Multi-GPU accelerated real-time retinal image segmentation

by

Maxwell Miao

B.A.Sc, Simon Fraser University, 2014

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Applied Science

in the
School of Engineering Science
Faculty of Applied Sciences

© Maxwell Miao 2016

SIMON FRASER UNIVERSITY

Fall 2016

All rights reserved.
However, in accordance with the *Copyright Act of Canada*, this work may be reproduced, without authorization, under the conditions for Fair Dealing. Therefore, limited reproduction of this work for the purposes of private study, research, education, satire, parody, criticism, review and news reporting is likely to be in accordance with the law, particularly if cited appropriately.

Approval

Name: **Maxwell Miao**

Degree: **Master of Applied Science**

Title: ***Multi-GPU accelerated real-time retinal image segmentation***

Examining Committee: **Chair: Dr. Ash M. Paramesawaran, P. Eng**

Dr. Marinko V. Sarunic, P. Eng, MBA
Senior Supervisor
Professor, School of Engineering Science

Dr. Yifan Jian
Supervisor
Associate Research Professor

Dr. Mirza Faisal Beg , P. Eng
Internal Examiner
Professor, School of Engineering Science

Date Defended/Approved: November 10, 2016

Abstract

In recent years, Optical Coherence Tomography (OCT) has become one of the dominant imaging technologies for ophthalmic diagnostics and vision research. The fast and high-resolution cross-sectional data that OCT provides has brought a new possibility in the role of intra-operative imaging. However, existing commercial OCT systems lack the automated real-time functionality for providing immediate feedback of changes in anatomical configuration as the result of surgical actions. The predominant reason for lacking such functionality is because high complexity algorithms are hard to implement in real-time imaging due to their computationally expensive nature.

In this thesis, we will present a Graphics Processing Unit (GPU) accelerated retinal layer segmentation for real-time intra-operative imaging applications. Modern GPUs has emerged as a strong tool for mass computation in scientific researches. The computational power of the GPU outpaces Central Processing Unit (CPU) significantly when the processing task is parallelizable. Image segmentation is a computationally expensive algorithm and traditionally implemented in sequential instructions. An example of a parallelizable segmentation algorithm is Push-Relabel (PR) Graph-Cut(GC), which can be implemented using GPU. The GPU Retinal Segmentation (GRS) presented in this thesis is built upon such an algorithm. To ensure the run time of the GRS meets the real-time requirement for its application, multiple GPUs are used to accelerate the segmentation processing further in parallel. As a result of using GRS, we were able to achieve the visualization of the retinal thickness measurement and the enhancement of retinal vasculature networks in real-time.

Acknowledgements

Through the course of my graduate studies, my senior supervisor, Dr. Marinko Sarunic, has provided me the chance of studying the medical imaging application of heterogeneous computing with graphics processing units. I would like to express my deepest gratitude for his kindness and patience he spent in training me to become an independent engineer in preparation for my future career, as well as the caring and support.

I would also like to than my other supervisor Dr. Yifan Jian for being helpful and supportive for explaining the abstract concepts of GPU programming. It has been a privilege to learn from and work with Dr. Jian throughout the course of my graduate program.

In addition, I would like to thank Dr. Faisal Beg for being in my supervisory committee and providing his guidance in helping me advance in my master program.

Last but not the least, I would like to thank all my fellow friends in BORG for their support throughout the course of my entire degree. It was truly my honor to work with these talented engineers.

Table of Content

Approval ii
Abstract iii
Acknowledgements iv
Table of Content v
List of Tables vii
List of Figures viii
List of Acronyms x

Chapter 1. Introduction 1
1.1. Optical Coherence Tomography 2
1.2. Research Motivations 6

Chapter 2. Heterogeneous computing 9
2.1. Central Processing Unit basics 9
2.2. Graphics Processing Unit Basics 11
2.3. Differences between CPU and GPU 13
2.4. Needs for Heterogeneous Computing 14
2.5. CUDA platform 15
2.6. Summary 20

Chapter 3. Segmentation theory 21
3.1. Graph-cut background 21
3.2. Push-relabel Algorithm for Graph Cuts 22
3.3. Operation of the algorithm 25
3.4. Parallel implementation: 27
3.4.1. Parallel push 27
3.4.2. Parallel relabel 28
3.5. Connected Component and Labeling 29
3.6. Summary 34

Chapter 4. Retinal Segmentation Pipeline 35
4.1. Asynchronous parallel computing 35
4.2. Batch processing and real-time requirement 37
4.3. Segmentation Initialization 38
4.4. Binary image segmentation with GPU 42
4.5. Layer extractions through CCL 43
4.6. Summary 45

Chapter 5. GPU-Based Retinal Layer Segmentation Result and Discussion 46
5.1. Environmental setup 46

5.2. Qualitative result of GPU retinal layer segmentation 48
5.2.1. Thickness measurement 48
5.2.2. Segmented SV angiography 53
5.3. Limitations 57
5.4. Speed of the GPU retinal layer segmentation 57
5.5. Performance due to contrast to noise ratio 65
5.6. Final evaluation of GRS against the CPU implementation 70
5.7. Summary 72

Chapter 6. Conclusion and Future Work 73
6.1. Future Work 73
6.1.1. Software and Algorithms 74
6.1.2. Hardware environment 74

References 77

List of Tables

Table 5-1 Hardware specification of testing bench 47
Table 5-2 Profiling result (SV05) 60
Table 5-3 Profiling result (H435) 62
Table 5-4 Profilling summary 64
Table 5-5 Representative time comparison between GPU and CPU 71
Table 6-1 Testing bench vs ideal system 75

List of Figures

Figure 1-1 Basic topologies of: (A) TD-OCT, (B) SS-OCT, (C) SD-OCT 3
Figure 1-2 Raster scanning pattern 5
Figure 1-3 Reconstruction process of OCT image 5
Figure 1-4 Target retinal layers of a representative cross-sectional OCT image centered at the fovea 6
Figure 2-1 Intel "Skylake" processor die layout 10
Figure 2-2 Die layout for NVIDIA Kepler™ architecture 12
Figure 2-3 OCTViewer thread distribution 14
Figure 2-4 Code compilation flow [20] 16
Figure 2-5 An example of a multithreaded program partitioned into blocks of threads that execute independently on each GPU [24] 17
Figure 2-6 CUDA program execution flow [24] 18
Figure 2-7 Sample code for kernel launch 19
Figure 3-1 Representative segmented binary image by PR GC 30
Figure 3-2 Generated labels after the first pass 31
Figure 3-3 Labeling result after second pass 32
Figure 3-4 Final result of CCL 33
Figure 4-1 processing work flow for segmentation pipeline for dual GPU computation system 36
Figure 4-2 Real-time deadline 38
Figure 4-3 Before and after frame averaging for B-scan intensity image 39
Figure 4-4 Bilateral filtered B-scan intensity image 40
Figure 4-5 Gradient image for graph construction 41
Figure 4-6 Binary image segmented by NPPI GC 42
Figure 4-7 Before and after applying CCL on the binary image 43
Figure 4-8 Line overlay for GPU retinal segmentation result 44
Figure 5-1 Thickness map of the representative healthy volume (SV05) 49
Figure 5-2 Thickness map for a pathological volume with AMD and NPDR (H468) 49
Figure 5-3 Thickness map for a pathological volume with central serous chorioretinopathy (H435) 50
Figure 5-5-4 Thickness map for a pathological volume with lamellar hole (H505) 51
Figure 5-5 Segmentation result corrupted by bright artifacts 52

Figure 5-6 Extreme case ... 52
Figure 5-7 Visualization of retinal vasculature network by naive region selection (SV05) ... 53
Figure 5-8 Visualization of vasculature network for a representative healthy volume. (SV05) ... 54
Figure 5-9 Visualization of vasculature network for a pathological volume with lamellar hole (H505) ... 55
Figure 5-10 Visualization of vasculature network for a pathological volume with central serous chorioretinopathy (H435) ... 56
Figure 5-11 Visualization of vasculature network for a pathological volume with AMD and NPDR (H468) ... 56
Figure 5-12 Representative per-batch timeline for entire processing pipeline captured by NVIDIA Visual Profiler ... 58
Figure 5-13 GC vs Segmentation pipeline vs SSVA (SV05) ... 59
Figure 5-14 CCL vs Segmentation pipeline vs SSVA (SV05) ... 60
Figure 5-15 CCL vs Segmentation pipeline vs SSVA (H435) ... 62
Figure 5-16 Images with different histogram scaling factors: (A) min: 7.5, max: 10.5 (B) min: 10, max: 16.5 (C) min: 10.5, max: 11 (D) min: 10.5, max: 12.5 ... 66
Figure 5-17 Running time comparison for segmentation pipeline between four different histogram range scaling factor settings ... 67
Figure 5-18 Running time comparison for segmentation pipeline between 6 settings of different histogram scaling range ... 68
Figure 5-19 Running time comparison for segmentation pipeline between 6 settings of different scaling factor levels with the same range ... 69
Figure 5-20 False segmentation result in the histogram scaling setting (F: 11-13) ... 69
Figure 5-21 Result of Chiu's implementation with our input A) GRS optimal histogram B) Histogram adjusted for Chiu's algorithm ... 71

List of Acronyms

ERM	Epi-Retinal Membrane
ILM	Inner Limiting Membrane
OCT	Optical Coherence Tomography
FD-OCT	Fourier Domain Optical Coherence Tomography
TD-OCT	Time Domain Optical Coherence Tomography
SS-OCT	Swept Source Optical Coherence Tomography
SD-OCT	Spectral Domain Optical Coherence Tomography
AMD	Age-related Macular Degeneration
DR	Diabetic Retinopathy
GPU	Graphics Processing Unit
CPU	Central Processing Unit
SDK	Software Development Kit
ALU	Arithmetic Logic Unit
DRAM	Dynamic Random Access Memory
FLOPS	Floating-point Operations Per Seconds
PCIE	Peripheral Component Interconnect Express
SM	Streaming Multiprocessors
HPC	High-Performance Computing
API	Application Programming Interface
CUDA	Compute Unified Device Architecture
SPMD	Single Program Multiple Data
VRAM	Video Dynamic Random Access Memory
GC	Graph Cut
PR	Push-Relabel
CCL	Connected Component and Labeling
BM	Bruch’s Membrane
SV	Speckle Variance
SS-svOCT	Swept Source speckle variance Optical Coherence Tomography
ECC	Eye Care Center
VGH	Vancouver General Hospital
NPPI	NVIDIA Performance Primitives Image

RPE	Retinal Pigment Epithelium
GRS	GPU retinal Segmentation
NPDR	Non-Proliferative Diabetic Retinopathy
INL	Inner Nuclear Layer
NFL	Nerve Fibre Layer
ONL	Outer Nuclear Layer
SSVA	Segmented Speckle Variance Angiography
SNR	Signal to Noise Ratio
CNR	Contrast to Noise Ratio

Chapter 1.

