Parallel Image Segmentation based on Quick Shift Method

Chen Wang*

University of Michigan, Ann Arbor

zoeumich@umich.edu

December 9, 2018

Abstract

I implement the quick shift method to segment images with different size on GPU. Memory access is the critical issue due to the size of image, variants of method involving global memory, texture catching, and shared memory are evaluated, yield to the conclusion that texture is the most suitable solution in this case. Compared with serial algorithm, the parallel one supported by texture fetching performs a 130 to 300 times speed up, promises a real time super pixel computation on modest size(several MB) images.

I. Introduction

Segmenting image into super pixels is one of the hot topic in image processing field today. Not only because it's one important preprocessing step which extracts essential information from images on a higher level instead of discrete pixels, but also it's naturally a challenging problem involves expensive computation and memory usage. Previous work on image segmentation pay less attention to speed since time consumption is not a critical problem in many image processing scenarios, however, with the development of artificial intelligence, real time interactive program plays a more and more important role.

In this article, I deliver a parallel quick shift method relying on Flux HPC Cluster, reach to a 130 to 300 times speed up compared to serial program, push the limit of real time super pixel segmentation, prove the feasibility of potential new application such as video understanding.

*Thanks to Prof. Stout, for his responsibility and work ethic.

II. Related Work

Quick Shift is a relative fast method to segments images into super pixels through mode seeking, while it's not the only one. Another popular method mean shift[1], on which quick shift based, delivers the idea of associating each data point to a mode of the underlying probability density function. This simple method possesses attractive advantages compared with tradition ways, that is, no need to predefine a chunk number, and segmentation result is highly possible to be arbitrary. However, mean shift is actually a gradient descent algorithm[6], the choice of step length and stopping criterion is a rather tricky aspect, resulting in not stable outcome on different images. Moreover, the common shortcoming of iterative algorithm is time complexity $O(In^2)$, where I is the number of iteration and n is the total number of pixels in input image. In contrast, quick shift[2] seeks the energy modes by connecting to nearest neighbor with exact higher energy levels, avoid the time consumed iterations.

III. Methods

i. Algorithm Introduction

Give N data points $x_1, x_2, ..., x_N \in \chi = R^d$, like other mode seeking clustering algorithms, quick shift conceptually starts with computing the isotropic Gaussian kernerlized *Parzen density estimate*[3]

$$P(x) = \frac{1}{2\pi\sigma^2 N} \sum_{i=1}^{N} e^{\frac{-\|x-x_i\|^2}{2\sigma^2}} \tag{1}$$

Then each data point x_i is linked into a mode of $P(x)$ with respect to some rules, all points that connects to the same mode form a pixel cluster. Quick shift method connects each point to the nearest neighbor in the feature space which has a higher $P(x)$, each connections is binded with a distance property, and the collection of all links and pixels forms a tree, with the pixel possessing highest density estimate as the root of the tree.

To obtain a segmentation from this tree, a distance threshold δ will be set, break all pixel pairs whose distance is larger than δ. Each smaller tree broken from the original large tree represents one super pixel.

In sake of this term project, we restrict the feature space to one suitable for image segmentation: a combined space composed of pixel coordinates (x, y) and RGB value (r, g, b). Then we will have a five dimensional feature space (r, g, b, x, y). In order to adjust the weights between spacial position and pixel values, we add a pre-scaled parameter λ to (x, y).

Due to the natural property of Gaussian-like kernel, data points that far away from center point x_i only have limited influence to $P(x_i)$, then we can set a window size parameter $r = 3\sigma$, to relieve the burden of computation, under the condition that almost no impact on the final result. For pixels near the boundary, we only consider their valid neighbors inside the image.

ii. Optimization Parameters

In this project, I set the value of trade-off parameter $\lambda \in [0.5, 1.5]$, the window size $r = 3\sigma$, *w.r.t.* $\sigma \in [2, 20]$, and the broken distance$\delta \in [10, 20]$. The pseudo-code of my program is as followed.

