EC527 Final Project

IMAGE SEGMENTATION WITH GRAPH-CUT AND MAXFLOW

Zuxiong Tan[1], Chenxi Li[2], Samyak Jain[3]

Contents

[1] Responsible for baseline part of the project

[2] Responsible for GPU part of the project

[3] Responsible for multithreaded part of the project

List of Figures and Tables

1. Introduction to Image Segmentation

Image segmentation is the process of partitioning a digital image into multiple segments, and the segment is a set of pixels, also known as image objects. The purpose of segmentation is to simplify and an image and acquire the features for more meaningful and easier analyzation. Image segmentation is typically used in locating objects and boundaries (lines, curves, etc.) in images. To be more specific, image segmentation is the process of assigning a label to every pixel in an image such that pixels with the same label share certain features.

The result of image segmentation is a set of segments that collectively cover the whole image or extract a set of contour lines(edge detection). The graph below shows the result of image segmentation. In our project, the method we use graph-cut algorithms, maxflow and mincut, to extract foreground image.

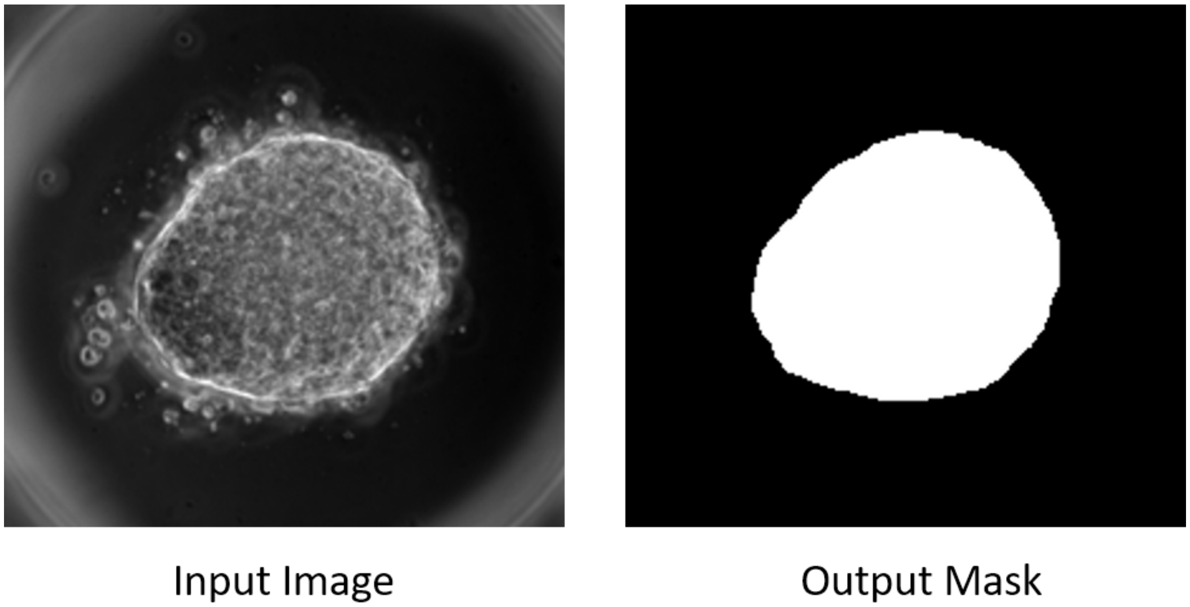


Fig 1. Image segmentation

a. Method used for Image Segmentation: Graph-Cut

A graph cut is the method of partitioning a graph (directed or undirected) into disjoint sets. The concept of optimality of such cuts is usually introduced by associating an energy to each cut i.e. build graph with image pixels as nodes with 2 imaginary nodes as *Source* and *Sink*.

After this, we find the cut that minimize total energy of the graph. Labelling resulting in minimum energy yields the most accurate segmentation.

Graph cut methods have been successfully applied to stereo image restoration, texture synthesis and image segmentation.