Introduction

The global aging of the human population in recent years has increased the prevalence of age-related diseases to the modern society, with vision loss being one of them. According to CNIB online report, one in 11 Canadians aged 65 or older are living with vision loss [**1**]. In preventing vision loss during clinical ophthalmic treatment, surgical procedures may be required. Ophthalmic surgical procedures such as peeling Epi-Retinal Membrane (ERM) rely on staining the ERM and Inner Limiting Membrane (ILM) with the use of an intra-operative microscope. However, the contrast generated by the dye stain is low, and the residual dye could cause post-operative complications. Moreover, the micro-anatomical changes to the retina caused by surgical procedure during the macular repair could potentially negatively affect the surgical outcome [**2**]. Thus, using an alternative imaging modality that capable of providing high-resolution images seamlessly could provide significant assistance to evaluating anatomical changes during surgery and greatly increase the success rate

In recent years, Fourier Domain Optical Coherence Tomography (FD-OCT) has emerged as a crucial diagnostic tool for clinical ophthalmic imaging. The structural information provided by high-resolution cross-sectional images is indispensable for detecting the presence of macular edema and retinal fluid, which are characteristics of dominant diseases leading to blindness such as Age-related Macular Degeneration (AMD) and Diabetic Retinopathy (DR) [**3**]. According to the JP Ehlers [**4**], integrating FD-OCT to existing surgical procedures would potentially have immediate feedback on completion of surgical objectives or new understanding of the anatomic configuration of the tissues. Namely, the real-time visualization of the microanatomic changes at the ILM layer during surgeries such as ERM peel. However, modern commercial retinal OCT lacks the support of automated retinal layer segmentation [**5**]. The existing methods for

retinal layer segmentation are often conducted in post-processing due to the high complexity of algorithms involved during the process. In this thesis, we investigate the performance of the retinal segmentation of FD-OCT data using a parallel processing approach by Graphics Processing Units (GPU), as well as the possibility for quantitative analysis of intra-operative OCT.

1.1. Optical Coherence Tomography

Optical Coherence Tomography (OCT) is a non-invasive imaging modality first introduced in 1991 by Huang *et al* [**6**]. OCT is often described as an optical analogue to ultrasound imaging due to their similarities in imaging principles. Both of them are non-invasive imaging modalities that use analogous terminology (A-scans, B-scans). The difference is that instead of sound, OCT is performed by measuring low coherence light. The basics of OCT will be covered in the following sections.

OCT uses the principle of low coherence interferometry to generate the structural information of the sample being imaged. The core configuration of an OCT system is based on a Michelson interferometer [**6**], which uses fibre couplers as optical waveguides. Briefly, in a simple 2 x 2 setup, light from a low coherence laser source is divided by the fibre coupler into two beam paths, with one traveling through reference arm and the other through the sample arm. The light scattered back from both pathways will combine and produce interferometric fringes that correspond to the optical path length mismatch between the two paths, which represents the signal of the sample tissue. There are two types of OCT: Time Domain OCT (TD-OCT) and Fourier Domain OCT (FD-OCT). Figure 1-1 demonstrates the setup for each type of OCT.

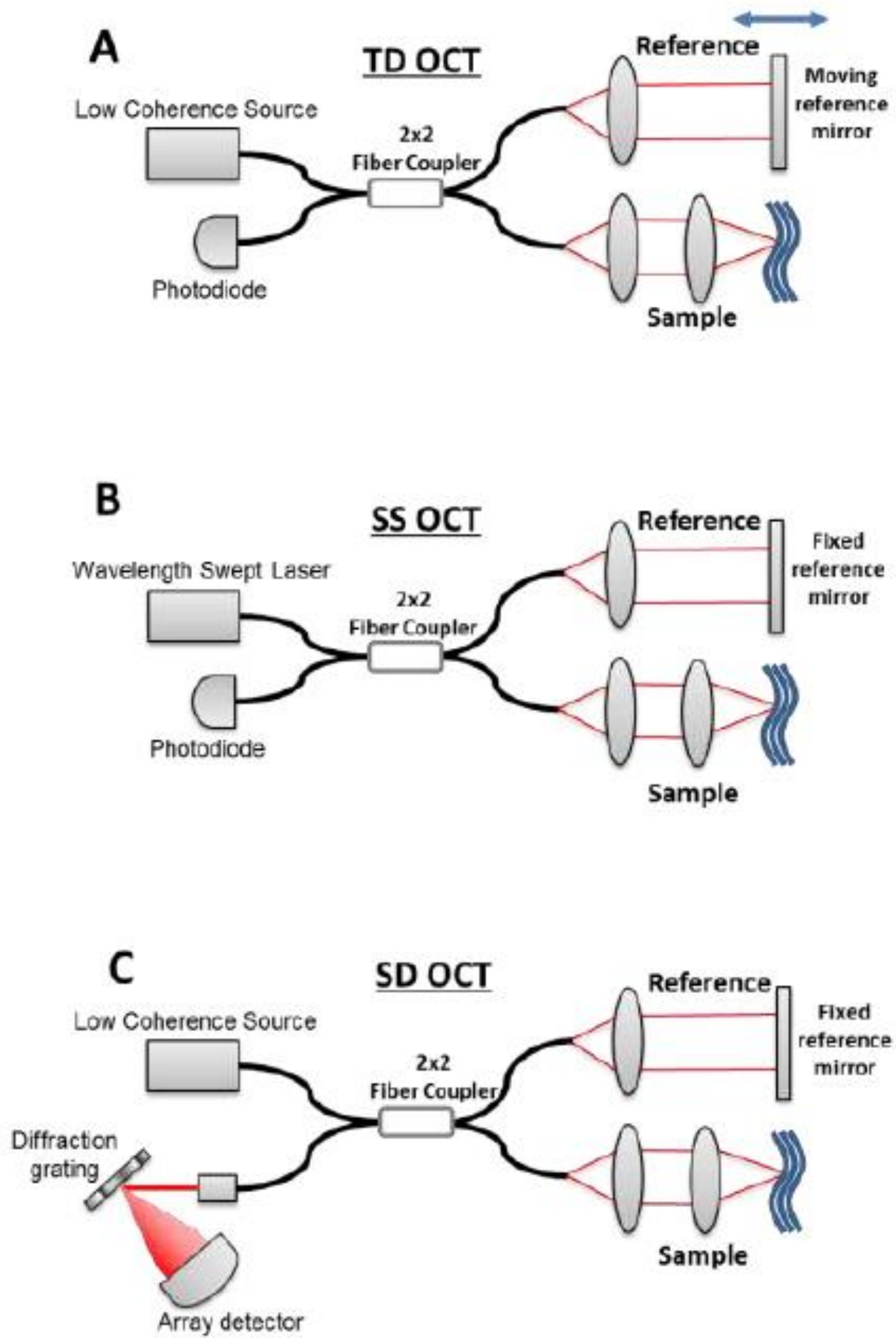


Figure 1-1 Basic topologies of: (A) TD-OCT, (B) SS-OCT, (C) SD-OCT

The first type of OCT developed was TD-OCT. The axial information is acquired by mechanically moving the reference arm and accumulating a longitudinal scan in time, corresponding to the depth direction of the sample. The data throughput of TD-OCT was low due to the moving mechanism in the reference arm. Later in 2003, the inception of FD-OCT revolutionized the ophthalmic imaging by providing high data throughput in real-time with high-resolution volumetric view of the retina for clinicians to identify the structural hallmarks of ophthalmic pathologies [**7**]. There are two sub-types for FD-OCT: Spectral Domain OCT (SD-OCT) and Swept Source OCT (SS-OCT), as presented in Figure 1-1 (B) and (C).

SD-OCT uses a broadband light source that is split into the reference arm and sample arm through fibre coupler, as with TD-OCT. The main difference is that the reference arm is kept stationary, and that the detector is replaced with a spectrometer, acquiring the interferogram as a function of wavelength. Then, performing a Fourier-Transform (FT) on the interferogram obtained by the spectrometer will generate a spatial representation of the sample tissue. Since the reference arm is stable, light returning from all depths in the sample are interrogated simultaneously, resulting in a higher overall sensitivity. This permits a significant increase in data throughput, and rapid A-scan rates relative to TD-OCT. However, the dispersive elements in the spectrometer detector do not distribute the light evenly spaced frequency. Therefore, the signal has to be resampled before processing, which could result in losses in signal quality. This is a computationally intensive procedure that has been demonstrated to be efficiently performed on GPU [**8**].

SS-OCT, which uses a narrowband light source that sweeps across a broadband spectrum. The wavelength of the narrowband light source is encoded as a function of time, and the interferometric signal is detected by a single pixel photodiode instead of a spectrometer. The result of the system remains the same as in a SD-OCT system. Performing the FT on the acquired spectrum will generate the corresponding depth information of the sample in the axial direction at a single location of the retina.

Irrespective of TD or FD-OCT, the acquisition of an A-scan represents the depth information at a single point on the sample. A volumetric image is commoly acquired by

scanning the laser beam across the sample in a raster pattern as presented in Figure 1-2.

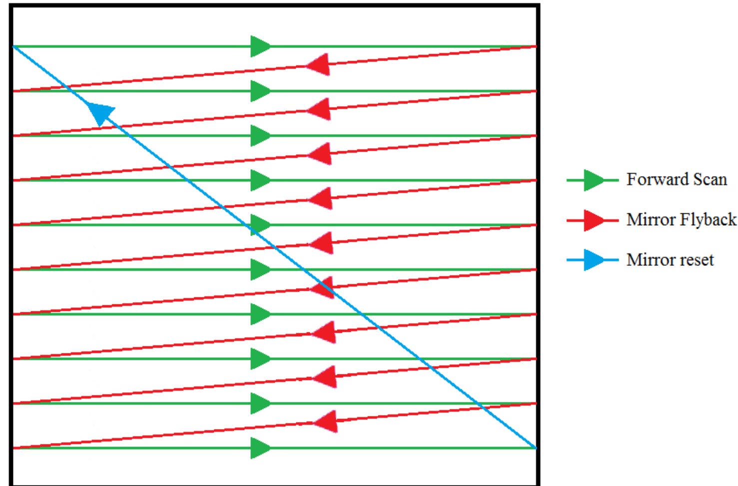


Figure 1-2 Raster scanning pattern

The raster scanning pattern is performed repeatedly until an entire volume of targeted region is acquired, then reset back to the initial location and the cycle repeats. A series of laterally adjacent A-scans along the forward scanning direction are combined into a cross-sectional view of the sample called a B-scan. Similarly, combining a sequence of laterally spaced B-scans yields a reconstruction of the sample volume. And finally, the *en-face* view of the retina is a 2D projection by summing up all intensity values along the axial directions. Figure 1-3 illustrates the volume reconstruction of the retina using OCT.

