

JF-Cut: A Parallel Graph Cut Approach for Large-Scale Image and Video

Yi Peng, Li Chen, Fang-Xin Ou-Yang, Wei Chen, and Jun-Hai Yong

Abstract—Graph cut has proven to be an effective scheme to solve a wide variety of segmentation problems in vision and graphics community. The main limitation of conventional graph-cut implementations is that they can hardly handle large images or videos because of high computational complexity. Even though there are some parallelization solutions, they commonly suffer from the problems of low parallelism (on CPU) or low convergence speed (on GPU). In this paper, we present a novel graph-cut algorithm that leverages a parallelized jump flooding technique and an heuristic push-relabel scheme to enhance the graph-cut process, namely, back-and-forth relabel, convergence detection, and block-wise push-relabel. The entire process is parallelizable on GPU, and outperforms the existing GPU-based implementations in terms of global convergence, information propagation, and performance. We design an intuitive user interface for specifying interested regions in cases of occlusions when handling video sequences. Experiments on a variety of data sets, including images (up to 15 K \times 10 K), videos (up to 2.5 K \times 1.5 K \times 50), and volumetric data, achieve high-quality results and a maximum 40-fold (139-fold) speedup over conventional GPU (CPU)-based approaches.

Index Terms—Graph cut, jump flooding, visibility, segmentation.

I. INTRODUCTION

GRAPH Cut can be employed to efficiently solve a wide variety of graphics and computer vision problems, such as segmentation, shape fitting, colorization, multi-view reconstruction, and many other problems that can be formulated to maximum flow problems [1].

In image segmentation, graph cut is advantageous compared with other energy optimization methods due to its high accuracy and high efficiency. By leveraging incremental or multi-level schemes [2], [3], the complexity of user interactions can be greatly reduced. However, the high computational

complexity limits its usage on high-definition images and videos, especially for real time interactions. Parallelization is an appropriate way to handle this, which motivates our work.

The basic idea of graph cut is to partition the elements into two disjoint subsets by finding the minimum cut using maximum flow algorithms. Existing graph cut methods can be classified into two categories: augmented path based methods that are solely workable in CPU [1], [4], [5]; push-relabel based that can be accelerated with GPU [6], [7]. The former normally requires a global data structure, like priority queue and dynamic trees [8], and thus is intractable for parallelization and handling large-scale data set. In contrast, using the push-relabel scheme is compatible with parallelization, but may lead to slow convergence speed, or yield non-global optimal results. It is also inefficient in tackling large-scale data sets.

Early work on image graph cut, such as lazy snapping [2] and grabcut [9] perform pre-segmentation to reduce the data scale. Analogously, video-cutout [10] introduces a hierarchical mean-shift based preprocess to support interactive segmentation, and volume-cutout [11] takes a watershed over-segmentation. All of them require a time-consuming preprocessing and may lead to approximated results.

Most of the methods use stroke-based interactions to help users specify foreground and background elements. However, for videos and volumetric data, it is hard to directly select the data due to perspective projection and occlusion. A more common way to do data selection is to mark frame by frame which is intuitive and tedious.

This paper introduces a parallel graph cut approach that is capable of handling large-scale image and video with fast convergence speed. The key idea is to accelerate the propagation of flow in the graph. More importantly, the entire process is parallelizable, making it an ideal solution for large-scale data sets. Compared with conventional graph-cut implementations [1], [4], [7], our method achieves a maximum 40-fold (139-fold for CPU) speedup. In summary, the main contributions of this paper are twofold:

- 1) A GPU-based graph cut scheme that combines jump flooding, heuristic push-relabel and convergence detection to achieve high performance and quality;
- 2) An interactive graph cut based data segmentation system that allows for intuitive data selection, interaction and segmentation for large-scale image and video.

In the remainder of this paper, we first take a brief review of the literature in Section II. Our approach is elaborately introduced in Section III, followed by the design of user interactions in Section IV. Experimental results and comparison are

Manuscript received March 24, 2014; revised July 6, 2014 and September 27, 2014; accepted November 20, 2014. Date of publication December 4, 2014; date of current version January 9, 2015. This work was supported in part by the 863 Program of China under Grant 2012AA041606, and in part by the National Science Foundation of China under Grant 61232012, Grant 61272225, Grant 91315302, and Grant 60773143. The associate editor coordinating the review of this manuscript and approving it for publication was Prof. Marios S. Pattichis. (*Corresponding author: Li Chen.*)

Y. Peng is with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China (e-mail: 15pengyi@gmail.com).

L. Chen, F.-X. Ou-Yang, and J.-H. Yong are with the School of Software, Tsinghua University, Beijing 100084, China (e-mail: chenlee@tsinghua.edu.cn; oyfx11@mails.tsinghua.edu.cn; yongjh@tsinghua.edu.cn).

W. Chen is with the State Key Laboratory of Computer Aided Design and Computer Graphics, Zhejiang University, Hangzhou 310027, China (e-mail: chenwei@cad.zju.edu.cn).

Color versions of one or more of the figures in this paper are available online at <http://ieeexplore.ieee.org>.

Digital Object Identifier 10.1109/TIP.2014.2378060

presented in Section V. Section VI concludes our approach and highlights the future work.

II. RELATED WORK

A. Augmenting Path Based Methods

The kernel of graph cut is to solve a maximum flow problem, which is typically addressed by finding augmented paths in the residual network. The pioneered Ford-Fulkerson algorithm has a computational complexity of $O(E \max |f|)$, where E and f denote number of edges and the maximum flow, respectively. It, however, can only handle weights with integer values. Thereafter, Edmonds-Karp algorithm takes the Breadth-First Search (BFS) to find the augmented path with a complexity of $O(VE^2)$, and is feasible for weights with floating point values. Similarly, Dinitz blocking flow algorithm also takes the BFS, but differs from others in that it searches multiple shortest path in each iteration, and has a computational complexity of $O(V^2E)$. The complexity can be further reduced into $O(VE \log V)$ by employing a dynamic tree technique. Further, a complexity of $O(VE)$ can be achieved by combining the binary blocking flow algorithm and the King-Rao-Tarjan (KRT) algorithm [12].

For most applications in computer vision, graph is sparse and structured (image and video). Thus, Boykov-Kolmogorov (BK) algorithm [4] leverages a search tree structure, and achieves almost 10 times faster than Dinitz blocking flow algorithm. The Grid-Cut algorithm [1] further refines the data structure and optimizes the locality usage, leading to 4 times speedup. The main limitation of augmented path based methods is that a global data structure needs to be maintained (like array, tree or sets), and thus has a low parallelization. This makes these methods inefficient for large-scale data sets, although they ensure accuracy.

There have been many solutions for this problem. For instance, Lazy Snapping [2] employs over-segmentation to reduce computation. Grab-Cut [9] adopts the GMM model to reduce the data size. Other solutions employ hierarchical structures (like hierarchical mean-shift [10] or narrow band algorithm [3]) to achieve multi-level cut. Note that, this kind of approaches improves the performance at the cost of the accuracy.

B. Push-Relabel Based Methods

The push-relabel scheme is an alternative scheme for solving the maximum flow problems. The basic version has a complexity of $O(V^2E)$. If an FIFO-based heuristic algorithm is used, the complexity becomes $O(V^3)$. Using a dynamic tree structure leads to a complexity of $O(VE \log(V^2/E))$. The HI-PR algorithm selects the highest-label nodes for discharge and achieves robust performance due to the combination of global and gap relabeling heuristics [13], [14]. The PAR algorithm maintains a preflow and a distance labeling and outperforms HI-PR on both DIMACS families and vision instances [15]. Generally, the serial push-relabel scheme is faster than Dinitz algorithm, but is slower than the Boykov-Kolmogorov algorithm [4], [16].

C. Parallelization Methods

Parallelization is a widely employed mechanism to speedup the computation. Because the push-relabel scheme is more compatible with parallelization than the augmented path scheme, it has been widely refined to be GPU implementations such as CUDA-Cut [6] and Fast-Cut [7], [17] approaches.