$$E(L) = \alpha R(L) + B(L)$$

where,

R[L] is the regional property

and

B[L] is the boundary property

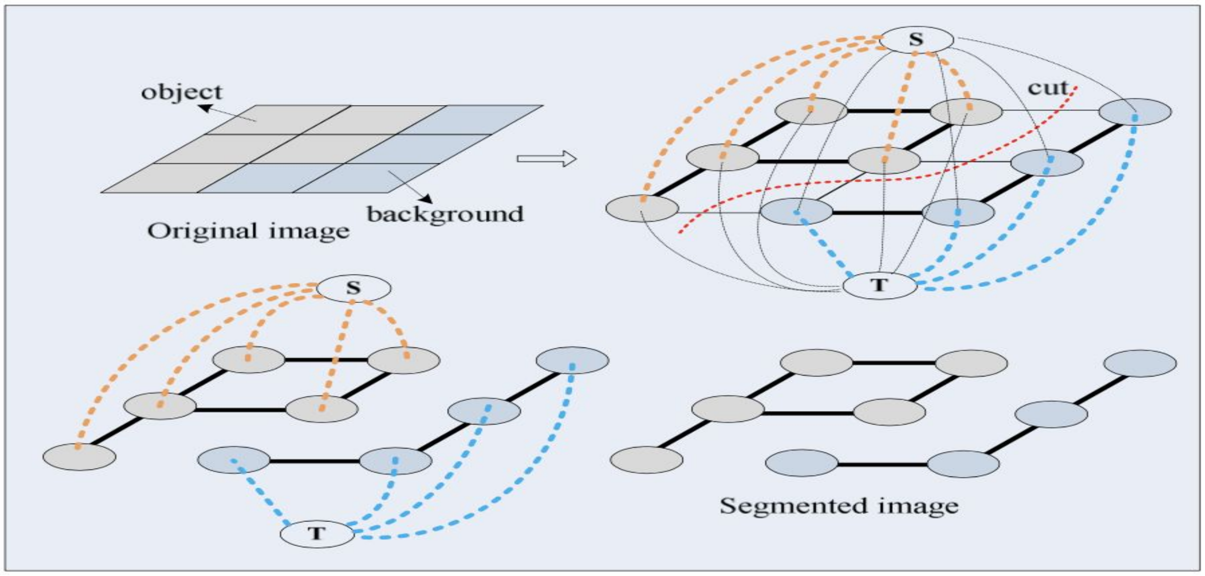


Figure 2. Graph-Cut for Image Segmentation

b. Max-flow min-cut Algorithm

The full name of maxflow is the max-flow min-cut theorem, it states that in a flow network, the maximum amount of flow passing from the source to the sink is equal to the total weight of the edges in the minimum cut. the smallest total weight of the edges which if removed would disconnect the source from the sink.

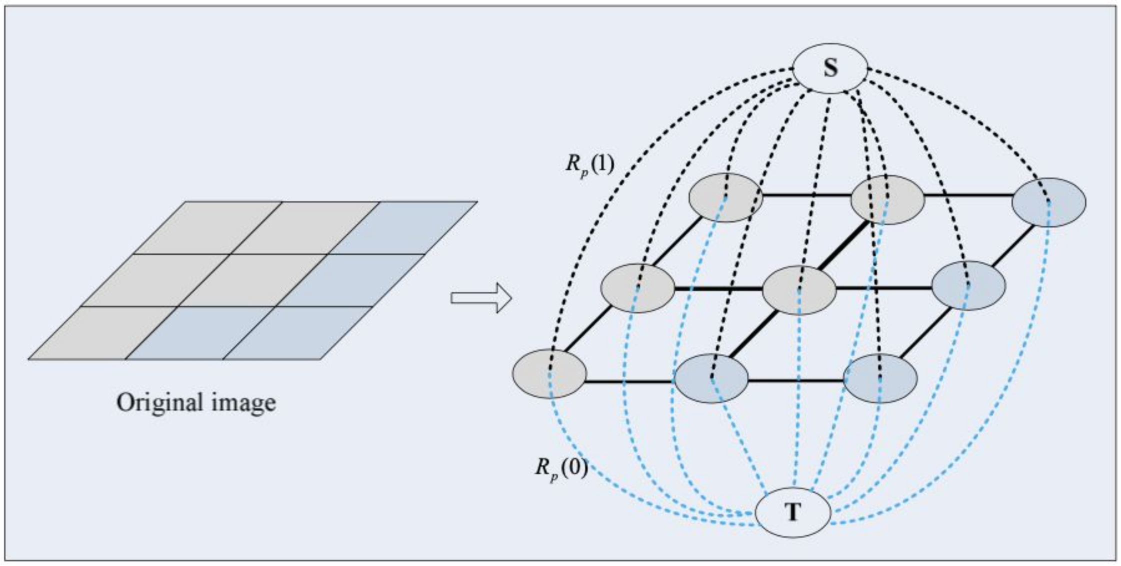


Figure 3. Maxflow diagram

The theorem relates two quantities: the maximum flow through a network, and the minimum weight of a cut of the network. To state the theorem, each of these quantities must first be defined. Let N = (V, E) be a directed graph, where V denotes the set of vertices and E is the set of edges. Let s ∈ V and t ∈ V be the source and the sink of N, respectively. The capacity of an edge is a mapping c: E → R+, denoted by c(u, v) where u,v ∈ V. It represents the maximum amount of flow that can pass through an edge. Figure 3. shows a simple example for max-flow network.

