Solving Multilabel MRFs using Incremental α-Expansion on the GPUs

Vibhav Vineet and P. J. Narayanan

Center for Visual Information Technology
International Institute of Information Technology
Hyderabad, India
{vibhavvinet@research., pjn@}iiit.ac.in

Abstract. Many vision problems map to the minimization of an energy function over a discrete MRF. Fast performance is needed if the energy minimization is one step in a control loop. In this paper, we present the incremental α-expansion algorithm for high-performance multilabel MRF optimization on the GPU. Our algorithm utilizes the grid structure of the MRFs for good parallelism on the GPU. We improve the basic push-relabel implementation of graph cuts using the atomic operations of the GPU and by processing blocks stochastically. We also reuse the flow using reparametrization of the graph from cycle to cycle and iteration to iteration for fast performance. We show results on various vision problems on standard datasets. Our approach takes 950 milliseconds on the GPU for stereo correspondence on Tsukuba image with 16 labels compared to 5.4 seconds on the CPU.

1 Introduction

Low-level vision problems like stereo correspondence, restoration, segmentation, etc., are usually modeled as a label assignment problem. A label from a given set is assigned to each pixel or block in an image. These problems are often modeled in a Markov random-field (MRF) framework. Geman and Geman [17] formulated the maximum a-posteriori (MAP) estimation of the MRF as an energy minimization problem. Minimization of these energy functions is computationally expensive. Several algorithms have been proposed to improve the computational performance [9, 4, 1, 2]. However, real time performance on regular images is still a challenge on the traditional hardware. Several vision applications like robot navigation, surveillance, etc., require the processing to be completed at close to video frame-rates.

Solving computationally expensive problems on parallel architectures is another way to improve the efficiency. However, not all algorithms are easily scalable to parallel hardware models. The contemporary graphics processing units (GPUs) are emerging as popular high performance platforms because of their huge processing power at reasonable costs. The Compute Unified Device Architecture (CUDA) [19] from Nvidia provides a general-purpose parallel programming interface to the modern GPUs. The emerging standard of OpenCL 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 this paper, we explore MRF optimization on the GPU. In particular, we propose a method to efficiently use the GPU and their atomic capability for high performance. We present the incremental α-expansion algorithm for multilabel MRFs. Our method retains the grid structure of the graph, common to low-level vision problems. We also propose a method to map dynamic energy minimization algorithms [4] to the GPU architecture. We recycle and reuse the flow from the previous MRF instances. A novel framework is also proposed based on the observation that the most of the variables in the MRF get the final labels quickly. Reuse of the flow 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 achieve a speedup of 5-6 times on different multi-labeling problems on standard datasets.

We tested our algorithm on different problems such as stereo correspondence, image restoration, and photomontage. Stereo correspondence results are shown on Tsukuba, Venus and Teddy images. Image restoration results are shown on Penguin and House images and photomontage on Panorama and Family images. All the datasets are taken from the Middlebury MRF page [14]. 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 with 16 labels compared to 5.4 seconds on the CPU.

1.1 Literature Review

Some of the key algorithms which are used to minimize energy functions defined over MRF include α-expansion and $\alpha\beta$-swap [3], max-product loopy belief propagation [20] and tree-reweighted message passing [12]. α-expansion involves constructing a graph over which maxflow/mincut algorithms are applied repeatedly. The Ford-Fulkerson's algorithm [21] to solve maxflow/mincut problem is popular and several fast implementations are available today [3, 10]. Push-relabel method [7] is more parallelizable and was implemented on the Connection Machines by Goldberg *et al.* [5].

Recently efforts have been made to solve the optimization algorithms on the parallel architecture. Push-relabel algorithms have been implemented on the GPU recenly [13, 8]. They demonstrate solution of bilabel problems on the GPU. Liang *et al.* [15] designed a new parallel hardware to solve belief propagation algorithm efficiently. Delong and Boykov [18] 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. Schmidt *et al.* [23] present an efficient graph cuts algorithm for planar graphs motivated by the work of Borradaile and Klein [22].

