GPUs are now very popular.

| Image | Size | Time (ms) BK | Time (ms) Non-atomic | Time (ms) Atomic | Time (ms) Stochastic |
|---|---|---|---|---|---|
| Sponge | 640× 480 | 142 | 28 | 16 | 11 |
| Flower | 608× 456 | 188 | 33 | 26 | 16 |
| Person | 608× 456 | 140 | 31 | 27 | 20 |
| Synthetic | 1K× 1K | 655 | 19 | 10 | 7 |

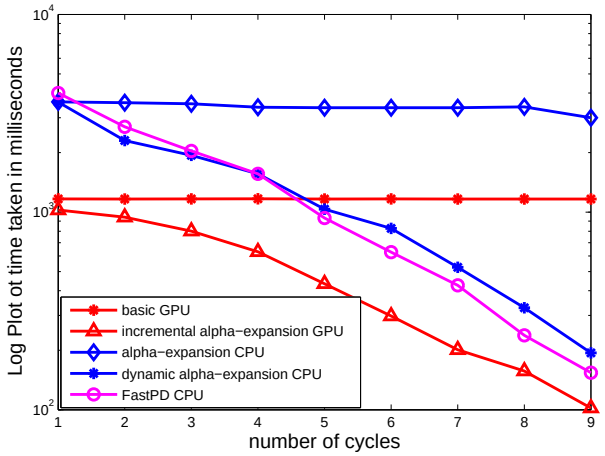
**Table 3.8** Comparison of running times of CUDA implementations on GTX 280 with that of Boykov on different images. Atomic: Push and Pull kernels are combined into one. Stochastic: Atomic functions are applied along with the stochastic operations.



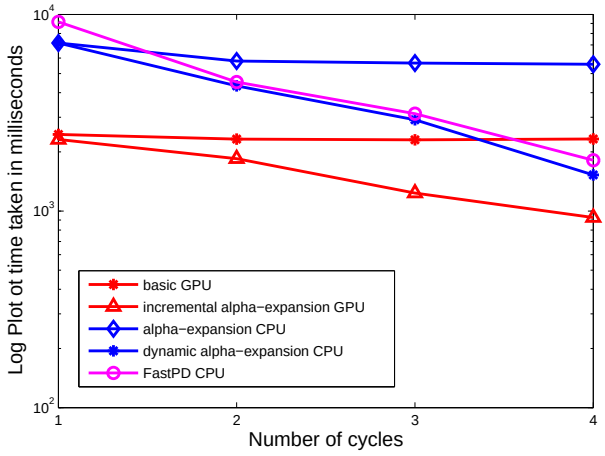

(a) Stereo: Tsukuba Image with 16 labels



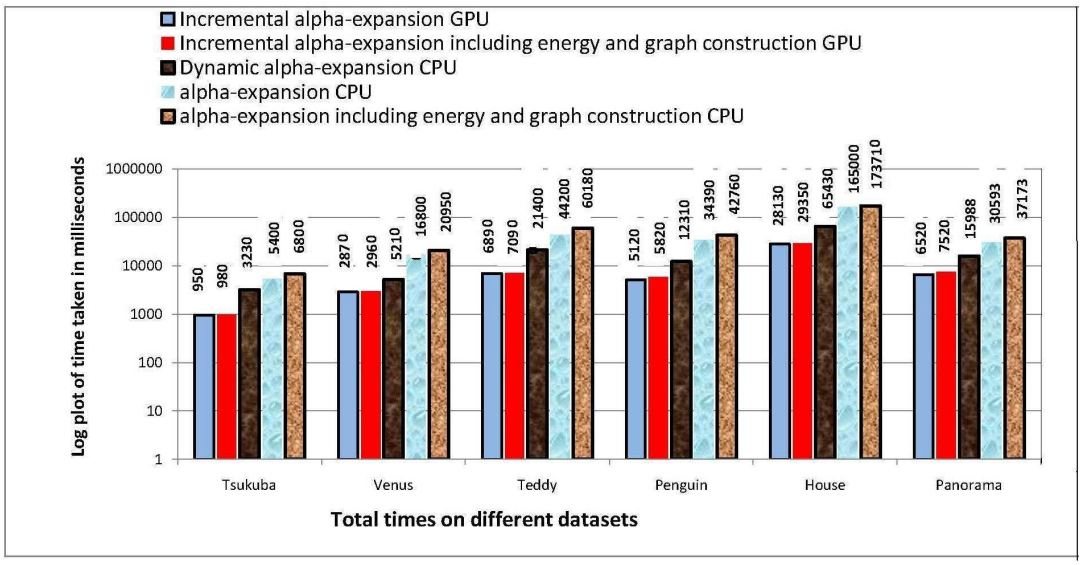

(b) Stereo: Teddy Image with 60 labels




(c) Restoration: Penguin Image with 256 labels




(d) Photomontage: Panorama Image with 7 labels




(e) Total times on different datasets

**Figure 3.8** Timings on different datasets from Middlebury MRF page [30]. (a)-(d) include only $\alpha$-expansion timings.

*Chapter 4*

# Globally Optimal Multiresolution MRFs

Advancement in camera culture and ever increasing demand from the entertainment industries lead to the acquisition of images involving millions of pixels. In this chapter, we focus on developing algorithms to process on these large images containing large number of pixels and large number of labels. We model the problem of optimization on multiresolution MRF framework where solution of one level dynamically updates the problem instance of the next level for better initialization.

We present *Pyramid Cuts*, a multiresolution approach to speed up graph cuts for energy minimization. We combine classical multiresolution or pyramid processing with graph cuts optimization. Our *Pyramid Reparameterization* exploits the optimal results on one resolution level of an image to initialize its graph at the next higher resolution level. Pyramid reparametrization involves an upsampling step to increase the resolution of graphs and a weight modification step to form a graph with the same mincut as the graph constructed at the higher resolution level. The mincut generated by pyramid cuts using multiple levels is identical to the mincut generated using graph cuts on the highest resolution image.

We apply pyramid cuts to several problems: segmentation of images upto 10 million pixels, stereo correspondence computation of images with over 1 million pixels, and denoising of images with upto 1 million pixels. It is 2-4 times faster than graph cuts on the highest resolution representation for segmentation problems. Speedups of 4-6 are observed on multilabelling problems like stereo and denoising. We also implement pyramid cuts on commodity GPUs to exploit their computation power. A speedup of 5-10 over graph cuts on the CPU was achieved on the GPU with pyramid cuts.

Section 4.1 overviews the related work briefly. Section 4.2 presents the pyramid reparametrization scheme and the overall pyramid cuts algorithm. Section 4.3 presents results of using pyramid cuts on segmentation, stereo, and denoising problems. Section 4.4 presents a few concluding remarks.

## 4.1 The Problem

We look at minimizing an energy function of the following general form over an image, commonly used in computer vision:

$$E(x) = \sum D_p(x_p) + \sum V_{(pq)}(x_p, x_q) \tag{4.1}$$

where $D_p(x_p)$ , the data term, measures the cost of assigning a label to a pixel $p$ and $V_{(p,q)}(x_p, x_q)$, the smoothness term, measures the cost of assigning pair of labels to the adjacent pixels $p$ and $q$. A graph is constructed over the image using the energy values for each edge. When the image is very large, computational effort of minimizing the energy is high. Multiresolution processing using an image pyramid is one way to process large images. We present our unified framework of dynamic graph cuts and pyramid processing in the coming sections.

## 4.2 Pyramid Reparameterization Scheme

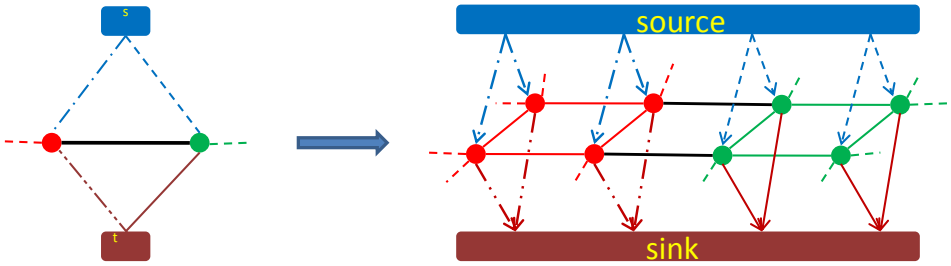



**Figure 4.1** Upsampling generates 4 nodes at level $i$ for each node at level $(i+1)$. The edges between them are given a small weight. The weight of each edge at level $(i+1)$ is copied to the pair of edges generated by it at level $i$. Weights of the 4 $t$-edges are copied from level $i$. Large positive weights are added to the $t$-edges based on the membership of a node at level $i+1$.

We construct an image pyramid $I^1, I^2, \cdots, I^L$ with the original image $I^1$ at the base and the coarsest resolution image $I^L$ at the top. Each pixel of $I^{i+1}$ has the average color of a $2\times2$ neighborhood of $I^i$. We use $G^i$ to denote the weighted graph constructed over the image at level $i$ and $G^i_r$ as the final residual graph after performing graph cuts on it. Pyramid reparametrization to initialize the graph at level $i$ using the results from the level $i+1$ has two steps: upsampling and weight modification. Upsampling generates graphs $\hat{G}^i$ and $\hat{G}^i_r$ from $G^{i+1}$ and $G^{i+1}_r$ respectively, each of identical dimensions of $G^i$. Weight modification adjusts the weights of $G^i$ using $\hat{G}^i$ and $\hat{G}^i_r$ to generate $\widetilde{G}^i$. We show that $G^i$ and $\widetilde{G}^i$ have the same mincut. Graph cuts on $\widetilde{G}^i$ converges to the same global solution faster compared to the graph cuts applied on $G^i$. Upsampling of $G^{i+1}$ is described below (see Figure 4.1).