To achieve the maxflow, there are several algorithms. For example, Ford–Fulkerson algorithm, Edmonds–Karp algorithm and push-relabel maximum flow algorithm and so on. In this project, we use the push-relabel algorithm.

Table 1 shows the equations we use to build edge capacities in the graph. P and q are pixels nodes, S is source and T is destination.

TABLE I. WEIGHT OF EDGES

edge	weight	condition
<p, q>	$B_{<p,q>}$	$\{p, q\} \in N$
{p, S}	$\alpha \cdot R_p(0)$	$p \in P$ (unknown)
	K	$p \in$ object
	0	$p \in$ background
{p, T}	$\alpha \cdot R_p(1)$	$p \in P$ (unknown)
	0	$p \in$ object
	K	$p \in$ background

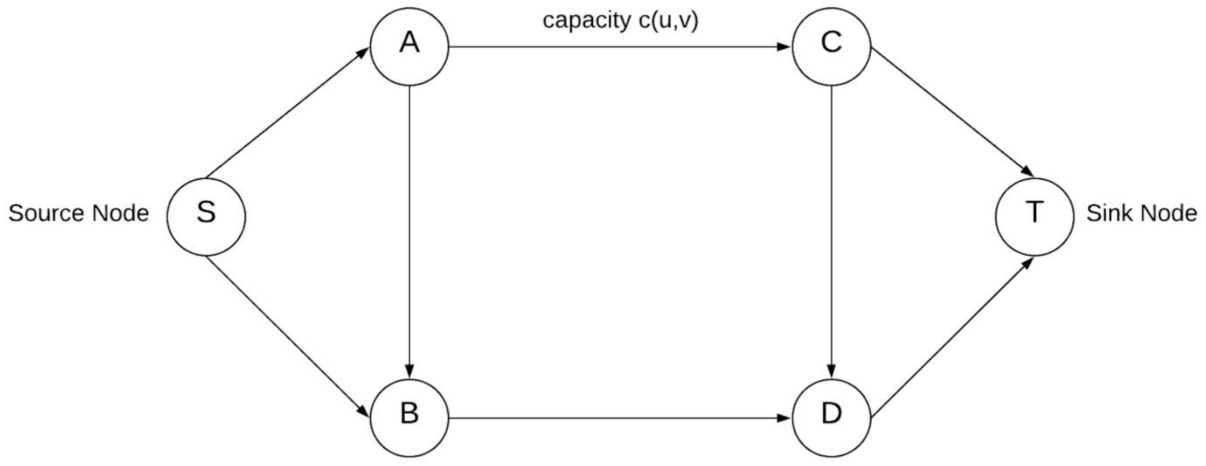


Figure 4. Simple example for Max Flow Network

c. Push-Relabel Method

The push-relabel algorithm is an algorithm to compute the maximum flows in a network. Its two basic operations are “push” and “relabel”. At first, the algorithm generates a “preflow” and store it for further use. Then using this “preflow” and gradually converts it into a maximum flow by moving flow locally between neighboring nodes using push operations under the guidance of an admissible network maintained by relabeling operations. The push operation increases the flow on a residual edge, and a height function on the vertices controls which residual edges can be pushed. The height function is changed with a relabel operation. The proper definitions of these operations guarantee that the resulting flow function is a maximum flow. Figure 4. shows a diagram of the basic push-relabel algorithm.

The time complexity of the algorithm is $O(V^2E)$.

