CUDA Cuts: Fast Graph Cuts on the GPU

Vibhav Vineet and P. J. Narayanan
Centre for Visual Information Technology
International Institute of Information Technology
Hyderabad, 500032. India
{vibhavvinet@students.,pjn@}iiit.ac.in

Abstract

Graph cuts has become a powerful and popular optimization tool for energies defined over an MRF and have found applications in image segmentation, stereo vision, image restoration, etc. The maxflow/mincut algorithm to compute graph-cuts is computationally heavy. The best-reported implementation of graph cuts takes over 100 milliseconds even on images of size 640×480 *and cannot be used for real-time applications or when iterated applications are needed. The commodity Graphics Processor Unit (GPU) has emerged as an economical and fast computation co-processor recently. In this paper, we present an implementation of the push-relabel algorithm for graph cuts on the GPU. We can perform over 60 graph cuts per second on* 1024×1024 *images and over 150 graph cuts per second on* 640×480 *images on an Nvidia 8800 GTX. The time for each complete graph-cut is about 1 millisecond when only a few weights change from the previous graph, as on dynamic graphs resulting from videos. The CUDA code with a well-defined interface can be downloaded for anyone's use.*

1. Introduction

Graph cuts have found applications in a large range of Computer Vision problems as a tool to find the optimal MAP estimation of images defined over an MRF lattice. Though the mincut/maxflow algorithm was introduced into Computer Vision early [17, 14], their potential was exploited only after the work of Boykov et al. [5, 6] and their characterization of functions that can be optimized using graph cuts [25]. Graph-cuts have since then been applied to several Computer Vision problems like image and video segmentation [29, 27], stereo and motion [5, 31], multi-camera scene reconstruction [24, 20], etc. Kolmogorov and Zabih characterized the energy functions which can be minimized via graph cuts [25].

Improving the computational performance of the maxflow algorithm has also been an active area of recent research. Boykov and Kolmogorov presented an algorithm to reuse the search trees [4]. Dynamic graph cut reparametrizes the graph and reuses the residual flow when only a few weights change from one minimization to the next [22, 23]. They could compute the graph cut on a new frame of a video in about 70 milliseconds by starting with the residual flow of the previous frame [23]. Active graph cuts, reuse the st-mincut solution corresponding to the previous MRF instance to generate the initialization for the next MRF [21]. The best implementation of it takes about 100 milliseconds on a 512×512 image.

The contemporary graphics processor unit (GPU) has huge computation power and can be very efficient on many data-parallel tasks. They have recently been used for many non-graphics applications [16] and many in Computer Vision. OpenVidia [13] is an open source package that implements different computer vision algorithms on the GPUs using OpenGL and Cg. Sinha et al. implemented a feature based tracker to the GPU [30]. SiftGPU implements the SIFT descriptor on the GPU [32]. The GPU, however, has had a difficult programming model that followed the traditional graphics pipeline. This made it difficult to implement general graph algorithms on them.

The potential of the GPU for non-graphic applications has resulted in more traditional parallel programming interfaces that treat the GPUs as massively parallel co-processors. The Compute Unified Device Architecture (CUDA) from Nvidia [9] and the Close-To-Metal (CTM) from ATI/AMD [8] are such interfaces for modern GPUs. These enable the acceleration of algorithms on irregular graphs [18] and other application involving graphs.

We present a fast implementation of the push-relabel algorithm for mincut/maxflow algorithm for graph-cuts in this paper using CUDA. Our implementation of the basic graph-cut can perform over 60 graph-cuts per second on images of size 1024×1024 and over 150 graph-cuts per second on images of size 640×480 on an Nvidia 8800 GTX GPU. Each graph cut can be computed in about 1 millisecond on

978-1-4244-2340-8/08/$25.00 ©2008 IEEE

images on dynamic graphs arising from videos. A shader based early implementation of graph cuts on the GPU was even slower than the CPU implementation [10]. Hussein et al. [19] report an implementation of the push-relabel algorithm on CUDA. They achieve a speedup of only 2-4.5 over the CPU implementation with a running time of 100 milliseconds per frames with a million pixels, as opposed to 6 milliseconds by our implementation.