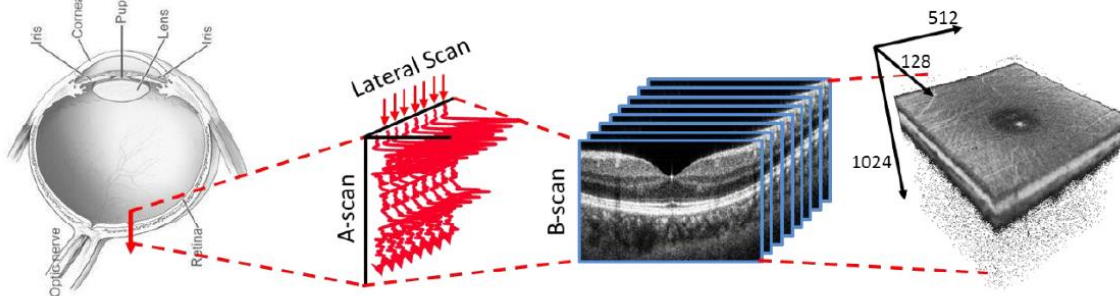


Figure 1-3 Reconstruction process of OCT image

1.2. Research Motivations

With the introduction of OCT in clinical ophthalmic imaging and the quantitative analysis of the retinal volume, clinicians have been able to detect the two major blinding diseases, namely AMD and DR, in the early stages and also track pathological changes as well as response to treatment. The progression of degenerative retinal diseases is monitored by measuring the thickness changes between retinal layers. Thus, extraction of the retinal layers through image segmentation is one of the first steps to the analysis of OCT data. A cross-sectional view of the OCT macular B-scan with marked layers is shown in Figure 1-4

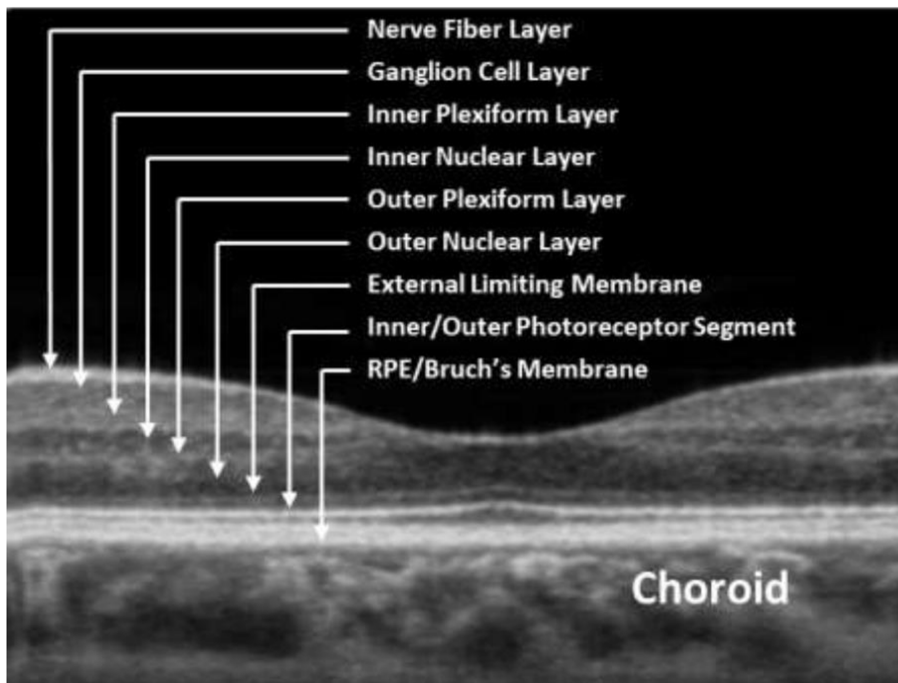


Figure 1-4 Target retinal layers of a representative cross-sectional OCT image centered at the fovea

In addition to the thickness measurement, the retinal image segmentation result can also be used to enhance the view of retinal vasculature network. Visualization of the retinal blood vessels is achieved by processing the flow contrast images using Speckle Variance OCT (svOCT) techniques [**9**]. The detail of svOCT is beyond the scope of this thesis. The basic idea for generating flow contrast images is by calculating the speckle variance from a set of B-scans acquired at the same location. This speckle variance

computation emphasizes particles in the retina that are in motion by generating contrast to locations where the speckle patterns change in the image. Thus, the blood vessels are pronounced stronger from the structural tissues in the retina.

Currently, the quantitative analysis is conducted in post-processing. Due to the high computational complexity and the large amount of data to process, the result of the retinal image segmentation cannot be displayed intra-operatively in the traditional method with existing technology. A method that permits quantitative real-time observation of the dynamic changes that occur in the retina during surgery will be helpful to reduce the risk of the real-time decision-making.

For intra-operative imaging, the reliability of the segmentation is one of the first things to consider since the result could have direct impact on the decision-making during surgery. In the past decade, the maximum-flow/minimum-cut (also known as Graph-Cut) segmentation method introduced by Boykov has become increasingly useful in the realm of image segmentation due to its robustness [**10**]. In addition to concerns in the quality of the segmentation, intra-operative image segmentation would also need to 'real-time' fast. Thus, computational complexity has to be carefully considered. The Graph-Cut (GC) has two subset algorithms: Augmenting-Path (AR) and Push-Relabeling (PR). Although PR does not have the best time complexity, in real world practice however, it has been proven to be the fastest running GC algorithms [**11**]. Moreover, PR is inherently parallelizable since the algorithm only considers local dependency during the push operation. If the parallelization of the PR is well applied, the overall processing time of segmentation can be reduced accordingly. With the advancement in microprocessors technology in recent years, the Graphics Processing Unit (GPU) has emerged as a powerful tool for massively parallel processing. The NVIDIA GPU Software Development Kit (SDK) has a built-in image library for PR GC segmentation [**12**].

The goal of this thesis is to verify the possibility of conducting retinal image segmentation in real-time for 200kHz laser source system by leveraging the computational power of GPU, and the qualitative result of its applications in measuring retinal thickness and enhancing the visualization of vasculature network. In this thesis, Chapter 2 provides the basics of parallel programming environment. Chapter 3

introduces the PR GC algorithm and its parallel adaptation. Chapter 4 describes the processing steps for achieving real-time segmentation. Chapter 5 will cover both quantitative benchmarking of the GPU implementation of real-time retinal segmentation and the qualitative analysis of the results of the thickness measurement and visualization of retinal vasculature network for the macular region. Lastly, chapter 6 will cover the future work and upcoming changes in parallel programming environment.

Chapter 2.

Heterogeneous computing

Medical image processing procedures are often very computationally demanding due to the large scale of medical datasets to process. Traditionally, Central Processing Units (CPUs) are the dominant choice for the scientific researchers. However, with the increase in the size of medical datasets, the serial computational nature of the CPU can no longer satisfy the ever increasing demand for real-time visualization. In recent years, the advancement of Graphics Processor Units (GPUs) architecture has enabled its capability for massively parallel computation in a wide range of medical imaging applications, including OCT imaging and retinal segmentation. Since medical imaging applications involve both algorithmic and parallelizable image processing, a heterogeneous solution that combines both CPU and GPU is required. This chapter will cover the fundamentals of both CPUs and GPUs, as well as the computational platform that employs heterogeneous computations.

2.1. Central Processing Unit basics

A CPU is essentially the brain of a computer. It is designed for wide range general purpose applications, ranging from the simplest logical operations to sophisticated algorithmic calculations. The principal components of CPU include the Arithmetic Logic Unit (ALU) that performs arithmetic and logic operations, the processor registers that supply operands to the ALU and store the results of ALU operations, and the control unit that fetches data and instructions from main memory to the processor registers [**13**]. Modern design of a CPU also includes system components on a single integrated circuit. The specific components vary depending on what purpose the CPU is designed for, but

generally speaking these system components include: a shared cache memory between cores, memory controller, peripheral interfaces, and main processing cores [**14**]. Figure 2-1 displays the general layout of a die map of the 6th generation intel Skylake™ processor.

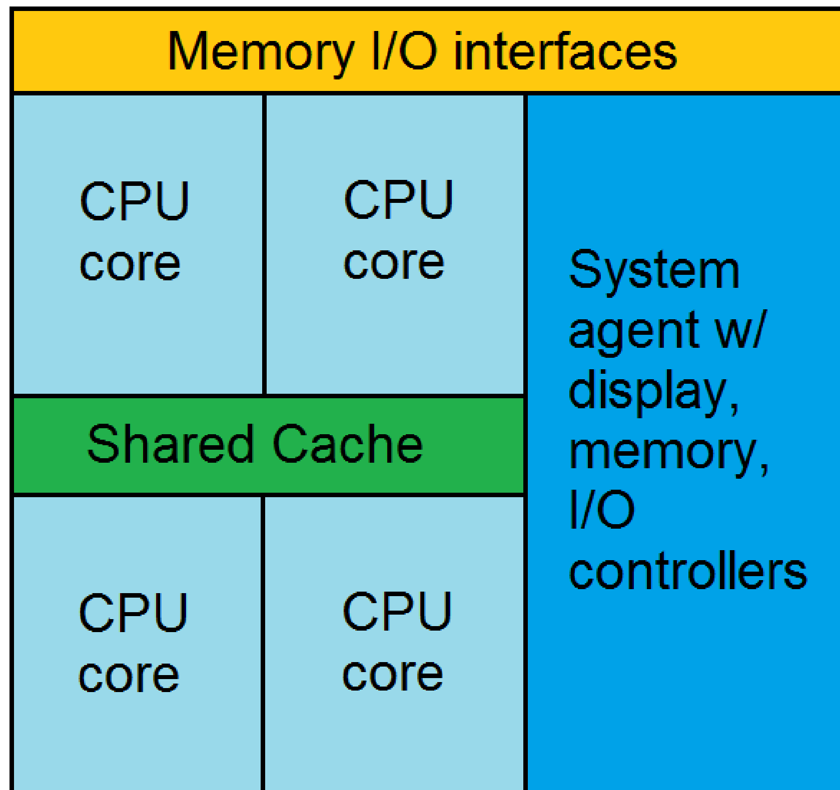


Figure 2-1 Intel "Skylake" processor die layout

CPU caches are special types of memories that speed up the instruction loading process. Typically there are 3 layers of caches in current CPU lineups [**14**]. L1 cache is core dedicated and can pre-store the information that CPU is most likely to use before the next instruction. It eliminates the data accessing time from CPU register to system memory (Dynamic Random Access Memory, DRAM). Every instruction and its data need to be available at the time they are requested so that the threads can be operated at full speed. If the information for the next instruction is not available in L1 at the time of request, the CPU will then look for the correct information from each level of cache, which results in many wasted cycles and a stalling in performance. Thus, CPUs are required to equip large local caches to keep themselves from stalling [**15**].