**Upsampling the Graph $G^{i+1}$:**

1. Each node $v \in G^{i+1}$ generates 4 nodes $\hat{v}_{1-4}$ in $\hat{G}^i$, giving $\hat{G}^i$ the same geometry as $G^i$.

2. The 4 "internal" edges between the nodes $\hat{v}_j$ due to $v \in G^{i+1}$ are given a small weight equal to $\epsilon$.

3. Each edge $(u, v) \in G^{i+1}$ generates 2 edges in $\hat{G}^i$, between adjacent nodes from $\hat{u}_j, \hat{v}_j$. They are given the same weight as $(u, v)$ in $G^{i+1}$.

4. Each $t$-edge in $G^{i+1}$ generates 4 $t$-edges in $\hat{G}^i$. They are given the same weight from $G^{i+1}$.

5. Let $S^{i+1}$ and $T^{i+1}$ be respectively the source and target sets of nodes after optimization at level $i+1$. Add a large positive weight to the edge from the source node $s$ to each $\hat{v}_j$ node generated by a node $v \in S^{i+1}$ and a large positive weight to the edge from each $\hat{u}_j$ node generated by a node $u \in T^{i+1}$ to the target node $t$.

**Claim 1:** The upsampled graphs $\hat{G}^i$ and $\hat{G}^i_r$ have the same mincut.

**Proof:** We show that $\hat{G}^i$ can be converted to $\hat{G}^i_r$ using a few steps that preserve the mincut. Let $\hat{S}^i$ and $\hat{T}^i$ be the set of nodes in $\hat{G}^i$ that are generated by nodes of the source and target sets $S^{i+1}$ and $T^{i+1}$ respectively. The graph $\hat{G}^i_r$ has no edge connecting a node in $\hat{S}^i$ with another in $\hat{T}^i$, as such pairs of nodes will belong to different partitions in $G^{i+1}_r$. $\hat{G}^i$ also has edges with large weights from the source node to nodes in $\hat{S}^i$ and from nodes in $\hat{T}^i$ to the target node by construction (Step 5 of upsampling).

Consider an edge $\hat{e} = (\hat{u}, \hat{v}) \in \hat{G}^i$ with $\hat{u} \in \hat{S}^i$ and $\hat{v} \in \hat{T}^i$. By construction, $\hat{u}$ is connected to the source $s$ and $\hat{v}$ to the target $t$ very strongly. Thus, in one step, $\hat{e}$ can be saturated by sending flow equal to its capacity from $\hat{u}$ to $\hat{v}$. The graph $\hat{G}^i$ can be turned into $\hat{G}^i_r$ by saturating all such edges. This proves the claim that $\hat{G}^i$ and $\hat{G}^i_r$ have the same mincut. We take one interesting case and explain how our upsampling rules maintain the same mincut for the graphs $\hat{G}^i$ and $\hat{G}^i_r$ as shown in Figure 4.2.

**Weight Modification of Graph $G^i$:**

1. Input: 3 graphs of identical geometry: upsampled graphs $\hat{G}^i, \hat{G}^i_r$ and input graph $G^i$. Output: Reparametrized graph $\widetilde{G}^i$.

2. Copy vertex list and edge list of $G^i$ to $\widetilde{G}^i$.

3. Given corresponding edges $e, \hat{e}, \hat{e}_r$, and $\tilde{e}$ in graphs $G^i, \hat{G}^i, \hat{G}^i_r$, and $\widetilde{G}^i$ respectively, set the weight $w(\tilde{e})$ as $w(e) + w(\hat{e}_r) - w(\hat{e})$. Weights are made non-negative as explained in [35].

**Claim 2:** The pyramid reparametrized graph $\widetilde{G}^i_r$ has the same mincut as the graph $G^i$.

**Proof:** Since $\hat{G}^i$ and $\hat{G}^i_r$ have the same mincut, the weight modification procedure is a reparametrization of $G^i$ that preserves the mincut, as shown by Kohli and Torr [35]. Thus, $\widetilde{G}^i_r$ has the same mincut as $G^i$.

Pyramid cuts generates results identical to graph cuts on the original graph $G^1$. This is true even if small foreground segments disappear at lower resolutions due to averaging. Banded graph cuts [16] cannot handle such cases and adaptive banded graph cuts [4] does so by maintaining a separated record of changes. Algorithm 9 gives the pseudocode of Pyramid Cuts.

**Practical points:** In practice, the large positive weight added to the $t$-edges cancel away in the weight modification step. Its role is limited to proving Claim 1. In the pyramid reparametrized graph $\widetilde{G}^i$, the "internal" edges between nodes $\hat{v}_j$ will have weights equal to their corresponding edges in $G^i$. Other $t$-edges and $n$-edges will have reduced or zero weights depending on the difference between $G^i$ and $G^i_r$. Thus, graph cuts on $\widetilde{G}^i$ will converge quickly to the global minimum of the graph $G^i$.

Pyramid cuts uses pyramid reparametrization to initialize the graph at one level using the optimum from the next coarser level. The graph so formed is optimized using regular graph cuts. We use the code

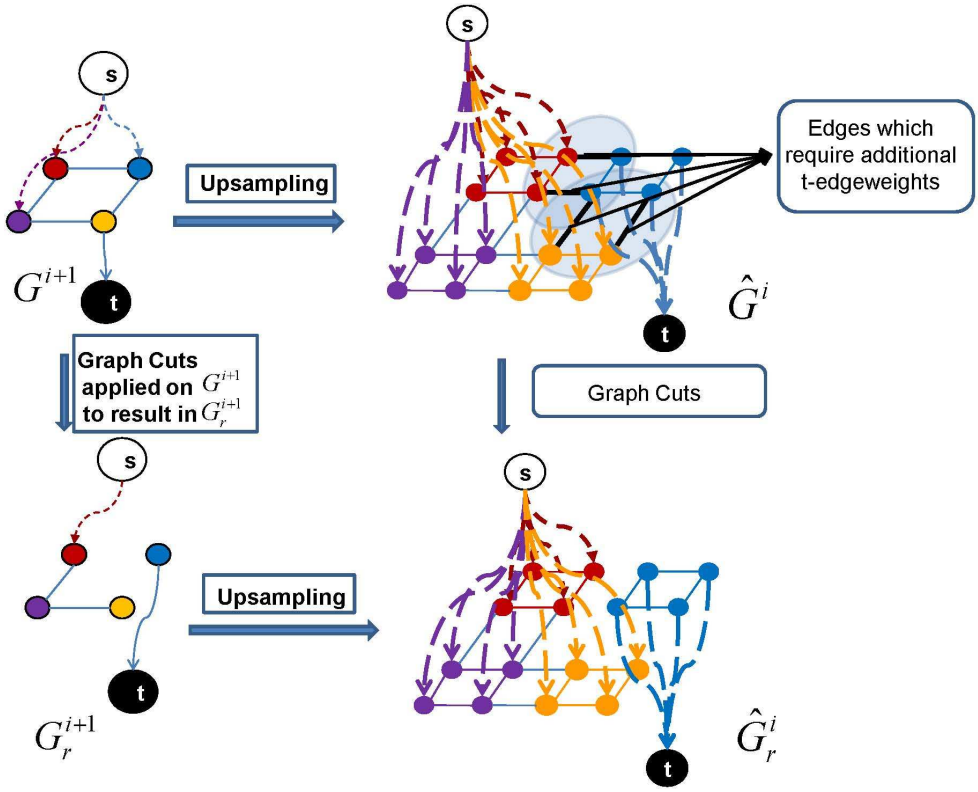



**Figure 4.2** $G^{i+1}, G_r^{i+1}$ and its upsampled versions $\hat{G}^i, \hat{G}_r^i$. $\hat{G}^i$ can be converted to $\hat{G}_r^i$ by saturating edges that connect $\hat{S}^i$ and $\hat{T}^i$ in the shaded region.

by Boykov and Kolmogorov [48], which is the fastest sequential implementation available, for this step on the CPU. Enhanced performance is given by the pyramid processing used instead of processing the image in the original (highest) resolution.

Pyramid cuts can also exploit the Graphics Processor Unit (GPU), when present. Commodity GPUs found on most systems provide high computation power of around 1 TFLOP. We use our own GPU implementation of graph cuts described in the previous chapter. Pyramid reparametrization is performed in parallel on the GPU when moving from one level to the next. The GPUs can also compute the energy functions in parallel.

Pyramid cuts runs 2-4 times faster than single level graph cuts on the CPU and 4-8 times on the GPU as shown in Figure 4.3(a). For our scheme, the optimization time continues to drop as more resolution levels are added to the pyramid initially. Adding more levels, however, deteriorates the performance beyond a point (Figure 4.3(b)). This is due to the higher overhead of processing additional levels compared to the gains. Experimentally, stopping at a resolution level where the image has 20 to 40 thousand pixels is most efficient.