Section 2 describes the GPU architecture as exposed by the CUDA programming model. Section 3 describes the the GPU implementation of the basic push-relabel algorithm for graph cuts. Section 4 presents the experimental results. Some concluding remarks and directions for future work are given in Section 5.

2. Compute Unified Device Architecture

General purpose programming on graphics processing units (GPGPU) tries to solve a problem by posing it as a graphics rendering problem, restricting the range of solutions that can be ported to the GPU. The GPU memory layout is optimized for graphics rendering. This restricts the GPGPU solutions as an optimal data structure may not be available. The GPGPU model provides limited autonomy to individual processors [28]. Creating efficient data structures using the GPU memory model is a challenging problem in itself [26]. Memory size on GPU is another restricting factor. A single data structure on the GPU cannot be larger than the maximum texture size supported by it.

CUDA is a programming interface to use the parallel architecture of Nvidia GPUs for general purpose computing. CUDA produces a set of library functions as extensions of the C language. A compiler generates executable code for the CUDA device. The CPU sees a CUDA device as a multi-core co-processor. All memory available on the device can be accessed using CUDA with no restrictions on its representation though the access times vary for different types of memory. This enhancement in the memory model allows programmers to better exploit the parallel power of the GPU for general purpose computing.

2.1. CUDA Hardware Model

At the hardware level, the GPU is a collection of multiprocessors, with several processing elements in each (Figure 1). For instance, the Nvidia 8800 GTX has 16 multiprocessors with 8 processing elements in each. Each multiprocessor has 16 KB of common shared memory accessible to all processors inside it. It also has a set of 32-bit registers, texture, and constant memory caches. Each processor in the multiprocessor executes the same instruction in every cycle. Each can operate on its own data, which makes each a SIMD processor. Communication between multiprocessors is only through the device memory, which is available to all the processors of the multiprocessors. The processing elements of a multiprocessor can synchronize with one another, but there is no direct synchronization mechanism between the multiprocessors. The GPUs provides only single-precision floating point numbers and 32-bit integers on native numeric data types, though this may change in near future.