2 MRF Energy Minimization on the GPUs

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 function is:

$$E(f) = \sum D_p(f_p) + \sum V_{p,q}(f_p, f_q)$$

where $D_p(f_p)$, the data term, measures the cost of assigning a label $f_p \in L$ to pixel $p \in P$ and $V_{(p,q)}(f_p, f_q)$, the smoothness term, measures the cost of assigning the labels f_p and f_q to the adjacent pixels p and q.

The MRF is modeled as a graph with a grid structure with fixed connectivity. We use the α-expansion algorithm to minimize the energy, which is posed as a series of two-label graph cuts. We use a flagged graph cuts using the push-relabel algorithm for these. 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.

2.1 α-Expansion Algorithm

The α-expansion [3] is a popular move-making energy minimization algorithm (Algorithm 1). Steps 2 to 4 form a cycle and the step 3 is an iteration within a cycle. The algorithm starts from an intial labelling and makes a series of moves, which involve label change of the random variables, until there is no decrease in the energy. After each iteration of α-expansion, the random variable in the MRF retains either its current label or takes a new label α. One cycle of α-expansion algorithm involves iterating over all the labels.

Algorithm 1 α-EXPANSION

```
1: Intialize the MRF with an arbitrary labelling $f$.
2: for each label $\alpha \in L$ do
3:     Find $f^{'}$=arg min $E(f^{'})$ among $f'$ within one $\alpha$-expansion of $f$, current labelling
4: end for
5: if $E(f^{'}) < E(f)$ then
6:     goto step 2
7: end if
8: return $f$
```

Given a current labelling f, there are exponential number of moves possible in Step 3. Graph cuts algorithm efficiently computes next configuration $f^{'}$ in polynomial time. It involves constructing a graph based on the current labeling and the label α. Vertices with label α do not take part in the iteration but all others attempt to relabel themselves with α. There are two ways to construct the graph involved. Kolmogorov *et al.* [10] construct the graph without any auxiliary vertices, while Boykov *et al.* [3] introduce auxiliary vertices. The α-expansion method involves constructing graph in each iteration. This is a time consuming step. It takes 1.2 seconds on the Tsukuba image for stereo correspondence problem with 16 labels on the CPU.

2.2 Flagged Graph Cuts on the GPU

The push-relabel algorithm finds the maximum flow in a directed graph. Each vertex u in the graph has two quantities associated with it: height value $h(u)$ and excess flow $e(u)$. The algorithm involves two basic operations: *Push* and *Relabel*. It has been implemented on the ealier version of CUDA, with and without the atomic capability [13, 8]. We extend prior work to handle α-expansion [13].

Push Operation: 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 2 explains the implementation of the push operation on the GPU.

Algorithm 2 KERNEL2_PUSH

```
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 neighbors 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
```

Relabel Operation: Local relabel operation is applied at a vertex u if it has positive excess flow but no push is possible to any neighbor due to height mismatch. The height of u is in-creased 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 3 explaines the implementation of local relabel performed on GPUs.

Algorithm 3 KERNEL3_RELABEL

```
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
```

Push-relabel algorithm finds the maxflow/mincut on an edge-capacitated graph. Graph construction is central to any energy minimization method based on the graph cuts. Our graph-construction exploits the grid-structures of the MRFs defined over images. We adapt the graph construction of Kolmogorov *et al.* [10], which maintains the grid structure. α-expansion repeatedly finds the mincut based on the current graph, the current labelling of the pixels, and the label α. Thus the structure of the graph changes for each iteration using α. We use a flagged graph-cuts method, which helps in retaining the grid structure of the graph. We keep a flag bit with each vertex in the graph, which is set if the vertex is part of the current graph. Each vertex participates in the computationonly only if the flag is set (Step 3 of Algorithm 2 and 3). This way, we maitain the grid structure and restrict the overall computation. The graph is constructed based on the energy functions which can change over iterations for some problems. To reduce the overall computation time, the evaluation of the energy functions and the graph construction are performed on the GPU. Our energy function and graph construction are same as in [14]. Table **??** compares the times for bilabel segmentation on some standard datasets on the GPU and the CPU.

