JF-Cut: A Parallel Graph Cut Approach for Large-scale Data

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 graphics community. The main limitation of conventional graph-cut implementations is that they can hardly handle large-scale datasets 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 rate (on GPU).
In this paper, we present a novel graph cut algorithm that leverages a parallelized jump flooding (JF) technique and a heuristic pushing and relabeling scheme to enhance the components of the graph cut process, namely, multi-pass relabel, convergence detection and block-wise push-relabel. The entire process is parallelizable on GPU, and outperforms 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 volumetric data or video sequences. Experiments on a variety of datasets, including images (up to $15K \times 10K$), videos (up to $2.5K \times 1.5K \times 50$), and volumetric data, achieve high quality results and a maximum 164-fold speedup over conventional approaches.

Index Terms—Graph Cut, Jump Flooding, Visibility, Segmentation

1 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]. The basic idea of graph cut is to partition the elements into two disjoint subsets by finding the minimum cut using maximum flow algorithms. 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.

- *Y. Peng (15pengyi@gmail.com) is with Department of Computer Science and Technology, Tsinghua University, Beijing 100084, P. R. China. http://cgcad.thss.tsinghua.edu.cn/jackiepang/*
- *L. Chen (chenlee@tsinghua.edu.cn), F.X. Ou-Yang (oyfx11@mails.tsinghua.edu.cn) and J.H. Yong (yongjh@tsinghua.edu.cn) are with School of Software, Tsinghua University.*
- *W. Chen (chenwei@cad.zju.edu.cn) is with State Key Lab of CAD&CG, Zhejiang University*

Manuscript received January 1, 2014;revised January 1, 2014.

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 dataset. On the other hand, using the push-relabel scheme may lead to slow convergence rate, or yield non-global optimal results. It is also inefficient in tackling large-scale datasets.

Early work on image graph cut, such as lazy snapping [2] and grabcut [9] perform pre-segmentation or use GMM(Gaussian Mixed Model) components 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 pre-processing and may lead to approximated results.

This paper introduces a parallel graph cut approach that is capable of handling large-

scale datasets with fast convergence rate. The key idea is to accelerate the propagation of flow in the graph with three schemes. First, a jump flooding technique is employed to improve Breadth-First Search (BFS) based operations. Second, a heuristic push-relabel technique that uses block-wise alternation and in-block iteration to facilitate quick convergence to the optimal result. Third, a convergence detection technique is designed to guarantee global convergence. The main gain of these schemes is that the information propagation speed throughout the network is greatly increased. The entire process is parallelizable and can be implemented in GPU, making it a quite challenging solution for large-scale datasets.

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 datasets (images, videos and volumetric data).

In the remainder of this paper, we first take a brief review of the literature in Section 2. Our approach is elaborately introduced in Section 3, followed by the user interaction in Section 4. Experimental results and comparison are presented in Section 5. Section 6 concludes our approach and highlights the future work.

2 Related Work

2.1 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 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 datasets. 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 improve the performance at the cost of the accuracy.

2.2 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 dynamic tree structure leads to a complexity of $O(VElog(V^2/E))$. The H-PRF algorithm select the highest-label nodes for discharge, and achieves the highest performance [13]. Generally, the serial push-relabel scheme is faster than Dinitz algorithm, but is much slower than the Boykov-Kolmogorov algorithm [4].

2.3 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 CUDA-Cut [6] and Fast-Cut [7], [14] 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 since this algorithm doesn't guarantee global optimum. The fast-cut algorithm [7] introduces wave push and global relabel to improve CUDA-Cut. However, it has a low convergence rate, thus is insufficient to handle large datasets.

3 JUMP FLOODING-CUT

3.1 Preliminary Knowledge

Push-Relabel: starts with a excess flow (or preflow, allowing a node to have more flow coming into it than going out), push excess flow closer towards sink (if excess flow cannot reach sink, push it backwards to 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 heuristics, such as *Global Relabel* and *Gap Heuristics*, can be used to further improve the performance.

Global Relabel: computes height values by performing Breadth-first Search (BFS) starting from the sink. This heuristic technique can greatly reduce the number of iterations, because it implicitly find out the augmenting-paths thus pushing a node in the optimal direction. However, it should be used in the earlier stage. Because more iterations leads to better height field (formed by the height of each node). If we use it later, the height field will be recomputed and may become worse. Moreover, the high cost of this technique doesn't allow us to use it frequently.

Convergence Detection: starts from source (or sink), performs BFS on the residual network to check whether a path from source to 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 starting from source is better, because we can do labeling at the same time. Otherwise, we should reverse the results to get the object and all the neutral nodes (belong neither to foreground, nor background) will be included.

Jump Flooding: computes the Voronoi Diagram in parallel. It can quickly compute the nearest seed point(NSP) of each point. The main idea is to replace the NSP of the current node with the nearest NSP of its neighbors' from a certain distance, which 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 all the neighboring points are connected with each other. Section 3.2 gives an example of using JF to accelerate BFS.

3.2 Overview

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

The first part is a basic module of our algorithm, which improves the performance of multi-pass global relabel and convergence detection.

The second part is the core algorithm of our approach. This part computes the maximum flow in an efficient way especially for large datasets, Section 3 presents more details. All these algorithms are parallelized with OpenCL (Open Computing Language), and a variety of datasets are used to evaluate our approach.

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.

