Real-Time Online Video Object Silhouette Extraction Using Graph Cuts on the GPU

Zachary A. Garrett and Hideo Saito

Department of Information and Computer Science, Keio University
3-4-1 Hiyoshi, Kohoku-ku, Yokohama, 223-8522 Japan
{zgarrett,saito}@hvrl.ics.keio.ac.jp

Abstract. Being able to find the silhouette of an object is a very important front-end processing step for many high-level computer vision techniques, such as Shape-from-Silhouette 3D reconstruction methods, object shape tracking, and pose estimation. Graph cuts have been proposed as a method for finding very accurate silhouettes which can be used as input to such high level techniques, but graph cuts are notoriously computation intensive and slow. Leading CPU implementations can extract a silhouette from a single QVGA image in 100 milliseconds, with performance dramatically decreasing with increased resolution. Recent GPU implementations have been able to achieve performance of 6 milliseconds per image by exploiting the intrinsic properties of the lattice graphs and the hardware model of the GPU. However, these methods are restricted to a subclass of lattice graphs and are not generally applicable. We propose a novel method for graph cuts on the GPU which places no limits on graph configuration and which is able to achieve comparable real-time performance in online video processing scenarios.

1 Introduction

Graph cutting is a technique that can be applied to energy minimization problems that occur frequently in computer vision. One of the most common applications of graph cuts is binary image segmentation, or object silhouette extraction. The groundwork for solving the push relabel maximum flow across graphs was laid through Goldberg and Tarjan's [1] reasearch, but the applicability to computer vision was not demonstrated until 1989 [2] and further expanded by Boykov et. al [3,4]. However, graph cuts have consistently been difficult to adapt to real-time scenarios due to the intensive computation required to construct and cut even a single graph.

Improving the speed of finding the *st*-mincut of a graph has been the goal of recent research on graph cuts. Research methods that have shown varying degrees of success include algorithms that re-use previous search trees [5], re-use of previous cuts to create a pseudo-flow which can be pushed and pulled through the graph [6], and finally implementations of push-relabel algorithms have been adapted to the GPU [7]. Currently, the fastest of these algorithms, presented by Vineet and Narayanan [8] in 2008, demonstrates a GPU implementation that

P. Foggia, C. Sansone, and M. Vento (Eds.): ICIAP 2009, LNCS 5716, pp. 985–994, 2009.

can compute the graph cut on a 640x480 image in approximately 6 milliseconds. This implementation exploits properties of the lattice graphs created from the input images and the hardware model in the graphics processor. Vineet and Narayanan admit that this approach makes the method only applicable to such graph types and not generally usable by any arbitrary maximum-flow calculation. Furthermore, these papers make no mention of the computation required for graph construction in the online video scenario.

In the case of online video, a graph must be constructed from each video frame as it arrives at the CPU. In contrast, offline methods layer all of the video frames into a 3D volume and cut once across the entire video sequence [9]. On traditional computing platforms, this necessitates techniques which speed up not only the cutting of the graph, but also the construction of the graph. One proposed method uses a laplacian pyramidal structure of decreasing resolutions of the input image to cut decreasingly coarse graphs [10]. This method works well for generally round objects without holes, but has difficulty with other input. We previously presented an online method for live video situations that uses the silhouette of the previous segmentation as a mask for graph construction, in effect shrinking the graph [11]. However, none of these methods compute faster than 100 milliseconds per QVGA frame. To realize real-time applications of graph cuts, large-scale parallel computing needs to be considered.

Recent advances in GPU hardware have given researchers access to low cost, large SIMD arrays which allow data level parallelization of algorithms. Research into how to harness the power of the graphics processor has lead to graphics card manufacturer's providing general purpose graphics processing unit (GPGPU) frameworks such as CUDA [12] by NVIDIA. In the field of computer vision, two such software libraries are CVGPU [13] and OpenVidia [14]. These libraries support many of the basic operations in computer vision with speed improvements of up to 100x, particularly in linear filtering tasks. Research has shown that the push-relabel algorithm is able to run in parallel due to the design of the method and various CPU based approaches having already been presented [15,16].

