On Implementing Graph Cuts on CUDA

Mohamed Hussein Amitabh Varshney Larry Davis
Department of Computer Science
University of Maryland
1103 A. V. Williams Building
College Park, Maryland 20742
{*mhussein, varshney, lsd*}*@cs.umd.edu*

***Abstract*—The Compute Unified Device Architecture (CUDA) has enabled graphics processors to be explicitly programmed as general-purpose shared-memory multi-core processors with a high level of parallelism. In this paper, we present our preliminary results of implementing the Graph Cuts algorithm on CUDA. Our primary focus is on implementing Graph Cuts on grid graphs, which are extensively used in imaging applications. We first explain our implementation of breadth first search (BFS) graph traversal on CUDA, which is extensively used in our Graph Cuts implementation. We then present a basic implementation of Graph Cuts that succeeds to achieve absolute and relative speedups when used for foreground-background segmentation on synthesized images. Finally, we introduce two optimizations that utilize the special structure of grid graphs. The first one is *lockstep BFS*, which is used to reduce the overhead of BFS traversals. The second is *cache emulation*, which is a general technique to regularize memory access patterns and hence enhance memory access throughput. We experimentally show how each of the two optimizations can enhance the performance of the basic implementation on the image segmentation application.**

I. Introduction

In recent years, graphics processing units (GPUs) have been progressively and rapidly advancing from being specialized fixed-function to being highly programmable and incredibly parallel computing devices. With the introduction of the Compute Unified Device Architecture (CUDA), GPUs are no longer exclusively programmed using graphics APIs. In CUDA, a GPU can be exposed to the programmer as a set of general-purpose shared-memory Single Instruction Multiple Data (SIMD) multi-core processors. The number of threads that can be executed in parallel on such devices is currently in the order of hundreds and is expected to multiply soon. Many applications that are not yet able to achieve satisfactory performance on CPUs can get benefit from the massive parallelism provided by such devices.

In this paper we present our preliminary results and findings on implementing the max-flow/min-cut algorithm (referred to just as *Graph Cuts* almost throughout the paper) on CUDA. The problem is defined as follows. Let $\mathcal{G}$ be a graph $\langle \mathcal{V}, \mathcal{E} \rangle$, where $\mathcal{V}$ is a set of nodes, and $\mathcal{E}$ is a set of links. Let s and t be two designated terminal nodes in $\mathcal{V}$. An s/t cut in $\mathcal{G}$ is a partitioning of $\mathcal{V}$ into two disjoint subsets $\mathcal{S}$ and $\mathcal{T}$ such that $s \in \mathcal{S}$ and $t \in \mathcal{T}$. Figure 1 shows an example of an s/t cut of a sample graph. Let $w(u, v)$ be a cost function that assigns a real value to every link $(u, v) \in \mathcal{E}$. The cost of a cut $\mathcal{C} = (\mathcal{S}, \mathcal{T})$ is defined as $c(\mathcal{S}, \mathcal{T}) = \sum_{u \in \mathcal{S}, v \in \mathcal{T}} w(u, v)$, which is basically the sum of the costs of all links linking a node in $\mathcal{S}$ to a node in $\mathcal{T}$. In Fig. 1, the cost of the shown cut is the sum of the costs of the thick links. The Graph Cuts algorithm finds the minimum cut in a graph, which is a cut with a minimum cost value. A cut in the graph defines a binary labeling over its nodes. If we are not interested in the cost of the minimum cut itself and interested instead in the best binary labeling of graph nodes according to some energy function, it can be shown that Graph Cuts can be used to exactly minimize a wide class of functions of binary variables, and approximately minimize a wide class of functions of discrete variables in general [1].

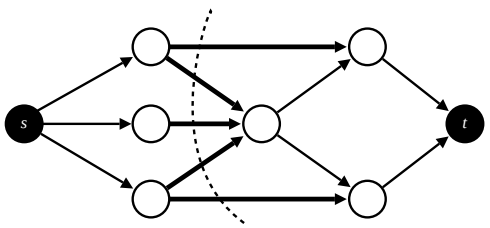

Fig. 1. An example of an s/t cut in a graph. Terminal nodes s and t are marked in black. The cost of the cut is the sum of the weights of the thick links, which are the links that connect nodes in $\mathcal{S}$ to nodes in $\mathcal{T}$.

The Graph Cuts algorithm has many applications in different areas of research. It is a fundamental graph algorithm thereof some other graph algorithms can be modeled as special cases, such as shortest paths, and bipartite graph matching. However, our focus and primary background are on applications related to computer vision, computer graphics, and machine learning. For example, in computer vision, binary labeling via Graph Cuts was applied in foreground/background segmentation such as in [2], [3], where labels are either foreground or background. Discrete labeling was applied in many other applications such as image restoration [4], where labels are discrete intensity values, stereo matching [5], where labels are discrete disparity values, and multi-camera scene reconstruction [6], where labels correspond to different scene elements. In computer graphics there are many applications as well. In [7], Graph Cuts was applied to interactive PhotoMontage, where different images for the same scene can be combined to form a better image based on interactively determined user's criteria. In [8], a method for texture synthesis using Graph Cuts was proposed. Authors of [9] proposed a method for labeling objects of interest in images, called SmartLabel, based on

Graph Cuts. An example application of Graph Cuts in machine learning is in [10], where a method for labeling a large unlabeled dataset based on a small labeled one, via Graph Cuts, was proposed. Among these many applications, we selected image segmentation [2] to evaluate our implementation on. That is primarily due to the simplicity of its implementation as well as non-reliance on field specific concepts. However, it is important to emphasis that our implementation of Graph Cuts is general and is not targeted to any particular application.