3.3 Jump Flooding Based BFS

We use JF to accelerate Global Relabel and Convergence Detection, because both of them

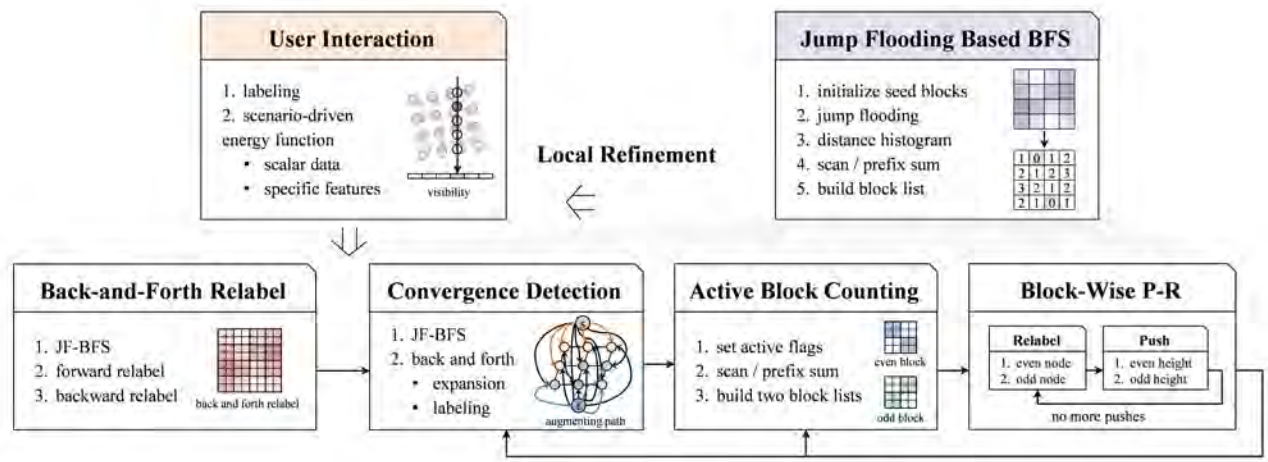
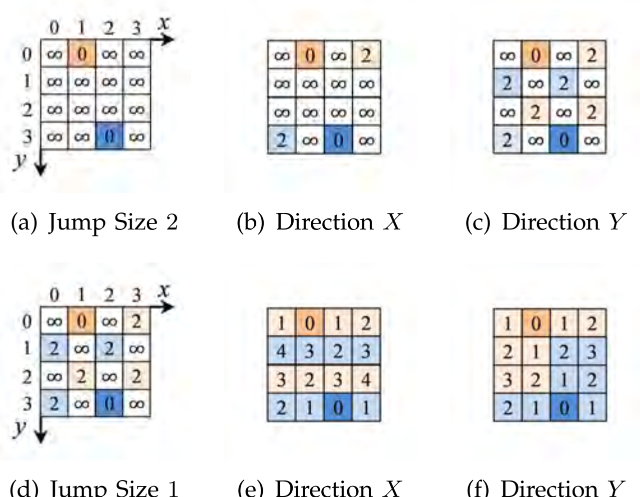


Fig. 1. Approach Overview. Our approach consists of two parts: jump flooding based BFS and heuristic push-relabel (the bottom part).

relies on BFS. In general, for GPU computing, the dataset is often divided into blocks to fit the hardware architecture for high efficiency. When the dataset is small, using CPU to finish BFS is enough. 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:

- **F-1** Initialize all the seed blocks that contain seed nodes;
- **F-2** Perform jump flooding;
- **F-3** For each block, compute its nearest distance to the seed blocks and the distance histogram;
- **F-4** Perform the scan primitive on the distance histogram;
- **F-5** Build the block list according to the prefix sum of distance histogram.

Note that, all the above steps can be performed on GPU. In the first step, 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 the next step, we perform the standard JFA. In the third step, for each block, we compute the distance to its NSB as the nearest distance to all the seed blocks. Then, we compute the distance histogram and label each block a corresponding order in the bin. Finally, we compute its prefix sum [15] and calculate the final positions of all the blocks in the list.



(a) Jump Size 2 (b) Direction X (c) Direction Y

(d) Jump Size 1 (e) Direction X (f) Direction Y

Fig. 2. Jump Flooding. 2(a)-2(c): the 1st flooding with $Size = 2$. 2(d)-2(f): the 2nd flooding with $Size = 1$.

Fig.2 illustrates the process 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:

Algorithm 1 Jump Flooding($Size, Nearest$)

$\mathbf{p} \leftarrow$ Global-ID
$\mathbf{q} \leftarrow Nearest[x_p, y_p]$
$d \leftarrow$ Taxicab-Distance($\mathbf{p}, \mathbf{q}$)
for each $\mathbf{p}_t \in$ Neighbors($\mathbf{p}, Size$) **do**
  $\mathbf{q}_t \leftarrow Nearest[x_{p_t}, y_{p_t}]$
  $d_t \leftarrow$ Taxicab-Distance($\mathbf{p}, \mathbf{q}_t$)
  if $d_t < d$ **then**
    $d \leftarrow d_t$
    $\mathbf{q} \leftarrow \mathbf{q}_t$
  end if
end for
$Nearest[x_p, y_p] \leftarrow \mathbf{q}$

3.4 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. 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, 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 compared to *Wave Push* which only handles one direction each time. Moreover, it allow us to take advantage of data locality thus jumping some of the directions which needn't to be processed.

