Efficient GPU-based Graph Cuts for Stereo Matching

Young-kyu Choi
Computer Science Department
University of California, Los Angeles
ykchoi@cs.ucla.edu

In Kyu Park
School of Information and Communication Engineering
Inha University
pik@inha.ac.kr

Abstract

Although graph cuts (GC) is popularly used in many computer vision problems, slow execution time due to its high complexity hinders wide usage. Manycore solution using Graphics Processing Unit (GPU) may solve this problem. However, conventional GC implementation does not fully exploit GPU's computing power. To address this issue, a new GC algorithm which is suitable for GPU environment is presented in this paper. First, we present a novel graph construction method that accelerates the convergence speed of GC. Next, a repetitive block-based push and relabel method is used to increase the data transfer efficiency. Finally, we propose a low-overhead global relabeling algorithm to increase the GPU occupancy ratio. The experiments on Middlebury stereo dataset shows that 5.2X speedup can be achieved over the baseline implementation, with identical GPU platform and parameters.

1. Introduction

Graph cuts (GC) has been successfully applied to many computer vision and pattern recognition tasks. Examples include stereo matching, segmentation, 3D reconstruction, and object recognition. In contrast to local decision-making algorithms, GC performs global optimization on the graph models, integrating both observation and the prior knowledge. This allows GC, along with belief propagation, to be one of the most accurate algorithms in solving many vision problems [13].

In spite of its high accuracy, one of the main obstacle in using GC is its high complexity. Even with high-end computers, it may take several seconds to process a single mid-sized frame. In order to increase the execution speed of such complex problem, it has become popular to use massively parallel solution [12] [14] on NVIDIA's general-purpose computing on graphics processing unit (GPGPU) platform called Compute Unified Device Architecture (CUDA). However, conventional GC implementation on GPGPU still suffers from problems such as memory bottleneck, low utilization ratio, and slow convergence speed. As a result, fully exploiting the computational power of GPGPU remains to be a challenging task.

In this paper, we propose a novel GC algorithm that is suitable for GPGPU framework. First, we present a new graph construction method that can be easily applied to accelerate the convergence speed of GC. A reordering heuristic and initialization method is employed to further improve the execution speed based on the proposed graph construction method. Next, we propose block-based push and relabel method to reduce the memory bottleneck. A repetitive scheme is used to increase the data transfer efficiency. Finally, we present a low-overhead global relabeling algorithm that increases the GPU occupancy by reducing the kernel launch overhead.

The effectiveness of the proposed algorithm is demonstrated using the stereo matching problem, since it is one of the most fundamental problems in computer vision. Note that the presented algorithm can also be applied to any other GC-related problems as well.

This paper is organized as follows. The background knowledge is explained in Section 2. Section 3 introduces our efficient graph construction method. In Section 4, block-based repetitive push and relabel is presented. Low-overhead global relabeling is explained in Section 4. The experimental result is shown in Section 6, and we conclude in Section 7.

2. Background

2.1. Graph Cuts: Basics

Graph cuts algorithm can solve the minimum energy cut problem by finding the maximum flow in a graph. Let $\mathbf{G} = (\mathbf{N}, \mathbf{E})$ be a directed graph with $\mathbf{N}$ node set and $\mathbf{E}$ edge set. Two special nodes exist: source, which emits flow into the graph, and sink, which accepts flow from the graph. Each edge between node p and q has a flow value $f(p, q)$, and it must be less than the edge capacity $c(p, q)$. Note that the flow has anti-symmetry property (*i.e.* $f(p, q) = -f(q, p)$). The net flow $\sum_q f(p, q)$ into a node p is called excess $e(p)$, and a node with excess value larger than zero is called an active node.

The objective of the algorithm is to send as much flow as possible from the source to the sink. This process of sending excess to another node is called *pushing*. In order to push flow to the correct direction, height $h(p)$ is introduced to obtain the distance from node p to the sink. The excess of a node may be pushed to other nodes only if the height of the sender node is larger than that of the receiving node. If pushing is no longer possible for an active node, the height of the node is promoted to a higher value in the *relabel* stage. Relabeling, however, often makes inefficient local decision; thus, *global relabeling* is periodically performed to assign correct height information [8]. Push, relabel, and global relabel are iteratively performed until no active node exists.