**Limitations:** The speed advantage of pyramid cuts comes at the cost of large memory requirements. The image pyramid can take need 33% more memory than the highest resolution copy. We also need

1: Construct a pyramid of images $I^1, I^2, I^3, ..., I^L$ with the original image $I^1$ at the base. Pixels of $I^{i+1}$ are averaged versions of a $2 \times 2$ neighborhood of $I^i$.
2: Construct and save the edge capacitated graph $G^L$ for the smallest image $I^L$.
3: Perform graph cuts to separate graph $G^L$ into two sets.
4: Save the final residual graph $G_r^L$.
5: **for** levels $i = (L-1)$ to 1 of the pyramid **do**
6: Upsample the initial and final graphs $G^{i+1}$ and $G_r^{i+1}$ to $\hat{G}^i$ and $\hat{G}_r^i$
7: Construct and save edgecapacitated graph $G^i$
8: Get the reparameterized graph $\widetilde{G}^i$ using $G^i$, $\hat{G}^i$ and $\hat{G}_r^i$
9: Perform graph cuts on $\widetilde{G}^i$ to separate graphs into two sets
10: Save the final residual graph $G_r^i$
11: **end for**

**Algorithm 9**: Pyramid Cuts Algorithm, unifying the concepts from dynamic graph cuts and multiresolution MRFs.

to store $G^i$ and $G_r^i$ for each level for use by the next level. This can triple the memory requirements to store the graphs. This is a problem especially on the GPU as its memory capacity is limited and cannot be enhanced.

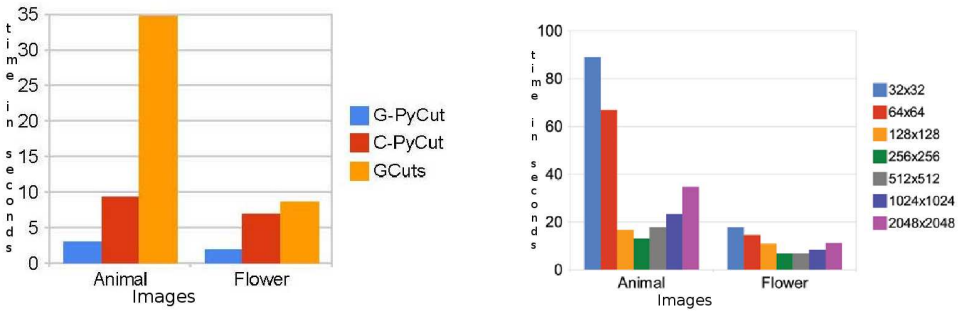
## 4.3 Pyramid Cuts: Applications

We show how different labelling problems in computer vision can be solved efficiently using pyramid cuts in this section. We demonstrate its applications on segmenting large images. We also use pyramid cuts to speed up the graph cuts used by $\alpha$-expansion for multilabel problems and show results on stereo correspondence and image denoising. Pyramid cuts is applied directly for segmentation problems, which assign binary labels.

The run times are computed on a dual-core CPU with 4 GB of memory. The GPU run times are computed on a quarter of the an Nvidia Tesla machine with 4GB of device memory and 240 computing cores. Our focus is on the performance enhancement brought by pyramid reparametrization. We therefore compare the multiresolution computation time of pyramid cuts with the equivalent processing at the base level of the pyramid directly. The same implementation of graph cuts is used for single level processing and for pyramid processing. The code by Boykov and Kolmogorov [48] is used for CPU comparison. We use our own GPU implementation of graph cuts described in the previous chapter.

### 4.3.1 Image Segmentation

We used pyramid cuts to segment images of different sizes and types of content on the types of images shown in Figure 4.9. All images had content complex boundaries between foreground and background. The segmentation framework used is similar to GrabCuts [8]. A user brushed the images to give foreground and background strokes. These were used to initialize Gaussian Mixture Models




(a) (b)

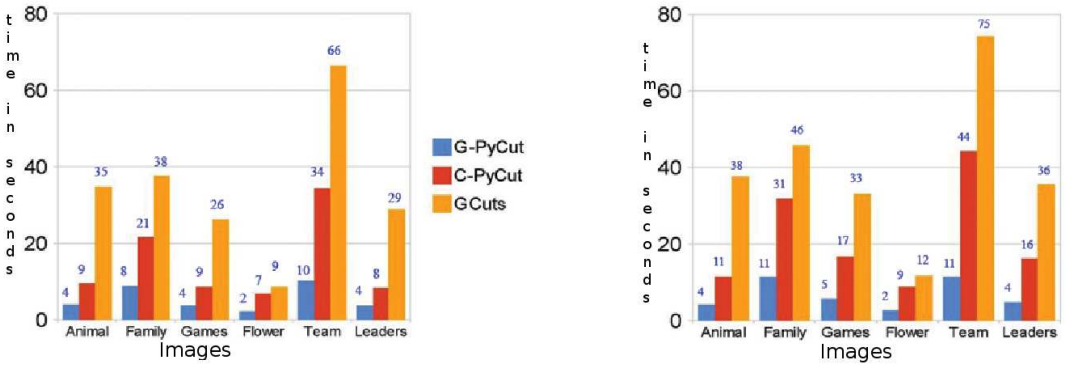
**Figure 4.3** (a) Comparison of the optimization time in seconds of graph cuts with Pyramid Reparameterization scheme on CPU and GPU on two images. (G-PyCut-GPU Pyramid Cuts, C-PyCut-CPU Pyramid Cuts, GCut-a single level graph cut). (b) Total running time in seconds of Pyramid Reparameterization on two $2048 \times 2048$ images as the number of pyramid levels is changed. The size of the lowest resolution is shown. All were run on CPU only.

(GMM) for the foreground and the background. The data terms for the energy came from the GMM and a Potts model was used for the smoothness term.

**Figure 4.4** Accuracy of segmentation improves as we use pixels from all levels to build the GMMs. Left half shows the image with a window marked on it. Right half shows the segmentation of the marked window when the GMM is built using pixels from: *top Left*: all images, *top right*: the largest image, *bottom left*: the same level only, and *bottom right*: the smallest image

**GMM Building** GMM model needs to be built carefully when processing image pyramids with large variations in the foreground and background pixels. We use more Gaussians in the mixture than comparable methods; 15-20 Gaussians in each mixture works best. The strokes made on one particular resolution can be transferred to corresponding pixesl in other levels. We build the GMMs using pixel colors from all levels of the pyramid. In practice, the low-pass filtering in the higher levels makes the models computed using all levels produce better segmentation than models built using pixels from the highest resolution alone (Figure 4.4). It also worked better than using separate GMMs for each level that are trained on pixels from that level as seen in the figure. Figure 4.4 also shows a comparison between

different GMM training options. The different ways of building GMMs only impacts the segmentation quality and has no effect on the computation time, however.




(a) (b)

**Figure 4.5** Running time in seconds for image segmentation using Pyramid Cuts on the GPU (G-PyCut), on the CPU (C-PyCut), and a single level graph cut (GCut). (a) The optimization time. (b) The total time, including energy calculation, graph construction, and optimization timings.

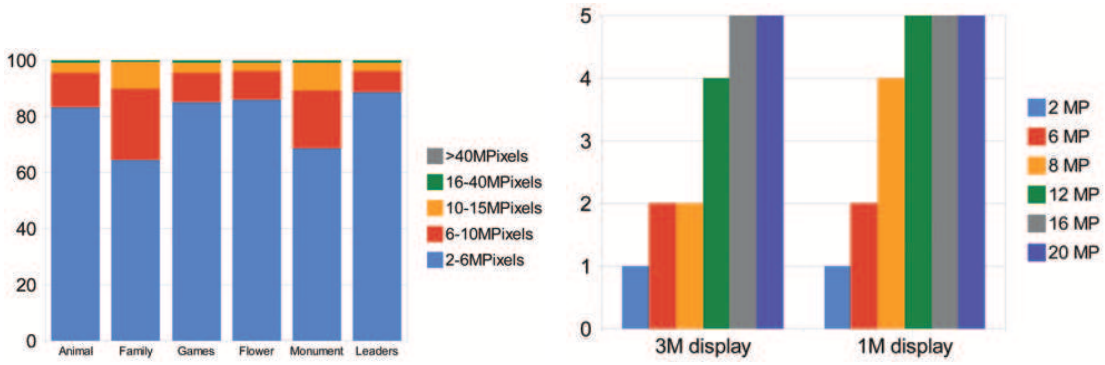
The running time of Pyramid Cuts on the CPU and the GPU and a single level graph cut on the CPU is given in Figure 4.5. The optimization time is the time to compute the mincut after the graph is setup. Pyramid Cuts performs the optimization 2-4 times faster than graph cuts on the highest resolution level on the CPU. The speed up on the GPU ranged from 4-9. We also show the total segmentation time, which includes the time to compute the energy values and to build the graph. The speed up drops to 2-4 on the CPU over but increases to 6-10 on the GPU. The GPU version evaluates the energy values and builds the graph in parallel, yielding higher performance. It must be noted that the graph building time for pyramid cuts includes the time for pyramid reparametrization. Our experiments using pyramid cuts on over 15 other images with 1 million to 10 million pixels gave an average speed up of 3.5 for optimization and 3 for complete segmentation on the CPU. The corresponding numbers on the GPU were 7 and 9 respectively.