While fast practical parallel implementations for Graph Cuts have already been accomplished before [11], [12], to the best of our knowledge, our implementation is the first implementation on CUDA. It is also the first implementation on graphics processors that succeeds to achieve relative and absolute speedups for the type of graphs targeted in this paper. An implementation on CUDA is particularly important due to the relatively low cost of CUDA-enabled devices. Moreover, many of the applications of Graph Cuts, as mentioned above, are targeted to desktop applications, e.g. [3], [7], where assuming existence of many CPUs or clusters of computers is not practical, while existence of a CUDA-enabled device is almost guaranteed.

The simplicity of CUDA's programming model, in fact, projects a number of challenges on implementing Graph Cuts on it compared to the other architectures on which Graph Cuts implementations were studied before. For example, mechanisms for memory locking, to prevent concurrent updates to the same memory location, cannot be taken for granted in CUDA [1]. Also, multiprocessors in CUDA work in a SIMD fashion, where best performance is achieved when all core processors perform the same operation at the same time. Divergence among core processors in the same multiprocessor results in serialization of the different paths taken and hence can cause a large performance penalty.

Our implementation addresses these issues by taking a rather unusual way of implementing Graph Cuts in parallel. Our algorithm is a push-relabel style algorithm [14]. Instead of implementing the global relabeling heuristic in parallel with push and relabel operations, such as in [11], [12], we exclusively rely on BFS graph traversals to assign optimal labels to graph nodes, as explained in section V.

Although implementation of Graph Cuts on general graphs is our ultimate goal, in this paper we focus only on grid graphs. In fact, implementing graph algorithms, such as Graph Cuts, where the complexity of processing a node in the graph is a function of its degree, is not straight forward on SIMD architectures, such as CUDA, where divergence among different processors has to be avoided as much as possible. Grid graphs have the attractive property of having a constant out-degree for almost all nodes in the graph. Therefore, the number of operations performed when processing a node in the graph is generally the same as processing any other node, which almost eliminates divergence. Moreover, the special structure of grid graphs allows us to apply two optimization techniques. The first technique is the *lockstep BFS*, which is utilized to mitigate the overhead of our CUDA implementation of BFS traversal. The second is *cache emulation*, which is a general technique to regularize memory access patterns. Restricting our implementation to grid graphs by no means nullifies its utility. Almost all the applications of Graph Cuts mentioned above, and many others, work on grid graphs.

The rest of the paper is organized as follows. Section II summarizes related work. In section III, an overview of CUDA is presented. Section IV explains our CUDA implementation of BFS graph traversal. Section V presents our approach for computing Graph Cuts and its parallel implementation on CUDA. Then, section VI introduces performance improvements that utilize the special structure of grid graphs. Section VII contains performance results. Finally, the paper is concluded in section VIII

II. Related Work

In [15], the first parallel algorithm for Graph Cuts was introduced. It was based on the augmenting paths approach. As many other researchers, our implementation is primarily based on the push relabel approach since it is more natural to implement in parallel. In [16], the first parallel algorithm based on push-relabel techniques was introduced and was implemented on a connection machine. The first parallel implementation on a shared memory architecture was introduced in [12], where they performed global relabeling concurrently with the main push and relabel operations. Recently, an extended and enhanced version of the same approach was implemented on a modern SMP architecture [11]. These implementations differ significantly from ours since they assume availability of a memory locking mechanism and assume asynchronous operation of different processors. In [17], the Graph Cuts algorithm was implemented on older GPUs. However, it was much slower than the CPU implementation.

III. Compute Unified Device Architecture

We briefly present the main features of CUDA. For a detailed description, refer to [13].

A. Hardware Architecture

In CUDA terminology, the GPU is called the *device* and the CPU is called the *host*. A CUDA device consists of a set of multi-core processors. Each multi-core processor is simply referred to as a *multiprocessor*. Cores of a multiprocessor work in a SIMD fashion. All multiprocessors have access to three common memory spaces. They are:

1) *Constant Memory:* A read only cached memory space.
2) *Texture Memory:* A read only cached memory space that is optimized for texture fetching operations.
3) *Global Memory:* A read/write non-cached memory space.

[1]Recently, atomic functions were introduced in CUDA devices with compute capability 1.1, namely, GeForce 8600 and 8500 series. However, the number of multiprocessors in these devices is much less than what is available in devices of compute capability 1.0, which do not support atomic functions yet [13].

Besides the three memory spaces that are common among all multiprocessors, each multiprocessor has an on chip *shared memory* space that is common among its cores. Furthermore, each core has an exclusive access to a read/write non-cached memory space called *local memory*.

Accessing constant and texture memory spaces is as fast as accessing registers on cache hits. Accessing shared memory is as fast as accessing registers as long as there is no bank conflict. On the other hand, accessing global and local memory spaces is much slower, typically two orders of magnitude slower than floating point multiplication and addition.

B. Execution Model

The execution is based on *threads*. A thread can be viewed as a module, called a *kernel*, that processes a single data element of a data stream. Threads are batched in groups called *blocks*. The group of blocks that executes a kernel constitutes one *grid*. Each thread has a two dimensional index that is unique within its block . Each block in a grid in turn has a unique two dimensional index. Knowing its own index and the index of the block in which it resides, each thread can compute the memory address of a data element to process.

A block of threads can be executed only on a single multiprocessor. However, a single multiprocessor can execute multiple blocks simultaneously by time slicing.

Threads in a block can communicate with one another via the shared memory space. They can also use it to share data fetched from global memory. There is no means of synchronization among threads in different blocks. The number of threads within a block that can execute simultaneously is limited by the number of cores in a multiprocessor. A group of threads that execute simultaneously is called a *warp*. Warps of a block are concurrently executed by time slicing.

C. Some Considerations

There are some important considerations that need to be taken into account to obtain good performance on CUDA. We refer to some of them later on in the paper.