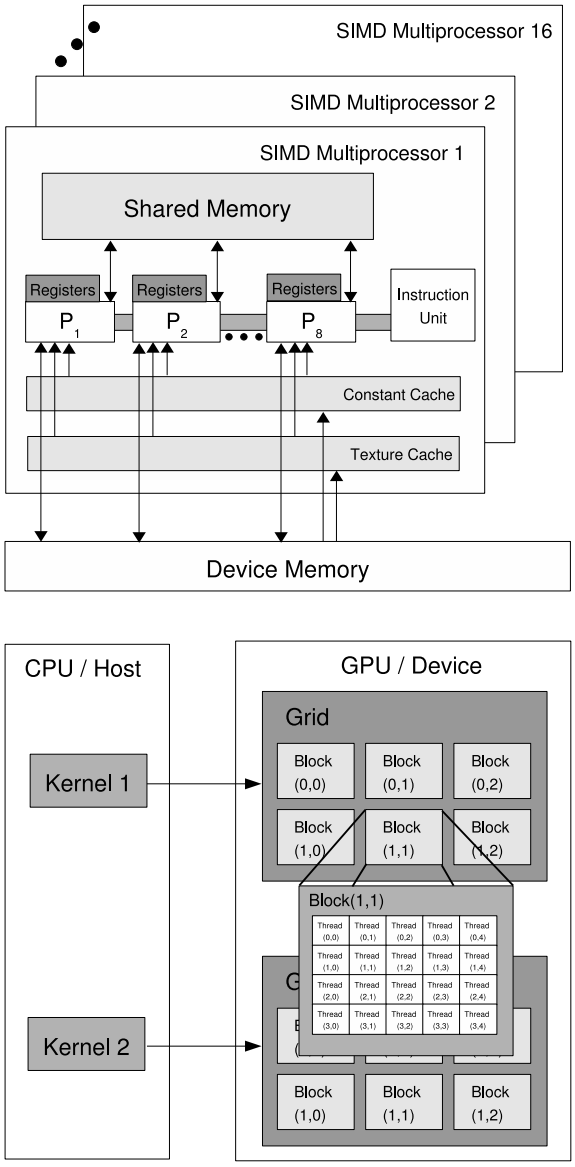

Figure 1. CUDA hardware model (top) and programming model (bottom) for Nvidia 8800 GTX

2.2. CUDA Programming Model

For the programmer, the CUDA consists of a collection of *threads* running in parallel. A *warp* is a collection of threads that are scheduled for execution simultaneously on a multiprocessor. The warp size is fixed for a specific GPU.

The programmer can select the number of threads to be executed. If the number of threads is more than the warp size, they are time-shared internally on the multiprocessor. A collection of threads called a *block* runs on a multiprocessor at a given time. Multiple blocks can be assigned to a single multiprocessor for time-shared execution. They also divide the common resources like registers and shared memory equally among them. A single execution on a device generates a number of blocks. The collection of all blocks in a single execution is called a *grid* (Figure 1). Each thread and block is given a unique ID that can be accessed within the thread during its execution. All threads of the grid execute a single program called the *kernel*.

The kernel is the core code to be executed on each thread. Using the thread and block IDs, each thread can perform the kernel task on different data. Since the device memory is available to all the threads, it can access any memory location. The CUDA programming interface presents a Parallel Random Access Machine (PRAM) architecture, if one uses the device memory alone. The performance improves with the use of shared memory which can be accessed in a single clock cycle. In contrast, the global or device memory access takes 200-400 cycles. Read only texture memory optimized for 2-D texture fetch and constant memory assigned by the CPU are also available. Their access is slow but the internal caching mechanism reduces the effective access times for coherent access. The hardware architecture allows multiple instruction sets to be executed on different multiprocessors. The current CUDA programming model, however, cannot assign different kernels to different multiprocessors, though this may be simulated using conditionals.

The Nvidia 8800 GTX graphics card has 768 MB memory. Large images can reside in this memory, given a suitable representation. The problem needs to be partitioned appropriately into multiple grids for handling even larger images and graphs.

3. GPU Graph Cuts

The mincut/maxflow algorithm tries to find the minimum cut in a graph that separates two designated nodes, namely, the source s and the target t. The mincut minimizes the energy of an MRF defined over the image lattice when a discontinuity preserving energy function is used [25]. The energy function used has the following form:

$$E(f) = \sum_{p,q \in N} V_{p,q}(f_p, f_q) + \sum_{p \in P} D_p(f_p), \qquad (1)$$

where, D_p is the data energy, $V_{p,q}$ is the smoothness energy, N the neighbourhood in the MRF, f_p is the label assigned to the pixel p, and P are all pixels of the lattice.

Two algorithms are popular to compute the mincut/maxflow on graphs. The first one, due to Ford and Fulkerson [12] and modified by Edmonds and Karp [11], repeatedly computes augmenting paths from source s to target t in the graph through which flow is pushed until no augmenting path can be found. The second algorithm, by Goldberg and Tarjan [15], works by pushing flow from s to t without violating the edge capacities. 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. There are two basic operations in a push-relabel algorithm: pushing excess flow from a vertex to one of its neighbors and relabelling a vertex. The algorithm is sped up in practice by periodically relabelling the vertexes using a global relabelling procedure or a gap relabelling procedure [7].