Note that the process explained above is based on the push-relabel method [8]. Initially, this method tries to send as much flow as possible, by saturating all edges from the source ($f(source, p) := c(source, p)$). Next, the excessive flows are pushed to the sink until all possible paths to the sink have been saturated. Then, the remaining flows are pushed back to the source and the algorithm terminates. In this paper, push-relabel approach is taken, because it is more parallelizable than those based on augmenting path (*e.g.* [5]).

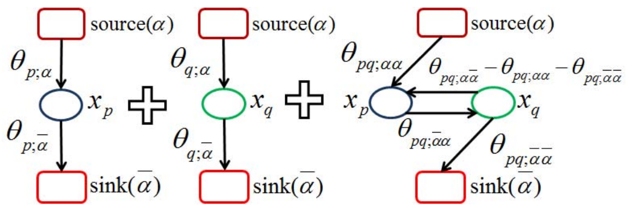
2.2. Stereo Matching Using Graph Cuts

Stereo matching is a process of comparing two images to find the distance to the objects. Closer object results in larger disparity in the image pair. Dense stereo matching algorithms measure this disparity by finding the correspondence for every pixel.

Graph cuts algorithm can be used to solve the stereo matching problem as follows, similar to [6]. The sink and the source are assigned to disparity labels, and the nodes are assigned to pixels. The capacity of the edges between sink/source to the nodes is assigned the likelihood of matching a disparity label to each pixel. The capacity of the edges between the pixel nodes reflects the penalty of violating smoothness among neighboring pixels. With such formulation, solving minimum cuts problem with GC becomes equivalent to partitioning pixels into labels that has the minimum energy. The labeling with minimum energy is then considered as the solution to the stereo matching problem.

For graph construction, Kolmogorov's method [11] is used as a baseline. In contrast to Boykov's method [6], Kolmogorov's method does not introduce any auxiliary nodes. Thus, a grid-like graph structure is maintained, which allows GPUs to make regular memory access. The graph formulation is shown in Figure 1.

Note that GC can only solve bi-label optimization problems. For multi-label problems, such as stereo matching, α-expansion heuristic is popularly used [6]. α-expansion breaks down multi-label energy optimization problem into a series of bi-label optimization problem. Each bi-label problem is solved by GC. For each GC, the current label α is assigned to a source node, and the previous label $\overline{\alpha}$ is assigned to the sink node. This labeling method is shown in Figure 1. After performing GC on label α, a pixel may retain its previous label $\overline{\alpha}$, or switch to label α if lower energy configuration can be obtained. Performing bi-label GC over all available labels are called a cycle. After GC is repeatedly performed for several cycles, the energy of the labeling decreases and converges to a minimum value.



source(α)
source(α)
source(α)
$\theta_{p;\alpha}$
$\theta_{q;\alpha}$
$\theta_{pq;\alpha\alpha}$
$\theta_{pq;\alpha\overline{\alpha}} - \theta_{pq;\alpha\alpha} - \theta_{pq;\overline{\alpha}\overline{\alpha}}$
x_p
x_q
x_p
x_q
$\theta_{p;\overline{\alpha}}$
$\theta_{q;\overline{\alpha}}$
$\theta_{pq;\overline{\alpha}\alpha}$
$\theta_{pq;\overline{\alpha}\overline{\alpha}}$
sink($\overline{\alpha}$)
sink($\overline{\alpha}$)
sink($\overline{\alpha}$)

Figure 1. Kolmogorov's graph construction method

2.3. CUDA Architecture

NVIDIA's CUDA [1] is briefly explained in this section. CUDA consists of an array of multithreaded Streaming Multiprocessors (SMs). Each SM has 32 processor cores and 48 KB of shared memory. The GTX 580 processor used in this paper has 16 SMs (512 cores) and 1.5 GB of global memory. Each SM can concurrently execute 32 threads, called a warp. Higher performance can be obtained if the threads in a warp execute the same instruction.

