CudaCuts: 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 has found applications in image segmentation, stereo vision, image restoration etc. The maxflow/mincut algorithm to compute graph cuts is computationally expensive. The best reported implementation of it takes over 140 milliseconds even on images of size 640×480 for two labels and cannot be used for real time applications.

The commodity Graphics Processor Unit (GPU) has emerged as an economical and fast parallel co-processor recently. In this paper, we present an implementation of the push-relabel algorithm for graph cuts on the GPU. We show our results on some benchmark dataset and some synthetic images. We can perform over 25 graph cuts per second on 640×480 size benchmark images and over 35 graph cuts per second on $1K \times 1K$ size synthetic images on an Nvidia GTX 280. The time for each complete graph-cut is few milliseconds when only a few edge weights change from the previous graphs, as on dynamic graphs. The CUDA code with a well-defined interface can be downloaded from http://cvit.iiit.ac.in/index.php?page=resources .

1. Introduction

Graph cuts have been used as a method to find the optimal MAP estimation of various Computer Vision problems defined over an MRF. Though the mincut/maxflow algorithm was introduced into Computer Vision early [17], 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.

Various algorithms and strategies have been proposed to improve the computational performance of maxflow/mincut algorithm. Boykov and Kolmogorov [4] reused the search trees towards improving the computational efficiency. MRFs can be initialized to the solution computed for the MRF instance in the previous frame, in a video, to converge to the solution quickly. Dynamic graph-cut [22, 23] and Active graph-cuts [21] use similar strategies. All the proposed implementations cannot be used for any real-time application.

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 non-graphics applications [16] and many in Computer Vision, e.g., OpenVidia [13], feature based tracker [30], SiftGPU [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 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.

In this technical report, we present a fast implementation of the push-relabel algorithm for mincut/maxflow algorithm for graph-cuts using CUDA. We use the global memory and the shared memory on the GPU for efficient computation. We use the atomic functions operating on the global memory only availabel for devices of compute capability 1.1 or above. We also propose stochastic cuts which improves the performance of push-relabel algorithm by factors for different problems on the GPU.

Our implementation of the basic graph-cut can perform over 30 graph-cuts per second on synthetic images of size 1024×1024 and benchmark images of size 640×480 on an Nvidia GTX 280. Each graph cut can be computed in few millisecond on 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 33 milliseconds by our implementation. Section 2 describes the the GPU implementation of the basic push-relabel algorithm for graph cuts. Different strategies to optimize the graph cuts on CUDA is described in section 3. Section 4 presents the experimental results. Some concluding remarks and directions for future work are given in Section 5.

2. Graph Cuts on GPU

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 to its distance to the sink. The algorithm is sped up in practice by periodically relabelling the vertices using a global relabelling procedure or a gap relabelling procedure [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.

2.1. 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 breadth first search 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.

2.2. Graph construction on CUDA architecture

Our graph-construction exploits the grid-structure that arises for MRFs defined over images. There are two popular method for constructing the graphs for the MRFs defined over images. Kolmogorov et.al. [25] constructs the graph which does not introduce any auxiliary vertices, which is in contrast to the graph construction of Boykov et.al. [6], which introduces auxiliary vertices. We adapt the graph construction of Kolmogorov et.al. [25], which maintains the the grid structure, suitable for the GPU/CUDA architecture. We constructs the grid-graph such that each pixel represents a non-terminal vertex in the graph. We assume fixed connectivity which could be 4 or 8 neighbors for each node. Consequently, 4 or 8 two-dimensional arrays store the weights along the n-edges. Two other arrays hold the excess

flow and the edge capacity to the target node for each node. Graph construction on GPU is very fast as shown in Table 1. An array to hold the heights and a mask array to hold the status of each node complete the representation. This representation can easily be extended to 3D grids for 3D graph cuts and other fixed connectivity patterns. Different strategies will have to be adopted for general graphs represented using adjacency list or adjacency matrix.

Image	Size	GPU Time(ms)	CPU Time(ms)
Sponge	640X480	0.151	61
Person	600X450	0.15	60
Flower	600X450	0.15	60

Table 1. Timings for constructing graphs from energy functions on different dataset on GTX 280 and CPU.