The sequential implementation of graph cuts by Boykov and others follow the Edmonds-Karp algorithm which repeatedly finds the shortest path from the source to the target using a breadth-first search (BFS) step, which is not easily parallelizable. The push-relabel algorithm was parallellized by Anderson and Setubal [2]. Bader and Sachdeva later produced a cache-aware optimization of it [3]. The target architecture is a cluster of symmetric multi-processors (SMPs) having from 2 to over 100 processors per node. Alizadeh and Goldberg [1] present a parallel implementation on a massively parallel Connection Machine CM-2. Two attempts to implement this algorithm on the GPU have also been reported [10, 19]. We implement the push-relabel algorithm on the GPU using CUDA.

3.1. Push-Relabel Algorithm

Let $G = (V, E)$ be the graph and s, t be the source and target nodes. The push-relabel algorithm constructs and maintains a residual graph at all times. The 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 $f(u, v)$, where $c(u, v)$ is the capacity of the edge (u, v). The push-relabel algorithm maintains two quantities: the excess flow $e(v)$ at every vertex and the height $h(v)$ for all vertexes $V' = V \cup \{s, t\}$ with $h(s) = n$ and $h(t) = 0$. The excess flow $e(v) \geq 0$ is the difference between the total incoming and outgoing flows at node v through its edges. The height $h(v)$, is a conservative estimate of the distance of vertex v from the target t. Initially all the vertexes have a height of 0 except for the source s which has a height $n = |V|$, the number of nodes in the graph.

Computation proceeds in terms of two operations. 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 neighbour $(u, v) \in E_f$. After the push, either vertex u is saturated (i.e., $e(u) = 0$) or the edge (u, v) is saturated (i.e., $c_f(u, v) = 0$). The 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 relabelling step by setting it to one more than the minimum height of its neighbouring nodes. Global relabelling needs a BFS to correctly assign the distances to the target. Gap relabelling needs to find any gaps in the height values in the entire graph. Both are expensive operations and are performed only infrequently. The algorithm stops when neither push nor relabelling can be applied. The excess flows in the nodes are then pushed back to the source and the saturated nodes of the final residual graph gives the mincut.

3.2. 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 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.

Our implementation of the push-relabel algorithm uses four kernels. The *Push* kernel pushes excess flow at each node to its neighbours and the *Pull* kernel updates the net excess flow at each node. The *Local Relabel* kernel applies a local relabelling operation to adjust the heights as stipulated by the algorithm. The *Global Relabel* kernel runs a BFS from the target t and updates the heights of all nodes to the correct distances.

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 or 8-neighbours. Different strategies will have to be adopted for general graphs represented using adjacency list or adjacency matrix. The grid has the dimensions of the image and is divided into $B \times B$ blocks. Each thread handles a single node or pixel. Thus, a block handles B^2 pixels and needs to access data from a $(B+2) \times (B+2)$ 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 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.

Push Kernel: A node can be active, passive, or inactive. Active nodes have the 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 $h(u)$ and $e(u)$ 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.

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. We divide the operation into two kernels, with the push kernel doing computing the changes in edge weights and the pull kernel subsequently updating each node's net excess flow. The push kernel updates the edge-weights of the possible edges $(u, v) \in E_f$ and the excess flow $e(u)$ of itself. Another option could be to combine the two kernels with the results of push kept in the shared memory to be used for pull in another part of the kernel. However, the nodes on the border of blocks will need results from other blocks. This cannot be done correctly as CUDA doesn't allow synchronization between blocks. The thread for node u of the kernel does the following.

*PushKernel (*node *u)*

1. Load $h(u)$ and $e(u)$ from the global memory to the shared memory of the block.
2. Synchronize threads (ensure completion of load).
3. Push $e(u)$ to eligible neighbours without violating the residual capacity of the edges.
4. Store the flow pushed to each edge in a special global memory array F.