For computation, each SM needs to transfer data from the global memory. However, the latency of accessing the global memory typically takes hundreds of cycles. Thus, it is important to reduce the access to the global memory. Commonly used strategy is to store frequently used data into the shared or local memory, which takes much less time to access.

In order to activate GPU, host CPU first copies data from main memory to the GPU global memory. Next, host CPU commits the GPU program, called a kernel, into the GPU work queue. Note that there is several microseconds delay between the CPU kernel commit and the GPU kernel execution.

3. Graph Construction for Fast Convergence

3.1. Related Works

As α-expansion heuristic progresses, almost identical cut is repeatedly obtained for many different labels and cycles. Thus, some acceleration techniques were proposed to

Cycle	Execution Time (sec)	
	Kolmogorov's	Proposed
1	1.60	1.54
2	0.92	0.63
3	0.88	0.56
4	0.86	0.54
Total	4.28	3.28

Table 1. Acceleration of later cycles using the proposed graph construction method for Teddy image

reuse the solution obtained from the previous GC cycle. The graph reparameterization technique reuses the flow [2] [10], whereas the active graph cuts method reuses the cuts [9].

Compared to these algorithms, the merit of the proposed method lies in its simplicity. Although it does not achieve the 2-6 speedup reported in these literatures, it also does not require dynamic cut/capacity modification features which involves extensive modification to the original push-relabel algorithm. It also does not suffer from the memory overhead of storing previous flows. As will be explained in the remainder of this section, only a simple modification to the graph construction or initialization is required to obtain a reasonable speedup of 2.37X.

3.2. Proposed Method

The first step is simple: modify Kolmogorov's formulation in Figure 1 by reversing the α label and the previous label ($\overline{\alpha}$). The experiment shows that such simple measure brings an average performance improvement of 1.24X. Table 1, which contains the execution comparison for Teddy image, shows that most of the speedup is obtained from later (2-4) cycles of α-expansion. The reason for this speedup can be found in the intrinsic characteristic of push-relabel algorithm when combined with α-expansion. As α-expansion progresses, most of the edges with smaller capacity tends to gather closer to the source ($\overline{\alpha}$) with the proposed graph construction method. Larger capacity is inclined to be assigned to the sink-connected edges. Although push-relabel GC starts by saturating the capacity of all source-connected edges, most of these flows from the source can be sent directly out to the sink in the first few iterations, because the edges connected to the source would have larger capacity than the edges connected to the sink. In the original Kolmogorov formulation, on the other hand, most of the nodes still have excess after first iteration, and it takes several additional iterations to find a path to push them out to the sink.

Figure 2 shows the comparison of the decrease rate of active nodes on Tsukuba test image. It confirms that most of the nodes become inactive in first few iterations with the proposed graph construction method.

These findings suggest that the proposed scheme allows

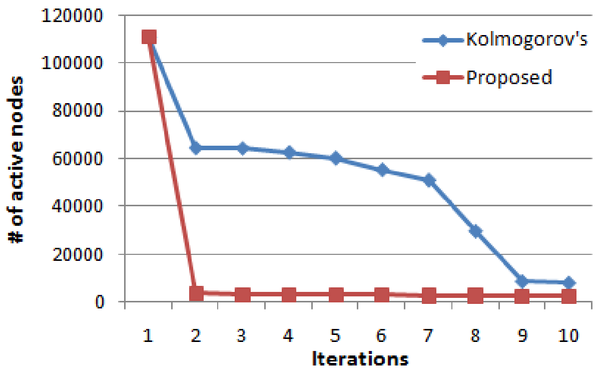


Figure 2. Comparison of decrease rate of active nodes (4th cycle, 16th label expansion of Tsukuba image)

execution time to decrease more rapidly, if $\overline{\alpha}$ label converges to the minimum energy label as soon as possible.