Our method includes 4 steps:

H-1 Perform JF-BFS to build the block list and perform back-and-forth relabel;
H-2 Repeat H-3 k_1 times, then use JF-BFS to build the list, then perform convergence detection;
  H-2-1 Perform JF-BFS to build the block list, repeat H-2-2 until no more nodes are marked;
  H-2-2 Perform back-and-forth detection;
    H-2-2-1 If any background node found, return to H-2-1, otherwise repeat H-2-1-2 until no more nodes are marked;
    H-2-2-2 Mark the current node according to the status of its neighbors.
H-3 Repeat H-4 k_2 times, count active blocks, use Scan to build two block lists for even blocks and odd blocks;
  H-3-1 For each block, set the flag by checking whether it contains active nodes;
  H-3-2 Perform scan primitive on the flags;
  H-3-3 Build two lists for even and odd blocks respectively;
H-4 Perform push-relabel for even blocks and odd blocks respectively;
  H-4-1 Repeat H-4-1 until no more nodes can be pushed;
  H-4-2 Relabel even nodes and odd nodes respectively, then push in the same way.

3.4.1 Back-and-Forth Relabel

In this step, 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 means first process the blocks in an ascending order and then in a descending order. It can help us to find out the paths that often change direction. Because after we finish relabel in one direction, the paths that pass along the same direction have already been found, and if we try an opposite direction then, we can easily find out all the paths that change their directions only once. For JF-BFS, the seed blocks are defined as including background nodes (with negative excess flow).

3.4.2 Convergence Detection

This step detects 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 have found the maximum flow. We perform JF-BFS staring from the foreground nodes because it can both 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 the nodes

should be connected, and all the edges belong to the minimum cut should be saturated. It means we can start from one part, repeat expanding to decide whether we have met the global optimum.

This step ensures that our algorithm can find the maximum flow thus get the optimal results. In the implementation, due to the expensive cost of this procedure, we repeat step 3 k_1 times and then perform it once. k_1 defaults to 4 and can be specified by users. Generally, it would be better to increase k_1 when data size becomes much larger.

3.4.3 Active Block Counting

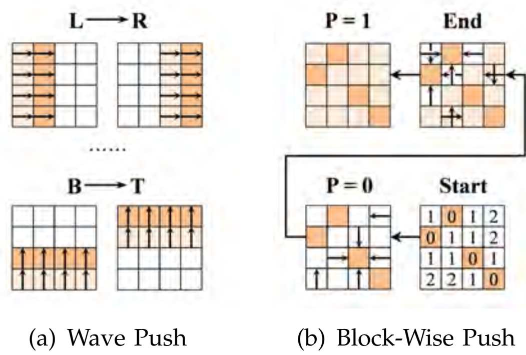
Active Block Counting is designed to reduce the computational cost by build a block list for active blocks, which is mainly based on the two facts: during the process only some of the nodes can be pushed or relabeled, and as more data is processed the number of active nodes will decrease. So only process active nodes efficiently improve the performance. Because active-blocks change slowly, we can do one counting after k_2 iterations. k_2 whose default value is 4 can also be specified by users. This procedure includes three sub-steps:

Here, we use two assistant 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 to be 0. After that, we perform scan to compact the flags into a scheduling lists.

3.4.4 Block-Wise Push-Relabel

As mentioned earlier, iterative push-relabel inside a block takes advantage of locality and increase 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. At each iteration, there are two stages: relabel and push. In the relabel stage, we first check whether a node is active, and then process even nodes and odd nodes respectively, because these two kinds of nodes have data conflict with each other. In the push stage, we



(a) Wave Push (b) Block-Wise Push

Fig. 3. Heuristic Push. 3(a): only handles one direction each time. 3(b): jumps some of the directions which needn't to be processed.