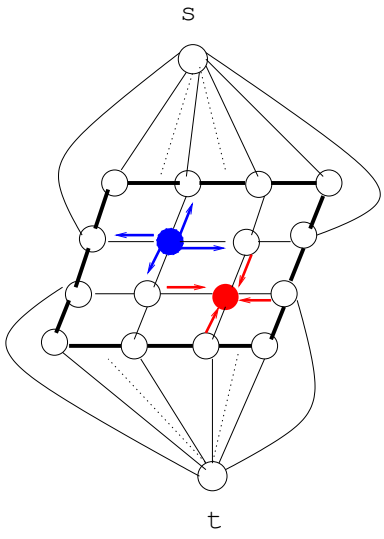

Figure 2. A 4×4 grid graph. Push kernel pushes flow along edges and pull kernel takes them into each node.

Pull Kernel: In the pull phase, a node receives all flow pushed to it in the previous step and computes its net excess flow. The kernel does the following.

*PullKernel (*node *u)*

1. Read the flow pushed to u from the F array of its neighbours.
2. Compute the final excess flow by aggregating all incoming flows. Store it as the $e(u)$ value in the global memory.

Figure 2 shows the basic Push and Pull operations. It shows an active vertex which pushes flow to all its neighbours, indicated using outgoing arrows. Similarly, a vertex in the pull phase updates its excess flow (shown with arrows coming in) by receiving flows from its neighbours and aggregating the net excess.

Local Relabel Kernel: The local relabelling step replaces the height of a node with 1 more than the minimum of the heights of its active or passive neighbours. This operation reads the heights of neighbouring nodes from the global memory and writes the new height value to the global memory. After the relabel operation, many passive nodes become active. The thread for node u of the kernel does the following.

*RelabelKernel (*node *u)*

1. Load $h(u)$ and $flag(u)$ from the global memory to the shared memory of the block.
2. Synchronize threads (ensure completion of load).
3. Compute the minimum height of active or passive neighbours of u.
4. Write the new height to global memory loication $h(u)$.

Global Relabel Kernel: The local relabelling step adjusts the heights based only on the local information. The true distance to the target node t can be computed using a Breadth First Search on the graph starting from t. For BFS, the t node is assigned a height of 0. In each iteration, each node looks at the minimum height of all its neighbours and sets own height as one more than that. The updation is done in parallel in each iteration. The kernel is invoked with the iteration number $k = 1, 2, \ldots$ and the partial results are stored in the global memory. The thread for node u of the kernel does the following for iteration k.

*GlobalRelabelKernel (*node *u,* iteration *k*)

1. If $k = 1$, all pixel nodes with non-zero residual capacity to t are assigned a height of 1.
2. Every unassigned node assigns itself a height of $k + 1$ if any of its neighbours have a height of k.
3. Update the new height values in the global memory array.

m	k	Number of Pulses	Time (in ms)
1	-	28	4.5
2	-	23	5.2
3	-	23	7.3
1	1	17	6.7
2	1	17	9.2
1	4	20	4.1
1	6	20	3.9
2	4	19	5.9

Table 1. Results for different m and k values. The algorithm performs m push/pull operations followed by a local relabelling per pulse. A global relabelling is done every k pulses. No global relabelling is used in the top 3 cases.

Overall Graph Cuts Algorithm: The overall algorithm applies the above steps in sequence, as follows. The CUDA grid has the same dimensions as the image, say, $M \times N$. The CUDA block size is $B \times B$, with B^2 threads in each.

GPUGraphCuts ()

1. Compute the edge weights and energies from the underlying image.
2. Invoke *PushKernel()* followed by *PullKernel()* on the whole grid.
3. Repeat step 2 for m times.
4. Invoke *RelabelKernel()* on the grid.
5. Repeat steps 2 to 4 for k times.
6. Apply *GlobalRelabelKernel()* on the grid.
7. Repeat steps 2 to 6 until convergence.

The computation terminates when no push or relabeling operation is possible.