3.3. Initialization

In the previous subsection, it was shown that less time would be taken to execute a single cycle of GC if $\overline{\alpha}$ label has lower energy label assignment. This suggests that employing accurate initial label assignment will reduce the overall execution time with the proposed scheme. This is one of the major advantages of the proposed method.

Thus as a second step, we modified the initial label assignment of all zeros to local winner-take-all (WTA) method with 9$\times$9 window aggregation [13]. Although WTA with 9$\times$9 windows takes longer initialization time, experiment shows that the overall execution time is reduced by 1.68X. The reason is that GC converges faster with accurate initial labeling with the proposed graph construction method.

3.4. α-Expansion Reordering

For multi-label problems such as stereo matching, the execution speed can be further accelerated by changing the order of α-expansion. Performing GC on a label that allows faster convergence to a smaller energy labeling will decrease the execution time.

We propose a heuristic of choosing the label in the order of most frequent occurrence. The reason is that it will allow more pixels to switch to that label in the early stage of α-expansion. This is shown in Figure 3. Experiment shows that the execution speed of the first cycle is improved by 1.40X (overall speedup: 1.14X). Adopting better heuristic could further decrease the execution time. Note that the accuracy of the final disparity map may slightly change as a result, since the result of α-expansion changes depending on the order of label expansion.

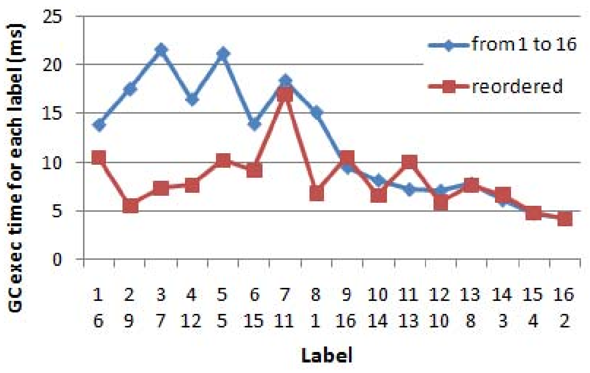


Figure 3. Execution time comparison between reordered label expansion (most frequent first) and sequential label expansion (from 1 to 16) (1st cycle of Tsukuba image)

4. Repetitive Block-Based Push & Relabel

4.1. Motivation

We use a repetitive block-based push & relabel method to reduce the memory bottleneck problem. Accessing the global memory takes a very long time in CUDA architecture. Accessing the shared memory, on the other hand, takes much less time. Thus, it is advantageous to move data into the shared memory, if the data is accessed frequently. A block size of 32×32 pixels has been chosen, and 512 threads are invoked to process a block.

In conventional parallel implementation of push-relabel algorithm, push or relabel is performed only once per global memory data fetch, before proceeding to the next iteration of push or relabel. As a result, flow and capacity data are accessed only a small number of times before getting evicted out of the shared memory. This leads to frequent global memory access and causes memory bottleneck problem.

4.2. Proposed Method

To increase the number of computation per data transfer, a repetitive block processing technique is employed. After performing a push operation, flow data is *not* written back to the global memory for subsequent relabel operation. Instead, relabeling is performed based only on the flow information inside the block. Next, push is performed with the relabeled height information. This process is repeated for some iterations, updating only the information inside a block. The excess flow toward neighboring blocks after each iteration is not directly written to global memory; they are temporary stacked on the block boundary. After the block processing is over, these outward flows are synchronously updated to the neighboring blocks.

Note that the flow and the height information from outside the block no longer needs to be fetched from the global memory on each iteration, because they are assumed to remain constant. One side effect is that excess may flow to a wrong direction due to the limited scope of local relabeling. However, the advantage of reducing the data transaction between the shared memory and the global memory surpasses such a disadvantage.

A strategy similar to our proposed method can be found in [7]. They suggest a block discharge scheme, which iterates until all excess is completely pushed out of a block. The problem with this method is that intra-block global relabeling is frequently performed, which is a very time-consuming operation. Moreover, the height information obtained from intra-block global relabeling is not accurate enough, compared to the global relabeling on the entire graph.