#### 4.3.1.1 Interactive Image Segmentation

User interaction plays an important role in foreground/background layer separation. Several methods have been presented to segment a foreground layer interactively from images. A graph-cuts based energy minimization is at the heart of most approaches. Global optimization was used in early methods [46, 1, 47, 8]. Local optimization using a virtual paint-brush has also been used recently, especially to provide interactive experience on large images [10, 21]. Processing large images is a necessity today as

even entry level digital cameras take pictures with multiple millions of pixels. Global optimization is computationally challenging on such images.

Large images are likely to contain multiple, distributed regions that constitute the foreground layer. Figure 4.9 presents several examples. In contrast, most previous methods focused on segmenting one dominant foreground object in the image. The distributed nature of foreground makes painting or boundary based methods tedious for the user. Global optimization based on a few scribbles to define the foreground and background will work best on such images. The computation requirements may challenge global optimization methods, however. Fast processing is needed to retain the user's interactive interest. A quick perusal on the web suggests that an overwhelming fraction of the public images is of size less than 10 million pixels (Figure 4.6(a)). Fast segmentation of such images can be very useful to a large number of users.




(a) Statistics of image sizes available on Google images for a few common search keywords such as "animal", "family", "soccer", "flower", "statue of liberty", and "Obama". An overwhelming fraction of images are of size 2 to 10 million pixels. Only 0.6% of fewer images had more than 40 mega pixels.

(b) Results of a user study on image size for comfortable manipulation for two display sizes. Average subjective response on a scale from 1 to 5 (1 best) is shown for six image sizes. Images that are larger than the display is disfavored by users.

**Figure 4.6** In our experiments, the large images for segmentation range from 2 MP to 10 MP. It is supported by the results of the quick perusal on the web.

We thus present *Pyramid Segmentation System*, an approach to separate a distributed foreground layer in large images using a global optimization method, which provides a quick and approximate segmentation feedback like painting based methods. At the core of our segmentation system is *pyramid reparametrization* technique discussed in section 4.2. We also exploit the computation power of the GPUs to provide a quick, interactive response. The bulk of the interaction is performed at a coarse pyramid level at which the entire image is visible on screen. Figure 4.6(b) shows that users prefer handling images that fit the screen and dislike scrolling for selection. Our system presents a novel user experience that presents a quick segmentation to the user, while the complete processing proceeds in the background. This exploits the gap between the responses of the human perceptual and motor systems to hide expensive computations while providing a satisfactorily quick response.

**Pyramid Segmentation System** Pyramid Segmentation system first loads the image and builds $L$ levels of the image pyramid. The choice of $L$ depends on computational considerations only. In practice, we stop at level $L$ when the number of pixels in $I^L$ is in the neighborhood of 40K or approximately $200 \times 200$ (Figure 4.3(b)). The image displayed to the user is of level $d$, at which $I^d$ has a size in the range of 160K to 250K. This balances the presented detail with ease of interaction. Larger images provide more detail, but is more tedious to interact with if panning or large mouse movement is necessary. User can increase the display resolution for finer interaction. We maintain the window size when doing so. The foreground and background specified using strokes or bounding box at the display resolution level are transferred up and down the pyramid to build or update the GMMs. The overall process of Pyramid Segmentation is described below.

1. Load the image and build $L$ levels of the pyramid using $2 \times 2$ averaging. Display $I^d$ to the user for interaction.
2. Get foreground and background strokes from the user. Transfer the strokes to all levels and update the GMMs using pixels of all levels.
3. Perform complete graph-cuts on graph $G^L$.
4. Build the graph $\widetilde{G}^i$ at level $i$ from the results of level $i+1$ using pyramid reparametrization. Perform graph cuts on it.
5. Quick Segment: Display the segmentation results to the user once the image $I^d$ is processed. This provides the user with a perceptual response to start planning further interactions.
6. Continue processing lower levels using Step 4 in the background till $I^1$ is processed.
7. Propagate the segmentation mask up the pyramid. Change the image displayed, if necessary.
8. Wait for further user input. Change $I^d$ if user changes the display resolution. Repeat processing from Step 2.

The system provides quick response to the user (Step 5) to boost the interactive experience. These results could change after complete processing. The changes are small and appear quickly, before any user action. Table 4.1 gives the time to quick segmentation results and the full segmentation result, available after the base image is segmented. We are clearly able to provide a quick response to the user's perceptual system, while exploiting the lag in the physical response time for complete computation. This is critical to provide interactive experience to images with 10 million or more pixels. Most of the distributed foreground layer is recovered in a few iterations. Local adjustment of specific parts can then proceed at higher resolution levels, if necessary.

Our system displays the image to be segmented at a display resolution of around 200K pixels. The user interacts with this image. The complete view of the image that the user has made this interaction more effective. The user has the ability to increase or decrease the display resolution to facilitate finer interaction at places. The impact of such interactions can be restricted to a local region of the image, typically the portion being displayed. Global impact is enabled by default.

| Image | CPU time | | GPU time | | Difference in pixels (%) |
|---|---|---|---|---|---|
| | Quick | Final | Quick | Final | |
| Animal | 3 | 7 | 0.82 | 3.2 | 5.2 |
| Games | 7.2 | 17.4 | 1.4 | 8.3 | 1.8 |
| Flower | 1.8 | 7.2 | 0.7 | 2.4 | 0.5 |
| Birds | 3.4 | 13 | 1.2 | 7 | 4.7 |

**Table 4.1** Time in seconds till the quick segmentation response and the final response after complete graph cuts on several images along with the differences in pixels between the two. Less than 5% of the pixels of the quick segmentation region change after complete processing.

We compare the performance of our pyramid segmentation scheme with GrabCut which uses global optimization and Quick Selection which uses local selection. The comparison was made on a computer with a dual-core CPU and 4 GB of memory. The GPU timings were obtained on the same machine with an Nvidia GTX280 card.

Speedy response is the advantage of our method over other global methods and ease of interaction on complex foregrounds is our advantage over painting based methods. Figure 4.8(a) shows the total response time, from the time when the user ends manual interaction to the final response. The total time to segment the foreground layer satisfactorily over multiple iterations is shown there. The interaction time of local methods will be high [21] on images with a distributed foreground layer, though they provide quick feedback. In our system, the quick segmentation provides fast feedback to the user's perceptual system to maintain interest, while complete optimization proceeds in the background. The average total response time of our method is less by a factor of 3-4 than GrabCut and Quick Selection. Complex images with a distributed foreground is particularly challenging for painting based methods as the image needs to be panned and interacted with everywhere. Paint Selection [21] would have been the closest tool to compare with, but we were not able to obtain code for it or run it on our images.

**User Study:** We conducted a user study to evaluate the response of users on different systems on three images (Animal, Flower, and Politician) as shown in Figure 4.9. We requested volunteers to segment three images using Quick Selection, GrabCut and Pyramid Segmentation after giving them a short tutorial on them. The results for 3 typical users are shown in Figure 4.7. We collected the total interaction time, segmentation response time, and an overall subjective score on the experience.

Users rated the Pyramid Segmentation system better than others on these images. The quick segmentation result was found to be useful as they could start planning for further interaction sooner. The users did not feel they were waiting for results when the final segmentation was still being computed in the background. The scribble based interaction was preferred on such images by the users as they didn't have to deal with each foreground region independently. Scrolling was found to be very distracting; the availability of the overall image in the display window, even at a lower resolution, was a factor in favor of Pyramid Segmentation. Figure 4.7 shows this study. One user found the change in output from the quick segmentation to the final segmentation on the "Games" image distracting as it interfered with the

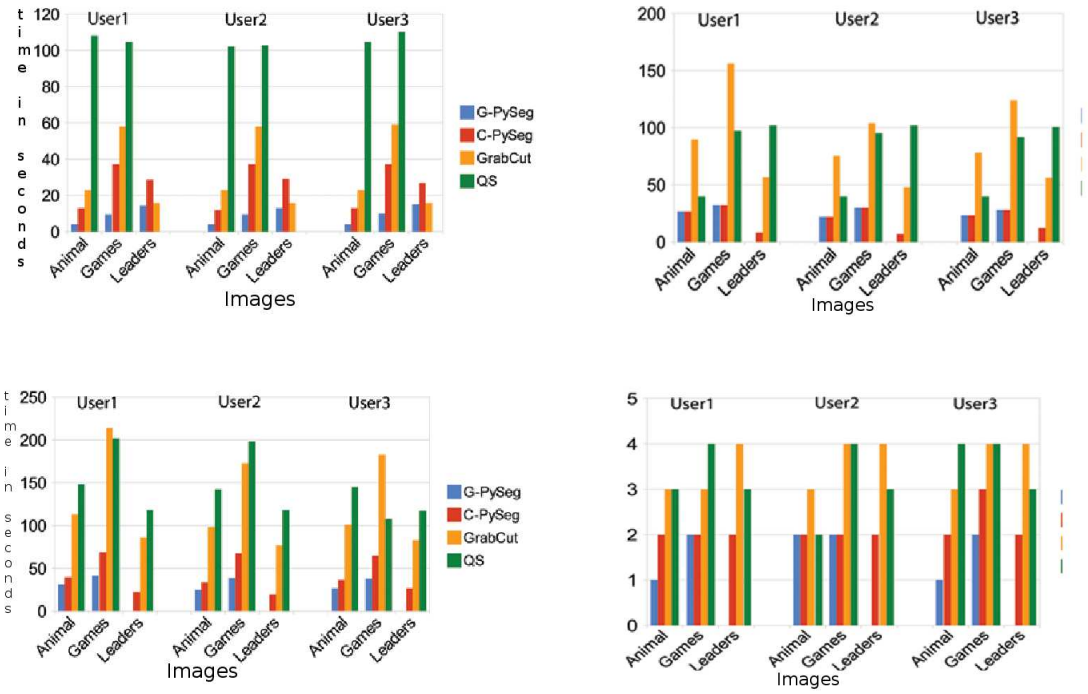



**Figure 4.7** Results of the user study on CPU and GPU versions of pyramid segmentation, as well as GrabCut and Quick Select on the CPU. Top row, left: Interaction time that includes the time for scribbling and image panning, right: response time of the system after interactions are over. Bottom row, left: total time for the interaction including everything, right: subjective feedback of the users on the overall experience, with 1 being the best. All times are given in seconds.