The earliest CUDA-Cut algorithm [6] is a straightforward implementation of the conventional Push-relabel algorithm. It performs push followed by pull to avoid data conflict and achieves 10-fold speedups over BK. The disadvantage is that it may yield inaccurate results because in some cases it stops iterating before reaching the global optimum which is unacceptable. In our method, we use convergence detection to do enough iterations to guarantee the global minima and ensure accuracy.

The fast-cut algorithm [7] introduces wave push and global relabel to improve CUDA-Cut. However, it has a low convergence speed, this is insufficient to handle large data sets. In order to solve this problem, we introduce a block-wise push-relabel technique to accelerate the propagation of flow and the convergence speed.

III. JUMP FLOODING-CUT

A. Preliminary Knowledge

Push-Relabel: Starts with an excess flow (or preflow, allowing a node to have more outwards flow than inwards flow) and pushes an excess flow closer towards a sink (if the excess flow cannot reach a sink, it is pushed backwards to a source), until the maximum flow is found. Because different active nodes (with excess flows) can be pushed or relabeled simultaneously, this algorithm has a high parallelism. Moreover, several heuristic schemes, such as *Global Relabel* and *Gap Heuristics*, can be used to further improve the performance.

Global Relabel: Computes height values (lower bound of the node's distance from the sink in the residual graph) by performing a Breadth-First Search (BFS) starting from the sink. This heuristic technique can greatly reduce the number of iterations, because it implicitly finds out the augmenting-paths and pushes a node in the optimal direction. This technique is used in the first stages of our approach because recomputing the height field (formed by the height of each node) in later stages is not effective and time-consuming.

Convergence Detection: Starts from a source (or a sink) and performs a BFS on the residual network to check whether a path from a source to a sink exists: no existence means the minimum cut is found, which partitions the graph into two parts; otherwise, the push-relabel should be continued. Generally, the search should start from a source, because it allows us to do labeling at the same time, instead of starting from a sink and reversing its results to get the set of all neutral nodes (belonging to neither foreground, nor background) which is useless. This technique ensures optimal results and avoids unnecessary iterations.

Jump Flooding: Was first used to compute Voronoi Diagram (similar to Euclidean Distance Transform) for various applications in Computation Geometry and Pattern

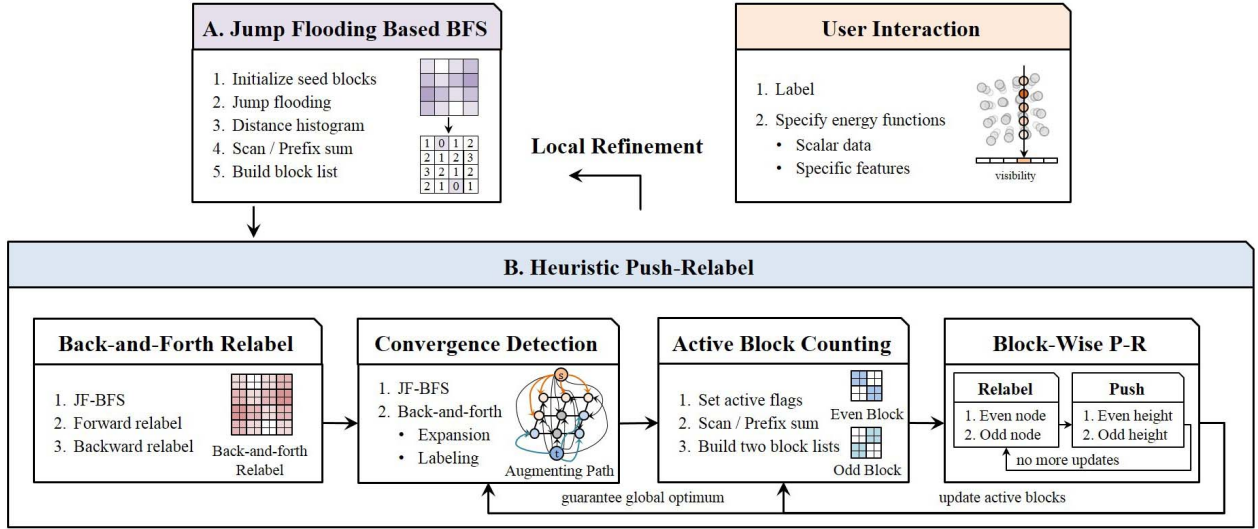


Fig. 1. Our approach consists of two parts: jump flooding based BFS and heuristic push-relabel.

Recognition [18], [19]. It can quickly compute the *Nearest Seed Point* (NSP) of each point. The main idea is to replace the NSP of the current point with the nearest one of the NSPs of all its neighboring points. The size of the neighborhood is halved in each iteration. *Jump Flooding* (JF) can be used to accelerate BFS in computing the nearest distance. Given the nearest point, the nearest distance can be derived, if each neighboring point is connected with each other. Section 3.2 gives an example of using JF to accelerate BFS.

B. Overview

The key idea of our method is to accelerate the propagation of flow in the graph. We propose a convergence detection technique to ensure accuracy that is not achieved in [6] and a block-wise push-relabel technique to achieve a faster convergence speed compared with [7].

Our approach consists of two parts as shown in Fig.1: jump flooding based BFS and heuristic push-relabel.

Part A (jump flooding based BFS) is a basic module of our algorithm, which improves the performance of multi-pass global relabel and convergence detection.

Part B (heuristic push-relabel) is the core algorithm of our approach. This part computes the maximum flow in an efficient way especially for large data sets, detailed in Section 3.4. All these algorithms are parallelized with OpenCL (Open Computing Language), and evaluated with a variety of data sets.

We also design an intuitive user interface to help users specify foreground and background elements based on visibility, described in Section 4. If the users are not satisfied with the results, they can do local refinement.

C. Jump Flooding Based BFS

We use JF to accelerate Global Relabel and Convergence Detection, because both of them on the BFS. In general, for GPU computing, the data set is often divided into blocks (each block can contain 64 to 512 threads) to fit the hardware

Algorithm 1 Jump Flooding (Size, Nearest)

```

p ← Global-ID
q ← Nearest[xp, yp]
d ← Taxicab-Distance(p, q)
for each pt ∈ Neighbors(p, Size) do
    qt ← Nearest[xpt, ypt]
    dt ← Taxicab-Distance(p, qt)
    if dt < d then
        d ← dt
        q ← qt
    end if
end for
Nearest[xp, yp] ← q

```

architecture for high efficiency. When the data set is small, using the CPU to fulfill BFS is adequate. However, when the size becomes much larger, the increasing cost should receive additional consideration. For instance, if the data size is $1024 \times 1024 \times 512$ and the maximum block size is 256, we have to deal with nearly four million blocks. Accordingly, we introduce a Jump Flooding based algorithm (JF-BFS), which includes five steps:

- F1** Initialize all the seed blocks that contain seed nodes;
- F2** Perform jump flooding;
- F3** For each block, compute its nearest distance to the seed blocks and the distance histogram;
- F4** Perform the scan primitive on the distance histogram;
- F5** Build the block list according to the prefix sum of distance histogram.