In the proposed scheme, on the other hand, a limitation is set on the number of iterations that an intra-block push and relabeling may perform. When this limitation has been reached, the content within a block is flushed to the global memory, and global relabeling is triggered. The reasoning behind this strategy is that relabeling with only information inside a block becomes gradually inaccurate after several iterations, and eventually, global relabeling over the entire graph becomes more efficient.

Experiment shows that our scheme over-performs the block discharge strategy [7] by 1.60X, and the conventional strategy with single iteration of push / relabel per global memory fetch by 1.91X.

Note that 'wave' push scheme [12] [14] is employed in this work. If no constraint is given, threads may compete with other threads when updating the flow and excess value. The wave push scheme avoids this conflict by pushing flows in only one direction. When all threads complete, they are synchronized and proceed to the next direction.

5. Low-Overhead Global Relabeling

Global relabeling may adopt a similar strategy as the push & relabel processing technique explained in the previous section. But for global relabeling, the performance can be further improved by increasing the GPU occupancy ratio. To do so, we explain a new way to launch kernels in this section. We define new terminology: the group of pixels that has been fetched into the shared memory of SM is called a *pixel block*, and SM's threads that process the pixels are called a *thread block*.

5.1. Conventional Method

The simplest way to launch thread blocks is to execute every block. This strategy may work in the early stage of iteration when most of the blocks are active. However, when blocks are sparsely active, this method suffers from poor work-efficiency.

To solve this problem, Narayanan and Vineet proposed a scheme to execute only active blocks [12]. After execut-

```
Algorithm 1 : Low overhead global relabeling
1 :  Launch global relabeling kernels for M thread blocks
2 :     Add 1 pixel block (that has been initially assigned)
          to each thread block's FIFO list
3 :     **while** FIFO list is not empty
4 :        Read flow and height from global memory
5 :        Generate new height information for all pixels
             in the pixel block
6 :        **if** height of neighbor blocks must be updated **then**
7 :           **if** the neighbor block is included in another
                thread block's FIFO **then**
8 :              Do nothing
9 :           **else**
10 :             Add that neighbor block to its own FIFO list
11 :          end **if**
12 :       end **if**
13 :       Write new height to global memory
14 :    end **while**
15 : Terminate current thread block and process next block
```

ing a single iteration, the list of active blocks is sent to the host CPU. Then, CPU only launches kernel for those block that are active. This method allows us to maintain work-efficiency.

However, some problems still remain with this approach. Since a list of active blocks are sent to CPU on every iteration, memory transfer overhead increases as a result. Another problem is that it requires several kernel launches, as illustrated in the top part of Figure 4. Frequent kernel launch decreases the GPU occupancy, because there is several microseconds delay between CPU kernel commit and GPU kernel execution.

5.2. Proposed Method

To address problems with the conventional method, we propose a low-overhead global relabeling algorithm which involves only a single kernel launch. In contrast to the conventional scheme where each *processor* holds a FIFO queue (*e.g.* [3][4][7]), the key idea of the proposed algorithm is that each *thread block* maintains a FIFO list of pixel blocks to be processed. After processing a pixel block, the thread block moves onto the next pixel block in its FIFO list. Unlike [12], it becomes possible to revisit pixel block in a single kernel launch, if another thread block attempts to update the information on the already-processed pixel block.

The kernel code for proposed global relabeling is shown in Algorithm 1. Kernel is launched only once (step 1). Initially, each thread block is assigned 1 pixel block, and this is added to its FIFO list (step 2). After reading information from the global memory (step 4), the kernel generates new height information by finding the minimum distance to the sink (step 5). Before the generation, the thread block stores a list of neighboring blocks' adjacent pixels that previously had no height information. If any pixels in current block adjacent to them obtained new height information after current block's global relabeling, it means that blocks neighboring the updated pixels should be updated (step 6). The thread block refers to a list of pixel block occupancy flags to check if that neighboring pixel block is included in another thread block's FIFO (step 7). If so, nothing is performed (step 8), because that pixel block is going to be processed by another thread block. If not, add it to its own FIFO list (step 10), so that it can be visited later.