A CPU core has a much higher clock rate compared to the GPU counterpart with heavy optimization towards serial instructions. This was due to the fact that traditionally computer algorithms were implemented as a serial stream of instructions. Nowadays a mainstream CPU can achieve a clock rate of more than 4.0 GHz per core. The processing power of a processor is measured in terms of Floating-point Operations Per Seconds (FLOPS). FLOPS can be calculated using the equation:

$$FLOPS = sockets \times \frac{cores}{sockets} \times clock \times \frac{FLOPs}{cycle} \qquad \text{Eq. 2.1}$$

Specifically for a state-of-the-art Intel Skylake i7 processor, the performance per clock is 32 FLOPS for single precision operations [**14**], which roughly translates to 500 GFLOPS maximum theoretical performance. According to the Linpack™ benchmarking tool, the processing power for arithmetic operations is roughly 240 GigaFLOPS [**16**]. Please note that data throughput is not measured in FLOPS. FLOPS measurement merely offers a measurement for maximum processing capability when the computational environment is ideal. A direct comparison between GPUs and CPUs will be presented in the next section.

2.2. Graphics Processing Unit Basics

A Graphics Processing Unit (GPU) is a specialized device that manipulates memory for an image buffer to accelerate the rendering process of the image output, traditionally used in video games where the application heavily focuses on rendering geometric objects for the gameplay. The geometric objects are displayed on the monitor as polygons that consist of pixels output by GPU. Each pixel is essentially a single entry of the output data, and the GPU uses a set of processors to compute such pixel output in parallel. The computational nature of parallelism made GPU suitable for large-scale data computation. As a result, nowadays the GPUs are designed to be a general purpose computational device to accelerate computational workloads in areas such as financial modeling, cutting-edge scientific research or even natural resource exploration [**17**].

Just like CPUs, GPUs also have ALU, registers and memory control units. However, these three components are built within a set of processors called Streaming Multiprocessors (SM) or Streaming Multiprocessors eXtreme (SMX) for recent generations. The major component of a GPU consists of SMs, global memory and other system agents. The global memory (Video-RAM) is a memory pool dedicated for GPU and separated from CPU space. This is because data access through the Peripheral Component Interconnect Express (PCIE) creates huge latency in running time. Hence it is computationally economical to have local memories for fast accessing, then transfer the result back to DRAM through PCIE once the computations are complete. In addition to the L1 cache within each individual SM, for recent generations of GPUs, there is also a L2 cache memory block shared by all SMs for faster memory access. Small L2 cache memories are provided to help control the bandwidth requirements of applications so that multiple threads that access the same memory data do not need to all go to the DRAM. Figure 2-2 displays the chip layout of a Kepler GPU GK110, of which the GPU used for this thesis.

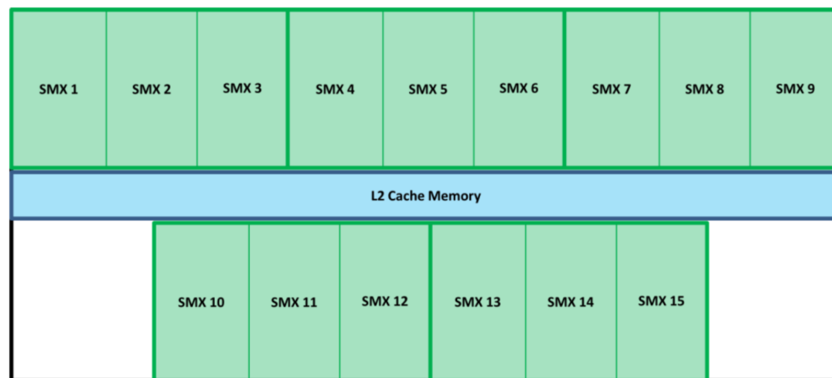


Figure 2-2 Die layout for NVIDIA Kepler™ architecture

Each SM is capable of supporting thousands of co-resident concurrent hardware threads, up to 2048 on modern architectures. All thread creation, scheduling and synchronization are performed entirely on hardware by the SM. To efficiently manage such large number of threads, the SM employs a unique architecture called Single Instruction Multiple Thread (SIMT), meaning that all stream processors (GPU cores) execute the same instructions simultaneously but with different data. SM's multithreaded instruction unit is

divided into warps with 32 threads for each warp [**18**]. A warp is the smallest entity for arithmetic operations. All instructions are issued in order and there is no branch prediction and no speculative execution.

2.3. Differences between CPU and GPU

As compared to their CPU counterparts, GPU cores are designed such that more transistors are devoted to data processing rather than data caching and flowing control. Since each thread executes unified operations, the need for a sophisticated flow control is negligible. Threads on a CPU are generally heavy-weight entities. The operating system must constantly be swapping threads on and off from the CPU execution channel to provide multithreading capability. In comparison, threads on GPUs are much more light-weight. Since separate registers are allocated to all active threads, no swapping of registers is needed when switching among GPU threads. Resources stay allocated to each thread until the thread completes its execution. However, allowing massively concurrent data throughput increases the latency for memory transfer significantly, as high data throughput and short latency are fundamentally in conflict [**19**]. Particularly, PCIE has a high data bandwidth up to 16GigaBytes per second (GB/s) for PCIE 3.0, but with latency up to milliseconds [**20**]. In comparison, the data accessing latency for the latest CPU DRAM DDR4 is at the level of sub-nanoseconds [**21**]. In short, CPU cores are designed to minimize latency for one or two threads at a time whereas the GPU cores are designed to handle large number of concurrent lightweight threads and sacrifice latency in order to maximize the data throughput [**22**].

With the heavy optimization towards massively concurrent throughput, the maximum computational performance of modern high-end GPUs can reach over 12 Tera Floating-point Operations Per Second (TFLOPS) [**23**]. For the NVIDIA Quadro™ K6000 GPU used in this research, the maximum computational performance is 5.2 TFLOPS, which is faster than the maximum performance of a high-end CPU by magnitudes [**23**]. In medical imaging where the vast amount of data throughput is parallelizable, the GPU is a clear winner in terms of processing powers.

2.4. Needs for Heterogeneous Computing

The GPUs outpace CPUs in terms of raw processing power thanks to the light-weight threads and large data throughput. However, GPUs operates poorly when it comes to algorithmic operations due to the very same reasons of light-weight threads and high latency induced by its large data throughput. As medical imaging programs do not solely process images, but also run complex multi-threaded tasks that controls the imaging system, a heterogeneous combination of CPU and GPU is required for optimal processing rates.

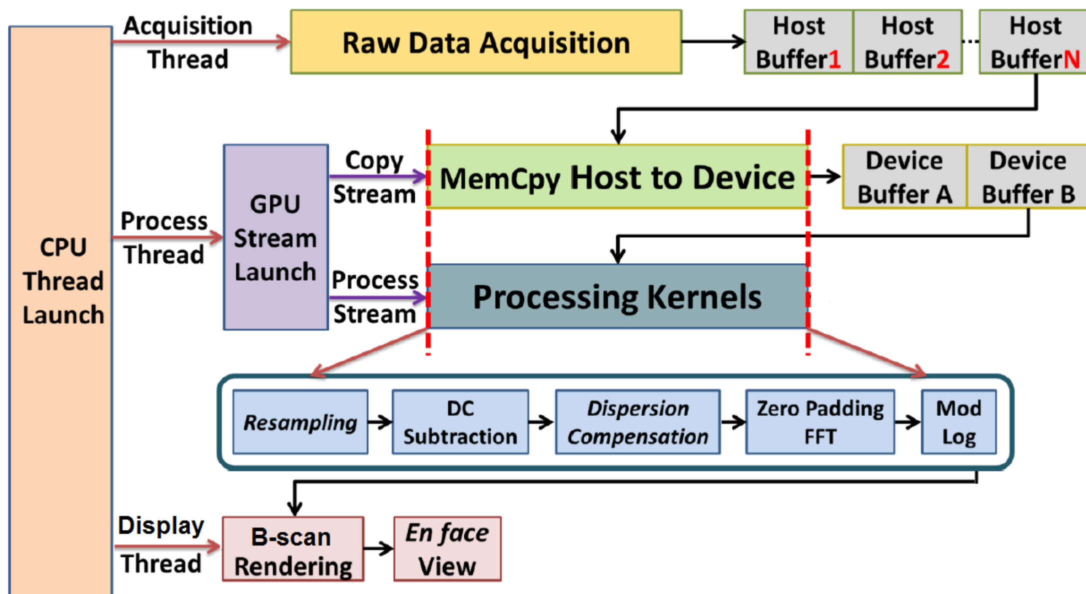


Figure 2-3 OCTViewer thread distribution

Particularly for Optical Coherence Tomography applications, the imaging software is responsible for at least three independent complex tasks: 1) laser scanning control; 2) camera synchronization and data acquisition; 3) signal and image processing. The thread number for the first two tasks is small but heavy-weight algorithms that require low operation latency, which cannot be provided by GPUs. Thus, for optimal High-Performance Computing (HPC), a heterogeneous solution is required for executing the sequential parts on the CPU and numerically intensive parts on the GPU.

2.5. CUDA platform

In November 2006, NVIDIA introduced Compute Unified Device Architecture (CUDA), a general purpose scalable parallel computing platform for GPUs. It allows the developer to bypass the low-level Application Programming Interfaces (API) and simply code in common computer languages for heterogeneous applications [**22**]. For the OCT imaging software, the CUDA language we used is CUDA C. The CUDA computing system consists of a *host*, traditionally a CPU, and one or more *devices* that are typically GPUs. Each CUDA source file can have a mixture of both host and devices code. Any traditional C/C++ code that runs on CPU will become host code by definition. Once device functions and data declarations are added to a source file, it is no longer acceptable to a traditional C compiler [**24**]. Thus, the mixed source file needs to be compiled by a compiler that recognizes these additional functions and declarations, which is NVIDIA C Compiler. (NVCC) Figure 2-4 shows the processing flow for CUDA program compilations.