Note that, all the above steps can be performed on the GPU. In **F1**, a seed block is defined to have at least one foreground or background node, whose *Nearest Seed Block* (NSB) is initialized as itself. Other blocks' NSBs are set to infinity. In **F2**, we perform the standard Jump Flooding Algorithm (JFA). In **F3**, for each block **p**, we compute the distance to its NSB as the nearest distance *h* to all the seed blocks

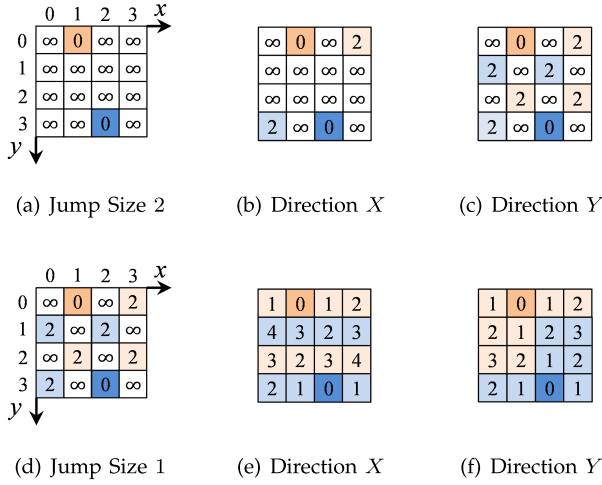


Fig. 2. Jump Flooding. Two seed blocks are colored in dark orange and dark blue. (a)-(c): the 1st flooding with $Size = 2$. Six blocks with light colors update their nearest seed blocks. Two of them are in x direction and four of them are in y direction. (d)-(f): the 2nd flooding with $Size = 1$. The rest of the blocks updates the distances to their nearest seed blocks (see the integers in pixels). (a) Jump Size 2. (b) Direction X. (c) Direction Y. (d) Jump Size 1. (e) Direction X. (f) Direction Y.

(see Algorithm 1). In **F4**, we compute the distance histogram and label each block a corresponding order in the bin. In **F5**, we compute its prefix sum [20] and calculate the final positions of all the blocks in the list.

Fig.2 illustrates the process by taking a 4×4 network as an example. The seed blocks are (1, 0) and (2, 3) and the number inside a block shows the current distance to the NSB. In this case, the JFA includes two iterations. The step size in the first iteration is 2 (Fig.2(a)-2(c)) while in the second iteration it is halved to be 1 (Fig.2(d)-2(f)). In the first iteration, according to the neighbors in the x direction, the NSBs of block (3, 0) and (0, 3) are updated to (1, 0) and (0, 3) respectively. After that, block (0, 1), (2, 1), (0, 3) and (2, 3) are updated by taking the vertical neighbors into consideration. In the second iteration, we process these blocks in the same way using a smaller step size. Finally, eight blocks, colored in yellow have the same NSB - block (1, 0) while the other blocks, colored in blue are the nearest to block (2, 3), see Fig.2(f). The corresponding pseudo code is:

D. Heuristic Push-Relabel

The basic idea of *Heuristic Push-Relabel* is to process a block like a node, meaning that the push and relabel operations are repeated inside a block until it is saturated (no more updates for push and relabel). Compared to the global push or relabel scheme, ours has two advantages. First, the synchronization operations in a block can avoid data conflict for push and relabel, and local memory can be efficiently used to cache intermediate data. Second, an alternate push and relabel improves the propagation speed of information because a node along each direction is pushed at the same time.

This scheme is more efficient than *Wave Push* which only handles one direction each time. Moreover, it allows us to take advantage of data locality and skip some of the directions.

Our method includes 4 steps:

- H1** Perform JF-BFS to build the block list and perform a back-and-forth relabel;
- H2** Repeat **H3** k_1 times and use JF-BFS to build the list and perform the convergence detection;
 - H2.1** Perform JF-BFS to build the block list starting from the foreground nodes and repeat **H2.2** until detection failed or no more nodes are marked;
 - H2.2** Perform back-and-forth detection;
 - H2.2.1** If any background node is found return to **H2.1** (detection fails, keep iterating), otherwise repeat **H2.1.2** until no more nodes are marked (detection succeeds, stop iterating);
 - H2.2.2** Mark the current node as visited according to the state (visited or not) of its neighbors.
- H3** Repeat **H4** k_2 times, count active blocks and use Scan to build two block lists for even blocks and odd blocks;
 - H3.1** For each block, set its flag as whether it contains active nodes;
 - H3.2** Perform scan primitive on the flags;
 - H3.3** Build two lists for even and odd blocks respectively;
- H4** Perform push-relabel for even blocks and odd blocks respectively;
 - H4.1** Repeat **H4.1** until no more nodes can be pushed;
 - H4.2** Relabel even nodes and odd nodes respectively, then push them in the same way.

1) *Back-and-Forth Relabel*: In **H1**, we first use JF-BFS to build the block list for subsequent scheduling. Based on the list, we perform a back-and-forth relabel, which processes the blocks in an ascending order and then in a descending order. It can help us find out the paths that often change directions. The reason is that after we finish relabel in one direction, the paths that pass along the same direction have already been found and we can easily find out all paths that change their directions only once if we try an opposite direction. For JF-BFS, the seed blocks include background nodes (with negative excess flow).

2) *Convergence Detection*: In **H2**, we detect global convergence while labeling the cutting results by checking whether there exists an augmenting-path in the residual network. If so, we need further actions to increase the flow, otherwise we found the maximum flow. We perform JF-BFS from the foreground nodes (source-connected) because it can mark the final results and avoid neutral nodes (see Section 3.1).

The principle implied by these sub-steps is that when we get the maximum flow (or minimum cut), the residual network will be divided into two parts, and for each part all nodes should be connected, and all edges belonging to the minimum cut should be saturated. It indicates that we can start from one part and repeat expanding nodes to decide whether we have found the global optimum.

This step ensures that our algorithm can find the maximum flow and get the optimal results. In our implementation, due to the expensive cost of this procedure, we repeat **H3** k_1 times and perform it once. k_1 defaults to 4 (performs best for median-size image) and can be specified by users.

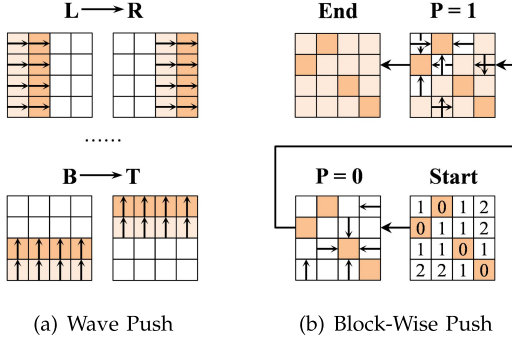


Fig. 3. Heuristic Push. (a): only handles one direction each time. (b): skip some directions which needn't to be processed. P denotes block parity. (a) Wave Push. (b) Block-Wise Push.

Generally, it would be better to increase k_1 when the data size increases (16 is better for large images and videos).

3) *Active Block Counting*: Active Block Counting is designed to reduce the computational cost by building a block list for active blocks, which is mainly based on the two facts: during the process only some nodes can be pushed or relabeled, and the number of active nodes decreases when more data is processed. So only process active nodes efficiently improve the performance. Because active-blocks change slowly, we can do one counting after k_2 iterations. k_2 is set to be 4 (better for median-size data sets) and is user-adjustable. This procedure includes three sub-steps (see H3.1 H3.3). We use two flags to record the detection results. If the current block contains active nodes, we set its corresponding flag (even flag corresponds to even block) to be 1, otherwise 0. After that, we perform a scan to compact the flags into a scheduling list.

4) *Block-Wise Push-Relabel*: As mentioned earlier, iterative push-relabel operations inside a block takes advantage of locality and increases the propagation speed of flow compared with *Wave Push* as shown in Fig.3.

The kernel code (2D-version) is shown in Algorithm 2. We first load the data from global memory to local memory, then perform iterative push-relabel, and finally write back the data. In each iteration, there are two stages: relabel and push. In the relabel stage, we first check whether $Node'[x_p, y_p]$ is active, and then process even nodes with $(x_p + y_p) \bmod 2 = 0$ and odd nodes with $(x_p + y_p) \bmod 2 = 1$ respectively, because these two kinds of nodes have data conflict with each other. In the push stage, we process the nodes in the same way. Both directions are handled one after another. When a node becomes inactive after being pushed in one direction, we skip the following directions. In this way, many unnecessary computations can be avoided. Then we process the nodes in the same way.

IV. USER INTERACTION

We design a user interface for both images and videos. It helps users mark foreground and background based on visibility, build an energy function according to the scenario, make local refinements and finally shows the graph cut results.