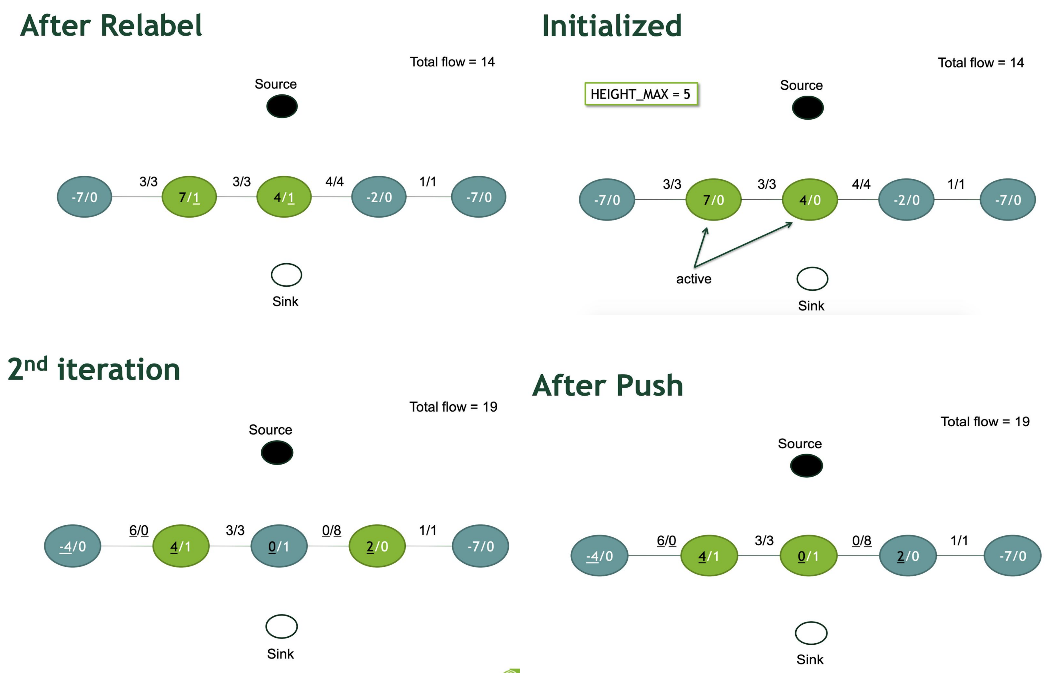


Figure 5. Baseline Implementation for push-relabel algorithm

2. Serial CPU version

Based on the theory of the algorithm, “push” and “relabel” are the two important parts of serial CPU code. We wrote the serial code based on python.

The “push” function is shown as follows, where F is the residual network and C is the original network. “u” and “v” are two connected notes.

a. Code

```
def push(u, v):
    send = min(excess[u], C[u][v] - F[u][v])
    F[u][v] += send
    F[v][u] -= send
    excess[u] -= send
    excess[v] += send
```

Then is the relabel part, the code is shown below. Relabel is to adjust the parameters(i.e. height) in each node.

```
def relabel(u):
    # find smallest new height making a push possible,
    # if such a push is possible at all
    min_height = float('inf')
    for v in range(n):
        if C[u][v] - F[u][v] > 0:
            min_height = min(min_height, height[v])
            height[u] = min_height + 1
```

In this program, both push and relabel are run simultaneously, until the height of the node becomes zero or the excess flow inside the node cannot push anymore. The input of the network is a matrix, shown in table 1 and for the sake of understanding the diagram is shown in figure 5.

Node:0	Node:1	Node:2	Node:3	Node:4	Node:5	
0	0	0	0	0	0	**Node:0**
16	0	4	0	0	0	**Node:1**
13	10	0	9	0	0	**Node:2**
0	12	0	0	7	0	**Node:3**
0	0	14	0	0	0	**Node:4**
0	0	0	20	4	0	**Node:5**

Table 1. Input Matrix

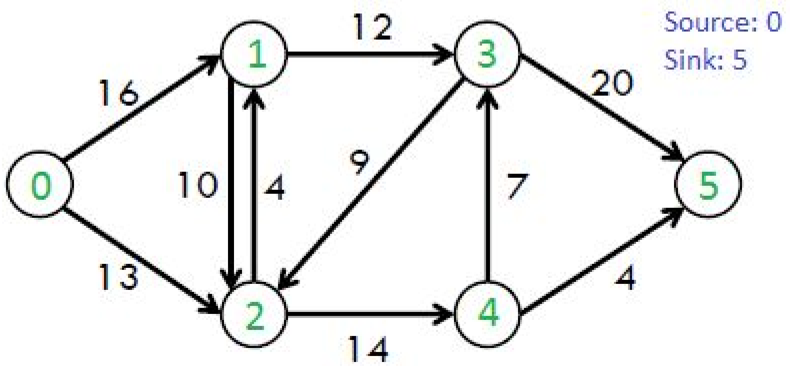


Figure 6. Input Network Diagram

With the input size increases, this matrix will eventually become a sparse matrix.

b. Output

The above shown serial code was run for different image sizes, ranging from 10x10 to 100x100. It was observed that as the image size increased, the time for computation increased which was expected. The results for the same are as below:

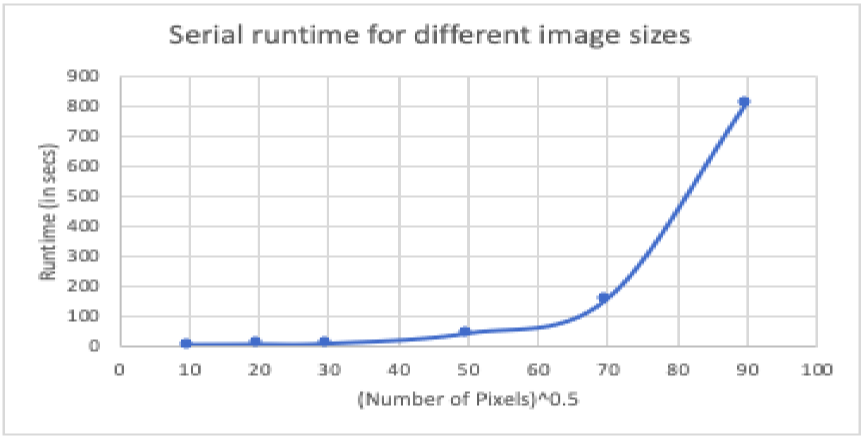


Figure 7. Serial CPU runtime for different image sizes

As it can be seen from the above results that the time required to compute max-flow increases exponentially when the image size exceeds the limit 30x30. Therefore, it can be deduced from this that the serial version of the push-relabel algorithm has really poor performance with large image sizes and can only work with small images.