Image	Graph Cuts		Graph Construction		Total	
	GPU	CPU [11]	GPU	CPU [11]	GPU	CPU [11]
Flower	37	188	0.15	60	37.15	248
Sponge	44	142	0.15	61	44.15	203
Person	61	140	0.15	60	61.15	201

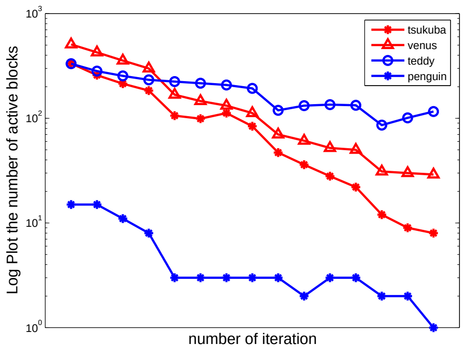
Table 1. Comparison of running times(in milliseconds) for one graph cut on GTX 280 with that of Boykov on the CPU on different images.

2.3 Stochastic Cut

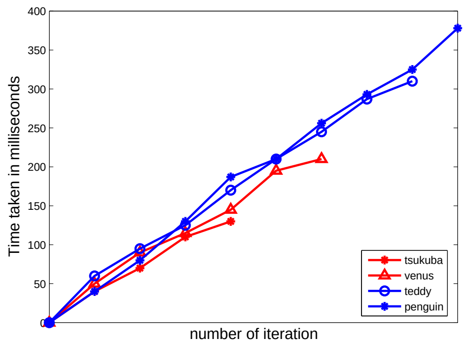
Most of the pixels get their final label after a few iterations on different datasets. This relates to the fact that the MRF constitutes both simple and difficult variables [16]. The simple variables settle and get their final label within few iterations of the graph cuts algorithm. Only few vertices exchange flows with their neighbours later. Processing vertices which are unlikely to exchange any flow with their neighbours results in inefficient utilization of the resources.

We determine blocks of vertices that are active at different stages of the graph cuts algorithm. An active block has at-least one pixel that has exchanged flow in the previous step. The activity is determined based on the change in the edge-weights and is marked for each block. Based on the active bit, the kernel executes the other parts of the algorithm. Figure 1(a) shows the number of active blocks as the computation progresses. This behaviour of the MRF is data dependent. In the case of the Tsukuba image for stereo correspondence problem, we see that the number of active blocks decreases significantly within a few iterations of the graph cuts. However, in the case of the Penguin image for restoration problem, the number of active blocks remains almost same throughout the computation. We delay the processing of a block based on its activity bit by a fixed amount. A block is processed in every iteration if it is active, otherwise the block is processed only after 10 iterations. This delaying has no effect on the final convergence as Goldberg *et al.* [7] proves that the convergence of the push-relabel algorithm is independent of the order of processing of the push and relabel operation for each vertex. However, to maintain the preflow condition at all time, we can apply the above approach only to the push operation. Algorithm 4 explains the working of the push-relabel algorithm with stochastic cuts on the GPUs. Figure 1(b) shows the time taken by each iteration averaged over all cycles.

```
Algorithm 4 KERNEL4_STOCHASTIC
Input: A residual graph, $G^r(V, E)$.
1: Check the active bit of the block.
2: Perform Flagged Graph Cuts every iteration on all the above blocks and every 10th
   iteration on the inactive blocks.
```



(a) active blocks vs. iteration graph



(b) average time vs. iteration graph

Fig. 1. (a) Number of active blocks vs. iteration number and (b) average time vs. iteration number on different datasets.