- *Effect of Branching:* If different threads of a warp take different paths of execution, the different paths are serialized, which reduces parallelism.
- *Global Memory Read Coalescing:* Global memory reads from different threads in a warp can be coalesced. To be coalesced, the threads have to access data elements in consecutive memory locations. Moreover, addresses of all data elements must follow the memory alignment guidelines. Details are in [13].
- *Shared Memory Bank Conflict:* Reading from shared memory is as fast as reading from registers unless a bank conflict occurs among threads. Simultaneous accesses to the same bank of shared memory are in most cases serialized.
- *Writing to Global Memory:* In CUDA, two different threads, in the same warp, can write simultaneously to the same address in global memory. The order of writing is not specified, but, one is guaranteed to succeed.

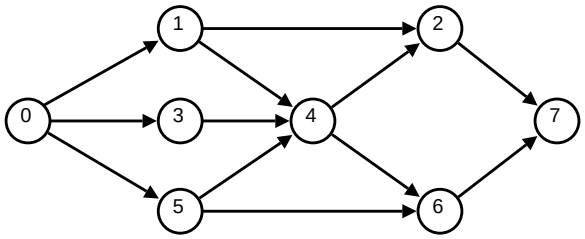

Fig. 2. An example graph.

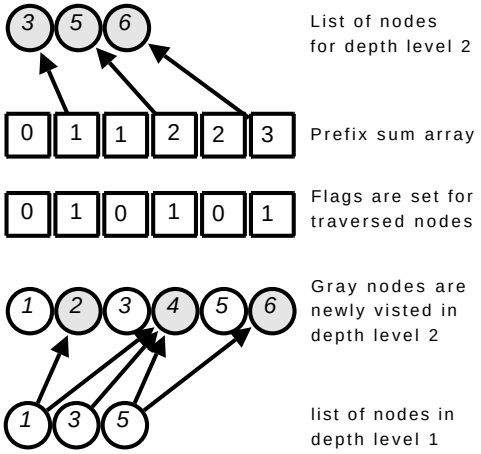

Fig. 3. Traversal of depth level 2 for the graph in figure 2 if traversal starts from node 0: creation of a list of nodes visited in depth level 2 given the list of nodes of depth level 1. Arrows at the bottom indicate traversals. Arrows at the top indicate moving nodes to their positions in the new list. Note that prefix sum values for traversed nodes correspond to their locations in the newly created list.

IV. Parallel BFS Graph Traversal on CUDA

Let $\mathcal{G}$ be a graph $\langle \mathcal{V}, \mathcal{E} \rangle$. In breadth first search graph traversal, we start from a designated node in the graph $s \in \mathcal{V}$. We visit all nodes at a specific depth level from s before visiting any node in the next depth level. The output of the algorithm is a label for each node reachable from s, that specifies its depth level with respect to s. In our implementation, to visit nodes at depth level $k+1$, we start with a list of nodes at depth level k. All nodes at level k are processed in parallel. In processing each node, all its neighbors that have not been visited yet are marked to be added to level $k+1$. After finishing this process, we end up having an array of flags each element therein indicates whether the corresponding node in the graph is added to level $k+1$ or not. The size of this array is n, the number of nodes in the graph. To compact this list of flags and construct the list of nodes for level $k+1$, we use a parallel prefix sum operation [18]. Figure 3 illustrates this operation for traversing depth level 2 of the graph in figure 2

This approach is not work efficient since the work complexity of visiting one level of the graph is $\Theta(n)$, which is the complexity of performing the prefix sum operation on n elements. That makes the overall complexity of the BFS traversal $O(n^2)$. In fact, it may be much more efficient to implement the BFS traversal on the host (the CPU) instead of performing

it inefficiently on the device. But, the overhead of transferring the results from the host to the device may be much more significant than the overhead of performing the traversal on the device. Therefore, we selected to implement the traversal on the device. In section VI, we introduce an optimization for grid graphs that reduces the overall complexity of the traversal to $\Theta(n)$.

V. Parallel Graph Cuts on CUDA

We first give an algorithmic background to Graph Cuts and then explain our approach to implement the algorithm in parallel on CUDA.

A. Background on Graph Cuts Algorithms

In a fundamental theorem in graph theory, Ford and Fulkerson [19] proved the duality between finding the maximum flow that can be pushed from a source node s to a target node t in a graph, and finding the minimum s/t cut in that graph. Based on this theorem, algorithms for solving the Min Cut problem typically do that by solving the dual problem, the Max Flow problem. In the Max Flow terminology, the cost of a link w is referred to as its *capacity*.

There are two main approaches to finding the maximum flow in a network, the *augmenting paths* approach [19], and the *push-relabel* approach [14]. We briefly explain the basic idea of the two approaches. We elaborate more on the push relabel approach since we believe it is more appropriate to implement in parallel, and our algorithm is based on it. In both approaches, a residual graph is constructed and used throughout the algorithm. A residual graph $\mathcal{G}_f$ of a graph $\mathcal{G}$ is a graph that has the same layout as $\mathcal{G}$, but the capacities of its links are *residual capacities*. The residual capacity $w_f(u,v)$ of a link (u,v) after pushing flow $f(u,v)$ through it is $w(u,v) - f(u,v)$, where $w(u,v)$ is the capacity of the link (u,v).

1) Augmenting Paths: An augmenting-paths style algorithm tries to find a path from the source to the target in the residual graph and then pushes the maximum possible flow through that path. The algorithm continues until no path remains from the source to the target. The differences between augmenting paths algorithms lie mainly in the way they select the path through which to push.