2.3. Push-Relabel Algorithm on CUDA

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

The basic implementation of the push-relabel algorithm requires three phases. The *Push* phase pushes excess flow at each node to its neighbours and the *Pull* phase updates the net excess flow at each node. The *Local Relabel* phase applies a local relabelling operation to adjust the heights as stipulated by the algorithm. These three basic phases use two kernels. The *Push* phase requires one kernel. The *Pull* phase and *Local Relabel* phase require another kernel. The heights of the nodes can also be adjusted by applying breadth first search starting from the sink. The breadth first search step is very slow and slows the computation overall.

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

There are multiple blocks running in parallel on the GPU. We organize them into a two-dimensional grid of geometry $B_x \times B_y$, where B_x and B_y are the number of thread blocks in x and y directions. Blocks are further divided into $D_x \times D_y$ threads. The maximum efficiency is achieved when B_x and B_y are multiple of D_x and D_y respectively. We represent an image as a two dimensional grid. Each thread block is of size 32×8. To achieve maximum efficiency, we pad the rows and columns to make them multiples of 32×8.

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 $h(u)$ 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. To maitain the preflow conditions without the read-after-write data consistency, we divide the operation into two kernels: Push Kernel and Pull kernel. However, atomic functions can perform read-modify-write atomic operations on 32-bit or 64-bit word residing in global or shared memory. So, we can combine push phase and pull phase without any inconsistency. Section 2.3.1 describes the implementation on hardware with

atomic capabilites and Section 2.3.2 describes the implementation on hardware without atomic capabilities.

2.3.1 CudaCuts on hardware with atomic capabilities

PushPull Kernel: The kernel updates the edge-weights of the edges (u, v) and (v, u) and the excess flows $e(u)$ and $e(v)$ of the vertices, u and v in the residual graph E_f.

*PushPullKernel (*node *u)*

1. Load $h(u)$ from the global memory to the shared memory of the block.
2. Synchronize threads to ensure completion of load.
3. Push flow to eligible neighbours atomically without violating the preflow conditions.
4. Update the edge-weights of (u, v) and (v, u) atomically in the residual graph E_f.
5. Update the excess flows of $e(u)$ and $e(v)$ atomically in the residual graph E_f.

Atomic writes to the global memory ensure synchronzation across the blocks of the grid.

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

*RelabelKernel (*node *u)*

1. Load $h(u)$ from the global memory to the shared memory of the block.
2. Synchronize threads to ensure completion of load.
3. Compute the minimum height of neighbours of u in the residual graph E_f.
4. Write the new height to global memory $h(u)$.

2.3.2 CudaCuts on hardware without atomic capabilities

Atomic functions are not available on such devices, so push and pull phases can not be combined into one. The synchronization is limited to the threads of a block. The border pixels of a block may not get the exact flow value. So, we perform the push operation in one kernel and pull and relabel operation in another kernel.

Push Kernel: The push kernel updates the edge-weights of the possible edges (u, v) and the excess flow of $e(u)$ in residual graph.

*PushKernel (*node *u)*

1. Load $h(u)$ from the global memory to the shared memory of the block.
2. Synchronize threads to ensure completion of load.
3. Push flow to the eligible neighbours without violating the preflow conditions.
4. Update the residual capcities of edges (u, v) in residual graph.
5. Update the excess flow $e(u)$ of the vertex.
6. Store the flow pushed to each edge in a special global memory array F.

PullRelabel Kernel: The kernel updates the excess flow $e(u)$ and edge-weights $e(u, v)$ residual graph.

*PullRelabelKernel (*node *u)*

1. Load $h(u)$ from the global memory to the shared memory of the block.
2. Synchronize threads to ensure completion of load.
3. Update the excess flow $e(u)$ of each vertex and the residual capacities of edges (u, v) in the residual graph E_f with the flows from global memory array F.
4. Synchronize threads to ensure completion of updation of edge-weights and excess flow.
5. Compute the minimum height of neighbours of u in the residual graph E_f.
6. Write the new height to global memory $h(u)$.

Figure 1 shows the effect of push and pull operations. It shows an active vertex which pushes flow to all its neighbours in the residual graph. Similarly, a vertex in the pull phase updates its excess flow by receiving flows from its neighbours and aggregating the net excess.