Algorithm 2 Push-Relabel (Node)

```

p ← Local-ID
copy global Node to local Node'
d' ← false
barrier()
while not d' do
  t_a ← Active(Node'[x_p, y_p])
  t_p ← (x_p + y_p) mod 2
  if t_a and t_p = 0 then
    Relabel()
  end if
  barrier()
  if t_a and t_p = 1 then
    Relabel()
  end if
  d' ← true
  barrier()
  t_a ← Active(Node'[x_p, y_p])
  if t_a and t_p = 0 and Push() then
    d' ← false
  end if
  barrier()
  if t_a and t_p = 1 and Push() then
    d' ← false
  end if
  barrier()
end while
copy local Node' to global Node

```

A. Labeling

For videos, it is quite difficult for users to specify the foreground and background elements. The conventional way is to mark the nodes frame by frame, which is time-consuming. In our system, we provide users a WYSI-WYG way to do labeling (both for images and videos). The basic idea is to mark what users actually see based on visibility. This technique is combined with a ray casting algorithm which requires users to specify a visibility threshold [21], [22].

B. Scenario-Driven Energy Function

An *Energy Function* is used to define the capacity of each edge in the graph, which measures the similarity of different subsets. The generic form of the energy function is given by:

$$E = \lambda \sum R(p) + \sum B(p, q) \quad (1)$$

where R and B denote the region properties and boundary properties respectively, λ specifies a relative importance of the region properties versus the boundary properties. We provide users with different ways to build the energy function for different scenarios.

Scalar Field Data: We simplify the energy function introduced by Boykov-Kolmogorov [4] which uses a probability distribution to measure the similarity and has a low accuracy.

We propose to focus on the continuity of a structure:

$$C_s(p, q) = \begin{cases} 0 & p, q \in \mathcal{F} \cup \mathcal{B} \\ \exp(-\frac{(I_p - I_q)^2}{2\sigma^2}) & \text{otherwise} \end{cases} \quad (2)$$

$$E_s(p) = \begin{cases} +E_{max} & p \in \mathcal{F} \\ -E_{max} & p \in \mathcal{B} \\ 0 & p \in \mathcal{U} \end{cases} \quad (3)$$

where the nodes are divided into three groups: \mathcal{F} (foreground), \mathcal{B} (background) and \mathcal{U} (unknown). C_s denotes the capacity of the neighboring edge, E_s denotes the excess flow of a node. I denotes the intensity and σ can be estimated as “sampling error”.

Vector Field Data: We define the region and boundary terms similar to *Lazy Snapping* which measures the similarity in the feature-space:

$$C_v(p, q) = \begin{cases} 0 & p, q \in \mathcal{F} \cup \mathcal{B} \\ \frac{1}{1 + \|\mathbf{v}_p - \mathbf{v}_q\|} & \text{otherwise} \end{cases} \quad (4)$$

$$E_v(p) = \begin{cases} +E_{max} & p \in \mathcal{F} \\ -E_{max} & p \in \mathcal{B} \\ \lambda \frac{\min \|\mathbf{c}_i^{\mathcal{B}} - \mathbf{v}_p\| - \min \|\mathbf{c}_i^{\mathcal{F}} - \mathbf{v}_p\|}{\min \|\mathbf{c}_i^{\mathcal{B}} - \mathbf{v}_p\| + \min \|\mathbf{c}_i^{\mathcal{F}} - \mathbf{v}_p\|} & p \in \mathcal{U} \end{cases} \quad (5)$$

where C_v denotes the neighbor-capacity, E_v denotes the excess flow and is defined by the minimum distance to the center of a cluster which the current node belongs to. These centers are computed by *K-Means Clustering* which is also parallelized. It helps us extract key features from the foreground and background, so that the noises can be effectively restrained and the accuracy is improved. In this algorithm, the number of clusters k is set to be 8, and the seed points are initialized using a histogram equalization technique.

Our system also supports features extraction, which can convert a scalar data into a multi-dimensional data [23]. These features include gradient, hessian matrix [24] and curvature [25].















V. RESULTS

We implement an integrated system using OpenCL 1.0 and QT 4.8. Our system supports across heterogeneous platforms. The experimental environment is: Windows 8 64bit with Intel (R) Core i7-3770 @ 3.40GHz and NVIDIA GeForce GTX TITAN @ 6GB (or AMD Radeon HD 7990 @ 2×3GB). Different kinds of data sets are used to evaluate our method, and we compare our algorithm with both CPU based methods and GPU based methods.

A. Datasets

We tested two kinds of data sets. One of them is a part of the benchmark provided by Grid-Cut [1]. The other is downloaded from the Internet, which includes high-resolution images, HD videos and volume data sets. Below is the detail information of the data sets and the scaled-up versions of the data sets. The performance of different algorithms is summarized in Table I.

TABLE I
BENCHMARK DATASETS

Name	Scale	Width	Height	Depth	Max Flow	Size (MB)
 Flower(M)	1 x 1 4 x 4 16 x 16	600 2400 9600	450 1800 7200	1	1208089 7856011 50618597	3.35 53.5 856
 Person(M)	1 x 1 4 x 4 16 x 16	600 2400 9600	450 1800 7200	1	880461 5042635 29042350	3.35 53.5 856
 Sponge(M)	1 x 1 4 x 4 16 x 16	640 2560 10240	480 1920 7680	1	343591 1910718 10136071	3.81 60.9 975
 Bone(U)	1 x 1 x 1 2 x 2 x 2	256 512	256 512	119 238	71344 1913780	111 892
 Liver(U)	1 x 1 x 1 2 x 2 x 2	170 340	170 340	144 288	625447 3886049	59.5 476
 Babyface(U)	1 x 1 x 1 2 x 2 x 2	250 500	250 500	81 162	222943 3316927	72.4 579
 Adhead(U)	1 x 1 x 1 2 x 2 x 2	256 512	256 512	192 384	589368 3594110	180 1440
 Madagascar(G)	1 x 1 x 1	10800	8100	1	4946409834	1413
 LTA(G)	1 x 1 x 1	9900	7500	1	6244836523	1198
 TimeScapes(Y)	1 x 1 x 1	2560	1440	30	80886496	1577
 The Croods(Y)	1 x 1 x 1	1920	1080	40	180301245	1178
 Life of PI(Y)	1 x 1 x 1	1920	1080	32	1178329527	949
 MRBrain(V)	1 x 1 x 1	256	256	109	29036	102
 Lobster(V)	1 x 1 x 1	301	324	56	238402	78

- 1) M - Middlebury College: <http://vision.middlebury.edu/MRF/>
- 2) U - University of Western Ontario: <http://vision.csd.uwo.ca/data/maxflow/>
- 3) G - Google Advanced Image Search: http://images.google.com/advanced_image_search
- 4) Y - YouTube: <http://www.youtube.com/>
- 5) V - VolVis: <http://www.volvis.org/>

We conduct experiments on these large data sets to show the practicality and efficiency of our method. In these experiments, the foreground and background are colored in magenta and green respectively, and the cut results are highlighted in yellow.

Image Segmentation: The results include two parts, as shown in Fig.4. The first part contains standard data sets (from Middlebury) with providing energy function. The *Lower Than Atlantis*¹ and *Madagascar*² data sets in the second part are much larger and we use the clustering based model to set up energy function.

¹Lower Than Atlantis: <http://fashionsoundtrack.com/wp-content/uploads/2013/08/A1posterLTA.jpg>.

²Madagascar: <http://wallpaper.imgcandy.com/images/233316-jessica-alba-wardrobe-malfunaction-original-source-of-image.jpg>.