In this paper we present a fast, generic implementation of the push-relabel algorithm based on the lock-free method proposed by Hong [15] using the CUDA framework and the power of the GPU. Using this method in the system we proposed in [11], we are able to track images in online and offline video and obtains the silhouette of the tracked object at each frame. Our method is adaptable to any graph construction and runs at a 30 Hz frame rate for QVGA images and 4-5 Hz frame rate for HD TV images . By accepting any graph construction, the method is not restricted to lattice graphs, allowing the technique to be applied to maximum flow solutions in other problem spaces, such as graph cuts across spaces of greater than three dimensions [17].

Section 2 presents the GPU framework provided by CUDA and describes how programs are parallelized within this framework. In Section 3 we describe our GPU implementation of a lock-less generic push-relabel graph cut algorithm. Section 4 describes our experimental setup and discusses our experimental

results. Finally, Section 5 gives concluding remarks and the direction for our future work.

2 GPU Programming Framework

GPGPU research initially focused on representing computations as graphics renderings. By encoding data arrays into textures and program kernels written in fragment or vertex shaders, computation could be performed by rendering the image. This method has limitations: read-back performance of the result was poor for early hardware, kernel operations were limited, and data structure sizes were limited to texture sizes [18]. To overcome these limitations, both NVIDIA and AMD released new high-level programming interfaces named CUDA and Stream SDK in 2007, respectively. These programming interfaces improved the memory access interface and created an easier to use programming environment, which led to an increase in popularity of porting existing programs to the GPU.

CUDA allows researchers to easily parallelize their computing task to take advantage of the GPU. CUDA provides a software library to interface with the graphics processor, and a specialized compiler to create executable code to run on the GPU. CUDA presents programmers with two other types of memory access, shared memory and global memory, on top of the texture cache units. Being able to access all the memory on the GPU (albeit with varying access speeds) has made it much easier to perform research into GPGPU computing. The method presented in this paper was built using the CUDA architecture and run on NVIDIA hardware. Vineet and Narayanan have previously explained the CUDA framework in detail [8], and there are many technical resources available on the NVIDIA CUDA homepage [19].

3 Graph Cuts

The traditional method for push-relabel graph cutting is outlined in [1]. The algorithm consists of two main operations: *discharge* (also called *push*) and *relabel* (also called *raise*). In push-relabel algorithms, vertices have two states: overflowing and inactive. Vertices contain information concerning the amount of *excess* and *height*. Edges hold information about their *residual capacity*. Typically a queue of overflowing vertices is checked each iteration, with the first vertex in the list being dequeued and either *discharged* or *relabeled*. The loop terminates once there are no more viable *discharge* or *relabel* operations to perform.

Discharging a vertex causes some of the excess to be pushed to neighboring vertices when the incident edge has not completely saturated (has residual capacity). The push decreases the excess in the current vertex, increasing excess in the destination, and decreases the residual capacity of the edge. The destination vertex is then enqueued into the overflowing list, along with the current vertex if it is still overflowing.

A vertex must be *relabeled* when it is overflowing but has no valid neighbors. A neighbor is valid if the edge incident to both vertices has remaining capacity,

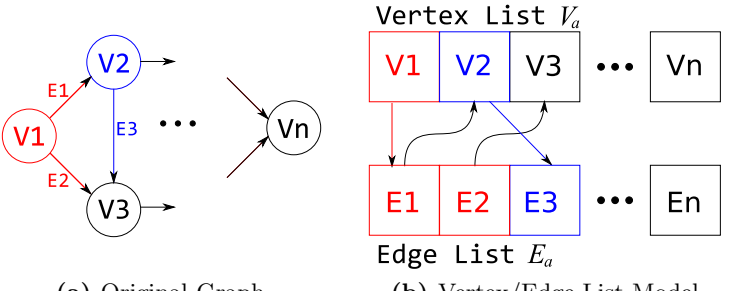
and the destination vertexs height is lower than the current vertex. The current vertex's height is increased to one more than the lowest neighbor, which allows further *discharging*.