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
```

process the nodes in the same way. All the directions are tried one after another. When a node becomes inactive after push in one direction, we skip over the following directions. In this way, a lot of unnecessary computations can be avoided. Then we process the nodes in the same way.

4 USER INTERACTION

We design a user interface based on our approach for different datasets, see Fig.4. It can help users mark foreground and background based on visibility, build energy function according to the scenario and finally shows the graph results. Based on the results, users can also make local refinements.

4.1 Labeling

For 3D datasets, it is quite difficult for users to specify the foreground and background. The conventional way is to mark the nodes slice by slice. However, it takes users a long time to specify. In our system, we provide users a WYSIWYG way to do labeling (both for 2D and 3D datasets). The basic idea is to mark what users actually see based on visibility. This technique is combined with ray casting algorithm and users need to specify a visibility threshold α (defaults to 0.25)

In ray casting algorithm, the integration formula of the *Absorption Plus Emission* model [16] is given by:

$$I(D) = I_0 \prod_{i=1}^{n} t_i + \sum_{i=1}^{n} g_i \prod_{j=i+1}^{n} t_j \tag{1}$$

Where t_i and g_i denote the transparency of the ith segment and its glow respectively. The corresponding front-to-back compositing algorithm can be rewritten as follow:

$$\mathbf{c}_i = \begin{cases} \mathbf{0} & i = 1 \\ \mathbf{c}_{i-1} + \alpha_i(1-\alpha_{i-1})(r_i, g_i, b_i, 1) & 2 \le i \le n \end{cases} \tag{2}$$

where c is the color of the ith segment, and $vi = \alpha_i(1-\alpha_{i-1})$ is the opacity contribution of this segment. We define it as the *Visibility*, which is different from [17], [18]. If it is larger than the visibility threshold α, we define its neighboring nodes as what users want to mark. This technique can be integrated with transfer functions, so users can select the nodes with different depths at one time, which significantly simplify the interactions.

4.2 Scenario-Driven Energy Function

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:

$$E = \lambda \sum R(p) + \sum B(p, q) \tag{3}$$

where R and B denote the region properties and boundary properties respectively [19]. We provide users different ways to build the energy function. For a specific scenario, users can choose a best way to set up the model.

For scalar field data, we simplify the energy function introduced by BK which uses probability distribution to measure the similarity and has a low accuracy. We prefer to focus on the continuity of a structure:

$$C_s(p,q) = \begin{cases} 0 & p,q \in \mathcal{F} \bigcup \mathcal{B} \\ exp(-\frac{(I_p - I_q)^2}{2\sigma^2}) & otherwise \end{cases} \tag{4}$$

$$E_s(p) = \begin{cases} +E_{max} & p \in \mathcal{F} \\ -E_{max} & p \in \mathcal{B} \\ 0 & p \in \mathcal{U} \end{cases} \tag{5}$$

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

For vector field data, we define the region and boundary terms similar to *Lazy Snapping* which measures the similarity in the feature-space:

$$C_v(p,q) = \begin{cases} 0 & p,q \in \mathcal{F} \bigcup \mathcal{B} \\ \frac{1}{1+\|\mathbf{v}_p - \mathbf{v}_q\|} & otherwise \end{cases} \tag{6}$$

$$E_v(p) = \begin{cases} +E_{max} & p \in \mathcal{F} \\ -E_{max} & p \in \mathcal{B} \\ \lambda \frac{\min \|\mathbf{c}_i^{\mathcal{B}} - \mathbf{v}_p\| - \min \|\mathbf{c}_i^{\mathcal{F}} - \mathbf{v}_p\|}{\min \|\mathbf{c}_i^{\mathcal{B}} - \mathbf{v}_p\| + \min \|\mathbf{c}_i^{\mathcal{F}} - \mathbf{v}_p\|} & p \in \mathcal{U} \end{cases} \tag{7}$$

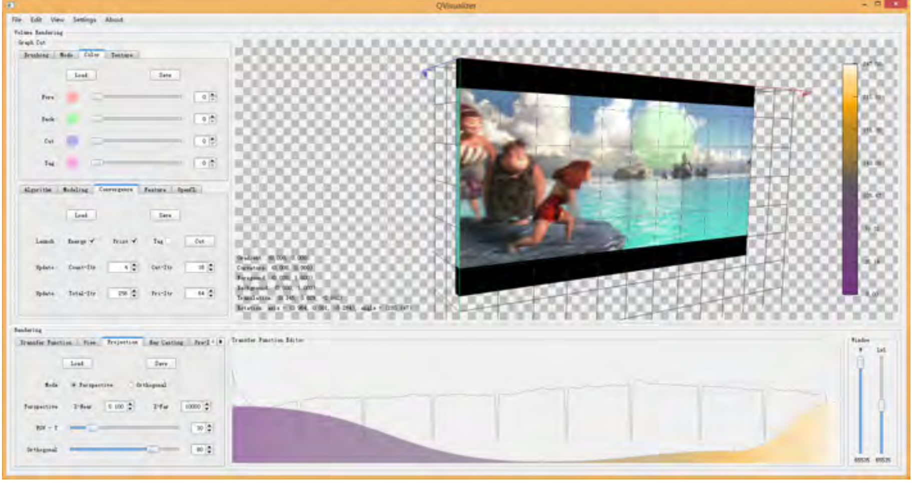


Fig. 4. User Interface. The JF-cut supports labeling and scenario-driven energy function.

Here 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 which improves the accuracy of results. In this algorithm, the number of clusters k defaults to 8, and the seed points are initialized using a histogram equalization technique.

Our system also supports feature extraction, which can convert a scalar data into a multi-dimensional data. These features include gradient, curvature [20] and so on.

5 Results and Discussions