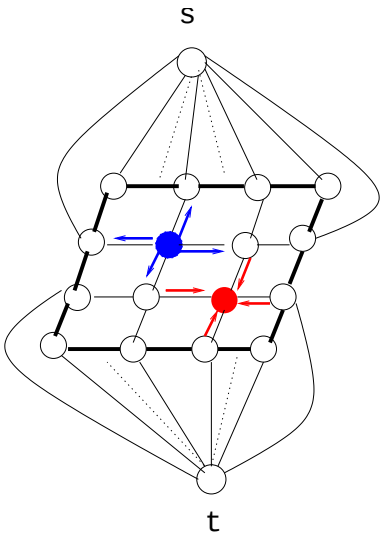

Figure 1. A 4×4 grid graph. Push kernel pushes flow along edges and pull kernel takes them into each node.

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 $B_1 \times B_2$ threads.

GPUGraphCuts ()

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

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

2.3.3 Stochastic Cut

We notice that most of the pixels get their actual label after a few iterations on all datasets. Figure 6 shows the labels after different numbers of iterations on sponge image. After 10 iterations only 643 pixels are labelled incorrectly for sponge image. Figure 2 and Figure 3 gives an estimate of error (pixels getting incorrect lebels) and energy, respectively, as the computation of graph cuts progresses. Only a few pixels exchange flow with their neighbours later. Processing nodes which are unlikely to exchange any flow with their neighbors results in inefficient utilization of the resources. We also explore the number of blocks that are active after each iteration. An active block has at-least one pixel which has exchanged flow in the previous iteration. The activity is determined based on the change in n edge-weights and t edge-weights in the previous iteration. The kernel marks whether each block is active. Based on the active bit, the kernel executes the other parts of the program. Figure 4 gives an illustration of the grid when some blocks are active and some are inactive. It is observed that after a few iterations, only 5-10% of the blocks are active, as shown in Figure 5. We delay the processing of a block based on its activity bit. A better model is to delay the processing of a block based on the likelihood and prior information, but we settle for a fixed delay for inactive blocks. We check the activity of a block after each 10 iterations. A block is processed for next 10 iterations if its active otherwise the block is not processed.

*StochasticCut (*node *u)*

1. Check the active bit of the block.
2. Perform step 2 or 3 of *GPUGraphCuts()* every iteration on all above blocks and every Kth iteration on inactive blocks.

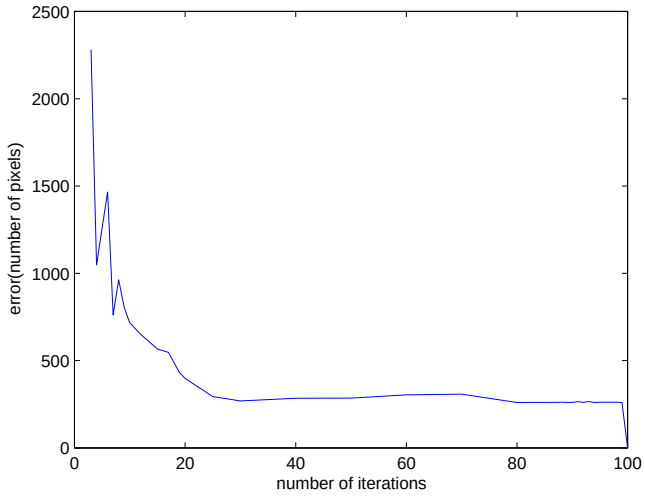

Figure 2. The plot shows the error(pixels) with iterations for Sponge image.

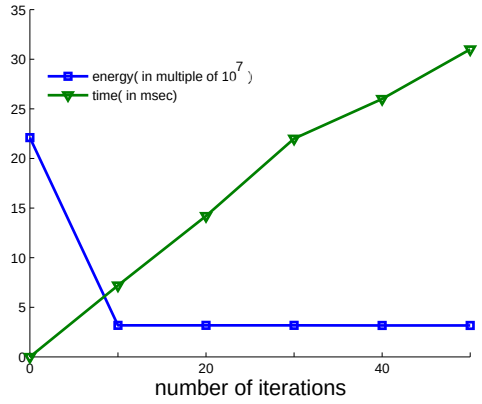

Figure 3. The plot shows the energy vs. iterations and time vs. iterations plot for Sponge image. It gives an estimate of energy drop as the graph cuts computation progresses.