3. Multithreaded CPU version

After implementing the serial version of the push-relabel algorithm, we multithreaded the serial version on CPU to get performance improvements. The code is written in Python and we used the **“Threading”** library of python to create multiple threads of the code.

We used a maximum of 8 threads for running the multithreaded version. As shown in Figure 7., it can be seen that the time taken for an image size of 30x30 with 8 threads is about 2.5 secs.

a. Code

The “push” and the “relabel” functions for this are the same as used in the serial function. The code that is added to multithread this algorithm is as below:

```
#Worker function
def worker(a, thread_num):
    p = a
    while p < len(nodelist):
        u = nodelist[p]
        old_height = height[u]
        discharge(u)
        if height[u] > old_height:
            nodelist.insert(a, nodelist.pop(p)) # move to front of list
            p = a # start from front of list
        else:
            p += thread_num
```

```
threads = []
thread_num = 8      #number of threads
for i in range(thread_num):
    t = threading.Thread(target=worker, args=(i, thread_num))
    threads.append(t)
    t.start()
for thread in threads:
        thread.join()
```

b. Output

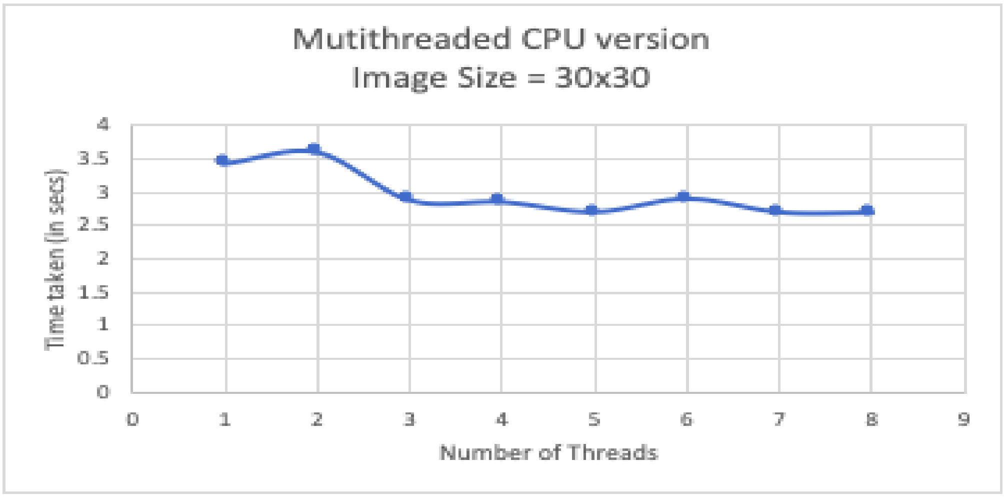


Figure 8. Multithreaded CPU Version

As seen in *Figure 7*., the image size here has been kept at 30x30 (which is very small) because the multithreaded version for the Push-Relabel algorithm can only work on small image sizes and for large image sizes, the performance worsens.

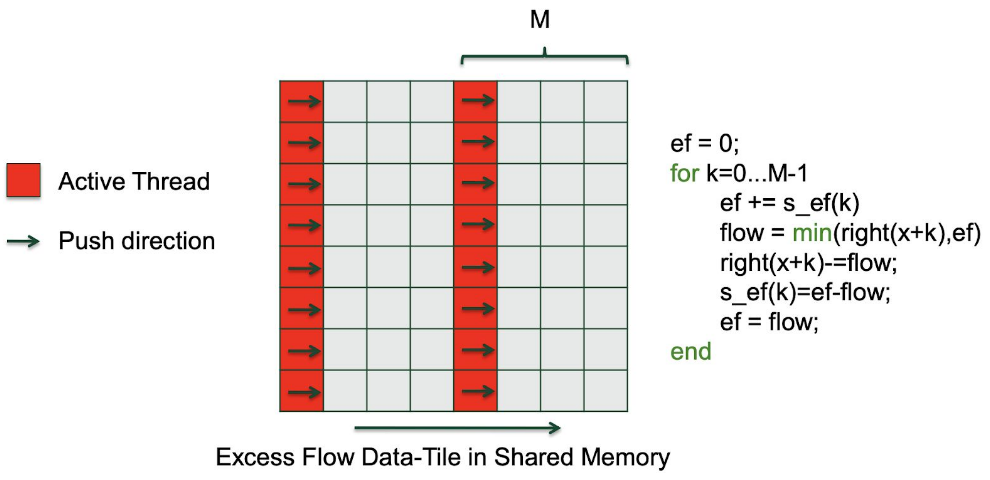
So, the multithreading of push-relabel on CPU also doesn't provide much improvement in the performance and is only beneficial when used for small images. Therefore, we move to GPU to further optimise the performance.