Efficiency Considerations: 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 PushKernel() and RelabelKernel() speeds up the operations by a factor of 25% . The binary segmentation of the flower image of size 600x450 takes almost 6.5 ms using the global memory access while it takes 4.9 ms with the shared memory access, as the global memory access is 200-400 times slower than shared memory access. The speedup is low because of the overhead of loading data into the shared memory and the thread synchronization. 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 even if any one succeeds. We also use 2D texture memory to store read-only data, such as the data-cost, the smoothness-cost and

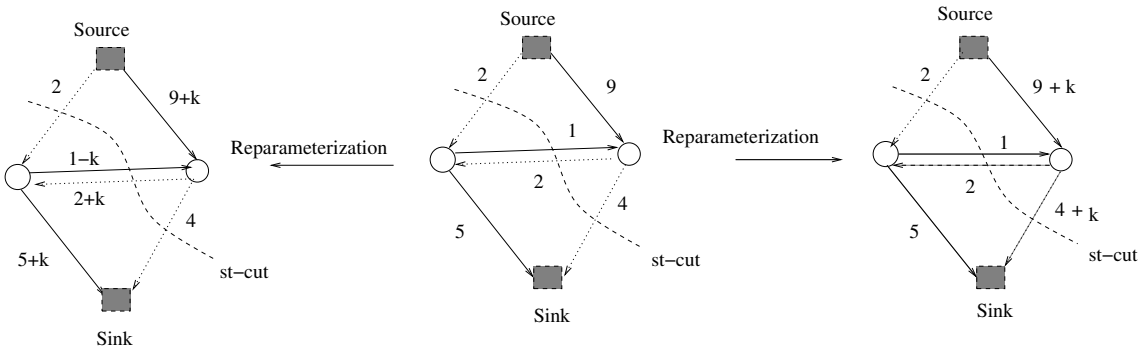

Figure 3. The graph reparameterization scheme for change in weights [23]

the intensity values of the pixels, for edge-weight calculations. The texture memory access is cached. This results in a typical improvement in speed of 50% . The edge-weight calculation for an image of size 600×450 takes 0.07 ms using the texture memory whereas it takes almost 0.12 ms for the same using the global memory access.

The push-relabel algorithm can perform multiple pushes before attempting any relabelling. The global relabelling is performed only occasionally. Table 1 gives the results for varying m and k of *GPUGraphCuts()* procedure on the Liberty-Bell image. The combination of a local relabelling after each push/pull and a global relabelling after six such pulses seems to do the best. More frequent global relabelling results in fewer overall pulses, but takes more time due to the slower BFS operation. The use of gap relabelling in combination with local and global relabelling resulted in very poor performance. This because the operation of identifying and counting the gaps in labels is not efficient on the SIMD architecture of the GPUs.

3.3. Dynamic Graph Cuts

Repeated application of graph cuts on graphs for which only a few edges change weights is useful 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 [23]. The resulting graph is close to the final residual graph and its mincut can be computed in a small number of iterations.

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 (Figure 3). These operations assign new weights/capacities as a modification of the final graph without violating any constraints.

4. Experimental Results

The CUDA Cuts algorithm was tested on several standard images. The running time also depends on the number of threads per block as that determines the level of parallelism. We experimented with different numbers of threads per block. A block size of $B = 16$ threads gives the best results with 256 threads per block. Each thread uses 14 registers for Push and Pull kernels and 15 registers for the Relabel kernel. An image of size 600×450 takes 5.7 milliseconds