2) Push Relabel: A push-relabel style algorithm assigns to each node in the graph an *excess* value and a *label*. The excess of a node is the total amount of flow it received from its neighbors minus the total amount of flow it sent to its neighbors. Initially all nodes have excess 0 except for those nodes that have links coming from the source. Each of the latter nodes initially has excess equal to the capacity of the link coming from the source. The label of a node is a non-negative integer that underestimates the node's distance to the target (in terms of link count.) Initially all nodes have label 0, except for s that is given label n, where n is the number of nodes in the graph. The algorithm alternates between two operations, push and relabel:

- *Push:* The push operation applies to a node u in the graph if u has positive excess. If u has label k, the push operation finds a neighbor v of u such that v has label $k-1$ and the link (u,v) has positive residual capacity. If such a node exists, the maximum possible flow is pushed from u to v. That push results in updating the excesses of u and v as well as the residual capacities of the links (u,v) and (v,u). After a push operation, either u loses all its excess or the link (u,v) is saturated. The criterion of selecting v can be understood based on the interpretation of a node's label as an estimate of the distance to the target. The push operation basically tries to push flow towards the target through a node that is one step closer. It does that relying only on local information of a node and its neighbors.
- *Relabel:* The relabel operation applies to a node u if u has positive excess and has outgoing links but a push operation does not apply. That happens when all outgoing links from u are towards nodes with labels greater than or equal to that of u. The relabel operation tries to enable u to eliminate its excess by increasing its label to the minimum possible value that makes a push operation applicable.

The algorithm can apply push and relabel operations in any order until none of them applies, which happens when all nodes have 0 excess. It is guaranteed that either of the two operations must apply for a node with positive excess [20]. Upon termination, all extra excess that was initially pushed from the source to its neighbors and did not find its way to the target will have been pushed back to the source. The algorithm is guaranteed to terminate in $O(mn^2)$ time regardless of the order in which push and relabel operations are applied, where n is the number of vertices and m is the number of links in the graph. However, it turns out that the order of such operations has a great impact on the performance of the algorithm in practice. Actually, the differences among push relabel methods lie mostly in the way this order is determined and the way nodes are labeled.

B. Our Approach to Implementing Graph Cuts on CUDA

One might think that a parallel implementation can process all nodes in parallel, and for each node with positive excess if a push operation applies it is performed, otherwise a relabel is performed. However, on CUDA, we would like to make all processors perform the same operation at the same time to avoid divergence. Also, a node cannot push flow and receive flow pushed to it at the same time since both operations update its excess value. Since, we do not assume any memory locking mechanism, we have to find a different way to prevent concurrent updates to the same value. In the following we explain how the push and relabel operations can be adapted to overcome these difficulties.

1) Parallel Labeling: As explained above, the label of a node is an underestimate of its distance to the target. But, in practice, relying only on the basic relabel operation results in very poor estimates and makes flow go back and forth between

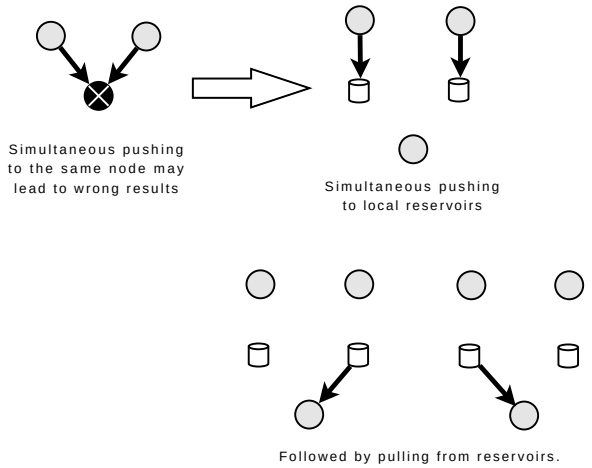

Fig. 4. Dividing parallel pushing into two steps.

```
procedure ParallelPush(LabelsLattice)
    EmptyReservoirs
                          ▷ initializes all node reservoirs to 0
    Level ← PopTopLevel(LabelsLattice)
    ParallelPushToLowerLevel(Level)
    while NumLevels(LabelsLattice) > 1 do
        Level ← PopTopLevel(LabelsLattice)
        ParallelPullFromUpperLevel(Level)
        ParallelPushToLowerLevel(Level)
    end while
    Level ← PopTopLevel(LabelsLattice)
    ParallelPullFromUpperLevel(Level)
    ParallelPushToTarget(Level)
end procedure
```

Fig. 5. Pseudo-code for the parallel push operation.

nodes many times before eventually reaching the target. That dramatically slows down the algorithm. A heuristic that was proposed to enhance the running time is *global relabeling* [14]. In this heuristic, the algorithm is frequently stopped and all nodes are labeled with their actual distances to the target. This is accomplished using a backward breadth first search traversal starting from the target. In our proposed algorithm, we use this heuristic as the only labeling scheme. In other words, this is the only way nodes get their labels. We employ the BFS traversal technique, explained in section IV. The traversal in this case starts from the target, and goes backwards in the graph. In this way, all nodes are optimally labeled in parallel without introducing expensive divergence among processors. Note that applying global relabeling on every iteration eliminates the need for the *gap relabeling* heuristic that was suggested to be combined with global relabeling in [21].

2) Parallel Pushing: The order of applying the push operations also impacts the performance of the algorithm. A heuristic that is used to enhance the performance is to apply push to nodes with higher labels before nodes with lower labels [22]. We apply this heuristic in our implementation. During the labeling phase, we store the nodes that are visited at each depth level in a separate list. This comes almost for free because of the way the breadth first search technique works (section IV). We call the resulting structure the traversal *lattice*. During pushing, we start from the top level of the lattice going downwards to the target. At each level pushing is done in parallel. All nodes in the current level in parallel push flow to their neighbors in the lower level.

Unfortunately, pushing in parallel in this way may not produce correct results. A node cannot receive flow from more than one neighboring node simultaneously. To resolve this problem, we divide the push operation into two phases, *push* and *pull*. In the push phase no node updates the excess of its neighbor. Instead, each node keeps a reservoir for each outgoing link in which it stores the amount of flow it pushes on that link in the current pushing phase. In the pull phase, all nodes at a given depth level in parallel collect flow pushed to them from their neighbors in the upper level by reading the values stored in the appropriate reservoirs. Each node then updates its excess value and residual capacities accordingly. Figure 4 clarifies this operation. The final parallel push algorithm is depicted in the pseudo-code in Fig. 5.