3. Different Levels of Optimizations

As the GPU has lower memory bandwidth, reducing global memory access is critical to performance. We explored the impact on the running time of different compact representaion. The compact versions have to be split into constituent terms after reading. The active flag takes values: $0, 1$ or 2 and 2 bits are sufficient to store them. We can compact 16 active bits in one word. In practice, heights can be represented using 16 bits or even 8 bits and edge weights using 32 bits or 16 bits. We also explored combining the flag bits along with the edge weights and heights. Some instances of data structure at a node are shown in Figure 7.

Table 2 and Table 3 show the different parameters (inco-

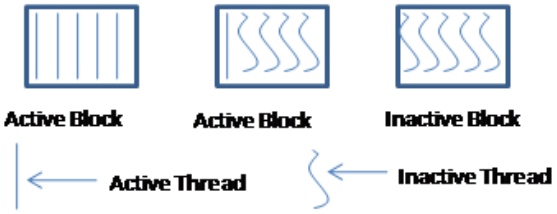

Figure 4. Active Vs Inactive Blocks as in Stochastic Cuts

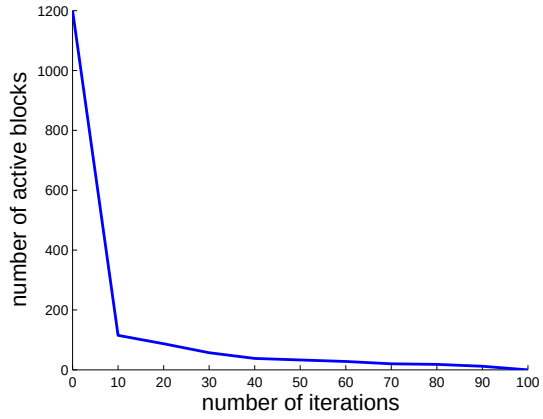

Figure 5. It shows the number of active blocks with the number of iterations of the graph cuts for sponge image.

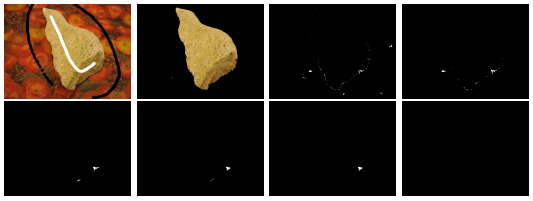

Figure 6. First two images are the sponge image and the segmented image. The other images shows the errors after 10, 20, 40, 80, 90, 100 iterations of graph cuts for Sponge Image.

herency, occupancy, shared memory uses, register counts) which effects the performance on the GPU. Table 5 shows the effect of compact representation. These tables show interesting results on 8800 and GTX 280, as far as global reads and global writes are considered. The Table 2 shows that there are almost 14% incoherent reads and 23% incoherent writes for *Non-Atomic CudaCuts* on 8800. However, there is no incoherent reads and writes on GTX 280. There is always a tradeoff between the shared memory used per block and the register count per thread. These two factors decide the occupancy. As we try to compact more data in a 32 bit word, the efficiecy decreases. When heights are in 8 bits and edgeweights are in 16 bits, we get the worst performance. Compacting the data reduces the global memory accesses at the cost of higher number of computations due

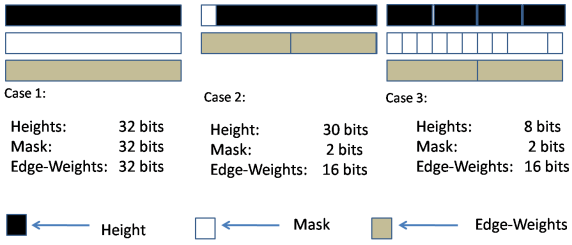

Figure 7. Data Structure at a node: Each node has height, mask and edge-weights. Each height value can be stored in 32 bits, 30 bits, 8 bits. Each mask can take 32 bits or 2 bits. Edge-weights are stored in 32 bits or 16 bits. Case 1: Height, mask and edge-weights are stored in 32 bits. Case 2: Height and mask are compressed in a single 32 bits word with height taking 30 bits and mask taking 2 bits. Edge-weights are stored in 16 bits. Case 3: Four height values are stored in a single 32 bits word with each taking 8 bits. Similarly, 16 masks are compressed in 32 bits with each mask taking 2 bits. Each edge-weight is stored in 16 bits.

to the shifting of the data. The register count per thread is also increased, which reduces the occupancy and so the efficiency. When height, edge-weights and mask all use 32 bits word, we get the best performance on GTX 280. Table 5 shows timings of two different implementations discussed in the previous section on the sponge image of size 640×480 on GTX 280.