2.4 Increamental α-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. Reusing the flow has been the method to initialize better. Kohli and Torr [4] describe a reparameterization of the graph to initialize it for later frames in dynamic graph cuts. Komodakis *et al.* [2] extends this concept to multilabeling problems. Alahari *et al.* [1] give a simpler model for the same using dynamic α-expansion.

We adapt these methods to get an incremental α-expansion algorithm. We make three modifications to speed the overall process.

– First, we adapt the re-parameterization 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 α-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 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 in Step 3 of Algorithm 1, leading to faster convergence. Cycle-to-cycle reuse of flow typically results in a speed up of 3 to 4 times in practice.

– Third innovation 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 2 and Algorithm 5 explain our approach for the incremental α-expansion. The incremental α-expansion algorithm needs to store L graphs G^j, 1≤j≤L, one

Algorithm 5 KERNEL5_INCREMENTAL

```
Input: A residual graph, $G^r(V,E)$.
 1: Initialize the graph
 2: for the first cycle:
 3:     Construct graph $\mathrm{G}_1^1$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 $\mathrm{G}_1^l$ 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 $\mathrm{G}_i^j$
14:     Reparameterize based on $\mathrm{G}_i^j$ and $\mathrm{G}_{(i-1)}^j$
15:     Perform $l$-expansion for label $l$ using flagged graph cuts
```

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.

3 Experimental Results

We conducted experiments on a single Nvidia GTX 280 graphics adapter with 1024MB memory on-board (30 multiprocessors, 240 steam processors) connected to an Quad Core Intel processor (Q6600 @ 2:4GHz) with 4GB RAM running Fedora Core 9. We tested our 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.

$$\begin{tikzcd}
\mathrm{C}_1: & \mathrm{G}_1^1 \arrow[r] \arrow[d] & \mathrm{G}_1^2 \arrow[r] \arrow[d] & {} \arrow[r, dashed, no head] & \mathrm{G}_1^l \arrow[d] \\
\mathrm{C}_2: & \mathrm{G}_2^1 \arrow[d] & \mathrm{G}_2^2 \arrow[d] & {} \arrow[r, dashed, no head] & \mathrm{G}_2^l \arrow[d] \\
 & {} \arrow[d, dashed, no head] & {} \arrow[d, dashed, no head] & & {} \arrow[d, dashed, no head] \\
\mathrm{C}_k: & \mathrm{G}_k^1 & \mathrm{G}_k^2 & {} \arrow[r, dashed, no head] & \mathrm{G}_k^l
\end{tikzcd}$$

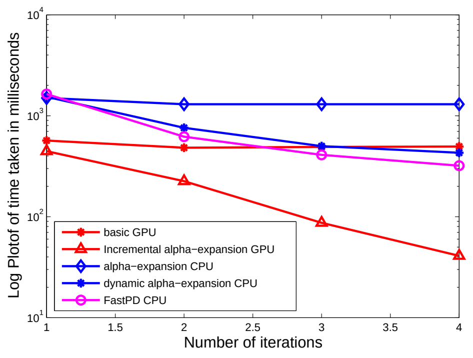
Fig. 2. Incremental α-expansion. Arrows indicate reparameterization based on the differences in graph constructed. G_i^j is the graph for iteration j of cycle i.

Image restoration results are shown on Penguin image and photomontage on the Panorama image. All the datasets are taken from the Middlebury MRF page [14]. The energy functions used are the same as used by them.