4. GPU version

a. Maxflow with Push-Relabel on GPU: How the algorithm works?

I. Solve the data dependency problem

With the help of GPU, we can be able to execute push and relabel operations in parallel. This benefits a lot when image size is astronomical and speedup with multithreading seems inconsequential. With each thread working on each pixel, GPU resources can be highly utilized. However, the update of data on nodes and weights in the residual graph comes with a lot of data dependencies. For example, when making a push from a certain node, it needs to check if it has access flow. But in the meanwhile, a different node may make a push to this node and update its excess flow value. Such read after Write hazard cannot be ignored. An atomic approach must be created to realize parallel push. Our solution, learnt from a NVIDIA post, pushes pixels in one direction at a time. If we push right, each thread will be responsible for updating the next few nodes on its right.



```
ef = 0;
for k=0...M-1
    ef += s_ef(k)
    flow = min(right(x+k),ef)
    right(x+k)-=flow;
    s_ef(k)=ef-flow;
    ef = flow;
end
```

Figure 9. Maxflow Optimisation with Push-relabel on GPU

II. Tile-based Push-Relabel

To take advantage of the full power of GPU, and also because of the limited number of threads in each block, our implementation with GPU uses two levels of parallelization: parallelization with threads inside a block and multiple tiles in an image. An image input is split into tiles, and tile size equals to block size. Between each iteration of the push-relabel process, the blocks need to sync up; because of the fact that nodes from one tile may push to nodes in another tile, each tile needs to fetch the excess flow pushed into itself before starting the next iteration.

For relabeling, we don't need to consider tile-based approach because the RAW hazard can be avoided by using double buffering on heights matrix. Each thread will update the height for each pixel node; it reads from global height matrix and write to a global buffer height matrix, which will be copied into global height matrix when all rebelling process are complete. The next iteration of push-relabel will therefore start with the correct heights for checking push availability.

III. Global Relabel

A separate relabeling operation is used for every six iterations of push-relabel. Because the previous relabel operation(which we call local relabel) only checks on a node's neighbors, it is a bad heuristic for long distance flow transportation. The flows may get pushed back and forth, which results in an infinite loop. A global relabel fixes this problem by running BFS from the sink, which all the flows are supposed to end up in. It relabels nodes that still have residual capacity to the sink 1, and relabel their unrelabled neighbors 2 in the next iteration if there is residual capacity connecting to the lower node. The relabelling keeps going when there is no more height changes. After global relabel process, the number of iterations till convergence will be significantly reduced.

IV. Convergence condition

The condition for a node to participate in push-relabel is that the node must be active; it must have excess flow and a height less than the number of nodes in the graph. We use a global array called ACTIVE_NODES to keep track of the active nodes, and ACTIVE_TILES to keep track of the active tiles. If any node inside a tile is active, the tile is active. When all the tiles are inactive, it means the graph converged and we have a final residual graph. Edges with their capacity reduced to zero are the minimum cuts, which segments image background and foreground. An alternative approach to decide if the graph reaches convergence is its total energy (explained on page 3) at the end of each iteration. The loop ends itself when an energy threshold is met. While this total energy approach can save the total number of iterations, we stick to "active nodes" method, because it works well in all cases so far.

Here is the main loop of the program.

```
Count=0
While has_not_converged:
    Push()
    Relabel()
    If count%6=0: Global_Relabel()
    count++
```

b. Memory

	Heights	Excess Flow	Weights	List of Active Nodes
Push (all directions)	Shared Memory	Shared Memory	Global Memory	Global Memory
Local Relabel	Global Memory	n/a	Global Memory	Global Memory
Global Relabel	Global Memory	n/a	n/a	n/a

Table 2. Memory usage chart. ***Note: all matrix sizes are image_size x image_size***

While we tried our best to reduce memory access time, shared memory is not applicable in too many places. It's used in all the push functions for reading and updating excess flow. After all, shared memory only saves time when it's used to store and write data to back and forth like a scratchpad.