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 X 870 @ 2.93GHz and Nvidia GeForce GTX TITAN @ 6GB (or ATI Radeon HD 7990). Different kinds of datasets are used to evaluate our method, and we compare our algorithm with both CPU based methods and GPU based methods.

5.1 Datasets

We use two kind of datasets, one of them is a part of the benchmark provided by Grid-Cut [1]. Below is the detail information of the datasets and the scaled-up versions of the datasets. These datasets are used to test the performance of different algorithms (TABLE 1).

TABLE 1
Benchmark Datasets

Name	Scale	Width	Height	Depth	Max Flow	Size (MB)
	1 x 1	600	450	1	1208089	3.35
	4 x 4	2400	1800	1	7856011	53.5
Flower(M)	16 x 16	9600	7200	1	50618597	856
	1 x 1	600	450	1	880461	3.35
	4 x 4	2400	1800	1	5042635	53.5
Person(M)	16 x 16	9600	7200	1	29042350	856
	1 x 1	640	480	1	343591	3.81
	4 x 4	2560	1920	1	1910718	60.9
Sponge(M)	16 x 16	10240	7680	1	10136071	975
	1 x 1 x 1	256	256	119	71344	111
Bone(U)	2 x 2 x 2	512	512	238	1913780	892
	1 x 1 x 1	170	170	144	625447	59.5
Liver(U)	2 x 2 x 2	340	340	288	3886049	476
	1 x 1 x 1	250	250	81	222943	72.4
Babyface(U)	2 x 2 x 2	500	500	162	3316927	579
	1 x 1 x 1	256	256	192	589368	180
Adhead(U)	2 x 2 x 2	512	512	384	3594110	1440

1) M - Middlebury College: http://vision.middlebury.edu/MRF/
2) U - University of Western Ontario: http://vision.csd.uwo.ca/data/maxflow/

Another set of datasets are downloaded from the internet as shown in TABLE 2, which in-

TABLE 2
Other Datasets

Name	Width	Height	Depth	Max Flow	Size (MB)
Madagascar	10800	8100	1	13536344426	83.4
Lower Than Atlantis	9900	7500	1	26054700795	70.8
TimeScapes	2560	1440	30	80886496	105.0
THE CROODS	1920	1080	40	180301245	79.1
Life of PI	1920	1080	32	1178329527	63.2
MRBrain	256	256	109	29036	13.6
Lobster	301	324	56	238405	5.2

cludes high-resolution images, HD videos and volume datasets.

We conduct experiments on these large datasets to show the practicality and efficiency of our method. Because our system only support 3D raw format currently, datasets need to be converted into this format. An image can be seen as a volume dataset that has only one slice, and a video can be processed as a series of images. In these experiments, the foreground and background are colored in magenta and green respectively, the cut results are highlighted in yellow.

Image Segmentation: The results of image segmentation includes two parts, as shown in Fig.5. The first part contains standard datasets (from Middlebury) with provided energy function. In the second part the data sizes are much larger, and we use the second clustering based model to set up energy function. These datasets come from NASA [1] [2]

Video Cutout: For video datasets, we use the same model (clustering based) to build energy function. These videos are clipped from official trailers of different films such as *THE CROODS* [4], *TimeScapes* [5]. *Life of Pi* [6]. Fig.6 shows the cutout results.

Volume Segmentation: For volume datasets (scalar field datasets), we use the first model to set up energy function. Because the augmenting-paths are much longer, the computing costs are much higher compared

1. Lower Than Atlantis: http://thecelebculture.files.wordpress.com/2013/08/a1posterlta.jpg
2. Madagascar: http://wallpaper.imgcandy.com/images/233316-jessica-alba-wardrobe-malfunction-original-source-of-image.jpg
4. TimeScapes - Trailer 2 4k 2560p: http://red.cachefly.net/TimeScapes4K2560p.mp4
5. THE CROODS - Official Trailer 3: http://www.youtube.com/watch?v=xrbwgn_kRBo
6. Life of Pi - Trailer 2 Official: http://www.youtube.com/watch?v=m7WBfntqUoA

TABLE 3
GPU Based Methods

Instance	CUDA-Cut		Fast Cut	JF-Cut	Speedup	
Name	%	Total	P/R	P/R	-/CUDA	-/Fast
flower	73.6	48.6	9.7	4.2	11.5	2.3
4 x 4	67.3	622.2	262.4	90.0	6.9	2.9
16 x 16	-	-	9636.7	2192.3	-	4.4
person	35.8	73.9	20.9	10.0	7.4	2.1
4 x 4	67.4	612.0	290.9	120.7	5.1	2.4
16 x 16	-	-	12131.0	3403.6	-	3.6
sponge	100.0	48.7	5.4	4.4	11.0	1.2
4 x 4	97.3	637.3	151.7	31.4	20.3	4.8
16 x 16	-	-	3689.5	595.3	-	6.2
bone	-	-	5643.6	616.9	-	9.1
2 x 2 x 2	-	-	68712.3	6749.9	-	10.2
liver	-	-	6211.4	585.7	-	10.6
2 x 2 x 2	-	-	165704.3	16441.2	-	10.1
babyface	-	-	7932.5	830.2	-	9.6
2 x 2 x 2	-	-	104277.6	11309.2	-	9.2
adhead	-	-	17084.4	1654.6	-	10.3
2 x 2 x 2	-	-	115991.0	10511.7	-	11.0

1) CUDA-Cut (v1.0) - http://cvit.iiit.ac.in/resources/cudacuts/