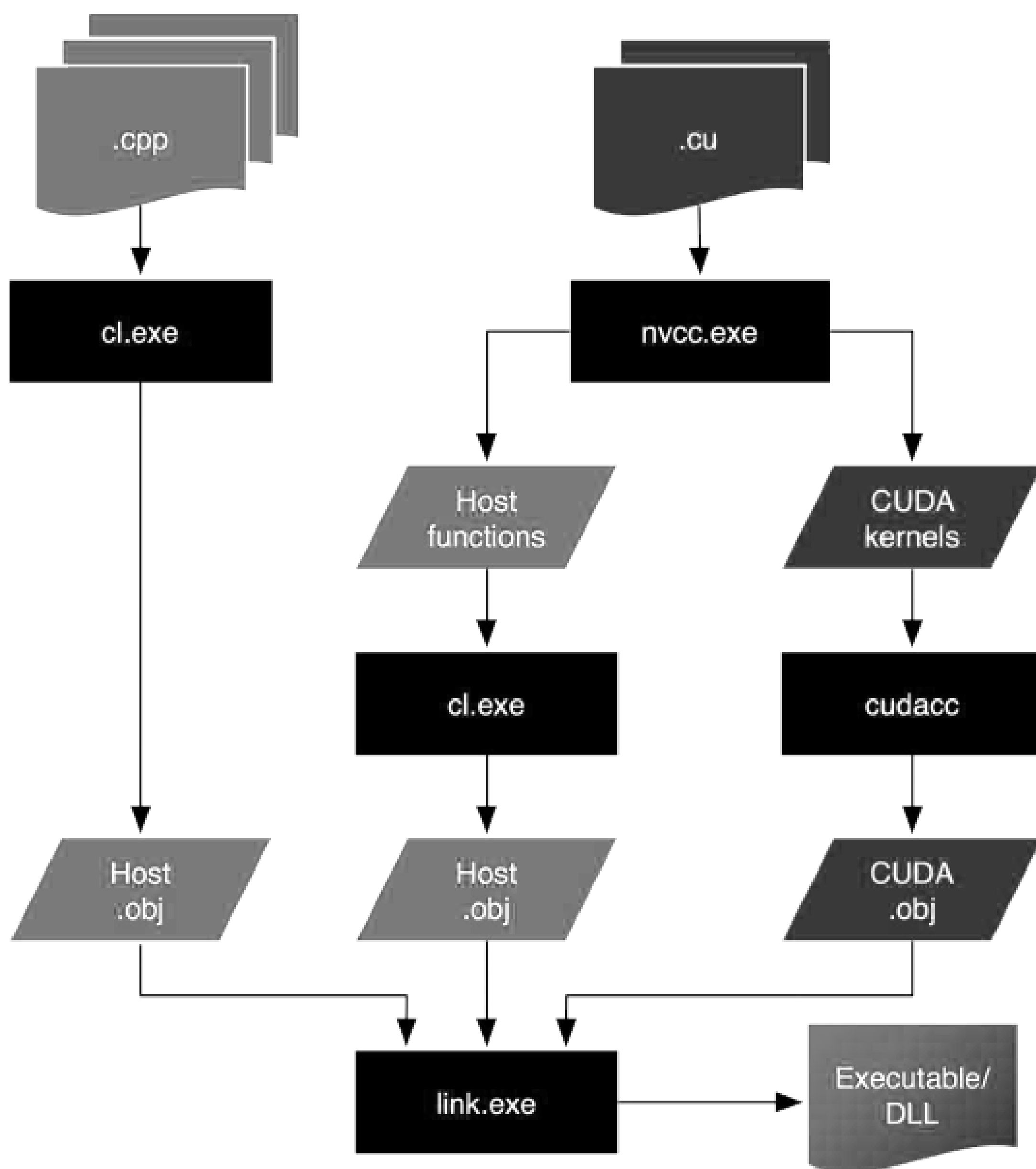


Figure 2-4 Code compilation flow [18]

The term “scalable” indicates the ability to scale the performance in parallelism through leveraging an increasing number of processor cores throughout the technology development. The platform enables data parallelism through providing hierarchies of thread groups and shared memories. This hardware architecture guides the programmer

to partition the big challenging problem in sub-problems that can be solved independently in parallel by blocks of threads, and each sub-problem into finer pieces that can be solved cooperatively and simultaneously by all threads within the block. All blocks of threads will be scheduled on available SMs as shown in Figure 2-5.

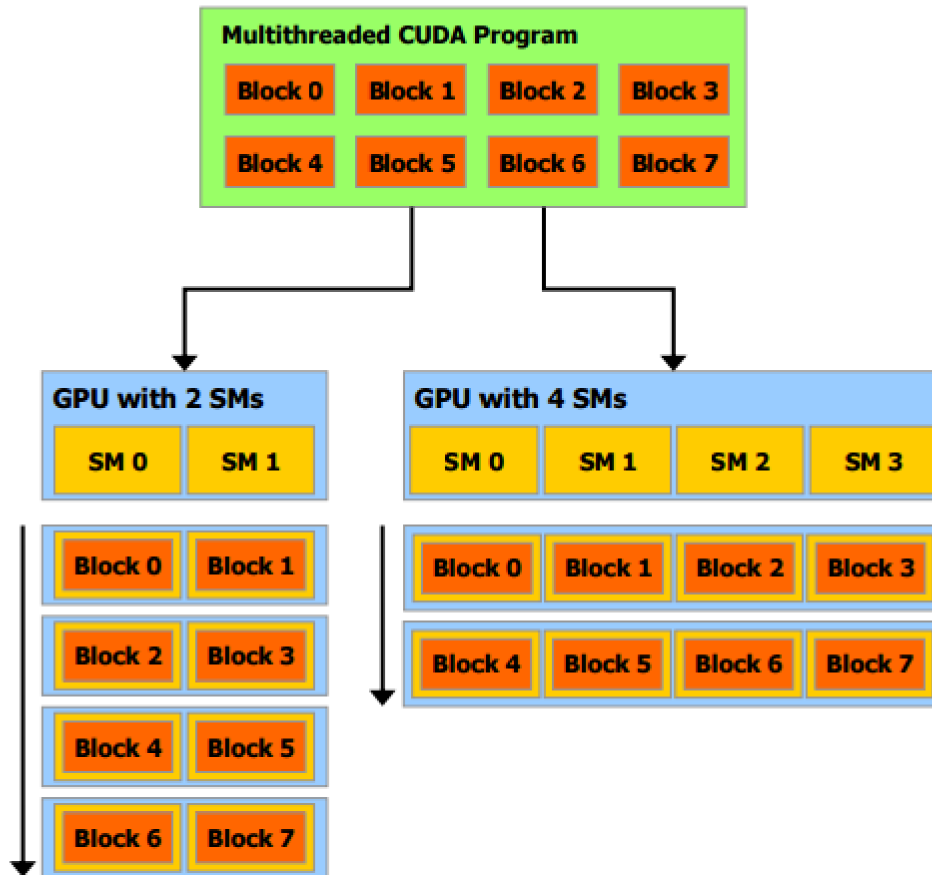


Figure 2-5 An example of a multithreaded program partitioned into blocks of threads that execute independently on each GPU [22]

CUDA employs a Single-Program-Multiple-Data (SPMD) programming model such that sequential instructions from one host thread instantiate many device threads in parallel. When executing a CUDA program, the kernel functions are launched by large number of threads on a device. All threads that are generated by a kernel launch are collectively called a *grid*. A grid consists of an array of thread blocks, and each thread block can contain up to 2048 threads for modern GPUs. Since CUDA uses sequential instructions, the program will start from the host and continues on the host until another kernel is

launched or the program is terminated. When all threads of a kernel complete their execution and the corresponding grid termination, the execution will then return back to the host and repeat. Figure 2-6 shows an example for a CUDA program execution.

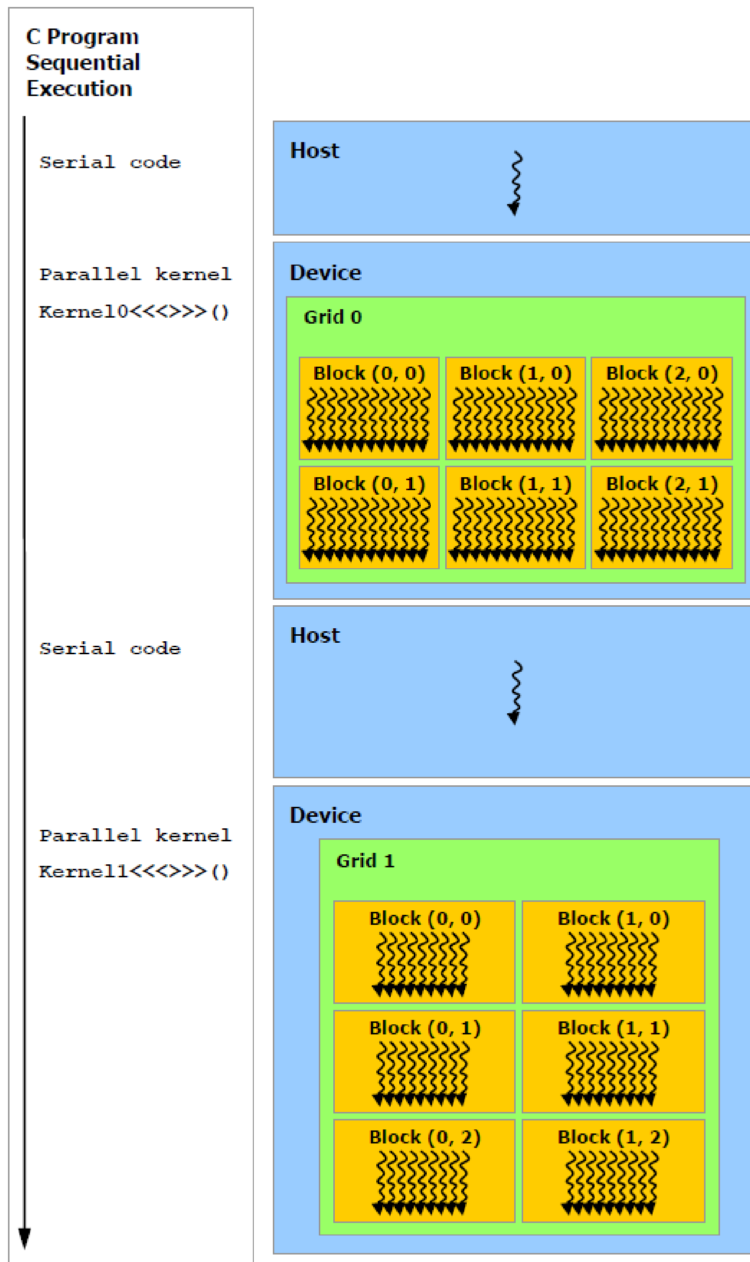


Figure 2-6 CUDA program execution flow [22]

For demonstration purpose, the CPU and GPU execution do not overlap in the example shown above. In real world applications, those two can overlap to reduce the overhead and improve the performance. Figure 2-7 is a sample code for a kernel definition and launch in CUDA C.

```
1  #include "cuda_runtime.h"
2  #include "device_launch_parameters.h"
3  #include <stdio.h>
4
5  __global__ void addKernel(int *c, const int *a, const int *b)
6  {
7      int i = threadIdx.x;
8      c[i] = a[i] + b[i];
9  }
10
11 int main()
12 {
13     const int arraySize = 5;
14     const int a[arraySize] = { 1, 2, 3, 4, 5 };
15     const int b[arraySize] = { 10, 20, 30, 40, 50 };
16     int c[arraySize] = { 0 };
17
18     // Add vectors in parallel.
19     addKernel <<< 1, arraySize >>>(c, a, b);
20
21     printf("{1,2,3,4,5} + {10,20,30,40,50} = {%d,%d,%d,%d,%d}\n",
22         c[0], c[1], c[2], c[3], c[4]);
23
24     return 0;
25 }
```

Figure 2-7 Sample code for kernel launch

The host and devices have separated memory due to the fact that devices have their own dedicated VRAM (global memory). To execute a kernel, the memory for the data needs to be allocated to the device memory space and the data itself needs to be transferred from the host memory space to the allocated device memory. Similarly, after device execution, the result needs to be transferred back from the device memory to the host memory and stored to free up the device memory as they are no longer needed.