3.1 Novel GPU Graph Cut Technique

To develop a GPU implementation of a traditional algorithm, both data structure layout (the graph representation in memory) and concurrency bottlenecks must be carefully considered before being able to realize the full potential of the GPU. CUDA implementations of graph cuts for image segmentation have traditionally paired the 2D lattice structure of graphs with the 2D grid structure of the CUDA programming model. However this leads to restrictions on the types of graphs that can be processed by the routine. Furthermore, traditional graph cut algorithms contain branching and looping. In CUDA, divergent code produces poor performance because the SIMD model cannot perform the instructions in parallel, making new techniques for graph cuts necessary.

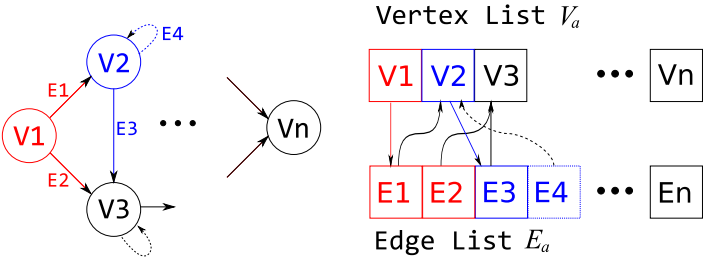
Graph Representation. Vineet and Narayanan presented a method of graph cuts in CUDA which has shown improved performance by pairing the 2D lattice graphs with the CUDA programming model. They conceded that this method would not be feasible for other types of graphs [8]. We structure our graphs similar to the technique presented by Harish et. al [20] that improved performance in distance calculations on arbitrary graphs using the GPU by representing graph $G(V, E)$ as a vertex array V_a and an edge array E_a. Edges in the edge list will be grouped so that all edges that originate at a vertex are contiguous, and the vertex will contain a pointer to the first edge in the group, as in Figure 1. Each edge structure holds information about its capacity, current flow, and destination vertex.

GPU Maximum Flow. Due to the design of the traditional push relabel algorithm (conditionals and the looping over a queue), the algorithm is not immediately useable on the GPU. The first step is to convert the algorithm loop to the CUDA grid. This is a fairly simple process of creating a CUDA kernel



(a) Original Graph **(b)** Vertex/Edge List Model

Fig. 1. Conversion of a direct graph into vertex and edge lists



(a) 2-regular Original Graph **(b)** Vertex/Edge List Model

Fig. 2. Addition of null edges to achieve a k-regular graph

that will perform either a *discharge* or *relabel* for a given vertex each invocation, and terminate once no discharging or relabeling is feasible. Another method is to create multiple CUDA kernels, typically consisting of a kernel that will push all possible flow from a vertex or mark it as needing relabeling, and another kernel that updates all the labels of vertices previously marked. In this method many different types of kernels must be coded and invoked many times.

3.2 Optimization Techniques

We improved upon the basic implementation of vertex and array lists. To make our list based graph representation more SIMD friendly, we first make the input graph *regular* by adding null edges to any vertex that has fewer edges than the vertex with the highest degree. Second, we simplify the code by only having one kernel, which performs both the discharging and relabeling steps.

Since the GPU is treated as an array of SIMD processors, it is important that the algorithm is able to perform the same operation across multiple data locations. However, we are assuming that any kind of graph configuration could be used as input, requiring that each vertex be treated uniquely, as the vertex set is not homogeneous. To overcome this problem, we propose adding extra edges to the graph so that each vertex has the same degree. In Figure 2, the dotted line of edge $E4$ is an example of a null edge. These null edges point back to the originating vertex and have a capacity of zero. This would cause the null edges to be ignored by the graph cut algorithm, as the destination of the edge would never have a height lower than the origin (a condition for discharging), and the capacity of the edge restricts the amount of flow discharged to zero.