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 different kernels speeds up the operations by over 20%. Heights can be stored in one-dimensional or two-dimensional shared memory block. Storing heights in one-dimensional block is efficient. We use a logical OR of the active bit of each node to check the termination condition. Logical OR is evaluated by all active nodes writing a 1 to a common global memory location. Though CUDA model doesn't guarantee an order of execution, OR can be computed quickly.

The push-relabel algorithm can be modified to perform m push operations before each relabel operations. The experimental results show if relabeling is done every other iteration, the speed increases. However, that does not extend to the higher values of m. The multiple push operations before each relabel operation exhausts excess flow quickly. In this way, the algorithm converges in fewer number of iterations. When there is bias towards data term, higher values of m will get efficient performance. Otherwise, value of m should be kept lower. The cuda cuts on flower image converges in 90 iterations when $m = 1$ and in 65 iterations when $m = 2$. Table 6 shows the effect of varying m on timings and number of iteration for convergence of the algorithm on flower image.

Kernel	Occupancy	Incoh(%) Load	Coh(%) Load	Incoh(%) Store	Coh(%) Store	Shared(Bytes) Memory Used	Registers Used
8800_Push	1	23.3	76.7	72.5	27.53	1448	9
280_Push	1	0	100	0	100	1448	9
8800_PullRelabel	0.67	0	100	0	100	1532	16
280_PullRelabel	1	0	100	0	100	1532	16

Table 2. Non-atomic: heights, edgeweights, masks are stored in a word. Push and Pull operations are in separate Kernel.

Kernel	Occupancy	Incoh(%) Load	Coh(%) Load	Incoh(%) Store	Coh(%) Store	Shared Memory Used	Registers Used
280_PushPull	1	0	100	0	100	1532	10
280_Relabel	1	0	100	0	100	1532	9

Table 3. Atomic: heights, edgeweights, masks are stored in a word. Push and Pull Kernels are combined.

Image	Size	Non-Atomic	Atomic
Sponge Image	640×480	61	49
Flower Image	608×456	73	51
Person Image	608×456	81	77

Table 4. The timings on standard images and synthetic image on GTX 280. Each push is followed by a relabel operation.

No. Of bit (ht/edgeweights/mask)	Occupancy	Shared Memory Used	Registers Used	Time(in ms) $(m = 1)$
32/32/32	1/1	1360/1360	13/10	49
32/32/2	1/1	2384/1360	13/10	51
30/32/2	1/1	2384/1360	13/10	54
30/16/2	0.75/1	2384/1360	20/10	56
8/32/32	1/1	1360/1360	13/10	56
32/16/32	0.75/1	1360/1360	20/11	52
16/16/32	0.75/1	1360/1360	20/11	67
8/16/32	0.75/1	1360/1360	20/11	67
8/16/2	0.5/1	2384/1360	23/11	73

Table 5. The table evaluates different parameters which determine the efficiency of implementation on the sponge image on GTX 280. First column gives the different possible combinations of heights, edgeweights and masks in one word. The second, third and forth columns give the occupancy, shared memory used and register used per thread respectively, for PushPull kernel and Relabel kernel as in Atomic case.