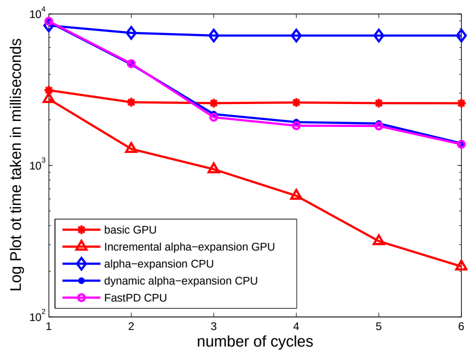
Figures 3(a), 3(b) shows the results of our approach on Tsukuba, Teddy images respectively for stereo correspondence. The results of restoration problem on Penguin image is shown in Figure 3(c) and of photomontage problem on Panorama image in Figure 3(d). Timings are shown on Middlebury code on the CPU, Fast-PD and dynamic α-expansion on the CPU, our basic implementation without flow reuse, and the complete incremental α-expansion. Our incremental α-expansion on the GPUs is 5-8 times faster than the α-expansion on the CPU using Middlebury code [14]. Impact of flow-reuse can also be seen from the graphs in Figure 4. Stereo correspondence on Tsukuba image with 16 labels takes 772 milliseconds on the GPU compared to 5.4 seconds on the CPU. Dynamic α-expansion [1] and Fast-PD [2] takes 3.23 seconds for the same. Figure 3(e) compares the total times for convergence for different levels of optimization discussed above on various datasets. Recently, Liang *et al.* [15] proposed belief propagation based optimization algorithm on the GPU to solve the energy minimization problems, which achieved 4 times speedup compared to the sequential algorithms. Our proposed algorithm achieves better performance than them.

4 Conclusion

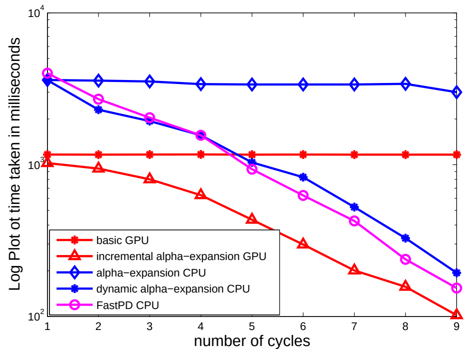
In this paper, we presented the incremental α-expansion algorithm for high-performance multilabel MRF optimization on GPU. We efficiently utilize the resources available on the current GPUs. We are able to get a speedup of 5-8 times on standard datasets on various problems. Our system brings a near-real time processing of MRF to the reach of most users as the GPUs are now very popular.



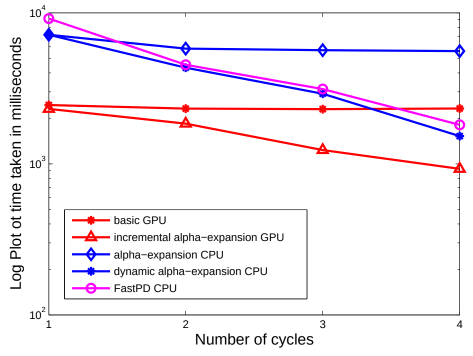
(a) Stereo: Tsukuba Image with 16 labels



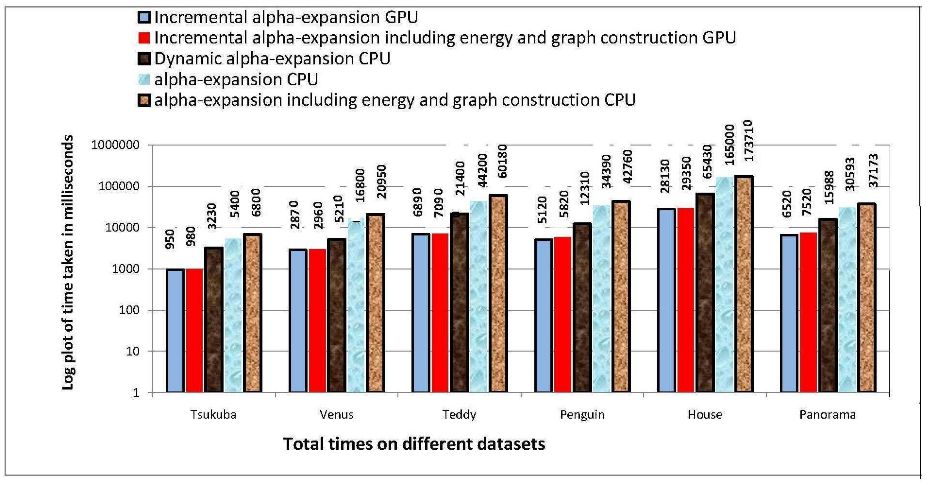
(b) Stereo: Teddy Image with 60 labels



(c) Restoration: Penguin Image with 256 labels



(d) Photomontage: Panorama Image with 7 labels



(e) Total times on different datasets

Fig. 3. Timings on different datasets from Middlebury MRF page [14]. (a)-(d) include only α-expansion timings.