3) Termination Criteria: After pushing flow all the way from the top level of the traversal lattice down to the target, the layout of the graph changes due to saturation of some links. So, the lattice has to be rebuilt before the next pushing phase. Therefore, the whole algorithm works by alternating between parallel labeling and parallel pushing phases. But, when should the algorithm be terminated? There are two conditions; reaching either of them causes the algorithm to terminate.

- *Failure to Construct the Lattice:* That happens when all links to the target are saturated. In this case, the resulting cut is $\mathcal{C} = (\mathcal{S}, \mathcal{T})$ such that $\mathcal{T} = t$ and $\mathcal{S} = \mathcal{V} - \{t\}$.
- *No Excess in Lattice:* In this condition, all nodes in the lattice, i.e. all nodes having at least a path to the target, have no excess. In this case, there is no flow to push down to the target. The cut in such a case is defined as $\mathcal{C} = (\mathcal{S}, \mathcal{T})$ such that $\mathcal{T} = \{u \in \mathcal{V}, u \text{ has a path to } t\}$ and $\mathcal{S} = \mathcal{V} - \mathcal{T}$.

The entire algorithm is depicted in the pseudo-code in Fig. 6. The pseudo-function ParallelBFS performs the BFS traversal and all proper initializations needed for it. The variable $ExcessFlag$ represents whether there is positive excess at any node in the traversal lattice or not. It is set during the parallel labeling phase.

VI. Optimizations for Grid Graphs

There are two main issues with the aforementioned algorithm:

- *Memory Access Pattern:* Either in labeling (building the lattice) or pushing, threads do not share data read from global memory. That is because two adjacent threads in a block could generally be processing two nodes in the graph that have no neighbors in common.

procedure PARALLELGRAPHCUT
  INITNODELABELS ▷ initializes all node labels to 0
  $ExcessFlag \leftarrow 0$
  $LabelsLattice \leftarrow$ PARALLELBFS
  while $ExcessFlag = 1$ **do**
    PARALLELPUSH($LabelsLattice$)
    INITNODELABELS
    $ExcessFlag \leftarrow 0$
    $LabelsLattice \leftarrow$ PARALLELBFS
  end while
end procedure

Fig. 6. Pseudo-code for the parallel Graph Cuts algorithm.

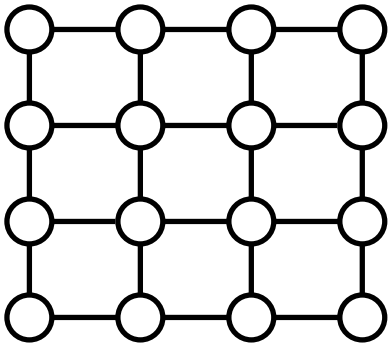

Fig. 7. An example of a grid graph.

- *Prefix Sum:* During constructing the lattice, in the labeling stage, prefix sum operations are performed over the entire set of nodes regardless of the number of nodes actually visited.

We propose two approaches to alleviate the effect of these problems by utilizing the special structure of grid graphs. For simplicity of presentation, we focus on two dimensional grids. The concept can easily be extended to higher dimensional grids. A two dimensional grid graph can be viewed as a matrix of nodes, where each node can be uniquely identified by a two dimensional index specifying its row and column in the graph. A general node in the graph has links with nodes only within a fixed neighborhood surrounding it in the grid. Figure 7 shows a sample 4×4 grid graph. In the following two sections, we explain the two proposed optimizations, *lockstep BFS traversal*, and *cache emulation*.

A. Lockstep BFS Traversal

The technique for BFS graph traversal explained in section IV utilizes an array of flags of length n, and performs prefix sum operation on it on traversing each depth level of the graph. That is important for general graphs, where the number of nodes traversed at each depth level is arbitrary and does not depend on the number of nodes in the preceding level. Also, each node can be traversed from several neighbors at the same time. Therefore, it is important to keep a unique flag for each and every node in the graph for the operation to produce correct results.

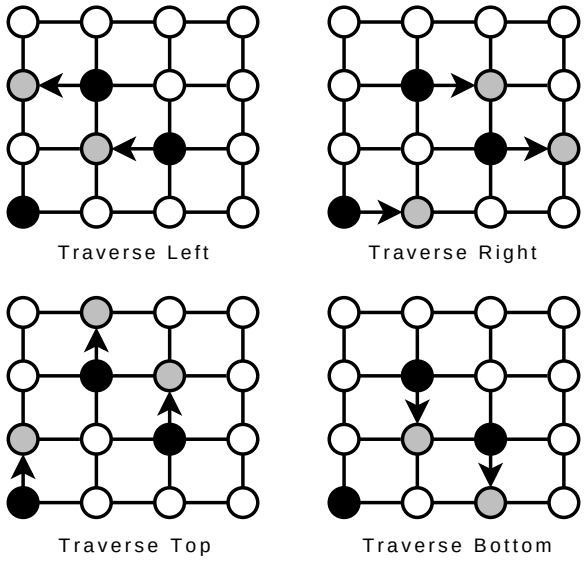

Fig. 8. Black nodes are on the same depth level. The illustration gives an example of how lockstep BFS traversal works to create the next depth level from these nodes. Gray nodes are the nodes traversed in each direction.

In grid graphs, on the other hand, each node has a fixed number of neighbors. Therefore, the number of nodes traversed at each depth level is at most a constant multiple of the number of nodes traversed at the preceding level. Moreover, if we represent the graph in a way where links to neighboring nodes have a fixed order based on their directions (e.g. left, right, top, and then bottom for a 4-connected neighborhood graph,) and during graph traversal only a single direction is traversed at a time, then the number of nodes traversed at a certain direction at a certain depth level is at most the same as the number of nodes traversed at the preceding depth level. Also, performing traversal in this way guarantees that a node is traversed-to from exactly one neighbor. This paradigm of traversal in which only one direction is traversed at a time is what we call the *lockstep BFS* traversal technique. For example, in a 4-connected neighborhood graph, constructing a given depth level is divided into four steps instead of being done all in one step. In each step neighbors along one direction are traversed. Figure 8 gives an illustration of this operation.