Algorithm 1: Parallel Quick Shift Based Image Segmentation Algorithm

Input : data collection χ
broken distance δ
window size r

Output: Collection of tree contains all pixels
with the property distance

```
function Quickshift (χ, δ, r);
for x_i ∈ χ do
    P(x_i) = 0
    for x_j ∈ pixels less than r away do
        P(x_i) += e^{-||x_i - x_j||^2 / (2×(r/3)^2)}
    end
end
for x_i ∈ χ do
    for x_j ∈ pixels less than δ away do
        if P(x_j) ≤ P(x_i) and ||x_i - x_j||^2 is smallest then
            distance(x_i) = ||x_i - x_j||^2
            parent(x_i) = x_j
        end
    end
end
```

IV. Results

i. Hardware

Because the *Parzen Density Estimation*[3] in quick shift algorithm is independent of data points in distant surroundings, and similar process has to be

performed on each pixel, meanwhile, the post-processing, breaking links farther than δ making it a good candidate for parallel computation.

I developed a program with CUDA 9.1 to handle all process after importing image into CPU memory. Roughly speaking, it contains steps as copying image data from host CPU memory to GPU device, compute *Parzen Density Estimate*[3] for each pixel simultaneously, find the nearest neighbor of each pixel with larger *Parzen Density Estimation*, and broken links distant than δ.

I compile the code under the architecture of SM_35 comm_35, and run the program on the NVidia K40 GPU of Flux HPC Cluster. Table 1 specifies detailed information about this platform[4]. Each allocation for GPU request comes with 2 compute cores and 8GB RAM. As for the detailed CPU information, the Flux HPC Cluster contains 4 kinds of Intel Xeon CPU with clock speed from 2.5GHz to 2.8GHz, users can request for a specific CPU model upon task scheduling and resource allocation via the command

```
#PBS -l feature=model_name
```

In this term project, I choose the Intel Xeon E5-2670 processor with clock speed 2.60 GHz to run the serial program.

Table 1: NVidia K40 Capacity (GK110B)

Peak single precision floating point perf.	4.29Tflops
Number and type of GPU	Kepler GK110B
Memory bandwidth(ECC Off)	288GB/sec
Memory size(GDDR5)	12GB
CUDA cores	2880
Compute capability	3.5

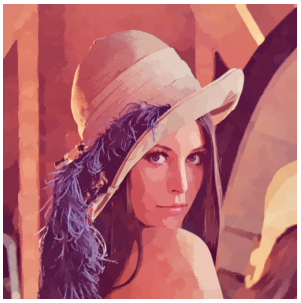
ii. Outcome

During the process, I set $\delta \in [10, 20]$, $\lambda \in [0.5, 1.5]$, $r \in [6, 60]$. In order to analyze the relationship between speed up and image size, I run the program to process 4 different size and shape images (378×478,494×494,922×1360,1018×1022). Due to limited spaces, here I display two groups of origin images, and segmented images under the condition of $r \in \{6, 30\}, \lambda = 0.5, \delta \in \{10, 20\}$, which performs similar segmented results compared with traditional mean shift method, and reach to hundreds of times speed up on GPU, compared with serial program on CPU.

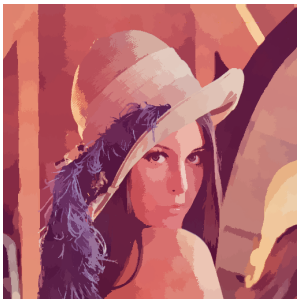
Table 2: Experiment Results

(a) Original: Lena (1018×1022)

(b) Lena: $r = 6, \delta = 10$



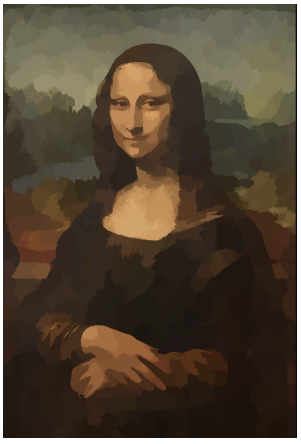
(c) Lena: $r = 30, \delta = 10$



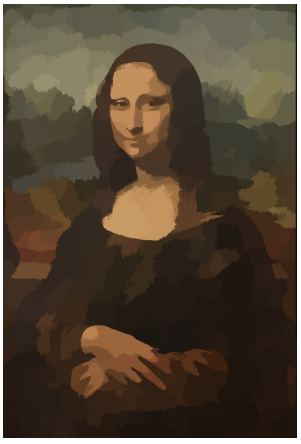
(d) Lena: $r = 30, \delta = 20$

(e) Original: Monalisa(922×1360)