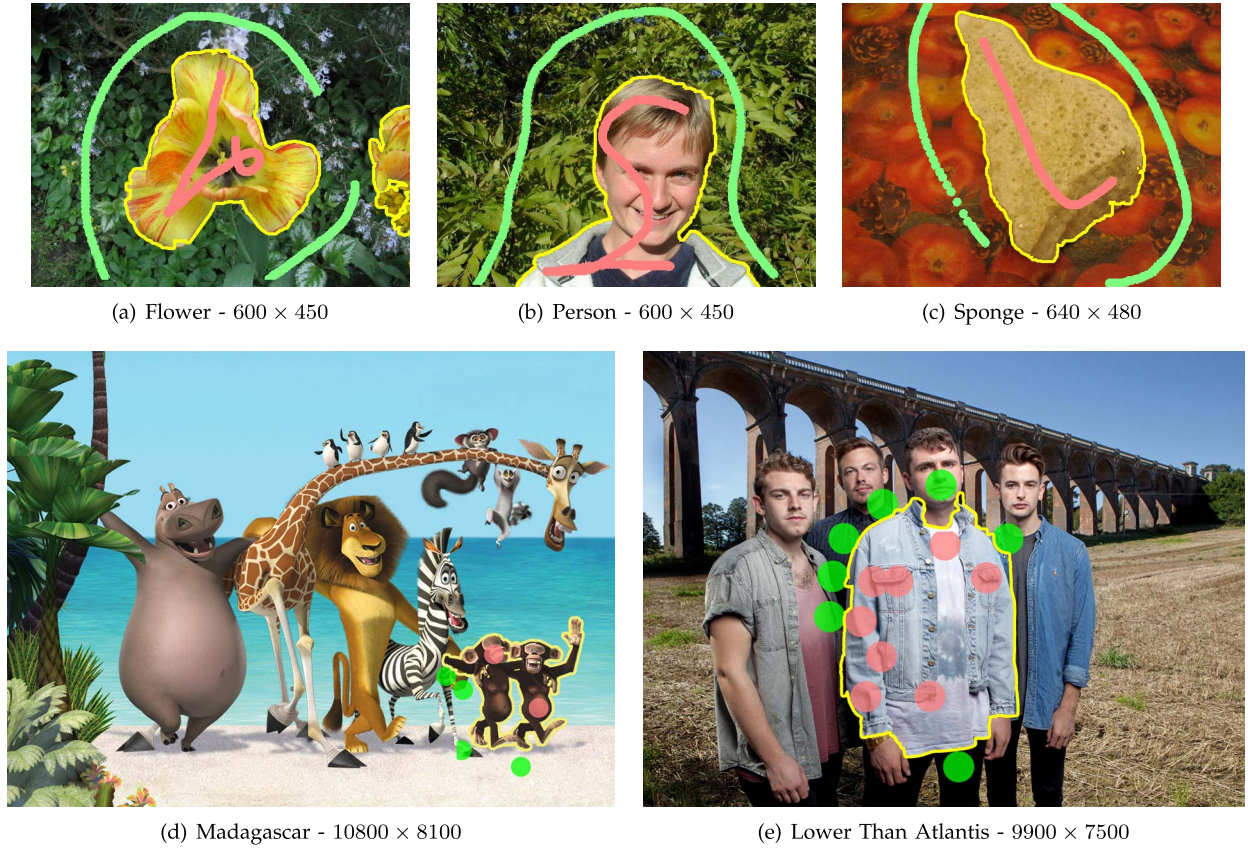


Fig. 4. Image Segmentation Results. 4(a)-4(c): benchmark data sets with small size. 4(e)-4(d): large images.^{1 2}



Fig. 5. Video Cutout Results. 5(a), 5(d): the cut-out bird.⁴ 5(b), 5(e): the cut-out Eep.⁵ 5(c), 5(f): the cut-out tiger⁶

Video Cutout: We use the clustering based model to build energy function. These videos are clipped from official trailers of different films such as *THE CROODS*,³ *TimeScapes*⁴ and *Life of Pi*.⁵ Fig.5 shows the cutout results.

Volume Segmentation: In terms of scalar field data sets, we use the first model to build an energy function. Because the augmenting-paths are much longer, the computing costs are

much higher compared with images. The segmentation results of CT⁷ and MRI⁸ data sets are shown in Fig.6.

B. Performance

We first compare our method with CUDA-Cut [6] and Fast-Cut [7], [17] which are GPU based methods, and then with CPU based methods BK [4] and Grid-Cut [1].

³TimeScapes - Trailer 2 4k 2560p: <http://red.cachefly.net/TimeScapes4K2560p.mp4>.

⁴THE CROODS - Official Trailer 3: http://www.youtube.com/watch?v=xrbwgn_kRBo.

⁵Life of Pi - Trailer 2 Official: <http://www.youtube.com/watch?v=m7WBfntqUoA>.

⁷MRBrain: <http://www-graphics.stanford.edu/data/voldata/>.

⁸Lobster: <http://www.volvis.org/>.

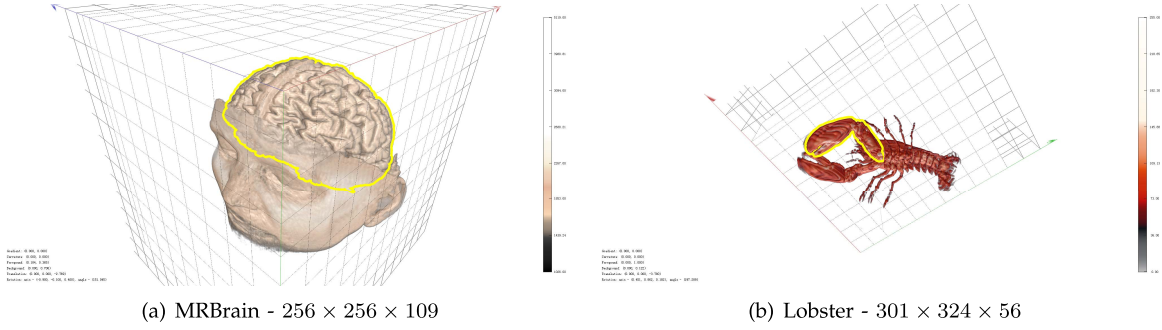


Fig. 6. Volume Segmentation Results. 6(a): MRBrain - MRI and the cutout brain.⁷ 6(b): Lobster - CT and the cutout claw.⁸

TABLE II
GPU BASED METHODS

Instance Name	CUDA-Cut		Fast-Cut	JF-Cut	Speedup	
	Accuracy	Time(ms)	Time(ms)	Time(ms)	-/CUDA	-/Fast
flower	0.736	48.6	9.7	4.2	11.5	2.3
4 × 4	0.673	622.2	262.4	90.0	6.9	2.9
16 × 16	-	-	9636.7	2192.3	-	4.4
person	0.358	73.9	20.9	10.0	7.4	2.1
4 × 4	0.674	612.0	290.9	120.7	5.1	2.4
16 × 16	-	-	12131.0	3403.6	-	3.6
sponge	1.000	48.7	5.4	4.4	11.0	1.2
4 × 4	0.973	637.3	151.7	31.4	20.3	4.8
16 × 16	-	-	3689.5	595.3	-	6.2
bone	-	-	5643.6	616.9	-	9.1
2 × 2 × 2	-	-	68712.3	6749.9	-	10.2
liver	-	-	6211.4	585.7	-	10.6
2 × 2 × 2	-	-	165704.3	16441.2	-	10.1
babyface	-	-	7932.5	830.2	-	9.6
2 × 2 × 2	-	-	104277.6	11309.2	-	9.2
adhead	-	-	17084.4	1654.6	-	10.3
2 × 2 × 2	-	-	115991.0	10511.7	-	11.0
madagascar	-	-	49559.9	24465.4	-	2.0
lta	-	-	95605.9	51279.0	-	1.9
timescapes	-	-	357.2	78.6	-	4.5
the croods	-	-	9410.7	1374.6	-	6.8
life of pi	-	-	743.4	80.9	-	9.2
mrbrain	-	-	9381.0	1226.2	-	7.7
lobster	-	-	2892.9	321.5	-	9.0

- 1) CUDA-Cut (v1.0) - <http://cvit.iit.ac.in/resources/cudacuts/>
2) Environment - Nvidia GeForce GTX TITAN