Algorithm 1 details the steps of the computation that take place on the CPU. The *initialize* function starts by discharging the maximum possible amount of flow from the source to the neighboring vertices. Then the GPU kernel is invoked so that one thread is excuted for each non-source and non-sink vertex. The CPU continues to invoke the GPU kernel until no discharge or relabel operations can be performed. The GPU kernel is detailed in Algorithm 2. Since our graph is regular, the loop over the neighbors is fully unrolled. The null edges will be ignored because the height of u will never be less than the current vertex's

Algorithm 1. GPU_GraphCut(Graph $G(V, E)$)

```
finished ← false
Initialize(V, E)
while not finished do
    // GraphCutKernel() is performed in parallel on the GPU
    finished ← GraphCutKernel(V,E)
end
```

Algorithm 2. GPU_GraphCut_KERNEL($G(V, E)$)

```
tId ← GetThreadID()
amtPushed ← 0
foreach Neighbor u of V[tId] do
    if u.height < V[tId].height and (V[tId], u) ∈ E has capacity > 0 then
        amtPushed ← amtPushed + Discharge(V[tId], u)
    end
end
if amtPushed > 0 and V[tId].excess > 0 then
    V[tId].height = FindLowestNeighbor(V[tId]).height + 1
end
```

height (since they are equal). In addition the discharge can be modified to push a conditionally assigned amount of flow. Conditional assignment allows us to use the ternary operator, which prevents divergent code since it can be optimized to a zeroing of either operand of an addition.

4 Experimental Results

To test the performance of our proposed technique, we performed a comparison of a CPU implementation of image segmentation using graph cuts versus our proposed GPU implementation. The CPU implementation uses the open source library provided by Vladimir Kolmogorov [5], using OpenCV for data capture and graph construction. The test computer used was running Windows XP. The CPU implementation ran on an Intel Quad Core Xeon processor, and the GPU implement ran on an NVIDIA GTX 280 graphics card. All of these components are regularly available at any electronics retailer. A video sequence scaled to four different resolutions (QVGA, VGA, HD720, and HD1080) was used in testing the speed for video object silhouette extraction.

The speed for the cut only is presented in Figure 3. At higher resolutions the parallelism achieved by the GPU significantly increases the performance, as much as 2.5x. However, we begin to see real performance gains when the whole process, from frame acquisition to graph building and finally silhouette recovery, is considered. Since all of these steps can be done in parallel, we see major speed improvements of up to 10x times in full HD 1080 resolution (1920x1080), as shown in Figure 4. This emphasizes the overhead caused by the non-cut