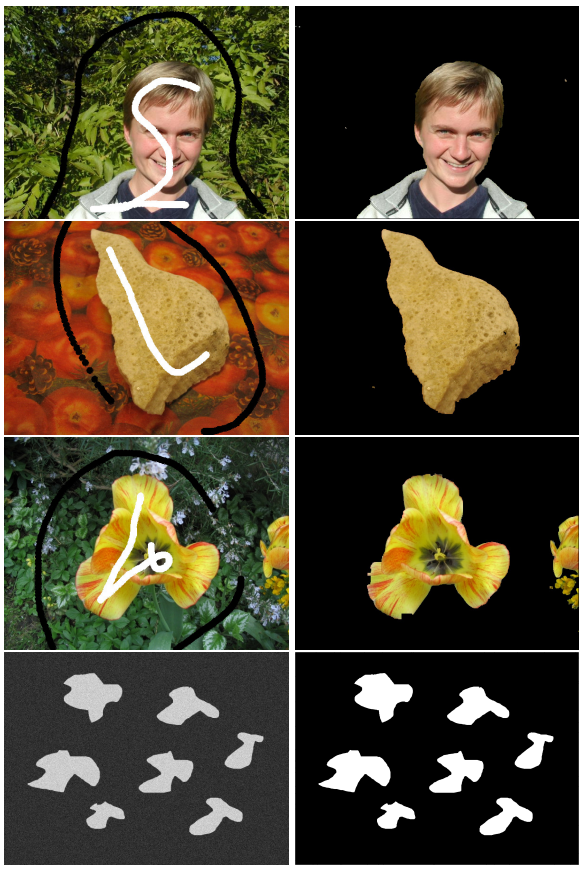

Figure 4. Binary Image Segmentation: Person, Sponge, Flower, and Synthetic images

Image	Size	Time (ms) Boykov	Time (ms) CUDA Cuts
Person	600× 450	140	4.86
Sponge	640× 480	142	5.76
Flower	600× 450	188	4.98
Synthetic	1K× 1K	655	16.5

Table 2. Comparison of running times of CUDA implementation with that of Boykov on different images

with a block size of 512 threads and 4.9 ms with a block size of 256 threads.

We tested our implementations on various real and synthetic images. Figure 4 shows the results of image segmentation using our implementation of the algorithm on the Person image, Sponge image and the Flower image. It also shows the results of image segmentation using our implementation of the algorithm on a noisy synthetic image. The energy terms used are the same as those given in the Middlebury MRF page [31]. The running times for these are tabulated in Table 2 along with the time for Boykov's sequential implementation of graph cuts. The reported times of the GPU algorithm includes the time to compute the edge weights. Figure 5 plots the running times on a noisy synthetic image of CUDA Cuts and the sequential graph cuts for different image sizes. The GPU implementation is faster by factor of 100 or better. This is about 8-10 times fraster than the previous best implementation on the GPU by Hussein et al. [19].

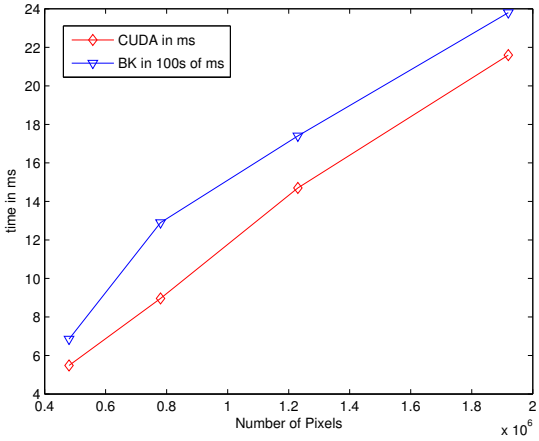

Figure 5. Comparing the running times of graph cuts on the GPU and the CPU for synthetic images. The GPU implementation is over 100 times faster.

Figure 6 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

Figure 6. Frames of a video segmented using dynamic cuts

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 less than 1 millisecond per frame. The running time depends on the percentage of weights that changed.

5. Conclusions and Future Work

In this paper, we presented an implementation of graph-cuts on GPU using CUDA architecture. We used the push-relabel algorithm for mincut/maxflow as it is more parallelizable. Periodic global relabelling improves the running time. We carefully divide the task among the multiprocessors of the GPU and exploit its shared memory for high performance. We perform over 150 graph cuts per second on 640×480 images. This is 30-40 times faster than the best sequential algorithm reported. We can process dynamic graphs in under a millisecond per frame, 70-100 times faster than the best sequential algorithm. More importantly, since a graph cut takes only 5 or 6 milliseconds, it can be applied multiple times on each image if necessary, without violating real-time performance. The code is available from our webpage and other relevant resources for download and use by other researchers. We are currently working on implementing multilabel graph cuts onto the GPU using a similar strategy.