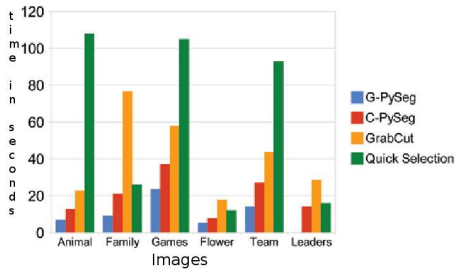
plans made for the next iteration. Pyramid Segmentation on the GPU was generally rated above it on the CPU due to the faster response times (Figure 4.8, right).

Overall, the total time spent by a user to extract the foreground using Pyramid Segmentation on these images was on the average less by a factor of 3 to 4 compared to GrabCut and Quick Select (Figure 4.7, bottom left). The use of GPU for the computations improved it further, whenever possible. The quick segmentation feedback kept the user interested in the process, while simultaneously reducing the segmentation time.

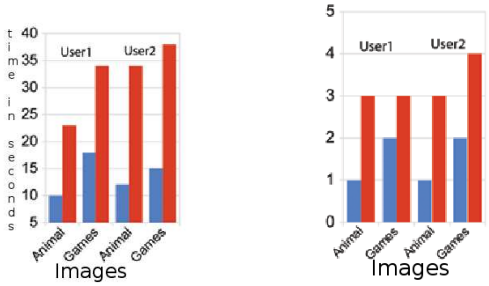
### 4.3.2 Stereo Correspondence

Stereo correspondence is modelled as a problem of assigning a label to each pixel corresponding to its disparity. It is solved using $\alpha$-expansion, which proceeds in cycles, each of which performs $L$ iterations, where $L$ is the number of labels. Each iteration performs a 2-label graph cuts on a suitably constructed graph [44]. We use the pyramid cuts procedure for each such step.

**Label-space compression:** The labels represent disparity between the left and right images. At higher levels of the pyramid, the range of labels shrinks by a factor of 2, along with the horizontal




(a) Comparison of the complete response time in seconds of pyramid segmentation on the GPU (G-PySeg) and the CPU (C-PySeg) with GrabCut and Quick Selection on the CPU. Total response time is measured from when the user removes the brush to the time at which the output is displayed. Time to segment all objects in the foreground layer is reported.




(b) Results of another user study to measure the interaction time in seconds (left) and subjective user experience on a scale of 1 to 5 with 1 being the best (right) with and without the quick segmentation response. Users updated the seed pixels and performed the segmentation. Quick segmentation option resulted in a positive experience consistently.

**Figure 4.8** Figures illustrate the total response timings and the subjective response of the users while conducting user studies.

resolution of the image. The number of iterations required is very small as a result at higher levels. Moreover, the disparity value $d^{i+1}$ computed at level $i+1$ is a good approximation of the true disparity. Thus, the search space in level $i$ can be restricted to the interval $[2d^{i+1} - \Delta, 2d^{i+1} + \Delta]$ for a suitably small $\Delta$ (Figure 4.11). The number of iterations in each cycle of level $i$ will then be $2\Delta + 1$, resulting in reduced computations.

Figure 4.10 shows the left image and the disparity for some of the pairs used for experimentation, taken from the Middlebury stereo data set [30]. Figure 4.12 shows the running times for a few pairs. The optimization time and the total time are shown separately. We use 4 or 5 resolution levels for the full-sized images. The $\Delta$ value varies from 5 in level 4 to 40 in level 1, when the disparity range was over 250 in level 1. Pyramid cuts achieve a speed up of 4-6 over the single level graph cuts on the CPU. The speed up on the GPU is 15-25 for the total time. This is due to computing the energy values and graph construction in parallel. Figure 4.13 shows the running times on images of different sizes. Pyramid cuts achieves greater speed ups on larger images. Similar timings were obtained on other examples from the Middlebury page as they are all of very similar sizes.

### 4.3.3 Image Denoising

Image denoising is modelled as a problem to assign a label corresponding to the noise free gray label to each pixel. It is solved using the $\alpha$-expansion algorithm that proceeds in cycles and iterations. Our implementation uses pyramid cuts for the basic optimization in each iteration.


**Figure 4.9** More images from the test data set, ranging from 2 to 13 mega pixels. Top row: original images, middle Row: Output of pyramid segmentation method, bottom row: Output of Quick Selection. Image category and the sizes are also given.

Figure 4.14 shows the input image, the noisy image, and the denoising results for a few of the images we experimented with. For this experiment, Gaussian noise with zero mean and a standard deviation of 40 was added to each pixel. Figure 4.15 shows the optimization and the total times for some of the images using pyramid cuts and single level graph cuts. Pyramid cuts achieves a speed up of 2-3 on the CPU and a speed up of 6-12 on the GPU on the total time. We also experimented with different levels of synthetically added noise. Experimentally, the running time is mostly independent of the noise level but increases with the size of the image being processed.

## 4.4 Conclusions

In this chapter, we presented Pyramid Cuts for multiresolution computation of graph cuts defined over regular MRFs used in computer vision. The pyramid reparametrization scheme ensures that the graph cuts computed over the pyramid has exactly the same energy as the minimum cut computed on the full graph. We achieved speed ups ranging from 4 to 25 on various problems involving segmentation, stereo correspondence, and denoising.

Performing graph cuts on large images in practical time is still a computational challenge. The basic maxflow computation is being optimized using multicore CPUs and GPUs. Pyramid cuts adds an independent dimension of multiresolution processing to them and can benefit from improvements in the basic algorithm.

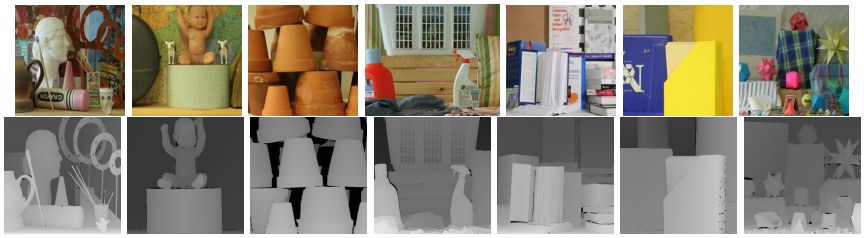


**Figure 4.10** Images from the stereo test data set. Top row: original images. Bottom row: output of stereo using pyramid cuts. Image labels, sizes and the number of disparity levels from left to right are: Art (1328×1104, 200), Books (1328×1104, 200), Laundry (1328×1104, 230), Plastic (1264×1104, 280), Bowling (1248×1104, 290), Flowerpots (1312×1104, 251), Moebius (1264×1104, 280).

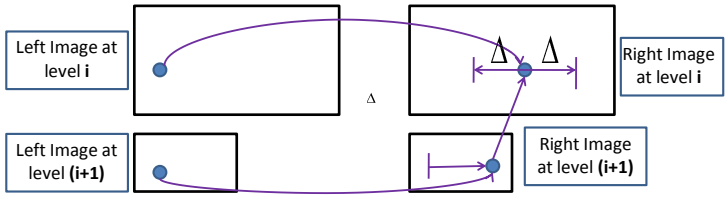
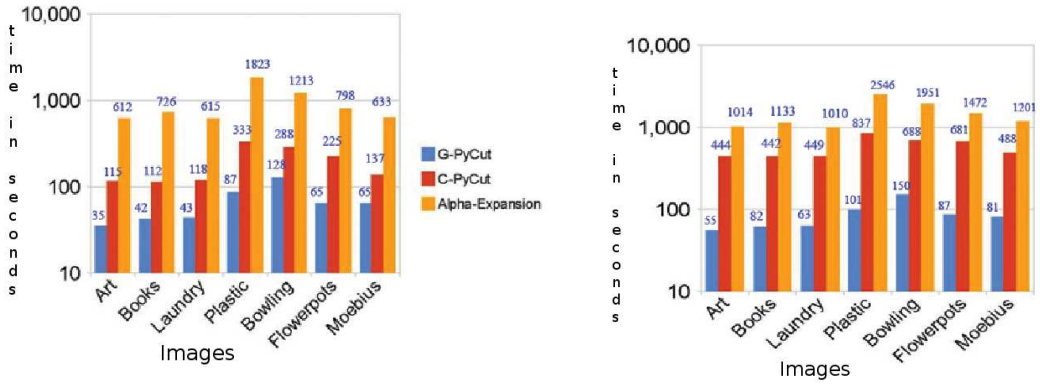



**Figure 4.11** The range of labels halves with each level of the pyramid. The search for labels is limited to a $\Delta$ neighborhood of the disparity found at the lower resolution level.