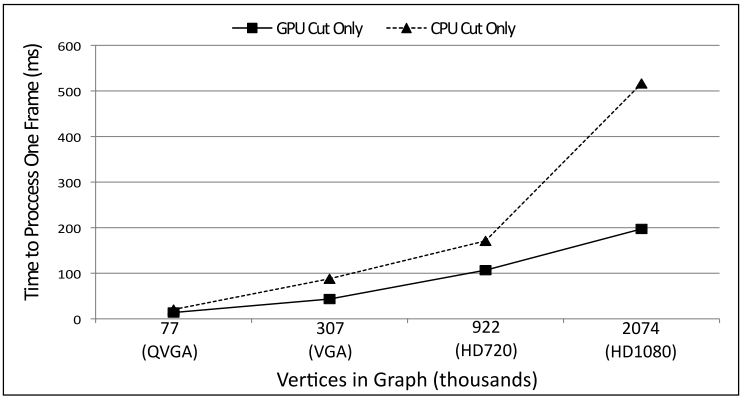


Fig. 3. Speed of graph cut (max-flow) calculation versus graph size

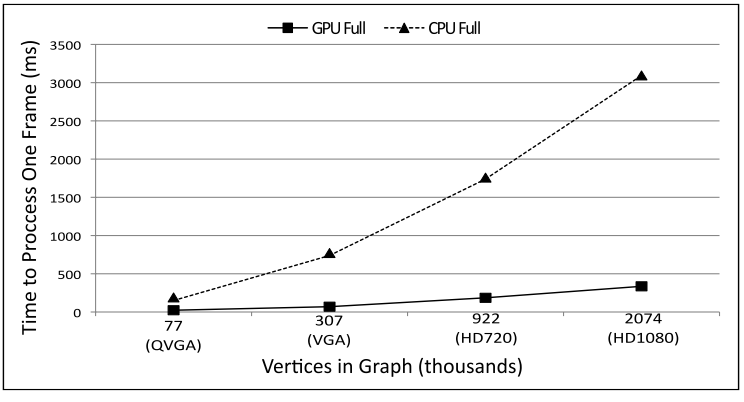


Fig. 4. Speed of frame silhouette retrieval versus graph size

operations, such as graph construction, in the cutting of image sequences. It is important to note that these tests do not use any kind of graph shrinking techniques, and that both implementations are cutting across the full HD TV image, which consists of over 2 millions vertices and 37 million edges.

In addition we chose a subset of images from the Berkeley Segmentation Dataset [21] to test the accuracy of the results between the two implementations. The dataset contains segmetations proposed by a set of human subjects, which we converted to silhouettes of the desired object so that they could be compared to our graph cut output. The test is designed to make sure that the silhouettes obtained are within a reasonable margin of error, and that the GPU algorithm is not returning erroneous results. Figure 5 gives three examples of the output from both the GPU and the CPU implementations. The border of the result silhouette is highlighted in cyan. Simple observation shows the cuts to be approximately visually equivalent.

Table 1. Accuracy of Graph Cut implementations

	Error	StdDev	Fp	Fn
GPU	4.3%	0.9%	2.3%	2.0%
CPU	3.4%	0.4%	1.9%	1.7%

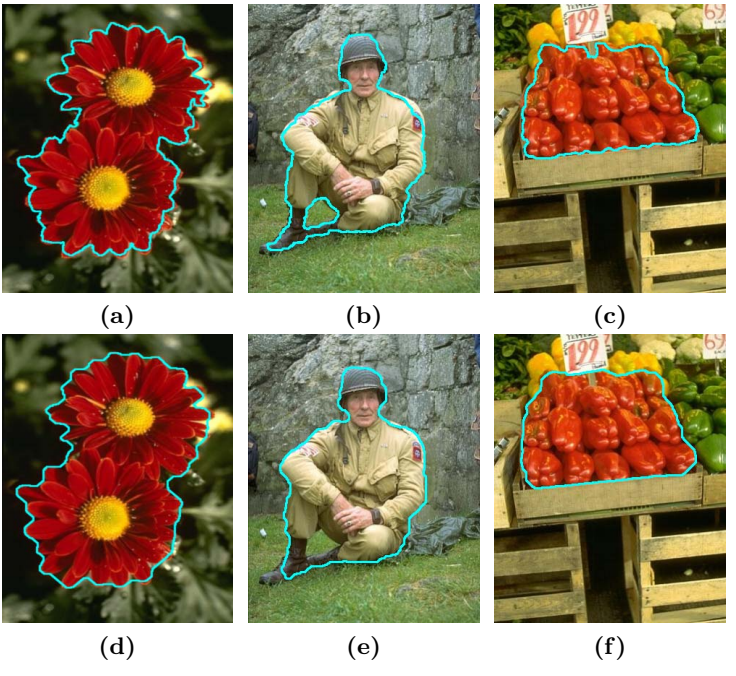


Fig. 5. GPU (top) and CPU (bottom) output (border highlighted)

For a numerical comparison of the silhouettes, Table 1 shows the percentage of mislabeled pixels (the ratio of mislabeled pixels to total image pixels), the standard deviation (StdDev) of mislabeled pixels, as well as the mean rate of false positives and false negatives. The results demonstrate that both implementations give the same highly accurate segmentations.

5 Conclusions and Future Works

In this paper we have presented a new method for performing graph cuts on the GPU using CUDA. The method described is capable of handling arbitrary graph structures and is able to optimize them for the SIMD processing model employed on the GPU. We have shown that this technique enables large speedups in processing time, particularly for graphs with millions of vertices, achieving 10 frames-per-second processing High Definition video. Our future research

directions include finding ways to use the results of this technique in higher end vision processing systems.