TABLE III
CONVERGENCE SPEED

Instance Name	Iterations			Ratios	
	CUDA	Fast-Cut	JF-Cut	-/CUDA	-/Fast
flower	155	41	10	15.5	4.1
4 × 4	232	174	81	2.9	2.1
16 × 16	-	516	195	-	2.6
person	232	100	26	8.9	3.8
4 × 4	232	203	88	2.6	2.3
16 × 16	-	692	366	-	1.9
sponge	155	31	14	11.1	2.2
4 × 4	232	114	27	8.6	4.2
16 × 16	-	234	84	-	2.8
bone	-	265	125	-	2.1
2 × 2 × 2	-	410	184	-	2.2
liver	-	498	170	-	2.9
2 × 2 × 2	-	1721	697	-	2.5
babyface	-	512	194	-	2.6
2 × 2 × 2	-	876	413	-	2.1
adhead	-	518	197	-	2.6
2 × 2 × 2	-	439	175	-	2.5
madagascar	-	1344	609	-	2.2
lta	-	1344	668	-	2.0
timescapes	-	36	13	-	2.8
the croods	-	68	34	-	2.0
life of pi	-	32	10	-	3.2
mrbrain	-	516	282	-	1.8
lobster	-	214	104	-	2.1

GPU Based Methods: CUDA-Cut provides two different implementations: atomic (PushPull + Relabel) and stochastic (Push + PullRelabel). We use the stochastic version for comparison, because it is faster. We also observe that CUDA-Cut doesn't guarantee global convergence and it often outputs larger energies. We divide the minimum energy by its output to measure the degree of approximation as listed in the second column of Table II. Because JF-Cut and Fast-Cut always yield global optimal results, their accuracies are not listed.

We implement Fast-Cut using OpenCL and extends it to support videos (some of the optimization techniques are deprecated for generality). To be fair, we perform global relabel once at the beginning by using the same block size and mainly compare the cost on push-relabel which is the major part of all GPU based methods (we perform convergence detection and active block counting much less frequently).

Table II shows the performance of CUDA-Cut, Fast-Cut and our method with millisecond precision. Because CUDA-Cut consumes too much memory, it cannot handle large data sets, even there are 6GB device memory. For all data sets listed in Table I, JF-Cut is faster than Fast-Cut and CUDA-Cut. Our method achieves a maximum 20-fold speedup, see

Sponge 4×4 . Larger data sets lead to higher speedups, which means our approach is more suitable for large data sets.

The convergence speed of different methods is shown in Table III. Because the augmenting paths in volume data sets are longer than in images, volume data sets need more iterations. Our method has higher convergence speeds which are more than twice the rates of Fast-Cut and almost 10-fold to CUDA-Cut.

CPU Based Methods: We download the latest version of BK (V3.03) and Grid-Cut (V1.1) and compile them to get the 64-bit programs. The reason why we choose 64-bit programs is that for large data sets, such as adhead $2 \times 2 \times 2$ and TimeScapes, BK requires a contiguous memory which is larger than 4GB. We use the same compiler settings for comparison, see the table notes in Table IV.

Table IV compares the performance of BK, Grid-Cut and our method. Our method's speedup over Grid-Cut increases with the data scale, except for some data sets. Compared with Grid-Cut, we achieve a maximum 3-fold speedup (sponge 16×16) for images, a 20-fold speed up (life of pi) for videos and a 40-fold speedup (babyface $2 \times 2 \times 2$) for volume data sets. With its parallel version (8 threads), the maximum

TABLE IV
CPU BASED METHODS

Instance	BK	Grid-Cut		JF-Cut	
		Total	1 Thread 8 Threads	Total -/BK -/GC-1 -/GC-8	
flower	29	18	6	6	4.7 2.9 1.0
4 x 4	495	131	89	101	4.9 1.3 0.9
16 x 16	9850	2392	1462	1575	6.3 1.5 0.9
person	29	9	7	14	2.1 0.6 0.5
4 x 4	494	121	92	118	4.2 1.0 0.8
16 x 16	9111	2069	1377	2417	3.8 0.9 0.6
sponge	28	7	6	13	2.2 0.6 0.5
4 x 4	466	93	79	41	11.5 2.3 1.9
16 x 16	7537	1531	1265	595	12.7 2.6 2.1
bone	5761	867	519	421	13.7 2.1 1.2
2 x 2 x 2	204874	32057	20402	4707	43.5 6.8 4.3
liver	10769	3254	3916	305	35.3 10.7 12.8
2 x 2 x 2	855279	237593	256249	7923	107.9 30.0 32.3
babyface	8940	2619	1501	459	19.5 5.7 3.3
2 x 2 x 2	809026	231592	233656	5825	138.9 39.8 40.1
adhead	23192	6517	4149	980	23.7 6.6 4.2
2 x 2 x 2	223934	41414	21274	7921	28.3 5.2 2.7
madagascar	114673	29658	12269	13730	8.4 2.2 0.9
lta	104006	32055	13139	24275	4.3 1.3 0.5
timescapes	18175	3640	2741	-	- - -
the croods	12603	2883	2115	906	13.9 3.2 2.3
life of pi	11700	3925	2162	209	55.9 18.8 10.3
mrbrain	3902	599	597	798	4.9 0.8 0.7
lobster	7028	1428	657	229	30.7 6.2 2.9

- 1) BK (v3.03) - <http://pub.ist.ac.at/~vink/software.html>
- 2) Grid-Cut (v1.1) - <http://gridcut.com/downloads.php>
- 3) Compiler Settings for GCC (v4.9.0) -O3 -march=native -mtune=generic -DNDEBUG
- 4) Environment - Intel (R) Core i7-3770 and AMD Radeon HD 7990

speedups are 2-fold for images, 10-fold for videos and 40-fold for volume data sets.

We also observe that there exists a notable variant. The decisive factor may be the graph topology. The reason is that in volume data sets most of the terminal-edges have zero capacities, while in images most of them are positive, which will greatly affect the convergence speed. These speedups are lower than of GPU based methods, because there exist big differences between CPU based methods (based on augmenting-path) and GPU based methods (based on push-relabel). In summary, for large data sets our method significantly outperforms other methods.

VI. DISCUSSIONS

We analyze our tuning techniques and discuss the quality and the limitations of our approach.

A. Convergence Proof

The standard push-relabel method [13] maintains a preflow and always keeps a valid labeling:

- *Definition 1 (Preflow)*: An assignment of a non-negative flow $f(u, v)$ to each edge (u, v) of a network $(G = (V, E), s, t, c)$ is a preflow if for each edge (u, v) , $f(u, v) \leq c(u, v)$ and for each vertex $v \in V - t$, $\sum_u f(u, v) - \sum_w f(v, w) \geq 0$.
- *Definition 2 (Excess Flow)*: The excess flow at v is defined as $e_f(v) = \sum_u f(u, v) - \sum_w f(v, w)$.
- *Definition 3 (Invariant)*: At every step, if there is an edge (u, v) that has a positive capacity $c'(u, v) > 0$ in the residual network, then $h(u) \leq h(v) + 1$ and this labeling is *valid*.

The standard method consists of two steps: initialization and the main loop. In the first step, it saturates all the edges

$(s, v) \in E$ from the source s and sets the original labeling $h(s) = n$, $h(v) = 0$ for all $v \in V$, $v \neq s$. This gives a valid preflow f_0 . In the main loop, for each active node u , perform the following push and relabel operations:

- **Push(u)** If $\exists v$ with admissible arc $(u, v) \in E_f$, then send flow $\delta \leftarrow \min(c_f(u, v), e_f(u))$ from u to v .
- **Relabel(u)** If for all arcs (u, v) out of u , either $(u, v) \notin E_f$ or $h(v) \geq h(u)$, then set $h(u) = 1 + \min_{v:(u,v) \in E_f} h(v)$.

In our method, we take the initialization step and do push or relabel in a specific order until there is no augmenting $s - t$ path (ensured by the convergence detection). Three lemmas are provided below to prove the convergence of our algorithm:

- *Lemma 1*: The labels h remain valid in **H4** (the last step of our method in Section 3.4).

Proof: The proof is done by induction. Initially h is valid with respect to f_0 . In **H4**, we perform push and relabel operations alternatively. When we push a node on the admissible arc (u, v) , the new arc $(v, u) \in E_f$ satisfies $d(v) = d(u) - 1 \leq d(u) + 1$ and the labels remain valid. When we relabel a node, the new label is the largest one while remaining valid. Because each operation is independent (no data conflict), the property remains true.