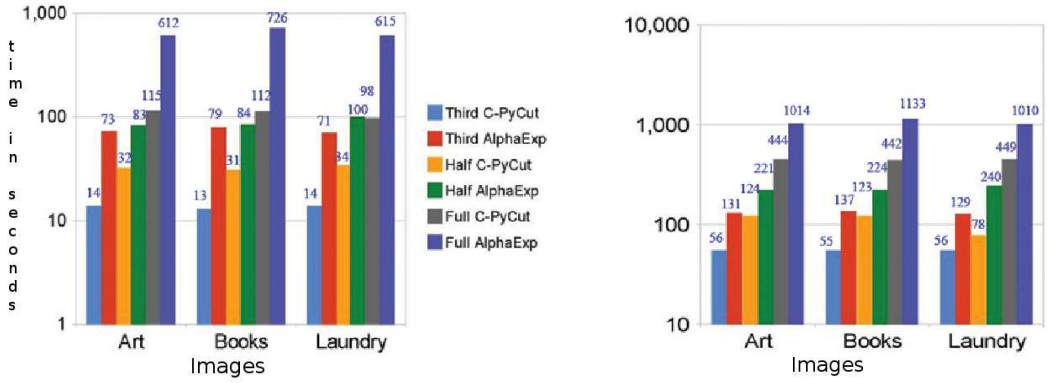



(a) (b)

**Figure 4.12** Running time in seconds for stereo correspondence using Pyramid Cuts on the GPU (G-PyCut), on the CPU (C-PyCut), and a single level graph cut (GCut). (a) The optimization time. (b) The total time, including energy calculation, graph construction, and optimization timings.

| Image | Size | EnFun | | GraphConst | | OptTime | | TotalTime | |
|---|---|---|---|---|---|---|---|---|---|
| | | Ours | $\alpha$-exp | Ours | $\alpha$-exp | Ours | $\alpha$-exp | Ours | $\alpha$-exp |
| Flower1 | 1600x1200 | 1.22 | 0.72 | 14.85 | 7.56 | 0.82 | 14 | 16.89 | 22.82 |
| Flower2 | 1600x1200 | 1.07 | 0.72 | 14.17 | 7.62 | 0.86 | 8.5 | 16.1 | 16.84 |
| Flower3 | 1600x1200 | 1.09 | 0.71 | 17.05 | 7.67 | 0.85 | 26.94 | 19 | 35 |
| Flower4 | 1600x1200 | 1.09 | 0.71 | 12.32 | 7.6 | 0.91 | 2.92 | 14.32 | 11.23 |
| Flower5 | 1600x1200 | 1.09 | 0.71 | 17.01 | 7.48 | 0.9 | 4.66 | 19 | 12.85 |
| Flower6 | 1600x1200 | 1.09 | 0.71 | 18.97 | 7.52 | 0.87 | 1.55 | 20.93 | 9.78 |
| Flower7 | 1600x1200 | 1.08 | 0.72 | 20.93 | 7.56 | 0.86 | 20.76 | 22.87 | 29.04 |
| Flower8 | 1600x1200 | 1.18 | 0.72 | 13.81 | 7.5 | 0.89 | 13.04 | 15.9 | 21.3 |
| Flower9 | 1600x1200 | 1.07 | 0.70 | 18.33 | 7.7 | 0.85 | 9.53 | 20.3 | 18 |
| Flower10 | 1600x1200 | 1.06 | 0.74 | 20.34 | 7.72 | 0.28 | 9.75 | 21.7 | 18.2 |
| Family1 | 1600x1200 | 1.09 | 0.7 | 10.73 | 8 | 6.79 | 7.2 | 18.61 | 15.9 |
| Family2 | 2048x1536 | 1.78 | 1.2 | 18.29 | 13 | 11.14 | 21.8 | 31.21 | 36 |
| Family3 | 1880x2816 | 3.78 | 2.02 | 33.08 | 22.14 | 7.74 | 74.8 | 44.6 | 98.9 |
| Family5 | 3008x2000 | 13.94 | 2.3 | 39.87 | 24.8 | 17.47 | 46.8 | 71.28 | 73.9 |
| Family6 | 2304x1728 | 2.27 | 1.5 | 23.04 | 16.44 | 11.45 | 98.3 | 36.76 | 117 |
| Animal4 | 3008x2000 | 3.16 | 2.28 | 38.8 | 2.28 | 3.44 | 56.89 | 45.4 | 61.5 |
| Animal5 | 4567x3104 | 31.9 | 5.31 | 154.4 | 5.31 | 97.8 | 98.8 | 284.1 | 109.42 |
| Animal6 | 3000x2278 | 3.6 | 2.6 | 43.4 | 2.6 | 7 | 31.5 | 54 | 37 |
| Animal7 | 2800x1800 | 2.8 | 1.9 | 33 | 1.9 | 33.4 | 19.89 | 69 | 24 |
| Animal8 | 3008x1960 | 3.19 | 2.23 | 37.5 | 2.23 | 52.8 | 39.94 | 93.5 | 44.4 |
| Animal9 | 3648x2736 | 14.3 | 3.8 | 64.55 | 3.8 | 211 | 83.45 | 289 | 92 |
| Animal10 | 1600x1200 | 1.03 | 0.73 | 11.4 | 0.73 | 7.36 | 3.14 | 19.8 | 4.6 |
| Animal11 | 2304x3072 | 3.69 | 2.68 | 43.34 | 2.68 | 39.82 | 137 | 86.85 | 142.5 |
| Animal12 | 1984x1488 | 1.55 | 1.13 | 18.12 | 1.13 | 9.4 | 8.4 | 29 | 10.7 |
| Animal13 | 2000x1370 | 1.45 | 1.04 | 16.81 | 1.04 | 8.19 | 15.76 | 26.45 | 17.84 |
| Animal15 | 2000x1332 | 1.41 | 1.01 | 16.11 | 1.01 | 16.7 | 11.56 | 34.22 | 13.6 |
| Games1 | 2774x2504 | 3.57 | 2.65 | 36.59 | 28.85 | 8.46 | 20.86 | 48.62 | 52.3 |
| Games3 | 3456x2304 | 4.12 | 4.58 | 45.7 | 50.86 | 20.83 | 13.88 | 70.65 | 69 |

**Table 4.2** Time in seconds for segmentation on various images. Large size images are collected randomly from web. It compares the time taken by our approach to $\alpha$-expansion for energy function calculation, graph construction and optimization steps.



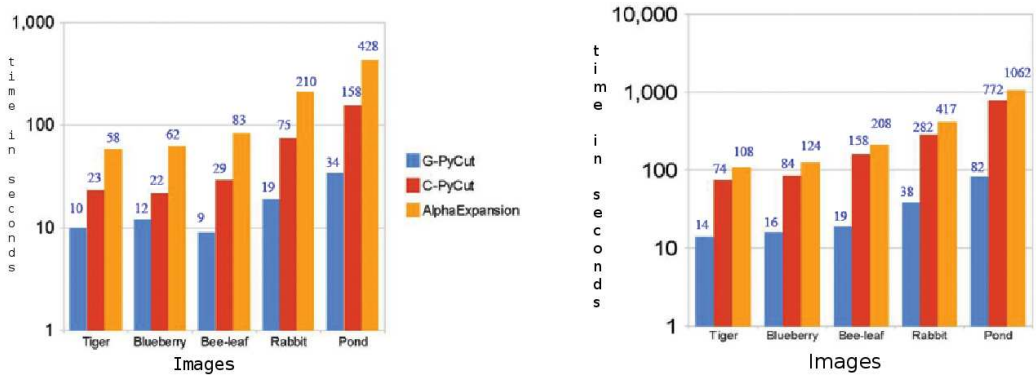

(a) (b)

**Figure 4.13** Running time in seconds for stereo using Pyramid Cuts on the GPU (G-PyCut), on the CPU (C-PyCut), and a single level graph cut (GCut) on images of full size, half size, and one third size. (a) The optimization time. (b) The total time, including energy calculation, graph construction, and optimization timings.