References

1. Goldberg, A.V., Tarjan, R.E.: A new approach to the maximum flow problem. Journal of the ACM 35, 921–940 (1988)
2. Greig, D.M., Porteous, B.T., Seheult, A.H.: Exact maximum a posteriori estimation for binary images. Journal of the Royal Statistical Society 51(2), 271–279 (1989)
3. Boykov, Y., Veksler, O., Zabih, R.: Markov random fields with efficient approximations. IEEE Conference on Computer Vision and Pattern Recognition, 648–655 (1998)
4. Boykov, Y., Veksler, O., Zabih, R.: Fast approximate energy minimization via graph cuts. IEEE Transactions on Pattern Analysis and Machine Intelligence 23 (2001)
5. Boykov, Y., Kolmogorov, V.: An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. IEEE Transactions on Pattern Analysis and Machine Intelligence 26, 359–374 (2004)
6. Juan, O., Boykov, Y.: Active graph cuts. IEEE Computer Society Conference on Computer Vision and Pattern Recognition 1, 1023–1029 (2006)
7. Hussein, M., Varshney, A., Davis, L.: On implementing graph cuts on cuda. First Workshop on General Purpose Processing on Graphics Processing Units (2007)
8. Vineet, V., Narayanan, P.: Cuda cuts: Fast graph cuts on the gpu. In: IEEE Conference on Computer Vision and Pattern Recognition: Workshop on Visual Computer Vision on GPUs, June 2008, pp. 1–8 (2008)
9. Boykov, Y., Funka-Lea, G.: Graph cuts and efficient n-d image segmentation. Int. J. Comput. Vision 70(2), 109–131 (2006)
10. Lombaert, H., Sun, Y., Grady, L., Xu, C.: A multilevel banded graph cuts method for fast image segmentation. In: Proceedings of the Tenth IEEE International Conference on Computer Vision (ICCV 2005), Washington, DC, USA, vol. 1. IEEE Computer Society, Los Alamitos (2005)
11. Garrett, Z., Saito, H.: Live video object tracking and segmentation using graph cuts. In: International Conference on Image Processing, pp. 1576–1579. IEEE, Los Alamitos (2008)
12. nVidia: NVIDIA CUDA Compute Unified Device Architecture - Programming Guide (2007)
13. Farrugia, J.P., Horain, P., Guehenneux, E., Alusse, Y.: Gpucv: A framework for image processing acceleration with graphics processors. In: 2006 IEEE International Conference on Multimedia and Expo., pp. 585–588 (2006)
14. Fung, J., Mann, S.: Openvidia: parallel gpu computer vision. In: Proceedings of the 13th annual ACM International Conference on Multimedia, pp. 849–852. ACM, New York (2005)
15. Hong, B.: A lock-free multi-threaded algorithm for the maximum flow problem. In: IEEE International Parallel and Distributed Processing Symposium, pp. 1–8. IEEE, Los Alamitos (2008)
16. Anderson, R., Setubal, J.C.: A parallel implementation of the push-relabel algorithm for the maximum flow problem. J. Parallel Distrib. Comput. 29(1), 17–26 (1995)

17. Yu, T., Xu, N., Ahuja, N.: Reconstructing a dynamic surface from video sequences using graph cuts in 4d space-time. In: 17th International Conference on Pattern Recognition, pp. 245–248. IEEE Computer Society, Los Alamitos (2004)
18. Harris, M.: Gpgpu: Beyond graphics. In: Game Developers Conference (2004)
19. nVidia: Cuda zone – the resource for cuda developers (2009), `http://www.nvidia.com/object/cuda_home.html`
20. Harish, P., Narayanan, P.J.: Accelerating large graph algorithms on the GPU using CUDA. In: Aluru, S., Parashar, M., Badrinath, R., Prasanna, V.K. (eds.) HiPC 2007. LNCS, vol. 4873, pp. 197–208. Springer, Heidelberg (2007)
21. Martin, D., Fowlkes, C., Tal, D., Malik, J.: A database of human segmented natural images and its application to evaluating segmentation algorithms and measuring ecological statistics. In: Proc. 8th Int'l Conf. Computer Vision, July 2001, vol. 2, pp. 416–423 (2001)