When traversing nodes at depth level k from nodes at depth level $k-1$, applying the lockstep BFS traversal technique allows us to use an array of flags whose size is equal to the number of nodes in level $k-1$, n_{k-1}. That reduces the order of work complexity of building level k of the traversal lattice to $\Theta(n_{k-1})$ instead of $\Theta(n)$. The overall complexity of the traversal becomes $\Theta(\sum_k n_k) = \Theta(n)$.

B. Cache Emulation

Because nodes' data – excesses, labels, and outgoing link capacities – are updated and then read during the pushing and labeling phases, we selected to store all nodes' data

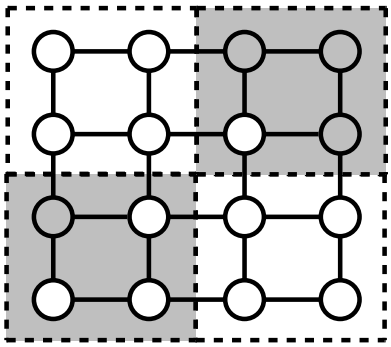

Fig. 9. A 4×4 grid graph divided into 2×2 tiles. Dotted lines indicate tile boundaries. Active nodes and active tiles are shown in gray. A tile is active if and only if it contains at least one active node.

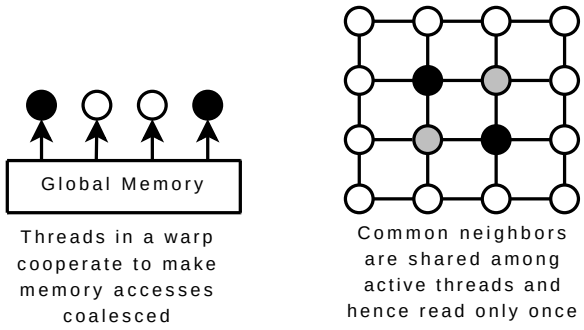

Fig. 10. Black circles represent active nodes. Gray circles are common neighbors to more than one active node. The illustration shows how active and non-active nodes can cooperate and how active nodes can share data to save memory access time.

in the global memory space, which is a read/write space [2]. The problem with the global memory space is that it is not cached. Moreover, threads have to respect a specific memory access pattern in order for reads from global memory to be coalesced (section III-C). Graph algorithms generally exhibit an irregular memory access pattern, which makes requirements for coalescing not guaranteed. The *cache emulation* technique aims at regularizing accesses to global memory by enforcing memory coalescing requirements on global memory accesses. It basically works by emulating the operation of a multi-dimensional cache memory unit. For the technique to work, data accessed have to be structured as a one dimensional or multi-dimensional array. For the technique to be useful, processing a data element has to rely only on its local neighborhood in the array; hence comes the restriction to grid graphs in this case. Unlike a hardware cache memory, whose operation is independent of any algorithm, the cache emulation technique in fact requires modifying the way the algorithm works in order to be used. We will explain the technique in the context of our Graph Cuts implementation on two dimensional grid graphs. Nevertheless, we believe the technique is fairly general and deploying it in other problems is straightforward.

Nodes' data of a two dimensional grid graph are assumed to be stored in two dimensional arrays. To process a node in the graph, we actually load from global memory its data and the data of the 2D *tile* of nodes in which it resides, and process all the nodes in the tile in parallel. In other words, the graph is divided into equally sized *tiles* of nodes. Figure 9 shows a simple 4×4 grid graph divided into 2×2 tiles. The algorithm proceeds exactly as explained earlier. The only difference is in building the breadth first search lattice. For each level of the lattice, instead of constructing a list of nodes, a list of tiles is constructed. Each tile added to a level of the lattice must have at least one node that belongs to that level. Figure 9 depicts the relationship between an active tile and an active node. Activity here means eligibility for the currently performed operation.

[2]In fact, since data updated during a kernel invocation are not read later on during the same kernel invocation, texture memory space can be used as well. However, in the current CUDA implementation this trick is restricted only to 1D arrays, which limits its utility. Also, according to our tentative experiments, the technique presented in this section performs much better.

For example, when pushing flow from level k of the lattice to level $k-1$, nodes that belong to level k are the active nodes, and the tiles containing them are the active tiles. Note that a node that belongs to an active tile is not necessarily active. But, at least one node in tile must be active for the tile to be active.

In the CUDA implementation, each block of threads corresponds to a tile of nodes. Each thread of a block loads data of one node in the tile from global memory to shared memory. Then, if a node is active for the current operation, its thread proceeds and processes the node, otherwise, the thread terminates. For example, if the operation is parallel labeling with label k, a thread processes a node only if it has label $k-1$, otherwise the thread terminates after loading the node's data. If the operation is parallel pushing from level k to level $k-1$ in the lattice, a thread processes a node only if it has label k.

When a thread terminates without processing a node, it actually does a useful job before termination. Indeed it helps in the most time consuming operation. It helps in reducing the overhead of memory access for other threads in the block that are processing active nodes. Specifically, the benefit from this technique is two fold:

- *Efficient Memory Access:* Two factors enhance memory access performance when using cache emulation. One factor is *cooperation* among threads by helping one another to achieve memory read coalescing. All threads in a warp of a thread block read adjacent global memory addresses with proper alignment. As explained in section III-C, that makes these reads coalesced together. The other factor is *sharing* among threads. Data read by one thread are stored in shared memory and become available for other threads in the block. Figure 10 depicts these notions.
- *Less Prefix Sum Overhead:* When using cache emulation, prefix sum operations are performed over tiles not individual nodes. If each tile on average contains b active nodes in the same level of the lattice, the number of elements processed by prefix sums is reduced by a factor of b on average.