with images. The segmentation results of CT [7] and MRI [8] datasets are shown in Fig.7.

5.2 Performance

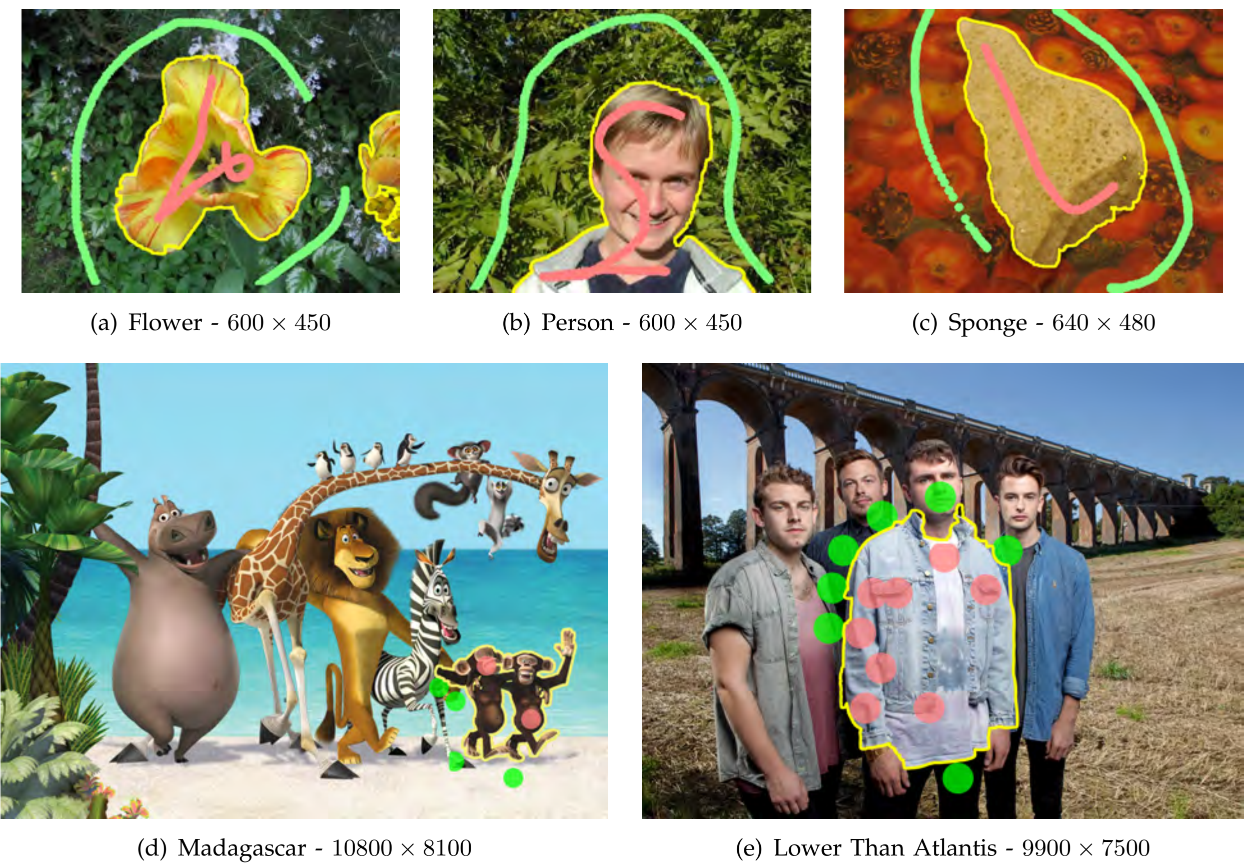
We first compare our method with CUDA-Cut [6] and Fast-Cut [7], [14] which are GPU based methods, and then with CPU based methods BK and Grid Cut.

GPU Based Methods: CUDA-Cut provides two different implementations: atomic (PushPull + Relbel) and stochastic (Push + PullRelabel). We use the stochastic version to compare, because it is faster. We also observe that CUDA-Cut doesn't guarantee global convergence and it often outputs larger energies. So we divide the minimum energy by its output to measure the degree of approximation as listed in the second column of TABLE 3.

We implement Fast-Cut using OpenCL and extends it to support 3D datasets (Some of the optimization techniques are deprecated for generality). To be fair, we perform global relabel once at the beginning, use the same block size and mainly compare the cost on push-relabel which is the major part of all GPU based methods.

TABLE 3 shows the performance of CUDA-Cut, Fast-Cut and our method with millisecond precision. Because CUDA-Cut consumes too much device memory, it can not handle

7. MRBrain: http://www-graphics.stanford.edu/data/voldata/
8. Lobster: http://www.volvis.org/



(a) Flower - 600 × 450 (b) Person - 600 × 450 (c) Sponge - 640 × 480

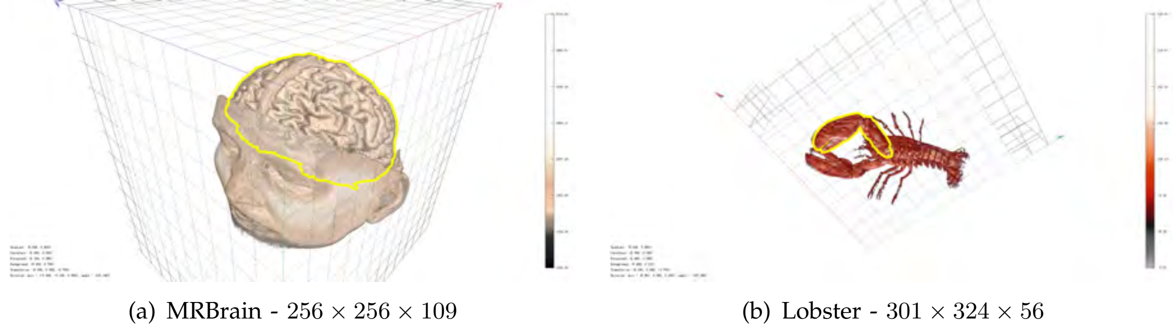
(d) Madagascar - 10800 × 8100 (e) Lower Than Atlantis - 9900 × 7500

Fig. 5. Image Segmentation Results. 5(a)-5(c): benchmark datasets with small size. 5(e)-5(d): large images. [1] [2]

(a) TimeScapes - 2560 × 1440 × 30 (b) THE CROODS - 1920 × 1080 × 40 (c) Life of PI - 1920 × 1080 × 36

(d) TimeScapes - Frame (e) THE CROODS - Frame (f) Life of PI - Frame

Fig. 6. Video Cutout Results. 6(a), 6(d): the cut-out bird [4]. 6(b), 6(e): the cut-out Eep [5]. 6(c), 6(f): the cut-out tiger [6].



(a) MRBrain - $256 \times 256 \times 109$ (b) Lobster - $301 \times 324 \times 56$

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

TABLE 4
Convergence Rate

Instance	Iterations			Ratios	
Name	CUDA	Fast	JF	-/CUDA	-/Fast
flower	155	41	10	15.5	4.1
4 x 4	232	174	81	2.9	2.1
16 x 16	-	516	195	-	2.6
person	232	100	26	8.9	3.8
4 x 4	232	203	88	2.6	2.3
16 x 16	-	692	366	-	1.9
sponge	155	31	14	11.1	2.2
4 x 4	232	114	27	8.6	4.2
16 x 16	-	234	84	-	2.8
bone	-	265	125	-	2.1
2 x 2 x 2	-	410	184	-	2.2
liver	-	498	170	-	2.9
2 x 2 x 2	-	1721	697	-	2.5
babyface	-	512	194	-	2.6
2 x 2 x 2	-	876	413	-	2.1
adhead	-	518	197	-	2.6
2 x 2 x 2	-	439	175	-	2.5

large datsets, even there are 6GB device memory. Our method achieve a maximum 19-fold speedup, see sponge 4×4. Larger datasets lead to higher speedups, which means our approach is more suitable for large datasets.

The convergence rate of different methods are shown in TABLE 4. Because the augmenting-path in volume datasets are longer than in images, volume datasets need more iterations. Our method has higher convergence rates which are more than twice the rates of Fast-Cut and almost 10-fold by CUDA-Cut.

CPU Based Methods: We download the latest version of BK (V3.02) and Grid-Cut (V1.1) and compile them to get the 64-bit programs. For large datasets, such as adhead (2 x 2 x 2), BK requires a contiguous memory which is larger than 4GB. We use the same compiler settings for comparison, see the table notes in TABLE 5.

TABLE 5
CPU Based Methods

Instance	BK	Grid-Cut	JF-Cut-N (TITAN)			JF-Cut-A (HD 7990)		
Name	Total	Total	Total	-/BK	-/Grid	Total	-/BK	-/Grid
flower	35	35	6	6.1	6.1	6	5.7	5.7
4 x 4	593	242	108	5.5	2.2	101	5.9	2.4
16 x 16	12615	4420	2560	4.9	1.7	1575	8.0	2.8
person	36	15	12	3.0	1.2	14	2.5	1.1
4 x 4	642	248	138	4.6	1.8	118	5.5	2.1
16 x 16	11736	3878	3930	3.0	1.0	2417	4.9	1.6
sponge	32	12	6	5.5	2.1	13	2.5	1.0
4 x 4	600	193	42	14.2	4.6	41	14.8	4.8
16 x 16	10358	3088	788	13.1	3.9	595	17.4	5.2
bone	6434	1262	731	8.8	1.7	421	15.3	3.0
2 x 2 x 2	256083	47977	7536	34.0	6.4	4707	54.4	10.2
liver	11589	5138	654	17.7	7.9	305	38.0	16.8
2 x 2 x 2	957459	306653	17470	54.8	17.6	7923	120.8	38.7
babyface	9524	3857	903	10.6	4.3	459	20.8	8.4
2 x 2 x 2	952648	310028	12257	77.7	25.3	5825	163.5	53.2
adhead	28757	9550	1828	15.7	5.2	980	29.3	9.7
2 x 2 x 2	282740	62222	12093	23.4	5.1	7921	35.7	7.9

1) BK (v3.02) - http://pub.ist.ac.at/~vnk/software.html