**Figure 4.14** Top row: Original images. Middle row: noisy images. Bottom row: Denoised versions using pyramid cuts on a few test images. Image labels, sizes, and the range of labels from left to right are: Bee_Leaf (640×600, 256), BlueBerry (440×432, 256), Pond (1600×1200, 256), Rabbit (800×800, 256), Tiger (400×396, 256).

| Image | Size | Labels | EnFun | | GraphConst | | OptTime | | TotalTime | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | Ours | $\alpha$-exp | Ours | $\alpha$-exp | Ours | $\alpha$-exp | Ours | $\alpha$-exp |
| Aloe | 1280x1104 | 224 | 28 | 179 | 420 | 396 | 97 | 317 | 545 | 892 |
| Baby1 | 1232x1104 | 256 | 29 | 192 | 443 | 434 | 109 | 754 | 581 | 1380 |
| Baby2 | 1232x1104 | 256 | 29 | 192 | 451 | 433 | 169 | 667 | 649 | 1292 |
| Baby3 | 1312x1104 | 256 | 31 | 207 | 491 | 461 | 116 | 706 | 638 | 1374 |
| Bowling1 | 1248x1104 | 240 | 28 | 225 | 417 | 513 | 341 | 1213 | 786 | 1951 |
| Bowling2 | 1312x1104 | 240 | 31 | 198 | 462 | 441 | 262 | 881 | 755 | 1520 |
| Cloth1 | 1248x1104 | 240 | 29 | 189 | 435 | 423 | 98 | 289 | 562 | 901 |
| Cloth2 | 1280x1104 | 208 | 24 | 171 | 372 | 378 | 104 | 536 | 500 | 1085 |
| Cloth4 | 1280x1104 | 256 | 30 | 209 | 485 | 469 | 106 | 420 | 621 | 1098 |
| Lampshade1 | 1280x1104 | 256 | 30 | 207 | 472 | 466 | 360 | 1212 | 862 | 1885 |
| Lampshade2 | 1280x1104 | 256 | 30 | 207 | 479 | 467 | 337 | 1233 | 846 | 1907 |
| Midd1 | 1376x1104 | 256 | 33 | 224 | 533 | 500 | 270 | 1830 | 836 | 2554 |
| Midd2 | 1344x1104 | 256 | 35 | 219 | 518 | 488 | 405 | 1684 | 958 | 2391 |
| Rocks 1 | 1248x1104 | 272 | 31 | 212 | 490 | 478 | 120 | 492 | 641 | 1182 |
| Rocks 2 | 1248x1104 | 272 | 28 | 212 | 542 | 478 | 125 | 656 | 695 | 1346 |
| Wood1 | 1344x1104 | 208 | 29 | 182 | 394 | 393 | 182 | 972 | 605 | 1547 |
| Wood2 | 1280x1104 | 256 | 31 | 203 | 479 | 454 | 429 | 1408 | 939 | 2065 |
| Art | 1376x1104 | 256 | 33 | 221 | 561 | 492 | 115 | 663 | 709 | 1376 |
| Books | 1376x1104 | 256 | 33 | 220 | 549 | 492 | 102 | 742 | 684 | 1454 |
| Computer | 1312x1104 | 256 | 31 | 210 | 513 | 468 | 155 | 701 | 699 | 1379 |
| Dolls | 1376x1104 | 256 | 33 | 221 | 549 | 490 | 144 | 538 | 726 | 1249 |
| Drumsticks | 1376x1104 | 256 | 33 | 222 | 588 | 493 | 160 | 617 | 781 | 1332 |
| Dwarves | 1376x1104 | 256 | 33 | 220 | 548 | 491 | 127 | 741 | 708 | 1452 |
| Laundry | 1312x1104 | 256 | 30 | 202 | 493 | 463 | 84 | 242 | 607 | 907 |
| Moebius | 1376x1104 | 196 | 26 | 174 | 366 | 377 | 286 | 627 | 678 | 1178 |
| Reindeer | 1312x1104 | 224 | 29 | 187 | 428 | 413 | 130 | 658 | 587 | 1288 |

**Table 4.3** Time in seconds for stereo correspondence on various standard images. Large size images are collected randomly from web. It compares the time taken by our approach to $\alpha$-expansion for energy function calculation, graph construction and optimization steps.




(a) (b)

**Figure 4.15** Running time in seconds for Pyramid Cuts on the GPU (G-PyCut), on the CPU (C-PyCut), and a single level graph cut (GCut). (a) The optimization time. (b) The total time, including energy calculation, graph construction, and optimization timings.

*Chapter 5*

# Conclusions

This dissertation addresses the issue of speeding up the MRF optimization defined over discrete variables using graph cuts. Last few decades have seen many MAP estimation problems in computer vision being mapped to energy minimization framework. The popularity has risen since the introduction of graph cuts based methods which provide accurate results at moderate times on certain class of energy functions. Unfortunately, however these methods have not been able to provide a real time performance which otherwise is required in many applications. Examples include robot navigations, surveillance, video processing etc. In this dissertation, we target towards speeding up the graph cuts algorithms.

The first part of this dissertation deals with speeding up the graph cuts and graph cuts based methods on hardware accelerators like GPUs with application to computer vision using CUDA. Preprocessing steps of building energy functions and graph constructions are also mapped onto the GPUs. We also target the problem of minimizing multiple similar functions on the GPU. We showed how a problem instance similar in structure can be solved efficiently on the GPU by reusing and recycling the flows from the solution of the previous MRF instance. In summary, we showed how different facilities provided by the GPU can be utilized to speed up the MRF optimization using graph cuts.

Advancement in camera culture and ever increasing demand from the entertainment industries have lead to the acquisition of images involving millions of pixels. The second part of the dissertation deals with the processing of high resolution images involving large number of pixels and large number of labels. We model our optimization problems on multiresolution framework and dynamic graph cuts which speed up the basic graph cuts by a factor without sacrificing the global optimality.

## 5.1 Our Contributions

The major contributions of this dissertation are listed below:

**Fast Graph Cuts on the GPUs** We proposed a fast implementation of the push-relabel algorithm for mincut/maxflow algorithm for graph-cuts using CUDA with applications to computer vision. The additional computation efficiency is achieved by exploiting the grid structure of the MRFs arising in vi-

sion problems. This makes the GPU an ideal platform for energy minimization problems. Our method adaptively manages the work load to improve the computation performance of the push-relabel algorithm by factors for different problems on the GPU. We also proposed methods to map dynamic graph cuts [35, 25] on the GPU. In addition, the edge weights computation and graph construction were also ported on the GPU, providing a factor of speedup. We also presented incremental $\alpha$-expansion, extending our basic and dynamic graph cuts implementation on the GPU to solve different multilabeling MRF optimization problems.

**Globally Optimal Multiresoltion MRFs** We presented fully dynamic algorithm based on multiresolution MRF and dynamic graph cuts to process on the high resolution images involving millions of pixels and large number of labels. We combined classical multiresolution processing with graph cuts optimization. Our Pyramid Reparameterization method exploited the optimal results at a resolution level of an image to initialize its graph at the next higher resolution level. Pyramid reparametrization involved an upsampling step to increase the resolution of graphs and a weight modification step to form a graph with the same mincut as the graph constructed at the higher resolution level. The mincut generated by pyramid cuts using multiple levels is identical to the mincut generated using graph cuts on the highest resolution image. We showed the results on image segmentation, stereo correspondence and image denoising problems on higher resolution images with large number of labels.

## 5.2 Directions for Future Work

We conclude this dissertation by providing some directions for future work. It was shown in the thesis how efficient implementation of the push-relabel algorithm can help in speeding up various computationally expensive MRF optimization problems arising in computer vision. Even then there is a clear need to extend it further to speed up the basic graph cuts to achieve realtime performance. One obvious way is to use global and gap heuristics along with the push and local-relabel steps in the push-relabel algorithm for large images to improve the computation efficiency. Boykov $et\ al.$ had proposed an augmenting path based method for graph cuts by starting two search trees from both $s$ and $t$ and reusing the trees from iteration to iteration. The development of an algorithm based on this augmenting path based method on the GPUs using the primitives like split, sort, segmented scan etc. is an important direction for future work.

Another promising and interesting direction is to use multiple GPUs for different MRF optimization problems involving large number of pixels and labels. The work can be adaptively divided and distributed over the these GPUs. Each GPU may process on a region of the image and a subset of the labels to improve the performance. This will require flawless merging of the results to get the final solution.

Higher order potentials and relations are generally used to model the rich statistics in the natural scenes. As the order increases, so the complexity of the problems increases too. GPU has been very successful in improving the performance of many complex problems. Further, our dynamic and hierarchical MRF framework can incorporate the higher order statistics efficiently. Thus, there is clear need

for extending the work to incorporate higher order potentials and non-submodular energy functions on the GPUs and on the multiresolution MRF framework.

Researchers have also proposed methods to reuse the solutions from previous cycles to the current cycle as in the dynamic $\alpha$-expansion [25], which uses graph cuts as intermediate step. One of the promising directions is to use our multiresolution graph cuts to further speed up the dynamic $\alpha$-expansion. We also showed how label space compression may be incorporated in our multiresolution framework for the stereo correspondence problem. Similar compression in label space may result in reducing the computation overhead for denoising problems too.

We conclude this dissertation by the following observations: Graph cut based methods have proved to be instrumental in solving various computationally challenging problems in computer vision. They provide an accurate and robust solutions at moderate times. Subsequently we have seen effort to improve the computation efficiency of the graph cuts algorithmically and on the parallel hardware accelerators. We have also seen many new problems in computer vision being solved using graph cuts based methods. These recent interests in the development of new graph cut based algorithms show that these methods will remain popular in computer vision for a long time.

# Related Publications

The following publications resulted from work in and related to this thesis

- P. J. Narayanan, Vibhav Vineet and Timo Stitch. Fast Graph Cuts on the GPU. *GPU Computing Gems (GCG), Volume 1 Dec. 2010* **(Book Chapter)**.
- Vibhav Vineet and P. J. Narayanan. Solving Multi-label MRFs using incremental alpha-expansion move on the GPUs. In Proceeding of *Ninth Asian Conference on Computer Vision. (ACCV-2009), China, 2009*.
- Vibhav Vineet and P. J. Narayanan. CUDA Cuts: Fast Graph Cuts on the GPU. In Proceeding of *CVPR workshop on Visual Computer Vision on GPUs (CVPR-2008), Alaska, USA, 2008*.
- Vibhav Vineet, Pawan Harish, Suryakant Patidar and P. J. Narayanan. Fast Minimum Spanning Tree for Large Graphs on the GPU. In Proceeding of *ACM SIGGRAPH Hight Performance Graphics (HPG-2009), New Orleans, LA, USA, 2009*.
- Pawan Harish, Vibhav Vineet and P. J. Narayanan. Large Graph Algorithms for Massively Multithreaded Architectures. *IIIT Tech Report, IIIT/TR/2009/74*.
- CUDA Cuts: Fast Graph Cuts on the GPU. *http://cvit.iiit.ac.in/index.php?page=resources*. (Software).
- Vibhav Vineet, Pawan Harish, Suryakant Patidar and P. J. Narayanan. Fast Minimum Spanning Tree for Large Graphs on the GPU. *GPU Computing Gems (GCG).* **(Under Submission)**.