Since the computations performed per node during the pushing and labeling phases are very simple computations, the time for global memory access is much more significant. That is proved by the superiority of the implementation that utilizes the cache emulation technique over others, as shown below.

VII. Experimental Results

We experimented our implementations on an image segmentation task. The segmentation algorithm we use is the one in [2]. It basically divide pixels in the image into two partitions, foreground and background, depending on user's input that marks some pixels as foreground and others as background. In the results shown here, we did not use hard constraints, i.e. enforcing some pixels to belong to the foreground or background. We set the weights of singleton and pairwise energy terms to 1 and 2.5, respectively, and the noise value to 50. Only 4-connected neighborhoods are used in the generated graphs.

We compare the running times of five different implementations: a CPU implementation of the proposed technique, a basic CUDA implementation without any optimization, a CUDA implementation using lockstep BFS, a CUDA implementation using cache emulation, and the CPU implementation introduced in [23], which is reported to be the fastest in practice for grid graphs, and whose implementation is available online. For the CPU implementation of our technique, BFS traversal is implemented in the regular sequential way without using prefix sum operations.

For all our CUDA implementations, the graph nodes' data are stored in three two-dimensional arrays of type `float4` (a structure of four floating point elements.) One array contains reservoirs, one array contains outgoing residual link capacities, and the last array contains for each node its excess, label, and link capacities to s and t. The results presented here for the proposed technique when implemented with cache emulation are for tile size of 8×4. We experimented with tile sizes 16×8, 16×12, and 16×16 as well. The best results we got were for the tile size 8×4. This is actually at odds with the recommended block sizes for CUDA devices [13]. This issue needs further investigation.

On running the algorithm on real images, it turned out not to be easy to adjust the user input, and weights for the singleton and pairwise terms to get desirable segmentation results. Alternatively, we report results here for experiments on synthetic images only. Each synthesized image is produced by drawing a foreground with intensity values generated from a Mixture of Gaussian density function, on a background with intensity values generated from a different mixture. Each mixture has three components. The foreground shape of each image is a collection of ellipses in random locations, orientations, and sizes. In each generated image, we enforce the condition that the 32×32 patch at the center of the image belongs to the foreground, and the 32×32 patch at the top left corner of the image belongs to the background. These are the patches based on which foreground and background intensity histograms are constructed. The enforcement here is in the distribution from which the random intensity values for these patches are generated. But, we do not enforce the resulting segmentation to assign these patches specific labels. Each generated image is resized to make a set of six different sizes. We compare the segmentation results from the five implementations and make sure they are exactly the same for each image. Each implementation is run for 10 times on each image. An example image with the resulting segmentation is shown in Fig. 11. The CPU implementations are run on an Intel Xeon 3.2 GHz processor with 1GB RAM. The GPU implementations are run on a GeForce 8800 GTX graphics card.

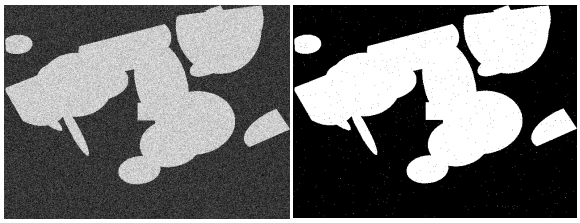

Fig. 11. Left is a sample randomly generated image. Right is its segmentation result.

The plot in Fig. 12 compares the running times of the five implementations with different image sizes. This is the time of running the Graph Cuts on the generated graph 10 times. The time to generate the graph itself is excluded. Also, for the CUDA implementations, the time to transfer the graph data from the host to the device and the time to transfer the result from the device to the host are excluded. That is because the input graph can actually be constructed on the device as well, which should be much faster than constructing it on the host. But, that is not done in our implementation. Also, the resulting cut might be postprocessed on the device, or directly rendered to the screen buffer in some applications.

The implementation of the proposed algorithm on CPU is not always faster than the implementation of Boykov-Kolmogrov's (BK) method [23]. The plot in figure 12 shows also that the basic CUDA implementation, without optimizations, consistently outperforms the two CPU implementations. The two optimizations proposed introduce another considerable speedup over the non-optimized CUDA implementation. The cache emulation technique in particular is consistently the fastest. That emphasis the importance of memory access optimization on such devices. That is particularly important in graph algorithms in general since they are typically memory intensive algorithms.

The plot in Fig. 13 shows the speed up of the CUDA implementation of the proposed algorithm with cache emulation when compared to the faster of the two CPU implementations. The speedups gained are in the range 1.7-4.5, depending on the image size.

VIII. Conclusion

In this paper, we presented our preliminary results and findings on implementing Graph Cuts on CUDA. To address

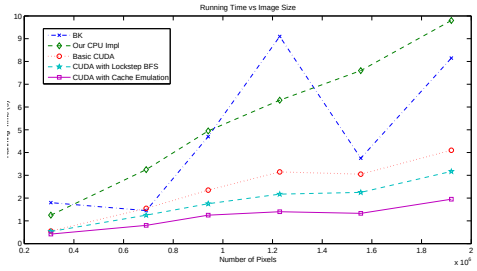

Fig. 12. Comparison of running times vs image size for different implementations of Graph Cuts. *BK* is Boykov-Kolmogriv's method and *Our CPU* is the implementation of the proposed approach on CPU

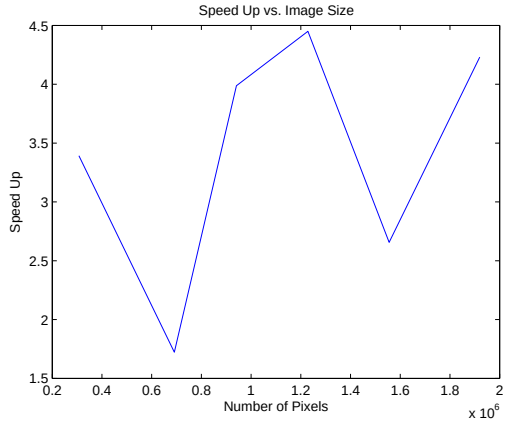

Fig. 13. The speed up of the the proposed method with tiling on CUDA over the faster CPU implementation for each image size.