However, please note that the data transfer latency through PCIE is huge and needs to be minimized as much as possible. Thus memory transfer between the host and devices shall be restricted to only large grouped raw data input or for displaying results. For OCT imaging software, after the retinal image is finished processing from raw spectral interferometric data, the result will be sent back to the host for OpenGL™ display. And then the cycle will repeat for another segment of spectral data input.

2.6. Summary

In this chapter, we covered the basics of CPUs and GPUs and compared them for their advantages and disadvantages of usage. The comparison laid down the narratives for the demand of heterogeneous computing for optimal HPC. A programming platform called CUDA is used and described to further explain the structure of heterogeneous computing. The core algorithms for retinal GC segmentation will be discussed in the next chapter.

Chapter 3.

Segmentation theory

In the previous chapter, the basics of heterogeneous computing for acceleration of retinal segmentation was presented. In this chapter, the details of Push-Relabel Graph-Cut algorithm (PR GC) and Connected Component and Labeling (CCL) that were used in the implementation of this thesis will be discussed.

3.1. Graph-cut background

In computer vision, segmentation is the process of partitioning digital images into multiple regions (sets of pixels), according to some homogeneity criterion. The problem of image segmentation is a well-studied subject. While there are a variety of approaches to solving this problem, minimum cut / maximum flow algorithms have emerged as the preferred tool over others due to their relative efficiency and accuracy. The general energy based function can be summarized as follows:

$$E(L) = \sum_{p \in P} D_p(L_p) + \sum_{(p,q) \in N} V_{p,q}(L_p, L_q) \,, \qquad \text{Eq.3-1}$$

where $L = \{L_p | p \in P\}$ is a labeling image of P; $D_p(\cdot)$ is a data penalty functions which indicates the individual label-preference of each pixel based on its observed pixel intensities; $V_{p,q}$ is an interaction potential which penalizes the discontinuity between neighbouring pixels; N is a set of pairs of neighboring pixels [**10**]. In the context of flow networks in image segmentation, those pairs of neighbouring pixels are defined as being connected by edges E with each pixel $v, w \in V \times V$ where V is the vertices of the graph. The penalties in the energy function are referred as ‘cut cost’ and a cost is assigned to

all edges in the graph. Image segmentation is achieved by cutting the flow network into partitions.

In practical applications, we want to extract the target region of a given image (defined as foreground) and treat the rest of the image as noise (defined as background). This will bring down the labeled image into just a binary image. Thus, there are two special terminal vertices added to the network: source s and sink t. The flow network constructed from the given image is then referred as a graph G. The cost of the cut C is the total cost of all edges separating the two groups. The cost of an edge is also called a capacity in terms of flow networks. The theorem of Ford and Fulkerson stats that a maximum flow from s to t saturates a set of edges in the graph dividing the vertices into two disjoint parts $\{S, T\}$ corresponding to a minimum cut. Thus finding the maximum flow problem is equivalent to finding a minimum cut. The process of image segmentation that uses algorithms based on maximum-flow/minimum-cut is often referred as graph cuts.

Generally speaking, the algorithm for graph cut can be categorized into two groups: Goldberg-Tarjan "push-relabel" and Ford-Fulkerson "augmenting paths" [**10**]. In this section, we will focus on the push-relabel algorithm as this approach exhibits good data parallelism and maps well to the CUDA programming model.

3.2. Push-relabel Algorithm for Graph Cuts

In the general maximum flow problem, a directed weighted graph $G = (V, E)$ consists of a set of vertices V and set of directed edges E that connects these vertices. The size of V is total number of pixels n and size of set E is total directed number of edges between each vertex m. A source s and a sink t are additional terminal vertices for data partitioning. Each edge $(v, w) \in E$ has a capacity $c(v, w)$. Directed edges indicate that the capacity of flowing is directional, meaning $c(v, w) = -c(w, v)$. For all edges $(v, w) \notin E$, we define $c(v, w) = 0$. The push-relabel algorithm introduces extra components to the equation: pre-flow, excess flow, residual graph and height labeling.

1. Pre-flow f_p

In Ford-Fulkerson's augment path algorithm, a flow f on the graph G is a real valued function that satisfies the following constraints:

a) $f(v,w) \leq c(v,w)$ given $(v,w) \in E$
b) $f(v,w) = -f(w,v)$ Eq.3-2
c) $\sum f_{in}(v) = \sum f_{out}(v)$ given $v \in V - \{s,t\}$ and all edges $(v,w) \in E$

The first constraint states that an edge on the graph can only carry flow less than or equal to its maximum edge capacity. The second constraint states that the flow is also directional. The last constraint states the flow conservation for all edges on the graph except the edges that directly connecting terminal vertices s and t.

The definition of a pre-flow f_p is same as the flow definition in Ford-Fulkerson's augment path algorithm except that the conservation constraints have been relaxed so that the amount of flow into a vertex is allowed to exceed the amount of flow out of the vertex. Hence, the pre-flow satisfies:

a) $f_p(v,w) \leq c(v,w)$ given $(v,w) \in E$
b) $f_p(v,w) = -f_p(w,v)$ Eq.3-3
c) $\sum f_{p,in}(v) \geq \sum f_{p,out}(v)$ given $v \in V - \{s,t\}$ and all edges $(v,w) \in E$

The relaxation of flow conservations introduces the second component, excess flow.

2. Excess flow f_e

For a vertex $v \neq \{s,t\}$, the excess flow $f_e(v)$ is defined as the net flow into vertex v:

$$f_e(v) = \sum f_{p,in}(v) - \sum f_{p,out}(v) \qquad \text{Eq.3-4}$$

The vertex $v \in V$ is active (overflowing) if the excess flow $f_e(v) > 0$. Note that a pre-flow becomes a flow if and only if the flow conservation is realized, in other words, the excess of every non-terminal vertex is zero. Thus transforming a pre-flow to flow that saturates the network involves reducing and eventually eliminating all excess flows.

3. Residual graph G_f

A residual capacity of an edge is defined as:

$$c_f(v, w) = c(v, w) - f_p(v, w) \qquad \text{Eq.3-5}$$

, where c is the edge capacity. The graph consists of residual edges is called a residual graph G_f. For an edge $(v, w) \in E$ that carries flow and capacity, the residual graph G_f includes a forward edge with the residual capacity c_f and a reverse edge with residual capacity of pre-flow f_p. Edges with zero residual capacities are omitted from the residual graph G_f.

4. Height function $h(v)$

The height function is used to ensure the regulation of push operation and termination of the algorithm. Imagine that each vertex in the flow network is a water tank, with unlimited water supply coming from source s. The goal is to push water flowing into sink t as much as possible. Since the water flow can only travel downwards, the height of each water tank needs to be labeled in order to regulate the push operation. The excess flow at vertex $v \in V$ can 'push' if and only if the following three constraints are met:

a) $h(s) = n$, where n is the number of vertices in the graph G_f
b) $h(t) = 0$
c) $h(v) \leq h(w) + 1$ for $(v, w) \in E_f$ where E_f is the collection of edges in residual graph G_f

Eq.3-6

The third constraint is stating that the pre-flow can be pushed downhill but cannot be pushed too fast. Since the source is starting at height n, t is at height 0, and each edge of the residual graph only goes downhill by at most 1, there cannot be any *s-t* path with more than *n-1* edges. Thus when there is a feasible pre-flow with no excess flow left, and all the labeling constraints are hold, then the flow must be a maximum flow.

3.3. Operation of the algorithm

The high-level strategy of the algorithm is to maintain the three invariants above while trying to zero out any remaining excesses. The operation consists of initialization, and then iteration between push and relabeling. The initialization starts by creating a residual graph G_f:

- Set $h(s) = n$;
- Set $h(t)$ and all $h(v)$ for $v \in V - \{s, t\}$ to $h(v) = 0$;
- Set $f_p = c$ for all edges outgoing from source s
- Set $f_p = 0$ for all other edges $(v, w) \in E$

The height constraints hold only during push operations. Thus the pre-flow can flow to vertices connecting to the source even though their height difference is more than 1. Then, the push operation is restricted by the height constraints:

- Choose an outgoing edge (v, w) of v in G_f with $h(v) = h(w) + 1$
- $\Delta = min\{ f_e(v), c_f(v, w)\}$
- Push Δ along the edge (v, w)

Once all outgoing edges (v, w) of v in G_f are saturated, and all residual edge c_f from v are not available for push due to the height constraint, the relabel operation is invoked to increase the height so that the flow can be continued.

In summary, the algorithm in pseudo code is as follows:

```
push_operation(excess_flow, residual_capacity, const height)
{
        for (each vertex in V)
                for (each w = neighbour(v))
                        Δ= min(residual_capacity(v,w), excess_flow(v))
                        excess_flow(v) = excess_flow(v) – Δ
                        excess_flow(w) = excess_flow(w) + Δ
                        residual_capacity(v,w) = residual_capacity(v,w) – Δ
                        residual_capacity(w,v) = residual_capacity(w,v) + Δ
}
```

```
relabel(height, const excess_flow, const residual_capacity)
{
        for(each vertex in V)
                temp_height = infinite
                for(each w = neighbour(v))
                        if(residual_capacity>0)
                                temp_height = min(temp_height, height(w)+1)
                height(v) = temp_height
}
while (vertex v != s or t)
{
        choose a vertex v with maximum height h(v)
        if(vertex v is overflowing)
        if(outgoing_edge(v,w) with h(v) = h(w)+1 exists)
                push_operation()
        else
                relabel()
}
```

Here is an intuitive example of the operation of the algorithms. Assuming the vertices in the network are water tanks and all edges connecting them are water pipes. We want to find the maximum flow of water from the source tank to the sink tank. Each of these water tanks are arbitrarily large and will be used for accumulating water. A tank is said to be overflowing if it has excess flow, or water, in it. Tanks are at height from ground level. The water can traverse from a higher lever to a lower level. We can push new flow from a tank to another which is downhill from the first one. Note that there still can be a flow from a lower tank to a higher tank, the height level only determines the direction of a new flow.