- *Lemma 2*: The push-relabel given by **H3** is equivalent to the original push-relabel algorithm.

Proof: In our method, each operation (push or relabel) is the same as the standard one and we perform these operations in parallel. In **H3**, we process even blocks and odd blocks alternatively. For each block, we relabel even nodes and odd nodes respectively, and then push them in the same way. This ensures that, there is no data conflict and all the operations are independent. And there exists at least one sequential way to perform these operations to get the same results. Note that in the original method, changing the processing order does not affect the convergence. Thus, our method is equivalent to the original push-relabel algorithm.

- *Lemma 3*: The final preflow resulted from **H2** is a maximum flow.

Proof: In **H2**, we start from the foreground nodes and repeat expanding them in the residual graph. If the algorithm terminates without finding any background node, then there is no augmenting-path. Hence the final preflow is a maximum flow.

Overall, these lemmas prove that our method can yield global optimal results like the standard push-relabel algorithm.

B. Complexity Analysis

The complexity of the original push-relabel algorithm is $O(V^2E)$. Most GPU based methods are based on it and use different strategies to process active nodes to improve the performance. However, the accurate analysis is quite difficult. As the access of global memory (400 – 600 cycles) is much slower than shared memory (20 – 40 cycles), we can use the workload per global memory read/write as metric to compare different methods.

TABLE V
COMPLEXITY ANALYSIS

Metric	Dimension	Block Size	Wave Push		Block-Wise
			CUDA	Fast-Cut	JF-Cut
-	-	-	-	-	-
N	2	-	0.25	0.25	4
N	3	-	-	0.13	6
D	2	16 × 16	16	16	256
D	2	32 × 32	32	32	1024
D	3	4 × 4 × 4	-	4	64
D	3	8 × 8 × 8	-	8	512

- **Metric 1** The first metric N is the number of operations (push or relabel) per global memory read/write. A larger N means a lower complexity.
- **Metric 2** The second metric D is the propagation distance of a flow per global memory read/write. A larger D means a lower complexity.

In terms of **Metric 1**, because CUDA-Cut and Fast-Cut perform a Wave Push, they can perform at most $N = \frac{1}{2d}$ operations (push or relabel in all the directions) within one global memory read/write, where d denotes the dimension. The reason is that, for each direction, one operation (push or relabel) in their method needs at least one global memory read/write. In our method, for one node, we can perform at least $N' = 2d$ operations within one global memory read/write (see Table V).

In terms of **Metric 2**, for one block, CUDA-Cut and Fast-Cut can send a flow along $D = \max\{x, y, z\}$ edges at most in one direction within one global memory read/write, where (x, y, z) denotes the block size. However, we can send a flow from one node to any other node in the same block, which means in one global memory read/write, the maximum propagation distance of a flow that we can achieve is $D' = x \times y \times z$.

Table II shows that the workload per global memory read/write in our method is much larger than the ones in others. Although the worst-case complexity of our algorithm is $O(V^2E)$ which is the same as CUDA-Cut and Fast-Cut. The average running time of our method is $n = \frac{N'D'}{ND} \approx \frac{4d^2xyz}{\max\{x, y, z\}}$ times faster than CUDA-Cut and Fast-Cut. We increase the information propagation speed throughout the block. Ours leads to a fast convergence and a lower complexity compared with CUDA-Cut and Fast-Cut (see Table III).

C. Quality

Compared with CUDA-Cut, our approach guarantees the global minimal, meaning that we can get the optimal results. For images such as Flower and Person (see Table II), CUDA-Cut yields inaccurate results. This means that the underlying flow is less than the maximum one (see the Person data set, for which the accuracy is only 0.358). This is because CUDA-Cut does not check the convergence and stops iterating after a specific number of iterations before reaching the global optimum. In contrast, JF-Cut, Fast-Cut, BK and Grid-Cut guarantee the global optimum and the maximum flow they get are the same, as shown in Table I.

In practice, given the same model (the way of defining excess flow and edge capacity), these methods get the

same segmentation boundaries due to the global convergence (see Fig.4 and Fig.5). Moreover, we use in-block push-relabel to improve the speed of information propagation, achieve a fast convergence compared with Fast-Cut. In this sense our approach can handle large data sets more efficiently and yields more practical results (<https://github.com/15pengyi/JF-Cut>).

D. Algorithm Tuning

For better performance, we develop a 2D version of our approach to handle images. And the parameter settings are different for small data sets and large data sets. When the data sets are small, we use $(k_1, k_2) = (4, 2)$ to increase the frequency of active block counting to ensure fewer iterations. For large data sets, (k_1, k_2) is set to $(16, 4)$ to reduce the cost of both active block counting and convergence detection.

Other optimizations include using local memory to cache the data which have multiple read or write per execution, checking if the value has changed before writing, using SOA (Structure of Array) instead of AOS (Array of Structures) and designing a compact structure (four bytes per unit) to support a coalesced read and write for global memory and avoid bank conflicts. We also use *AMD APP Profiler* and *NVIDIA Visual Profiler* to help us identify performance bottlenecks.

E. Limitations

Our approach has several limitations compared with CPU-based methods. First, the feasible data size of our approach is limited by GPU memory, i.e., *CL_DEVICE_GLOBAL_MEM_SIZE* and *CL_DEVICE_MAX_MEM_ALLOC_SIZE* which are defined by the OpenCL environment. For *GeForce GTX TITAN* and *AMD Radeon HD 7990* these parameters are (6GB, 1.5GB) and (3GB, 512MB) respectively. Instead, the size of contiguous-memory on the CPU can be much larger, e.g. 32GB. In addition, GPU based methods have to copy data from host memory to device memory, which consumes extra time. Compared with BK techniques that are affordable for unstructured data, our method currently only supports structured data.

To overcome the limitation of GPU memory, extremely large data can be partitioned and distributed on the GPU clusters. In terms of unstructured data, we need to convert the unstructured grid into a hierarchy of regular grids and perform JF-Cut in different levels.

VII. CONCLUSION

In this paper, we introduce a parallel graph cut approach named JF-Cut to handle large data sets which has two main advantages:

- 1) improving the performance of graph cut for large images and videos using a GPU-based graph cut scheme based on jump flooding, heuristic push/relabel and convergence detection;
- 2) providing users an interactive graph cut based data segmentation interface that allows for intuitive data selection, interaction and segmentation for large data sets.

We use a variety of data sets (both a benchmark and different large data sets) to evaluate the performance of our approach. The results show that our method achieves a maximum 40-fold (139-fold) speedup over conventional GPU- (CPU-) based approaches and can be effectively used in different scenarios. The source code of JF-Cut will be soon available at <https://github.com/15pengyi/JF-Cut>.

Our future work includes extending our approach to support unstructured data, optimizing the codes and making an independent library for end users to use.

ACKNOWLEDGMENT

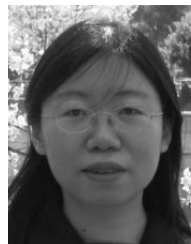
The authors would like to thank the anonymous reviewers for their valuable comments, Hongsen Liao, Min Wu, Zhu Zhu, Wenshan Zhou, Chaowei Gao and Kan Wu for their enthusiastic help and helpful suggestions.