the unique architectural features of CUDA, we resorted to an unusual way of implementing Graph Cuts in parallel. Our approach relies on BFS traversals solely to assign node labels. This computationally inefficient approach facilitates turning around limitations of CUDA, such as the simple SIMD execution model and the unavailability of memory locking constructs. Nevertheless, the performance of this approach provides both relative and absolute speedups when experimented on image segmentation of synthesized images. We proposed two improvements to this technique that make use of the special structure of grid graphs to deliver better performance, lockstep BFS and cache emulation. The lockstep BFS utilizes the special structure of grid graphs to make BFS traversal implementation on CUDA work efficient. The cache emulation technique is fairly general. It aims at regularizing memory access patterns to enforce memory read coalescing as much as possible through emulating the operation of a cache memory unit. The experimental results showed that the proposed techniques indeed enhance the performance, especially the cache emulation technique. That was expected since graph algorithms in general are memory bound and enhancing memory throughput is crucial to enhance their overall performance.

We are investigating how to enhance the speed further by applying the global relabeling heuristic concurrently with the push and relabel operations. We would like also to experiment our implementation on a wider range of images and imaging applications. In particular, having a fast implementation for Graph Cuts is much more important when the algorithm is applied to general discrete labeling, instead of binary labeling, since in the former the algorithm is actually run for many times until it converges. Therefore, we would like to experiment our techniques on such applications. Finally, we need to further investigate and understand the effect of changing the block size on the performance of the algorithm.

ACKNOWLEDGMENT

This research was supported by the U.S. Government under the VACE project, and the NSF CPU-GPU grant (NSF CNR 04-03313).

REFERENCES

[1] V. Kolmogrov and R. Zabih, "What energy functions can be minimized via graph cuts?" *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 26, pp. 147–159, 2004.

[2] Y. Boykov and M.-P. Jolly, "Interactive graph cuts for optimal boundary & region segmentation of objects in n-d images," in *International Conference on Computer Vision (ICCV)*, 2001.

[3] C. Rother, V. Kolmogorov, and A. Blake, "grabcut interactive foreground extraction using iterated graph cuts," in *SIGGRAPH*, 2004.

[4] Y. Boykov, O. Veksler, and R. Zabih, "Fast approximate energy minimization via graph cuts," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 23, pp. 1222–1239, 2001.

[5] S. Roy and I. Cox, "A maximum-flow formulation of the n-camera stereo correspondence problem," in *Proceedings of the Sixth International Conference on Computer Vision (ICCV)*. Washington, DC, USA: IEEE Computer Society, 1998, p. 492.

[6] V. Kolmogorov and R. Zabih, "Multi-camera scene reconstruction via graph cuts," in *ECCV '02: Proceedings of the 7th European Conference on Computer Vision-Part III*. London, UK: Springer-Verlag, 2002, pp. 82–96.

[7] A. Agarwala, M. Dontcheva, M. Agrawala, S. Drucker, A. Colburn, B. Curless, D. Salesin, and M. Cohen, "Interactive digital photomontage," in *SIGGRAPH*, 2004.

[8] V. Kwatra, A. Schodl, I. Essa, G. Turk, and A. Bobick, "Graphcut textures: Image and video synthesis using graph cuts," in *SIGGRAPH*, 2003.

[9] W. Wu and J. Yang, "Smartlabel: An object labeling tool using iterated harmonic energy minimization," in *ACM Multimedia*, 2006.

[10] A. Blum and S. Chawla, "Learning from labeled and unlabeled data using graph mincuts," in *18th International Conference on Machine Learning*. Morgan Kaufmann, San Francisco, CA, 2001, pp. 19–26.

[11] 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 *18th ISCA International Conference on Parallel and Distributed Computing Systems (PDCS 2005)*, 2005.

[12] R. J. Anderson and J. Setubal, "On the parallel implementation of goldbergs maximum flow algorithm," in *4th Ann. Symp. Parallel Algorithms and Architectures (SPAA-92)*, 1992.

[13] *CUDA Programming Guide ver 1.0*, NVIDIA, 2007.

[14] A. Goldberg and R. Tarjan, "A new approach to maximum-flow problem," *Journal of the Association for Computing Machinery*, vol. 35, pp. 921–940, 1988.

[15] Y. Shiloach and U. Vishkin, "An $o(n^2 \log n)$ parallel max-flow algorithm," *Journal of Algorithms*, vol. 3, pp. 128–146, 1982.

[16] A. Goldberg, "Efficient graph algorithms for sequential and parallel computers," Ph.D. dissertation, MIT, 1987.

[17] N. Dixit, R. Keriven, and N. Paragios, "Gpu-cuts: Combinatorial optimisation, graphic processing units and adaptive object extraction," CERTIS, Tech. Rep., 2005.

[18] Blelloch, "Prefix sums and their applications," in *John H. Reif (Ed.), Synthesis of Parallel Algorithms, Morgan Kaufmann*, 1993. [Online]. Available: citeseer.ist.psu.edu/blelloch90prefix.html

[19] L. R. Ford and D. R. Fulkerson, *Flows in Networks*. Princeton University Press, 1962.
[20] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms*, second edition ed. MIT Press and McGraw-Hill, 2001.
[21] B. Cherkassky and A. Goldberg, "On implementing the push-relabel method for the maximum flow problem," *Algorithmica*, vol. 19, no. 4, pp. 390–410, 1997.
[22] J. Cheriyan and S. N. Maheshwari, "Analysis of preflow push algorithms for maximum network flow." *SIAM Journal on Computing*, vol. 18, p. 10571086, 1989.
[23] Y. Boykov and V. Kolmogorov, "An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 26, pp. 1124–1137, 2004.