Acknowledgements: We gratefully acknowledge the contributions of Nvidia through generous equipment donations.

References

[1] 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.

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

[3] D. A. 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 *ISCA PDCS*, pages 41–48, 2005.

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

[5] Y. Boykov, O. Veksler, and R. Zabih. Markov random fields with efficient approximations. In *CVPR*, pages 648–655, 1998.

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

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

[8] A. Corporation. Ati ctm (close to metal) guide. Technical report, AMD/ATI, 2007.

[9] N. Corporation. Cuda: Compute unified device architecture programming guide. Technical report, Nvidia, 2007.

[10] N. Dixit, R. Keriven, and N. Paragios. GPU-cuts: Combinatorial optimisation, graphic processing units and adaptive object extraction. Technical report, CERTIS, 2005.

[11] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19(2):248–264, 1972.

[12] L. R. Ford and D. R. Fulkerson. *Flows in Networks*. Princeton Univ. Press, NJ, 1962.

[13] J. Fung, S. Mann, and C. Aimone. OpenVidia: Parallel GPU computer vision. In *Proc of ACM Multimedia 2005*, pages 849–852, 2005.

[14] S. Geman and D. Geman. Stochastic relaxation, gibbs distributions, and the bayesian restoration of images. *IEEE Trans. Pattern Anal. Mach. Intell.*, 6:721–741, 1984.

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

[16] N. K. Govindaraju. GPUFFTW: High performance GPU-based fft library. In *Supercomputing*, 2006.

[17] D. Greig, B. Porteous, and A. Seheult. Exact maximum a posteriori estimation for binary images. *J. Royal Statistical Society., Series B*, 51(2):271–279, 1989.

[18] P. Harish and P. J. Narayanan. Accelerating large graph algorithms on the GPU using CUDA. In *Intnl. Conf. on High Performance Computing (HiPC), LNCS 4873*, pages 197–208, December 2007.

[19] 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.

[20] H. Ishikawa and D. Geiger. Occlusions, discontinuities, and epipolar lines in stereo. In *ECCV (1)*, pages 232–248, 1998.

[21] O. Juan and Y. Boykov. Active graph cuts. In *CVPR (1)*, pages 1023–1029, 2006.

[22] P. Kohli and P. H. S. Torr. Effciently solving dynamic markov random fields using graph cuts. In *ICCV*, pages 922–929, 2005.

[23] P. Kohli and P. H. S. Torr. Dynamic graph cuts for efficient inference in markov random fields. *IEEE Trans. Pattern Anal. Mach. Intell.*, 29(12):2079–2088, 2007.

[24] V. Kolmogorov and R. Zabih. Computing visual correspondence with occlusions via graph cuts. In *ICCV*, pages 508–515, 2001.

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

[26] A. E. Lefohn, S. Sengupta, J. Kniss, R. Strzodka, and J. D. Owens. Glift: Generic, efficient, random-access GPU data structures. *ACM Trans. Graph.*, 25(1):60–99, 2006.

[27] Y. Li, J. Sun, and H.-Y. Shum. Video object cut and paste. *ACM Trans. Graph.*, 24(3), 2005.

[28] P. J. Narayanan. Processor Autonomy on SIMD Architectures. In *Proceedings of the Seventh International Conference on Supercomputing*, pages 127–136, 1993.

[29] C. Rother, V. Kolmogorov, and A. Blake. Grabcut: interactive foreground extraction using iterated graph cuts. *ACM Trans. Graph.*, 23(3):309–314, 2004.

[30] S. N. Sinha, J.-M. Frahm, M. Pollefeys, and Y. Genc. Feature tracking and matching in video using graphics hardware. In *Proc of Machine Vision and Applications*, 2006.

[31] 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.

[32] C. Wu and M. Pollefeys. Siftgpu library. Technical Report http://cs.unc.edu/ ccwu/siftgpu/, UNC, Chapel Hill, 2005.