2) Grid-Cut (v1.1) - http://gridcut.com/downloads.php
3) Compiler Settings for GCC (v4.7.1) -O3 -march=native -mtune=generic -DNDEBUG

TABLE 5 compares the performance of BK, Grid-Cut and our method. The speedup increases with the data scale, except for some datasets. Because Grid-Cut is an optimized version of BK, its speedups lower than BK's. We achieve a maximum 6-fold speedup (flower - TITAN) for images and 164-fold speedup (babyface - HD 7990) for volume datasets. We also observe that there exists a notable variability, the decisive factor may be the graph topology. The reason is that in volume datasets most of the terminal-edges have zero capacity, while in images most of them are positive, which will greatly affect the convergence rate. 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 datasets our method significantly outperforms other methods.

5.3 Discussions

We analyze the tuning techniques that are used in our implementation and discuss the limitations of our approach.

Algorithm Tuning: For better performance, we develop a 2D version of our approach to handle images. Other optimizations includes using local memory to cache the data which has 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 compact structure (four bytes per unit) to support 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.

Quality: Compared with CUDA-Cut, our approach guarantees the convergence, meaning that we can get the optimal results. Compared with Fast-Cut, we use in-block push-relabel to improve the speed of information propagation, thus achieving fast convergence. And because of that, our approach can handle large datasets. More details of JF-Cut are available at https://github.com/15pengyi/JF-Cut.

Limitations: Our approach has several limitations compared with CPU-based methods. First, the feasible data size of our approach is limited by *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 CPU can be much larger, e.g. 32GB is available. The GPU based methods also have to copy data from host memory to device memory, which takes up extra time. Compared with BK our method currently only support structured data.

6 Conclusion

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

1) improving the performance of graph cut for large datasets 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 system that allows for intuitive data selection, interaction and segmentation for large-scale datasets.

We use a variety of datasets (both a benchmark and different large datasets) to evaluate the performance of our approach. The results show that our method achieve a maximum 164-fold speedup, and it be effectively used in different scenarios. The source code of JF-Cut will be soon available at https://github.com/15pengyi/JF-Cut.

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

Acknowledgments

We would like to thank Hongsen Liao, Min Wu, Zhu Zhu, Wenshan Zhou, Chaowei Gao and Kan Wu for their enthusiastic help and helpful suggestions.

This research is supported by Major Program of National Natural Science Foundation of China (61232012), National Science Foundation of China (61272225, 51261120376, 81172124), Chinese 863 Program (2012AA041606, 2012AA040902, 2012AA12090), Chinese 973 Program (2010CB328001) and the Important National Science & Technology Specific Projects (2011ZX02403).

References

[1] O. Jamriska, D. Sykora, and A. Hornung, "Cache-efficient graph cuts on structured grids," in *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, 2012, pp. 3673–3680.

[2] Y. Li, J. Sun, C.-K. Tang, and H.-Y. Shum, "Lazy snapping," in *ACM SIGGRAPH 2004 Papers*, ser. SIGGRAPH '04. New York, NY, USA: ACM, 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 *Computer Vision, 2005. ICCV 2005. Tenth IEEE International Conference on*, vol. 1, 2005, pp. 259–265 Vol. 1.

[4] Y. Boykov and V. Kolmogorov, "An experimental comparison of min-cut/max- flow algorithms for energy minimization in vision," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 26, no. 9, pp. 1124–1137, 2004.

[5] J. Liu and J. Sun, "Parallel graph-cuts by adaptive bottom-up merging," in *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, 2010, pp. 2181–2188.

[6] V. Vineet and P. J. Narayanan, "Cuda cuts: Fast graph cuts on the gpu," in *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW '08. IEEE Computer Society Conference on*, 2008, pp. 1–8.