References

1. K. Alahari, P. Kohli, and P. H. Torr. Reduce, Reuse & Recycle: Efficiently Solving Multi-Label MRFs In *CVPR (2)*, pages 16–29, 2008.
2. N. Komodakis, G. Tziritas and N. Paragios. Fast, Approximately Optimal Solutions for Single and Dynamic MRFs In *CVPR (2)*, pages 1-8, 2007.
3. Y. Boykov, O. Veksler, and R. Zabih. Fast approximate energy minimization via graph cuts. *TPAMI*, 23(11):1222–1239, 2001.
4. P. Kohli and P. H. S. Torr. Dynamic graph cuts for efficient inference in markov random fields. *TPAMI*, 29(12):2079–2088, 2007.
5. F. Alizadeh and A. Goldberg. Implementing the push-relabel method for the maximum flow problem on a connection machine. Technical Report STAN-CS-92-1410, Stanford University, 1992.
6. R. J. Anderson and J. C. Setubal. On the parallel implementation of goldberg's maximum flow algorithm. In *SPAA*, pages 168–177, 1992.
7. A. V. Goldberg and R. E. Tarjan. A new approach to the maximum-flow problem. *J. ACM*, 35(4):921–940, 1988.
8. 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.
9. O. Juan and Y. Boykov. Active graph cuts. In *CVPR (1)*, pages 1023–1029, 2006.
10. V. Kolmogorov and R. Zabih. What energy functions can be minimized via graph cuts? *TPAMI*, 26(2):147–159, 2004.
11. 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 *ECCV (2)*, pages 16–29, 2006.
12. Vladimir Kolmogorov. Convergent Tree-Reweighted Message Passing for Energy Minimization In *TPAMI*, pages 1568-1583, 2006.
13. V. Vineet and P. J. Narayanan. CUDA cuts: Fast graph cuts on the GPU In *IEEE CVPR workshop on Visual Computer Vision on GPUs*, pages 1-8, 2008.
14. Middlebury MRF Page. http://vision.middlebury.edu/MRF/
15. C. Liang, C. Cheng, Y. Lai, L. Chen and H. Chen. Hardware-Efficient Belief Propagation In *CVPR*, 2009.
16. Ivan Kovtun. Partial Optimal Labeling Search for a NP-Hard Subclass of (max, +) Problems. In *DAGM-Symposium*, 2003.
17. S. Geman and D. Geman. Stochastic relaxation, Gibbs distributions and the Bayesian restoration of images In *TPAMI*, 1984.
18. A. Delong and Y. Boykov. A Scalable graph-cut algorithm for N-D grids. In *CVPR*, 2009.
19. N. Corporation. CUDA: Compute unified device architecture programming guide. Technical report, Nvidia, 2007.
20. William T. Freeman and Egon C. Pasztor Learning Low-Level Vision In *Computer Vision, IEEE International Conference on*, 1999.
21. L.R. Ford and D.R. Fulkerson. Flows in Networks. In *Princeton University Press*, 1962.
22. Glencora Borradaile and Philip N. Klein. An $O(n \log n)$ algorithm for maximum st-flow in a directed planar graph In *Proceedings, 17th ACM-SIAM Symposium on Discrete Algorithms*, 2006.
23. F. R. Schmidt and E. Töppe and D. Cremers. Efficient Planar Graph Cuts with Applications in Computer Vision In *CVPR*, 2009.