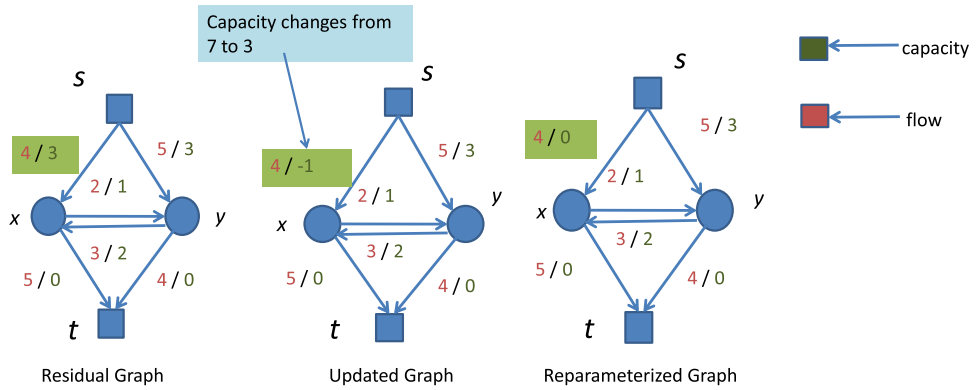

Figure 8. The graph updation and reparameterization scheme for change in weights as in dynamic graph cuts.

When using atomic CUDA Cuts, performing 2 pushes before each relabel performs the best. However, starting with $m = 1$ and increasing it to 2 after about 40 iterations performs the best on non-atomic CUDA Cuts. Table 7 gives timings after these optimizations.

m	Number of iteration	Time (ms)
1	231	77
2	187	64

Table 6. Comparison of running times of CUDA Atomic implementation without stochastic operations on GTX 280 when value of m is changed on person image.

3.1. Dynamic Graph Cuts

Repeated application of graph cuts on graphs for which only a few edges change weights is common in applications like segmenting frames of a video. Kohli and Torr describe a reparametrization of the graph that maintains the flow properties even after updating the weights of a few edges [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 8). These operations assign new weights/capacities as a modification of the final graph without violating any constraints. The frame-to-frame change in weights is computed for each edge first and the final graph from the previous iteration is reparametrized using the changes. It finds the pixels which change their labels with respect to the previous frame. This operation is performed in kernels in parallel. The two basic operations, updation and reparameterizations, are performed by these kernel. So, the maxflow algorithm terminates quickly on them, giving a running time of few milliseconds per frame. The running time depends on the percentage of weights that changed.

4. Experimental Results

The CUDA Cuts algorithm was tested on several standard and synthetic images. The running time also depends on the number of threads per block as it determines the level of parallelism. We experimented with different numbers of threads per block. A block size of 32×8 threads gives the best results with 256 threads per block.

We tested our implementations on various real and synthetic images. Figure 9 shows the results of image segmentation on the Person image, Sponge image and the Flower image. The energy terms used are the same as those given in the Middlebury MRF page [31]. It also shows the results of image segmentation on a noisy synthetic image. The running times for these are tabulated in Table 7 along with the time for Boykov's sequential implementation of graph cuts. The reported times of the GPU algorithm does not include the time to compute the edge weights. Figure 11 plots the running times on a noisy synthetic image of CUDA Cuts and the sequential graph cuts for different image sizes.

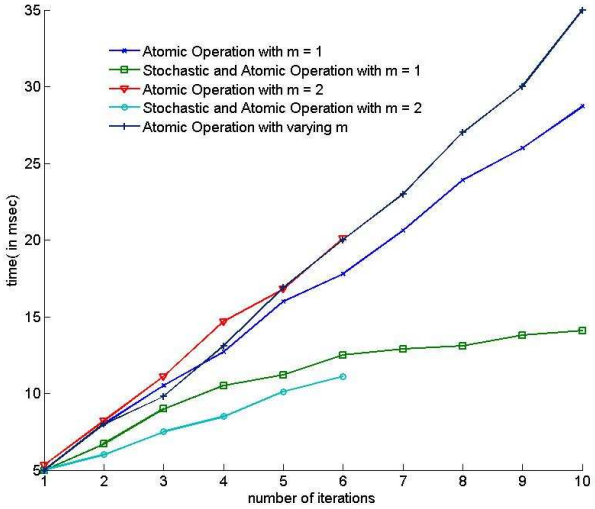

Figure 10. The plot compares the performance of different methods for Sponge image as graph cuts computation progresses on GTX280.

The figure 10 evaluates different optimization methods on GTX280 on sponge image. The stochastic cuts performs the best when $m = 2$.