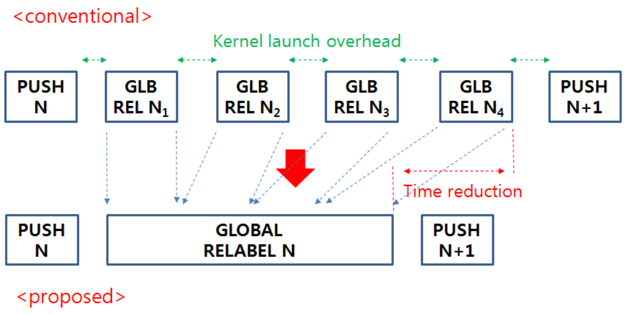


Figure 4. Kernel launch overhead reduction using proposed strategy. The execution time is reduced since only 1 kernel launch is needed for global relabeling.

The advantage of thread block, rather than processor, having FIFO list is that thread blocks are more flexible. They can be processed by any processor. Also, it gives finer granularity, making it easier to load-balance pixel blocks among SM automatically. Another important advantage is that only a single kernel launch is needed. As a result, the kernel launch time overhead is reduced and the GPU occupancy ratio can be increased, as shown in Figure 4.

Experimental result shows that the proposed kernel launch scheme reduces the execution time of global relabeling part by 1.64X compared to the active block only scheme [12]. Note that push & relabel part cannot adopt similar single-kernel strategy, because global relabeling must be performed periodically in between.

6. Experimental Result

6.1. Experimental Setup

The experiment is conducted on NVIDIA GTX580 GPU. It has 16 SMs, each with 32 cores and 48 KB of shared memory. The cores run at 1.5 GHz.

We used 7 stereo image pairs (Tsukuba, Venus, Teddy, Cones, Cloth3, Rocks2, Aloe) from Middlebury dataset [13]. Test parameter is optimized for each dataset, as shown in Table 2. For matching cost, we use combination of SAD (sum of absolute difference), SSD (sum of squared difference), or SGRAD (sum of absolute gradient

Image	Size (pixels)	λ	Matching Cost	Static Cue
Tsukuba	384×288	4.0	SAD	no
Venus	434×383	10.0	SSD	no
Teddy	450×375	2.0	SAD + SGRAD	yes
Cones	450×375	3.0	SAD + SGRAD	yes
Cloth3	626×555	5.0	SAD + SGRAD	yes
Rocks2	638×555	5.0	SAD + SGRAD	yes
Aloe	641×555	5.0	SAD + SGRAD	yes

Table 2. Parameter setting for each stereo matching dataset

difference) of the pixel intensity. For smoothness cost, we use the truncated linear function to model the slanted surface. In order to encourage similar labeling in low-textured regions, larger smoothness cost is assigned on pixels with similar intensity (called static cue [6]). Note that we use 4 cycles of α-expansion.

6.2. Performance Comparison

The measure the effect of the proposed techniques, we constructed a baseline GPU implementation which is based on the following conventional GPU techniques: It uses Kolmogorov's graph formulation in Figure 1. Push/relabel are performed once per global memory data fetch, and proceeds to the next push/relabel. Global relabeling is based on Narayanan's method that processes only active blocks [12]. The same GPU platform and the same parameter setting for each test images are used for both the baseline GPU and the proposed GPU implementation.

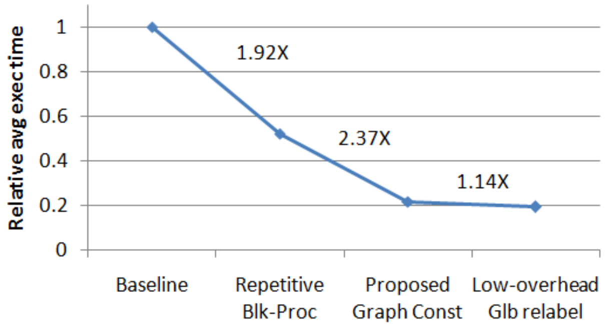


Figure 5. Cumulative execution time improvement (averaged over Tsukuba, Venus, Teddy, Cones)

The execution speed improvement can be observed in Figure 5. The speedup factor was obtained by averaging the relative execution time over 7 images. The proposed graph construction method increases the total execution speed by 2.37X, because it takes fewer iterations to push excess to the sink. Repetitive block-based processing brings 1.92X speedup, due to the reduced memory transfer.