Texture memory could be used for local relabel and global relabel, but then, it's lifecycle would only be one iteration, so we use global memory instead.

c. Numba

I. Runtime

The source code is written in Python, which is a scripting language and cannot be compiled by default. Therefore we used Numba library, which offers decorators for implementing CUDA kernels. The code structure is similar to CUDA on C, except programmer does not have to explicitly implement the transfer of data between host and device. During runtime of the program, first, Numba runtime will extract the kernel functions and precompile them into C code. When the kernel is launched from host in Python, corresponding GPU operations will be initiated by Numba. However, initial kernel launch brings a significant overhead (600ms), and when measuring the performance of our graph-cut algorithm, we want to count out this initial overhead when measuring GPU performance of the graph-cut program.

II. Why we use it

We chose to build the program in Python because of the algorithm's reliance on gaussian model. It'd be hard to implement a Gaussian Mixture Model and Predictor in C from scratch. The model calculates a pixel's probabilities to be part of image foreground and background, which we need in order to compute edge capacities.

d. Performance on GPU

Before measuring the GPU time on graph cut, I referred to the paper "A Survey of Graph-cut Methods" by Faliu Yi, Inkyu Moo, who claim a GPU results of 4ms on 600x450 image, which I find very unbelievable. Researchers at NVIDIA[4] gives a results of 92ms for the same image. Because we use numba, the initial overhead for setting up kernel itself takes at least 400ms.

Graph below shows the time it takes for different number of iterations for the same input image. Fortunately, run time does not increase too much for each iteration added, roughly 50ms. On average, it takes an 600x600 image 8 iterations to converge. Therefore, the whole graph cut takes 400ms if count out kernel launch time.

[4] Please see in "Reference" section

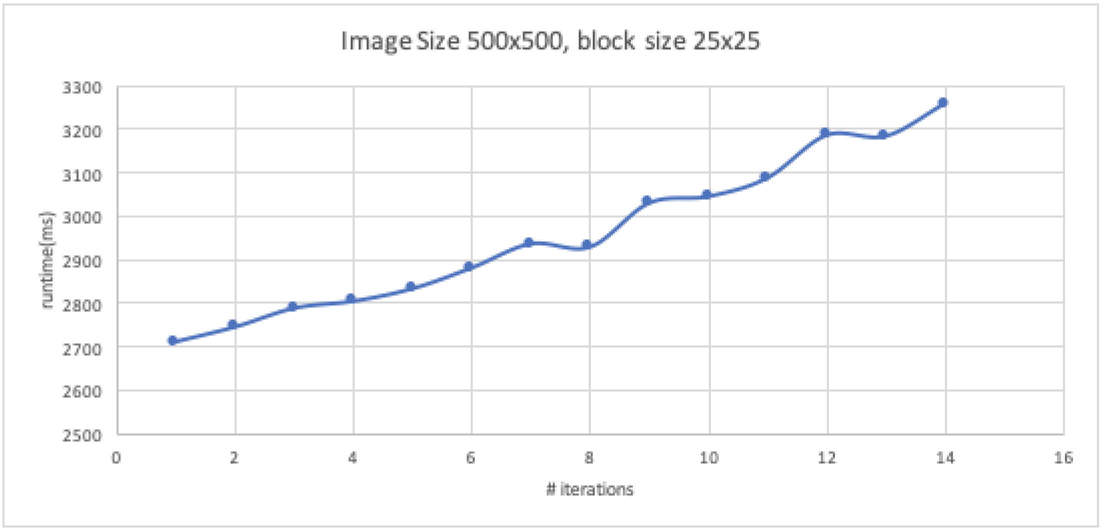


Figure 10. GPU runtime for different number of iterations

This graph shows the total time taken for running maxflow on different image sizes. Interestingly, run time decreased dramatically when image size exceeds 250. This proves that most of the time is spent on launching CUDA kernel with Numba.