[7] W. H. Wen-mei, *GPU Computing Gems Emerald Edition*. Morgan Kaufmann, 2011, vol. 1.

[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 *ACM SIGGRAPH 2005 Papers*, ser. SIGGRAPH '05. New York, NY, USA: ACM, 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," *The Visual Computer*, vol. 21, no. 8-10, pp. 745–754, 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 *Proceedings of the 45th annual ACM symposium on Symposium on theory of computing*, ser. STOC '13. New York, NY, USA: ACM, 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] T. Stich, "Graphcuts with cuda and applications in image processing," in *GPU Technology Conference*, 2009.

[15] D. MERRILL and A. GRIMSHAW, "High performance and scalable radix sorting: A case study of implementing dynamic parallelism for gpu computing," *Parallel Processing Letters*, vol. 21, no. 02, pp. 245–272, 2011. [Online]. Available: http://www.worldscientific.com/doi/abs/10.1142/S0129626411000187

[16] N. Max, "Optical models for direct volume rendering," *IEEE Transactions on Visualization and Computer Graphics*, vol. 1, no. 2, pp. 99–108, Jun. 1995. [Online]. Available: http://dx.doi.org/10.1109/2945.468400

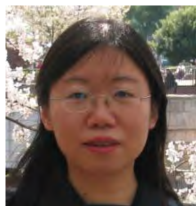
[17] U. D. Bordoloi and H.-W. Shen, "View selection for volume rendering," *Visualization Conference, IEEE*, vol. 0, p. 62, 2005.

[18] C. Correa and K.-L. Ma, "Visibility histograms and visibility-driven transfer functions," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 17, no. 2, pp. 192 –204, feb. 2011.

[19] Y. Boykov and M.-P. Jolly, "Interactive graph cuts for optimal boundary amp; region segmentation of objects in n-d images," in *Computer Vision, 2001. ICCV 2001. Proceedings. Eighth IEEE International Conference on*, vol. 1, 2001, pp. 105–112 vol.1.

[20] G. Kindlmann, R. Whitaker, T. Tasdizen, and T. Moller, "Curvature-based transfer functions for direct volume rendering: methods and applications," in *Visualization, 2003. VIS 2003. IEEE*, oct. 2003, pp. 513 –520.

Yi Peng is currently a Ph.D. student in Department of Computer Science and Technology at Tsinghua University. He received his B.S. in school of software from the Tsinghua University, China, in 2010. His research interests include data visualization, computer graphics and parallel computing.



Li Chen received the PhD degree in visualization from Zhejiang University in 1996. Currently, she is an associate professor in the institute of CG & CAD, School of Software, Tsinghua University. Her research interests include data visualization, mesh generation and parallel algorithm.

Fang-Xin Ou-Yang is currently a undergraduate student in School of Software, Tsinghua University. Her research interests include visualization and computer graphics.

Wei Chen Dr. Wei Chen is a professor in State Key Lab of CAD & CG at Zhejiang University, P.R.China. From June 2000 to June 2002, he was a joint Ph.D student in Fraunhofer Institute for Graphics, Darmstadt, Germany and received his Ph.D degree in July 2002. His Ph.D advisors were Prof. Qunsheng Peng, and Prof. Georgios Sakas. From July. 2006 to Sep. 2008, Dr. Wei Chen was a visiting scholar at Purdue University, working in PURPL with Prof.David S. Ebert. In December 2009, Dr.Wei Chen was promoted as a full professor of Zhejiang University. He has performed research in computer graphics and visualization and published more than 60 peer-reviewed journal and conference papers in the last five years. His current research interests include visualization, visual analytics and bio-medical image computing.

Jun-Hai Yong is currently a professor in School of Software at Tsinghua University. He received his B.S. and Ph.D. in computer science from the Tsinghua University, China, in 1996 and 2001, respectively. He held a visiting researcher position in the Department of Computer Science at Hong Kong University of Science & Technology in 2000. He was a post doctoral fellow in the Department of Computer Science at the University of Kentucky from 2000 to 2002. He obtained a lot of 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 Associate Editor of the Computers & Graphics Journal by Elsevier, and several National Excellent Textbook Awards. His main research interests include computer-aided design and computer graphics.