The initial height of the source s is set at n and sink t is set at 0. All other tanks have initial height 0, and their height increases as water flows in. There will be an infinite amount of flow coming into the source s, pushed towards the sink t until the whole flow network is saturated. Each outgoing pipe carries flow at its maximum capacity. The flow will be pushed downhill gradually from an overflowing tank to another tank. If an overflowing tank is at the same level or below the tanks to which it can push flow, then the current tank will be raised in level to a height just high enough to push more flow. If, after all the push operations are complete and the sink t is not reachable from any overflowing tank, all the excess flow will be sent back to the source s. Since the height of the source is n and there are only n tanks besides the source and sink, eventually all the tanks except the source and sink will stop overflowing. At that point, the maximum flow state is reached.

3.4. Parallel implementation:

3.4.1. Parallel push

The push operation at arbitrary tile v is dependent on edges connecting neighboring tiles and its excess flow and height. Thus, in parallel structure, the intra-tile dependencies need to be handled cautiously. For instance, a push from v to neighbor w will update the excess flow for both v and w, as well as the residual edge capacities (v, w) and (w, v). To avoid potential read-after-write or write-after-read hazard, all the variables can be processed within the current neighbor direction where the pushing operation occurs:

```
push_single_direction(row_index, excess_flow, residual_capacity, const height)
{
        for(each vertex from row_index to block coverage)
                w = v+1 //right direction for instance
                if(vertex v is overflowing)
                        Δ= min(residual_capacity(v,w), excess_flow(v))
                        excess_flow(v) = excess_flow(v) – Δ
                        excess_flow(w) = excess_flow(w) + Δ
                        residual_capacity(v,w) = residual_capacity(v,w) – Δ
                        residual_capacity(w,v) = residual_capacity(w,v) + Δ
}
```

Thus, for single direction operation, there are no intra-tile dependencies from tiles that are orthogonal to the pushing direction. Therefore, the push operation can be divided into directional based sections. One single direction processing can be further sub-divided into multiple groups to process in parallel. There is a potential hazard only when assessing the excess flow between the terminal tiles of each chunk. This problem is dealt by assigning one CUDA block with four warps (128 threads) where each thread transports flow over a chunk of 8 pixels per direction. The update of the neighbors outside of the assigned chunk is done after all threads are synchronized. This step ensures that no data hazard will occur during the operation [**25**]. Updates outside the tiles will be stored into a special border array and use a separate kernel to add the border array to the excess flow after each push kernel to avoid data hazard.

3.4.2. Parallel relabel

The parallelization of the relabel operation is simpler in comparison to push operation. All height variables can be updated in parallel by launching one thread per vertex. Since the height value stands by itself for each vertex, there will not be any racing condition occur during relabel operation.

3.5. Connected Component and Labeling

No algorithm is perfect. Sometimes the PR GC could segment unwanted regions as foreground. These unwanted regions would lead to a corrupted result for retinal segmentation. Thus, further processing steps are required to ensure the removal of all unwanted segmentation artifacts. Connected Component Labeling (CCL, also known as Connected Component and Analysis “CCA”) is an algorithmic application of graph theory that detects connected elements in input data and finds a unique label for every set of connected elements [**26**]. CCL is often used as a complementary algorithm to identify which of those results extracted by segmentation is the region of interests. For the result of retinal segmentation, the extracted retinal layers are in much larger connected groups than those of small segmentation artifacts. CCL will then identify the two largest connected groups as retinal layers (ILM and BM) and remove all other unwanted artifacts.

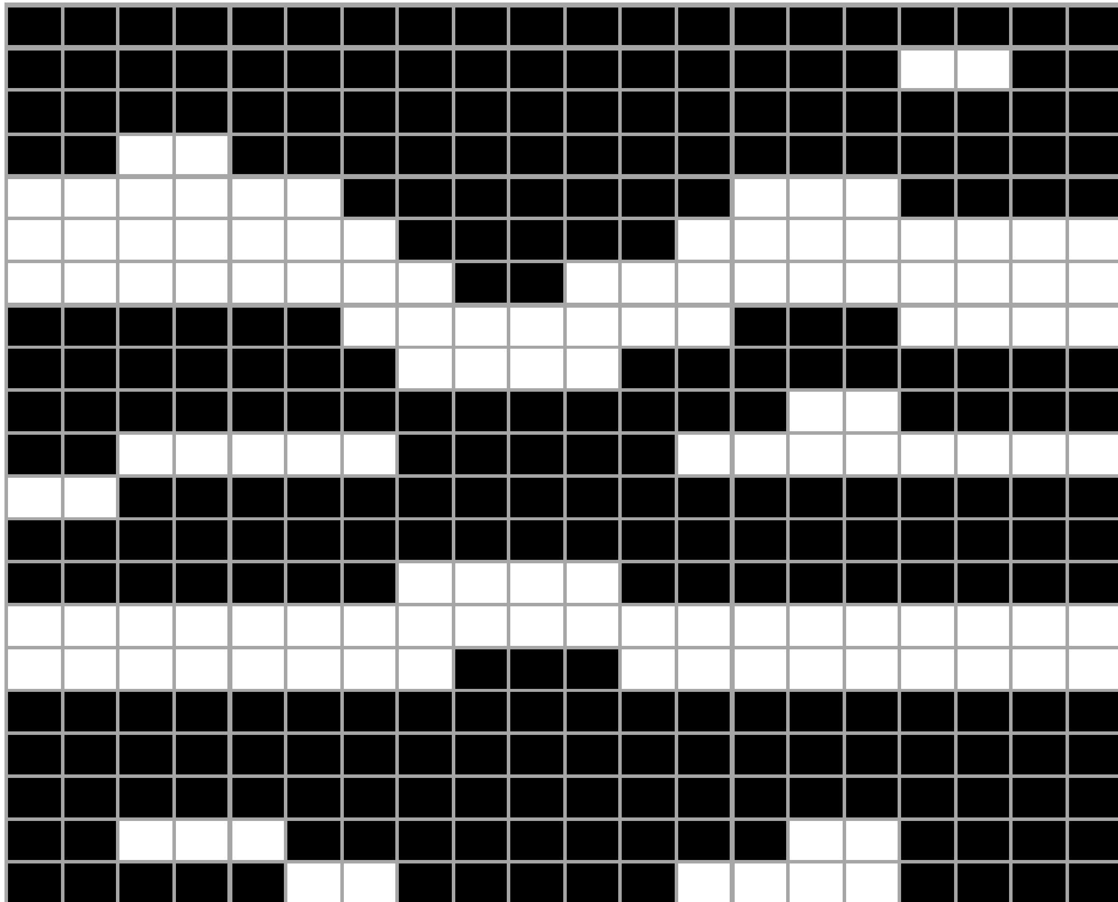
The core of the CCL is a two-pass method where the first pass assigns temporary labels and stores equivalences between all connected neighbouring elements, and the second pass analyzes the equivalences and replaces each temporary label by the smallest label of its equivalent class. One additional step is added for extracting the two largest connected groups to deliver the final segmentation result. The flow of the algorithm is as follows:

1. First Pass:
    a) Iterate through each element of the data in a raster scanning pattern
    b) If there are no labeled neighbours around the current element, uniquely label the element and continue. Only North-East, North, North-West and West are checked and needed for label look-ups.
    c) If there is a labeled neighbour, assign the current element with the smallest label that was found in the neighbour elements.
    d) Store the equivalence between neighbour labels in the labeling stack.

2. Second Pass
    a) Iterate through each element of data in a raster scanning pattern
    b) Relabel the element with smallest equivalent label from the stack
    c) Add the current element to the element counts for current label

3. Region of interest extraction
    a) Find the two connected groups with largest element counts, then set all other groups to background (set the pixel value to zero in terms of gray scale images)

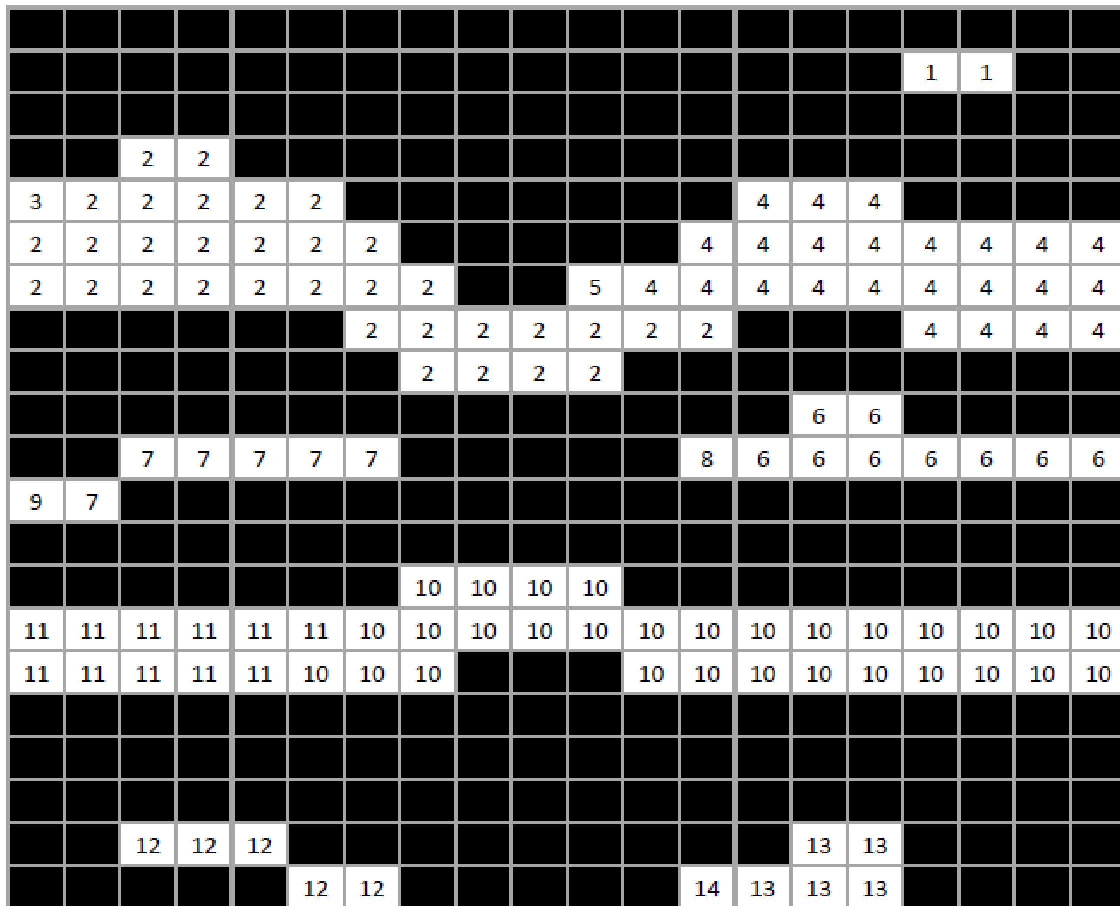
The term “neighbour” is defined as the non-background elements around the current tile(8-way connectivity). The foreground is the extracted objects from the retinal segmentation. Figures 3-1 to 3-4 provide a graphical example of the labeling process.