	Push + Relabel	Global Relabel	Total
Tsukuba	0.25s	0.11s	0.40s
	3.1X	5.0X	3.3X
Venus	0.59s	0.35s	1.0s
	2.8X	4.7X	3.3X
Teddy	1.2s	0.53s	2.0s
	3.7X	6.7X	4.1X
Cones	1.0s	0.43s	1.7s
	3.9X	8.0X	4.5X
Cloth3	2.9s	2.1s	5.7s
	5.0X	10.9X	6.7X
Rocks2	2.6s	1.4s	4.7s
	6.2X	19.2X	9.4X
Aloe	4.5s	2.6s	8.0s
	3.9X	8.8X	5.1X
AVG	4.1X	9.0X	5.2X

Table 3. Detailed profiling result for the proposed GC method (Time (sec), Speedup over baseline GPU implementation (X))

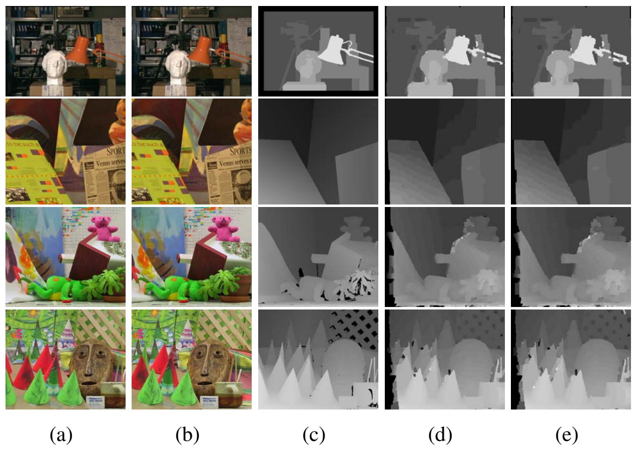


Figure 6. Test input and result images (a) left image (b) right image (c) ground truth (d) baseline disparity map (e) final disparity map, for Tsukuba, Venus, Teddy, and Cones

Low-overhead kernel launch scheme increases the speed of global relabeling by 1.64X (overall 1.14X). The detailed profiling result of the final proposed GC is shown in Table 3. Note that the total execution time also includes various minor parts such as initialization or graph construction time, in addition to push, relabel, and global relabeling. Overall, the comparison with the baseline implementation shows that total of 5.2X speedup can be obtained by employing the proposed GC algorithm.

The qualitative comparison on the accuracy of the disparity map between the baseline and the proposed method can be made in Figure 6. Quantitatively, it is measured by counting the number of pixels that has more than 1 label difference from the ground truth (called bad pixel %). This

Image	Baseline	Proposed GC	Difference
Tsukuba	1.84%	2.07%	+0.23%
Venus	0.62%	0.73%	+0.11%
Teddy	5.23%	5.31%	+0.08%
Cones	3.22%	3.29%	+0.07%
Cloth3	1.48%	1.45%	-0.03%
Rocks2	2.40%	2.44%	+0.04%
Aloe	5.25%	5.61%	+0.36%
AVG	–	–	+0.12%

Table 4. Comparison of bad pixel percentage

	Vineet [14]	Proposed GC	
	Time (sec)	Time (sec)	Speedup (X)
Tsukuba	0.95s	0.40s	2.4X
Venus	2.87s	1.0s	2.9X
Teddy	6.89s	2.0s	3.4X
AVG	–	–	2.9X

Table 5. Comparison with state-of-the-art GPU-based GC [14]

value is presented in Table 4. It shows an average degradation of 0.12% in accuracy. Considering high speedup of 5.2X, the degradation can be considered as negligible.