# Bibliography


[1] A. Blake, C. Rother, M. Brown, P. Pérez and P. Torr. Interactive Image Segmentation Using an Adaptive GMMRF Model. In *European Conference on Computer Vision*, 2004.

[2] Adobe Photoshop. http://www.adobe.com/support/photoshop.

[3] A. Delong and Y. Boykov. A Scalable graph-cut algorithm for N-D grids. In *IEEE Conference on Computer Vision and Patter Recognition*, 2008.

[4] A. K. Sinop and L. Grady. Accurate Banded Graph Cut Segmentation of Thin Structures Using Laplacian Pyramids. In *International Conference on Medical Image Computing*, pages 896-903, 2006.

[5] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum-flow problem. *J. ACM*, 35(4):921–940, 1988.

[6] B. V. Cherkassky and A. V. Goldberg. On implementing push-relabel method for the maximum flow problem. In *IPCO*, 157-171, 1995.

[7] C. Liang, C. Cheng, Y. Lai, L. Chen and H. Chen. Hardware-Efficient Belief Propagation. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2009.

[8] C. Rother, V. Kolmogorov and A. Blake. "GrabCut": interactive foreground extraction using iterated graph cuts In *ACM Trans. Graph.* , 2004.

[9] D. M. Greig, B. T. Porteous A. H. Seheult. Exact Maximum A Posteriori Estimation for Binary Images. *Journal of the Royal Statistical Society Series B*, 51, 271279, 1989.

[10] D. Olsen and M. Harris. Edge-respecting brushes. In *ACM Trans. Graph.* , 205–213, 2008.

[11] D. S. Hochbaum. The pseudoflow algorithm for the maximum flow problem. Manuscript, UC Berkeley, revised 2003. Extended abstract in: The pseudoflow algorithm and the pseudoflow-based simplex for the maximum flow problem. In *Proceedings of IPCO98, June 1998. Lecture Notes in Computer Science, Bixby, Boyd and Rios-Mercado (Eds.) 1412, Springer,*, 325-337, 2003.

[12] E. Lombaert Mortensen and W. Barrett. Intelligent scissors for image composition. In *International Conference on Graphics and Interactive Techniques*, 1995.

[13] F. Alizadeh and A. Goldberg. Implementing the push-relabel method for the maximum flow problem on a connection machine. In *Technical Report STAN-CS-92-1410, Stanford University*, 1992.

[14] F. R. Schmidt, E. Töppe and D. Cremers. Efficient Planar Graph Cuts with Applications in Computer Vision. In *International Conference on Computer Vision and Pattern Recognition*, 2009.

[15] G. Borradaile and P. N. Klein. An $O(n \log n)$ algorithm for maximum $st$-flow in a directed planar graph. In *Proceedings of 17th ACM-SIAM Symposium on Discrete Algorithms*, 2006.

[16] H. Lombaert, Y. Sun, L. Grady and C. Xu. A multilevel banded graph cuts method for fast image segmentation. In *IEEE International Conference on Computer Vision*, pages 259-265, 2005.

[17] I. Kovtun. Partial Optimal Labeling Search for a NP-Hard Subclass of (max, +) Problems. In *DAGM-Symposium*, 2003.

[18] J. Besag. Spatial interaction and the statistical analysis of lattice systems. In *Journal of the Royal Statistical Society Series B*, 36:192–236, 1974.

[19] J. Chen, S. Paris and F. Durand. Real-time edge-aware image processing with the bilateral grid. In *ACM Trans. Graph.*, 2007.

[20] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. In *J. ACM*, 1972.

[21] J. Liu, J. Sun and H. Y. Shum. Paint selection. In *ACM Trans. Graph. 28*, 3, 1–7, 2009.

[22] J. M. Hammersley and P. Clifford. Markov field on finite graphs and lattices. *unpublished.*

[23] J. Wang, M. Agrawala and M. Cohen. Soft scissors: an interactive tool for realtime high quality matting. In *ACM Trans. Graph.*, 2007.

[24] J. Worby and W. J. Maclean. Establishing Visual Correspondence from Multi-Resolution Graph Cuts for Stereo-Motion. In *CRV*, 2007.

[25] K. Alahari, P. Kohli and P. H. Torr. Reduce, Reuse & Recycle: Efficiently Solving Multi-Label MRFs. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 16–29, 2008.

[26] Khronos Group Std. The OpenCL Specification, Version 1.0, Online, Khronos Group Std., Rev. 33, April 2009. Online. *http://www.khronos.org/registry/cl/specs/opencl-1.0.33.pdf*.

[27] L. R. Ford and D. R. Fulkerson. Flows in Networks. In *Princeton University Press*, 1962.

[28] L. Yin, S. Jian, T. Chi-Keung and S. Harry. Lazy snapping. In ACM Trans. Graph., 302–308, 2004.

[29] M. Hussein, A. Varshney and L. Davis. On implementing graph cuts on cuda. In *First Workshop on General Purpose Processing on Graphics Processing Units*. Northeastern University, October 2007.

[30] Middlebury MRF Page. *http://vision.middlebury.edu/MRF/*.

[31] N. Corporation. CUDA: Compute unified device architecture programming guide. *Technical report, Nvidia*, 2007.

[32] N. Dixit, R. Keriven and N. Paragios. GPU Cuts: Combinatorial optimisation, graphic processing units and adaptive object extraction. In *Technical report, CERTIS*, 2005.

[33] N. Komodakis, G. Tziritas and N. Paragios. Fast, Approximately Optimal Solutions for Single and Dynamic MRFs. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 1-8, 2007.

[34] O. Juan and Y. Boykov. Active Graph Cuts. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2006.

[35] P. Kohli and P. Torr. Dynamic graph cuts for efficient inference in markov random fields. In *IEEE Transaction on Pattern Analysis and Machine Intelligence*, 29(12):2079–2088, 2007.

[36] P. Kohli and P. Torr. Effciently Solving Dynamic Markov Random Fields Using Graph Cuts In *IEEE International Conference on Computer Vision*, 2005.

[37] R. J. Anderson and J. C. Setubal. On the parallel implementation of goldberg's maximum flow algorithm. In *SPAA*, pages 168–177, 1992.

[38] R. Szeliski, R. Zabih, D. Scharstein, O. Veksler, V. Kolmogorov, A. Agarwala, M. F. Tappen and C. Rother. A comparative study of energy minimization methods for markov random fields. In *European Conference on Computer Vision*, pages 16–29, 2006.

[39] R. Szeliski, R. Zabih, D. Scharstein, O. Veksler, V. Kolmogorov, A. Agarwala, M. F. Tappen and C. Rother. A Comparative Study of Energy Minimization Methods for Markov Random Fields with Smoothness-Based Priors. In *IEEE Trans. Pattern Anal. Mach. Intell*, 2008.

[40] S. Geman and D. Geman. Stochastic relaxation, Gibbs distributions and the Bayesian restoration of images. In *IEEE Trans. Pattern Anal. Mach. Intell.*, 1984.

[41] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein. Introduction to Algorithms. In *MIT Press*, 2001.

[42] T. Vodrey, D. Gruber, A. Wedel and J. Klappstein. Space-Time Multi-Resolution Banded Graph-Cut for Fast Segmentation. In *DAGM*, 2008.

[43] V. Kolmogorov. Convergent Tree-Reweighted Message Passing for Energy Minimization. In *IEEE Trans. Pattern Anal. Mach. Intell*, 2006.

[44] V. Kolmogorov and R. Zabih. What energy functions can be minimized via graph cuts? In *IEEE Trans. Pattern Anal. Mach. Intell.*, 26(2):147–159, 2004.

[45] W. T. Freeman and E. C. Pasztor. Learning Low-Level Vision In *IEEE International Conference on Computer Vision*, 1999.

[46] X. Bai and G. Sapiro. A Geodesic Framework for Fast Interactive Image and Video Segmentation and Matting. In *International Conference on Computer Vision*, 2007.

[47] Y. Boykov and M. Jolly. Interactive Graph Cuts for Optimal Boundary and Region Segmentation of Objects in N-D Images. In *International Conference on Computer Vision*, 2001.

[48] Y. Boykov and V. Kolmogorov. An Experimental Comparison of Min-cut/Max-flow Algorithms for Energy Minimization in Vision. In *Energy Minimization Methods in Computer Vision and Pattern Recognition*, 2001.

[49] Y. Boykov, O. Veksler and R. Zabih. Fast approximate energy minimization via graph cuts. In *IEEE Trans. Pattern Anal. Machine Intell*, 23(11):1222–1239, 2001.