REFERENCES

- [1] O. Jamriska, D. Sykora, and A. Hornung, "Cache-efficient graph cuts on structured grids," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2012, pp. 3673–3680.
- [2] Y. Li, J. Sun, C.-K. Tang, and H.-Y. Shum, "Lazy snapping," in *Proc. ACM SIGGRAPH*, New York, NY, USA, 2004, pp. 303–308. [Online]. Available: <http://doi.acm.org/10.1145/1186562.1015719>
- [3] H. Lombaert, Y. Sun, L. Grady, and C. Xu, "A multilevel banded graph cuts method for fast image segmentation," in *Proc. 10th IEEE Int. Conf. Comput. Vis. (ICCV)*, vol. 1, Oct. 2005, pp. 259–265.
- [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.*, vol. 26, no. 9, pp. 1124–1137, Sep. 2004.
- [5] J. Liu and J. Sun, "Parallel graph-cuts by adaptive bottom-up merging," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2010, pp. 2181–2188.
- [6] V. Vineet and P. J. Narayanan, "CUDA cuts: Fast graph cuts on the GPU," in *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit. Workshops (CVPRW)*, Jun. 2008, pp. 1–8.
- [7] W. H. Wen-Mei, *GPU Computing Gems Emerald Edition*, vol. 1. San Mateo, CA, USA: Morgan Kaufmann, 2011.
- [8] E. A. Dinic, "Algorithm for solution of a problem of maximum flow in a network with power estimation," *Soviet Math Doklady*, vol. 11, pp. 1277–1280, 1970. [Online]. Available: <http://www.cs.bgu.ac.il/~dinitz/D70.pdf>
- [9] C. Rother, V. Kolmogorov, and A. Blake, "'GrabCut': Interactive foreground extraction using iterated graph cuts," *ACM Trans. Graph.*, vol. 23, no. 3, pp. 309–314, Aug. 2004. [Online]. Available: <http://doi.acm.org/10.1145/1015706.1015720>
- [10] J. Wang, P. Bhat, R. A. Colburn, M. Agrawala, and M. F. Cohen, "Interactive video cutout," in *Proc. ACM SIGGRAPH*, New York, NY, USA, 2005, pp. 585–594. [Online]. Available: <http://doi.acm.org/10.1145/1186822.1073233>
- [11] X. Yuan, N. Zhang, M. X. Nguyen, and B. Chen, "Volume cutout," *Vis. Comput.*, vol. 21, nos. 8–10, pp. 745–754, Sep. 2005. [Online]. Available: <http://dx.doi.org/10.1007/s00371-005-0330-2>
- [12] J. B. Orlin, "Max flows in $O(nm)$ time, or better," in *Proc. 45th Annu. ACM Symp. Theory Comput. (STOC)*, New York, NY, USA, 2013, pp. 765–774. [Online]. Available: <http://doi.acm.org/10.1145/2488608.2488705>
- [13] A. V. Goldberg and R. E. Tarjan, "A new approach to the maximum-flow problem," *J. ACM*, vol. 35, no. 4, pp. 921–940, Oct. 1988. [Online]. Available: <http://doi.acm.org/10.1145/48014.61051>
- [14] B. V. Cherkassky and A. V. Goldberg, "On implementing push-relabel method for the maximum flow problem," in *Integer Programming and Combinatorial Optimization* (Lecture Notes in Computer Science), vol. 920, E. Balas and J. Clausen, Eds. Berlin, Germany: Springer-Verlag, 1995, pp. 157–171. [Online]. Available: http://dx.doi.org/10.1007/3-540-59408-6_49
- [15] A. V. Goldberg, "The partial augment–relabel algorithm for the maximum flow problem," in *Algorithms—ESA* (Lecture Notes in Computer Science), vol. 5193, D. Halperin and K. Mehlhorn, Eds. Berlin, Germany: Springer-Verlag, 2008, pp. 466–477. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-87744-8_39
- [16] T. Verma and D. Batra, "MaxFlow revisited: An empirical comparison of maxflow algorithms for dense vision problems," in *Proc. Brit. Mach. Vis. Conf. (BMVC)*, 2012, pp. 61.1–61.12.
- [17] T. Stich, "Graphcuts with CUDA and applications in image processing," in *Proc. GPU Technol. Conf.*, 2009. [Online]. Available: <http://developer.download.nvidia.com/compute/cuda/docs/GTC09Materials.htm>
- [18] C. R. Maurer, Jr., R. Qi, and V. Raghavan, "A linear time algorithm for computing exact Euclidean distance transforms of binary images in arbitrary dimensions," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 25, no. 2, pp. 265–270, Feb. 2003.
- [19] G. Rong and T.-S. Tan, "Jump flooding in GPU with applications to Voronoi diagram and distance transform," in *Proc. Symp. Interact. 3D Graph. Games (I3D)*, New York, NY, USA, 2006, pp. 109–116. [Online]. Available: <http://doi.acm.org/10.1145/1111411.1111431>
- [20] D. Merrill and A. Grimshaw, "High performance and scalable radix sorting: A case study of implementing dynamic parallelism for GPU computing," *Parallel Process. Lett.*, vol. 21, no. 2, pp. 245–272, 2011. [Online]. Available: <http://www.worldscientific.com/doi/abs/10.1142/S0129626411000187>
- [21] U. D. Bordoloi and H.-W. Shen, "View selection for volume rendering," in *Proc. IEEE Vis. Conf.*, Oct. 2005, pp. 487–494.
- [22] C. Correa and K.-L. Ma, "Visibility histograms and visibility-driven transfer functions," *IEEE Trans. Vis. Comput. Graphics*, vol. 17, no. 2, pp. 192–204, Feb. 2011.
- [23] Y. Peng and L. Chen, "Multi-feature based transfer function design for 3D medical image visualization," in *Proc. 3rd Int. Conf. Biomed. Eng. Inform. (BMEI)*, vol. 1, Oct. 2010, pp. 410–413.
- [24] J. Kniss, G. Kindlmann, and C. Hansen, "Multidimensional transfer functions for interactive volume rendering," *IEEE Trans. Vis. Comput. Graphics*, vol. 8, no. 3, pp. 270–285, Jul./Sep. 2002. [Online]. Available: <http://dx.doi.org/10.1109/TVCG.2002.1021579>
- [25] G. Kindlmann, R. Whitaker, T. Tasdizen, and T. Moller, "Curvature-based transfer functions for direct volume rendering: Methods and applications," in *Proc. IEEE Vis. (VIS)*, Oct. 2003, pp. 513–520.



Yi Peng is currently pursuing the Ph.D. degree with the Department of Computer Science and Technology, Tsinghua University, Beijing, China, where he received the B.S. degree from the School of Software, in 2010. His research interests include scientific visualization, image processing, parallel computing, and computer graphics.



Li Chen received the Ph.D. degree in visualization from Zhejiang University, Hangzhou, China, in 1996. She is currently an Associate Professor with the Institute of Computer Graphics and Computer Aided Design, School of Software, Tsinghua University, Beijing, China. Her research interests include data visualization, mesh generation, and parallel algorithm.



Fang-Xin Ou-Yang is currently pursuing the bachelor's degree with the School of Software, Tsinghua University, Beijing, China. Her research interests include visualization and computer graphics.



Wei Chen is currently a Professor with the State Key Laboratory of Computer-Aided Design and Computer Graphics, Zhejiang University, Hangzhou, China. He received the Ph.D. degree with the Fraunhofer Institute for Graphics, Darmstadt, Germany in 2002. His Ph.D. advisors were Prof. Q. Peng and Prof. G. Sakas. From 2006 to 2008, he was a Visiting Scholar with Purdue University, West Lafayette, IN, USA, working in PURPL with Prof. David S. Ebert. In 2009, he was promoted as a Full Professor with Zhejiang University. He has performed research in computer graphics and visualization and has authored over 60 peer-reviewed journal and conference papers in the last five years. His current research interests include visualization, visual analytics, and biomedical image computing.



Jun-Hai Yong is currently a Professor with the School of Software, Tsinghua University, Beijing, China, where he received the B.S. and Ph.D. degrees in computer science, in 1996 and 2001, respectively. He held a visiting researcher position with the Department of Computer Science, Hong Kong University of Science and Technology, Hong Kong, in 2000. He was a Post-Doctoral Fellow with the Department of Computer Science, University of Kentucky, Lexington, KY, USA, from 2000 to 2002. He received several awards, such as the National Excellent Doctoral Dissertation Award, the National Science Fund for Distinguished Young Scholars, the Best Paper Award of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation, the Outstanding Service Award as an Associate Editor of the *Computers and Graphics* (Elsevier) journal, and several National Excellent Textbook Awards. His main research interests include computer-aided design and computer graphics.