Figure 12 shows the results of independent segmentation of the frames of a video using our implementation of dynamic graph cuts. The frame-to-frame change in weights is computed for each edge first and the final graph from the previous iteration is reparametrized using the changes. The CUDA implementation of the dynamic graph cuts is efficient and fast. It finds the pixels which change their labels with respect to the previous frame. This operation is performed in kernel in parallel. The two basic operations, updation and reparameterizations, are performed by this kernel. So, the maxflow algorithm terminates quickly on them, giving a running time of 4 milliseconds per frame. 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. We carefully divide the task among the multiprocessors of the GPU and exploit its shared memory for high performance. We perform over 90 graph cuts per second on 640×480 images. This is 10-12 times faster than the best sequential algorithm reported. More importantly, since a graph cut takes only 30 to 40 milliseconds, it can be applied multiple times on each image if necessary, without violat-

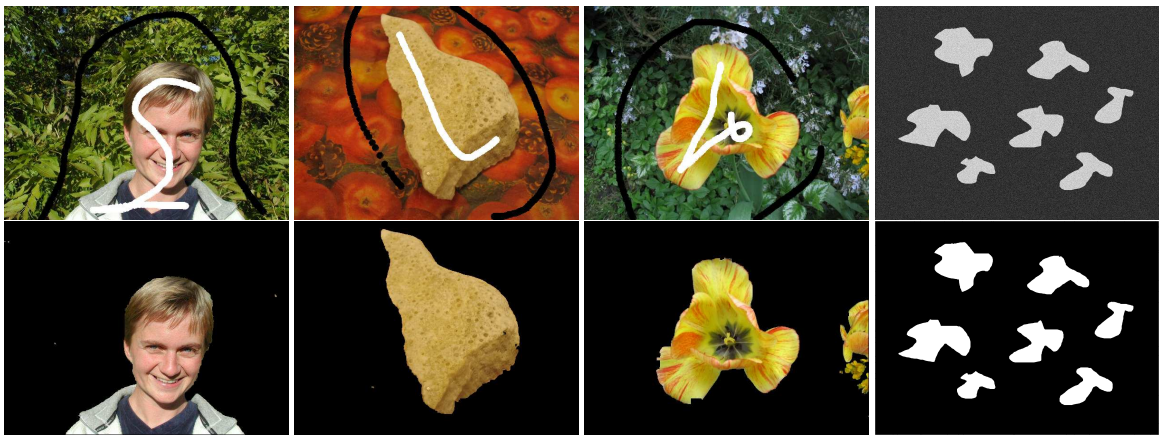

Figure 9. Binary Image Segmentation: Person, Sponge, Flower, and Synthetic images

Image	GC Time(ms) BK	GC Time(ms) Non-atomic	GC Time(ms) Atomic	GC Time(ms) Stochastic	Graph Construct CPU/GPU Time(ms)	TotalTime CPU/GPU (ms)
Flower	188	73	51	37	60/0.15	248/37.15
Sponge	142	61	49	44	61/0.151	203/44.15
Person	140	81	64	61	60/0.15	201/61.15
Synthetic	480	39	37	33	170/1.2	650/34.2

Table 7. Comparison of running times of CUDA implementations on GTX 280 with that of Boykov on different images. Non-atomic: Pull and Relabel kernels are combined into one kernel. Atomic: Push and Pull kernels are combined into one. Stochastic: Atomic functions are applied along with the stochastic operations.

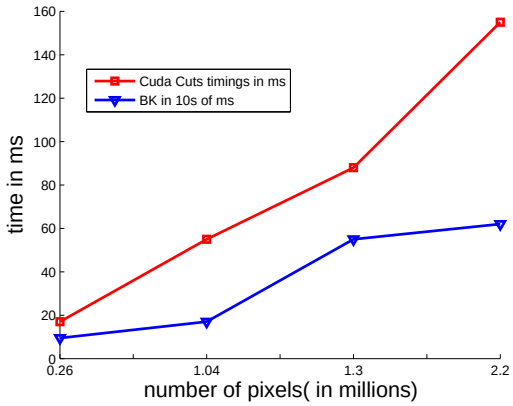

Figure 11. Comparing the running times of graph cuts on the GPU and the CPU for synthetic images.

ing 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.

Figure 12. Segmenting frames of a video using dynamic graphs

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.