(f) Monalisa: $r = 6, \delta = 10$



(g) Monalisa: $r = 30, \delta = 10$



(h) Monalisa: $r = 30, \delta = 20$

From the images generated by different r, δ, λ, it's easy to see that, increasing Gaussian kernel radius r smooths the underlying *Parzen Density Estimation*[3], leading to less super pixels in outcomes. Increasing the broken distance δ increases the average size of super pixels, at the same time result in more improper merge.[2]

V. Discussion

i. Memory Access

One crucial point worthy discussion here is the memory access method on device. The *Parzen Density Estimation* of each pixel is dependent on the surrounding $(r+1)^2$ neighbors, In order to reduce the influence of slicing a window of isotropic Gaussian Kernel to the final segmentation result, we intends to set a larger window radius. The global memory access is slow, require many circles to access for hundreds of times, even with a small radius like 20.

To solve this problem, one option is utilizing the shared memory on each streaming multiprocessor. For a modern GPU, the maximum size of shared memory for each streaming multiprocessor is about 48KB, which contains as most 3 single precision float data matrix no larger than 64×64. Suppose we luxuriously set up a series of single thread block, even given a mediate radius $r = 31$, shared memory will be exhausted. Besides, the total number of pixels in a modern image is usually between tens of thousands and millions, it not feasible to work out efficiently since a common GPU today only consists of thousands of cores. In conclusion, shared memory is filtered out of consideration.[5]

Another method is relying on texture catching to read image data. I map the image, and *Parzen Density Estimation* data into 3D and 2D textures respectively. The speed of fetching a texture from memory is absolutely slower than shared memory access, however it provides a possible solution to boost the data access, compared with global memory method.

ii. Time Result

The parallel program works well in terms of speed up. The following graphs intuitively shows the time consumption given different parameters such as image size, kernel size r, and broken distance δ. To figure out the relationship between time consumption and different parameters, I take the average time from all relevant experiments.

Figure 1: Time results $w.r.t$ r and δ

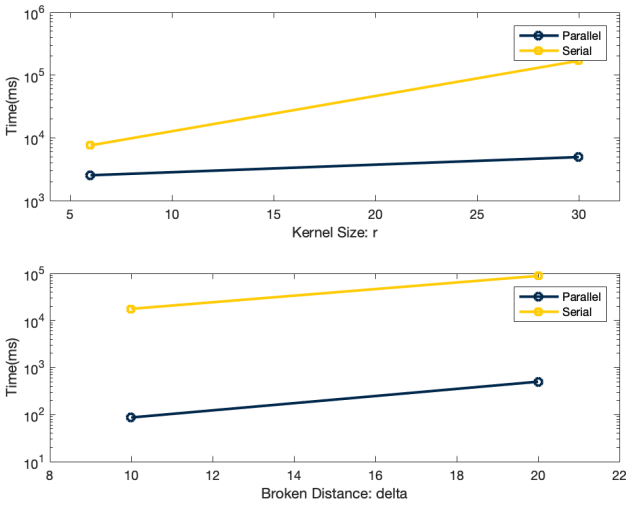
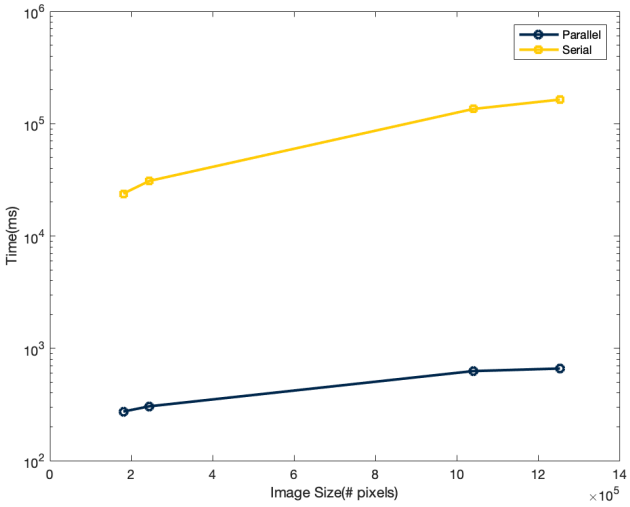