6.3. Comparison with Related Work

A comparison with current state-of-the-art GPU-based GC [14] is made in Table 5. It shows that the proposed GC has better performance by an average of 2.9X on three images. Note that a fair comparison is difficult due to 3 reasons. First, the referenced work uses GTX280 (G80 architecture [1], 933 GFLOPS) whereas we use GTX580 (Fermi architecture [1], 1581 GFLOPS). Second, it contains stereo matching result for only smaller images (Tsukuba, Venus, Teddy). Third, their work heavily utilizes graph reparameterization technique [2] [10] for speedup, at the cost of increased global memory consumption.

7. Conclusion

We have presented a novel GPU-based GC algorithm for stereo matching. A new graph construction method was proposed to reduce the number of iterations. The proposed method allows accurate initialization and label expansion reordering to further accelerate the convergence speed. Next, a repetitive block-based push and relabel strategy was employed to increase the number of computation per global to shared memory data transfer. Finally, a new global relabeling algorithm that reduces the kernel launch overhead was presented. Experiments on Middlebury dataset show that, compared to conventional GPU implementation techniques, 5.2X speedup in execution time can be obtained with 0.12% degradation in accuracy.

Acknowledgement

This research was supported by the Basic Science Research Program through the National Research Foundation of Korea funded by the Ministry of Education, Science and Technology (2012R1A1A2009495).

References

[1] CUDA C programming guide. NVIDIA Corporation, 2012. `http://docs.nvidia.com/`. 2, 7

[2] K. Alahari, P. Kohli, and P. Torr. Reduce, reuse & recycle: Efficiently solving multi-label MRFs. In *Proc. IEEE Conf. Computer Vision and Pattern Recognition*, pages 1–8, June 2008. 3, 7

[3] R. Anderson and J. Setubal. On the parallel implementation of Goldberg's maximum flow algorithm. In *Proc. 4th Annual ACM Symp. Parallel Algorithms and Architectures*, pages 168–177, June 1992. 5

[4] D. Bader and V. Sachdeva. A cache-aware parallel implementation of the push-relabel network flow algorithm and experimental evaluation of the gap relabeling heuristic. In *Proc. 18th ISCA Intl. Conf. Parallel and Distributed Computing Systems*, September 2005. 5

[5] Y. Boykov and V. Kolmogorov. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *IEEE Trans. Pattern Anal. and Mach. Intell.*, 26(9):1124–1137, September 2004. 2

[6] Y. Boykov, O. Veksler, and R. Zabih. Fast approximate energy minimization via graph cuts. *IEEE Trans. Pattern Anal. and Mach. Intell.*, 23(11):1222–1239, November 2001. 2, 6

[7] A. Delong and Y. Boykov. A scalable graph-cut algorithm for N-D grids. In *Proc. IEEE Conf. Computer Vision and Pattern Recognition*, pages 1–8, June 2008. 4, 5

[8] A. Goldberg and R. Tarjan. A new approach to the maximum flow problem. In *Proc. the 18th Annual ACM Symp. Theory of Computing*, pages 136–146, May 1986. 2

[9] O. Juan and Y. Boykov. Active graph cuts. In *Proc. IEEE Conf. Computer Vision and Pattern Recognition*, pages 1023–1029, June 2006. 3

[10] P. Kohli and P. Torr. Dynamic graph cuts for efficient inference in Markov random fields. *IEEE Trans. Pattern Anal. and Mach. Intell.*, 29(12):2079–2088, December 2007. 3, 7

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

[12] P. Narayanan, V. Vineet, and T. Stich. Fast graph cuts for computer vision. *GPU Computing Gems*, pages 439–450, 2011. 1, 4, 5, 6

[13] D. Scharstein and R. Szeliski. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. *Intl. Journal of Computer Vision*, 47(1–3):7–42, April 2004. 1, 3, 5

[14] V. Vineet and P. Narayanan. Solving multilabel MRFs using incremental α-expansion on the GPUs. *Lecture Notes in Computer Science*, 5996 (ACCV 2009):633–643, September 2010. 1, 4, 7