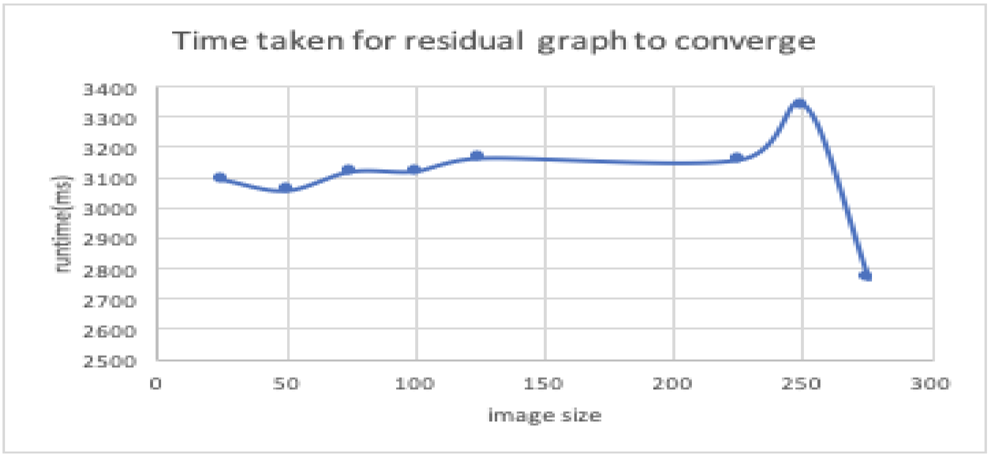


Figure 11. Run time vs image size

5. Segmentation Results

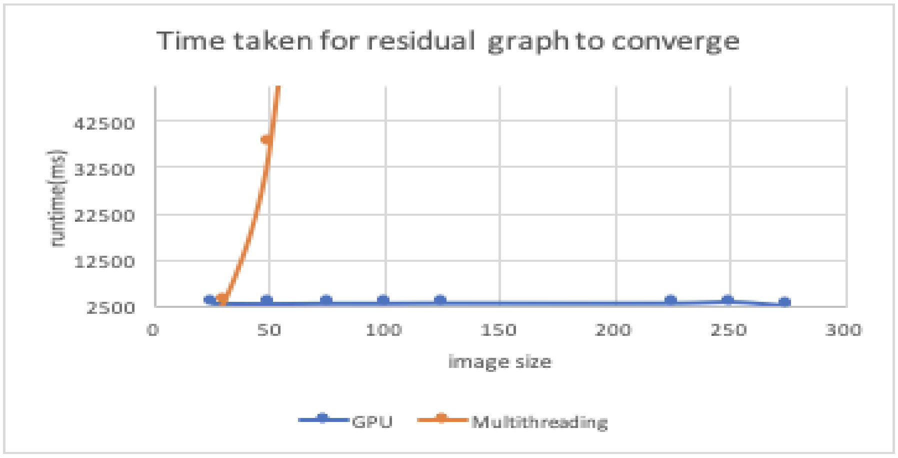


Figure 12. GPU runtime vs Multithreading

Different optimization method should end up with the same residual graph. However, only GPU version can generate the result in a reasonable amount of time. The white dots in the right image are predicted boundaries between foreground and background. We did not apply the Gaussian Mixture Model, each pixel's probability to be foreground or background is randomly generated.

As we can see, maxflow algorithm can still make rough prediction based on boundary property, which means the program works properly.

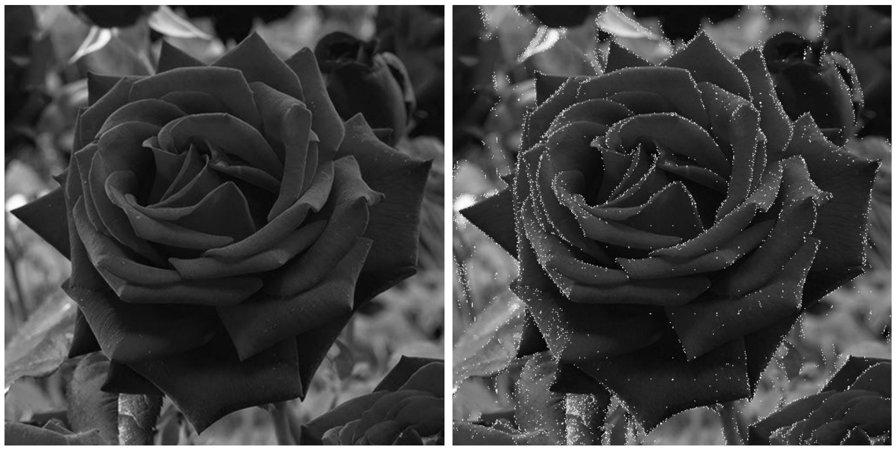


Figure 13. Segmentation result

6. Conclusions

Graph-cut with GPU turns out to be the far better method. In comparison, Push-Relabel on CPU has very limited performance. The advantage of Push-Relabel algorithm itself lies on “distributed” push, where there is no necessary specific sequence for the pushes. Taking advantage of such attribute with data and instruction parallelization is very beneficial. Also, we learnt how to create a variation of this algorithm to fully optimize it with available hardware.

Please see our source code in the project folder.

7. References

- Image Segmentation: A Survey of Graph-cut Methods -- Faliu Yi, Inkyu Moon
- https://www.nvidia.com/content/GTC/documents/1060_GTC09.pdf
- https://www.geeksforgeeks.org/minimum-cut-in-a-directed-graph/
- https://www.geeksforgeeks.org/push-relabel-algorithm-set-2-implementation/