Figure 2: Time results $w.r.t$ image size



From **Fig. 1**, we can see that the time consumption is positively correlated with broken distance δ and kernel size r, because the larger r or δ we pick, the more surrounding pixels involved when computing *Parzen Density Estimation* or generating trees, respectively. However, the speed up is not stable, it tends to be positive correlated with input size, which we can tell from **Fig. 2**.

iii. Full Capacity Speed Up

Table 3: Speed Up vs Image Size (Number of Pixels)

Number of Pixels	r=6	r=30
180k	7.32	140.26
244k	12.43	181.53
1040k	29.32	268.73
1254k	38.34	277.05

Table. 3 displays the speed up of different size input images. Speed up changes a lot with kernel size r, actually it's in order of $O(r^2)$ if perfectly speed up. In practice, we tend to choose a larger kernel size to ensure more tidy segmentation. That's why I claim the speed up lies in 130 to 300.

Table 4: Max capacity for Kepler architecture GPU

Warp size	32
Max warps per SM	64
Max threads per SM	2048
Max threads per block	1024
Max blocks per SM	16
Max registers per block	64k
Max registers per thread	255

In order to thoroughly utilize the computation capability of GPU, I need super high resolution image. Since it's nonsense to spend time on seeking for input image, I decide to generate random matrix to work as input. Consequently, I skip the step of generating output image, which is actually a mess. The Kepler architecture GPU NVidia K40 I rely on in this project is made up of 15 streaming multiprocessors, and each of them have 192 streaming processors(cores). According to **Table. 4**, I set the block size as 32× the warp size 32, so the occupancy per block is 100%. If I have fully control over the Flux Cluster, I'd like to allocate myself one whole K40 GPU for this project, however, each user will be assigned only 2 cores per request. As a result, three 4000× 4000 matrix will absolutely make use of the capability in one flux request, and the timing results is listed here.

Table 5: Full Capacity Speed Up

r=6	r=30
130.89	294.73

Given a larger input size like 4200×4200, I receive the assert "Invalid Arguments" on kernel launch, this may result from exhausted register on GPU, because quick shift algorithm require a bunch of matrix to store intermediate data, like distance, parent, and energy for each pixel. That is to say, the above table lists max capacity speed up for quick shift algorithm.

VI. Conclusion

I implement the parallel quick shift method on GPU which yields to 130 to 300 times speed up over the serial implementation on CPU, making it possible to process image segmentation on super pixel level in real time. More powerful NVidia Titan V GPU is accessible in the Flux HPC Cluster either, however, instead of persuading extreme speed up through high tech devices, I prefer to evaluating a program in a median level device, in order to get the speed up brought by parallelizing algorithm. In the future, this program can be further enhanced and improved through sub-sampling input image, simplify *Parzen Density Estimation* from a mathematical perspective, etc.

References

[1] Li, Hua and Zhang, Ming-Xin and Zheng, Jing-Long(2009). Color image segmentation based on mean shift and multi-feature fusion *Journal of Computer Applications*, 29:2074–2076.

[2] Andrea Vedaldi and Stefano Soatto (2008). Quick shift and kernel method for mode seeking *Computer Vision -ECCV2008*, Chapter 52.

[3] R. Malladi, J. Sethian, and B. Vemuri(1995). Shape modeling with front propagation: A level set approach *IEEE Trans. Pattern Anal. Mach. Intell.*, 17:158–175.

[4] NVidia volta architecture white page. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf Accessed: Dec 8 2018

[5] Brian Fulkerson and Stefano Soatto (2010). Really Quick Shift: Image Segmentation on a GPU *Proceedings of the Workshop on Computer Vision Using GPUs*

[6] Stefan Klein, Josien P.W. Pluim, Marius Staring, Max A. Viergever(2009) Adaptive Stochastic Gradient Descent Optimisation for Image Registration *Int J Comput Vis*, 81: 227–239.