Stack ID	Equivalent labels	Element counts
-	-	-

Figure 3-1 Representative segmented binary image by PR GC

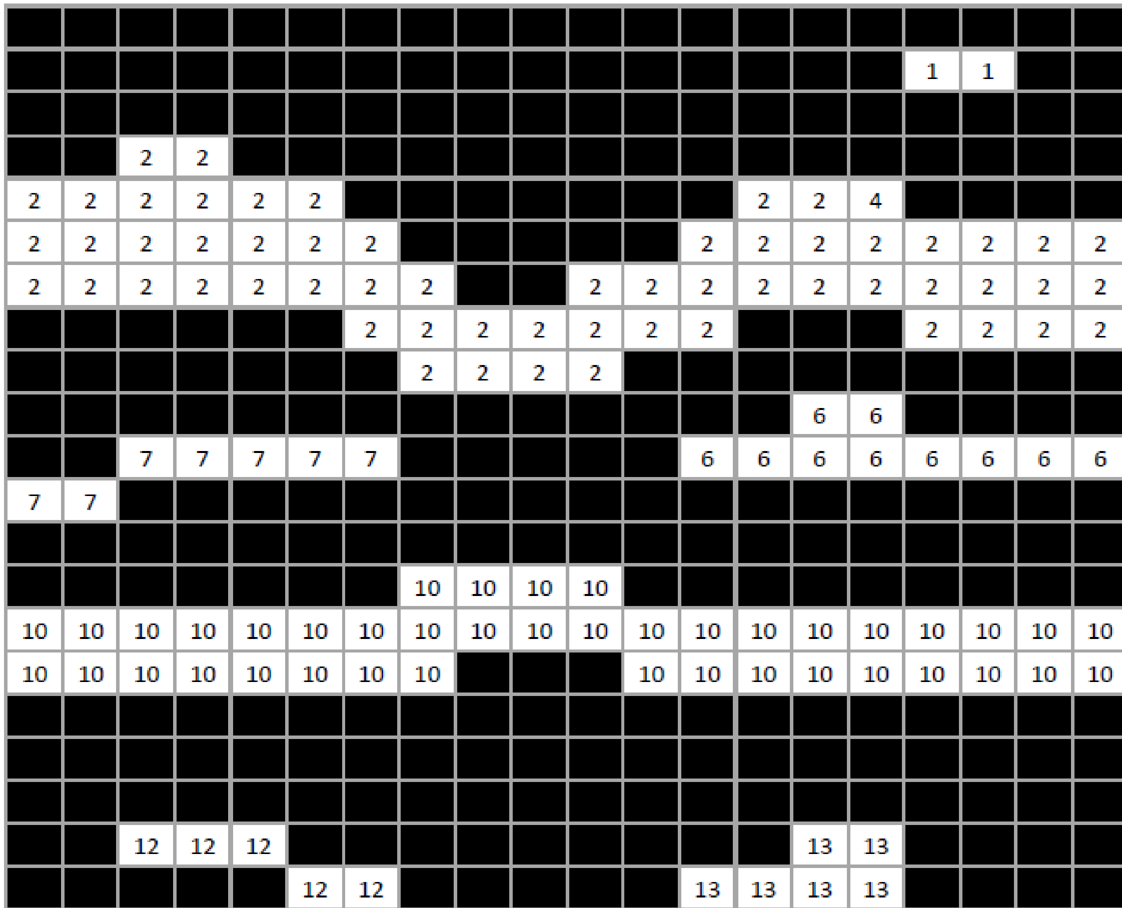
The CCL will take the segmented binary image as input, after running the first pass, the following labels are generated as shown in Figure 3-2.



Stack ID	Equivalent labels	Element counts
1	1	2
2	2,3,4,5	59
3	6,8	10
4	7,9	7
5	10,11	41
6	12	5
7	13,14	6

Figure 3-2 Generated labels after the first pass

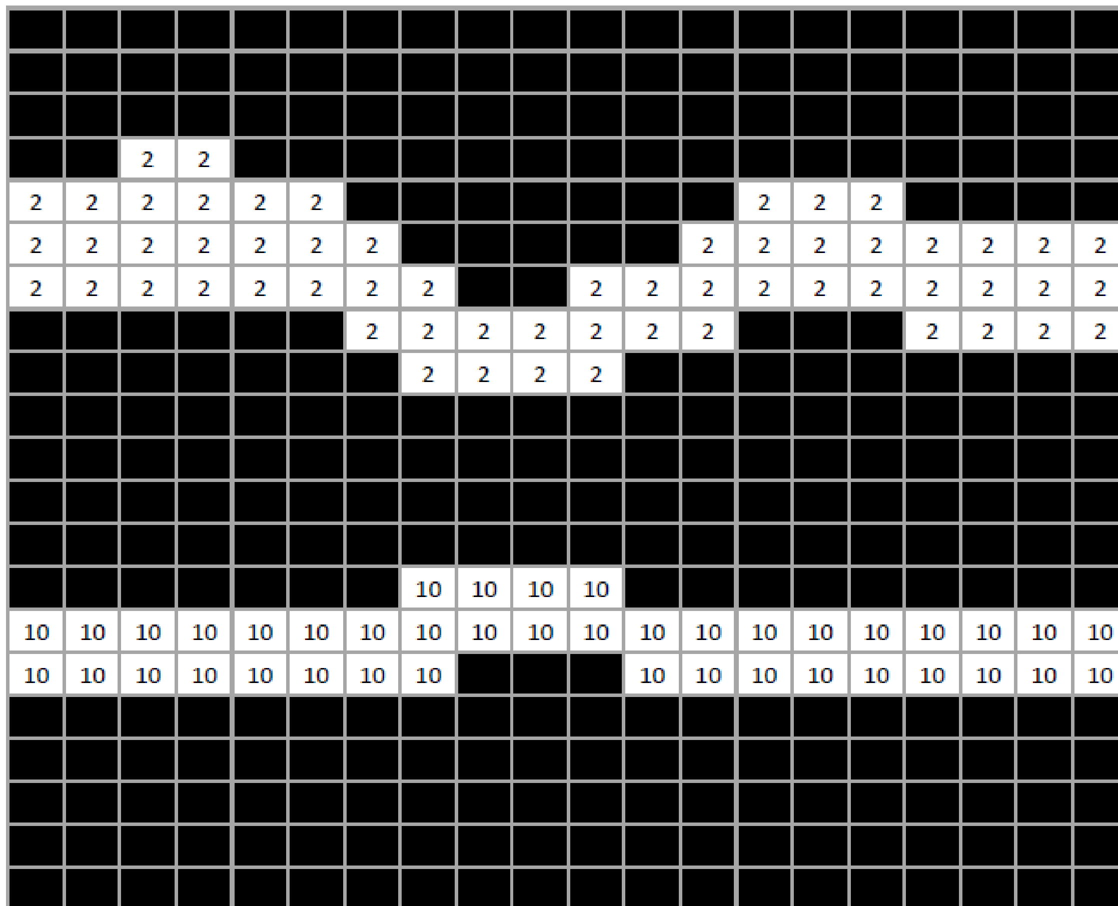
Then, in the second pass, each element label will be rearranged based on the lowest neighbouring label value, and the number of connected element will then be stored in the stack for each equivalent group.



Stack ID	Labels after merge	Element counts
1	1	2
2	2	59
3	6	10
4	7	7
5	10	41
6	12	5
7	13	6

Figure 3-3 Labeling result after second pass

In the final step, the CCL algorithm will extract the two largest connected groups and zero out all other groups. The result is shown in Figure 3-4



Stack ID	Labels after merge	Element counts
2	2	59
5	10	41

Figure 3-4 Final result of CCL

Hence, after GC segmentation, the CCL will remove all unwanted artifacts and deliver the correct result for ILM and BM layer. However, for the current revision, the CCL is implemented in traditional serial instruction processing with a CPU. This implementation slows down the segmentation pipeline by a considerable amount. This is because the CCL is inherently a sequential instruction algorithm, which is not trivial to program in parallel. In the result and discussion of Chapter 5 there will be a detailed benchmark analysis on the effect of CCL.

3.6. Summary

In this chapter, we discussed the details of Graph-Cut segmentation. The GC segmentation is based on the maximum-flow/min-cut problem introduced by Boykov et al [**10**]. There are two variants of GC algorithm, Ford Fulkerson's Augmented Path and Goldberg-Tarjan's Push-Relabel. The PR algorithm configures the flow network with pre-flow mechanism. The cut of the graph after complete iterations of push and relabeling operations represent the segmented foreground, which may contain unwanted artifacts. An additional Connected Component and Labeling step is applied to refine the final result. In the next chapter, we will apply this PR GC algorithm in our retinal imaging program.

Chapter 4.

Retinal Segmentation Pipeline

This chapter describes the implementation of GPU-accelerated Graph-Cut (GC) retinal segmentation. The result of the retinal segmentation can be used for calculating the retinal thickness between ILM and Bruch's Membrane (BM), as well as enhancing the view of vasculature networks. The SV visualization described in the following chapters was based on the work published by Xu *et al* [**27**].

4.1. Asynchronous parallel computing

The definition for real-time applications states that the program must guarantee a response within a specific time constraint to meet the process deadline. The Swept Source speckle variance OCT (SS-svOCT) acquisition system in the Eye Care Center (ECC) of Vancouver General Hospital (VGH) is equipped with a laser source with line rate of 200kHz, meaning the acquisition time for a volume at the size of 1024 x 300 x 900 frames takes around 1.575 seconds. A frame is defined by one complete scan over the transverse plane of the retina. The required timeline is very challenging to meet with single GPU solution. Thus, for real-time image segmentation, a multi-GPU solution is needed to meet the processing deadline. Data input will then be divided into sub-volumes and distributed to each individual GPU.

In the CUDA processing pipeline, data are transferred and processed in small segments rather than an entire volume. This way the program will start to load the next segment of data while processing the current ones. After current segment finished processing, the next segment will be transferred to device memory space and ready to be processed. Essentially this is a data flow control mechanism that reduces the processing lag when

new data comes in, making the imaging system more responsive to the data input. The segments of data are referred as “batches” in the rest of this article. Details of how batch processing works are documented by *Jian et al* [**8**]. With multiple GPUs installed in the system, each individual GPU will operate in the same batch structure. The flow of the processing tasks will be distributed as described in Figure 4-1.

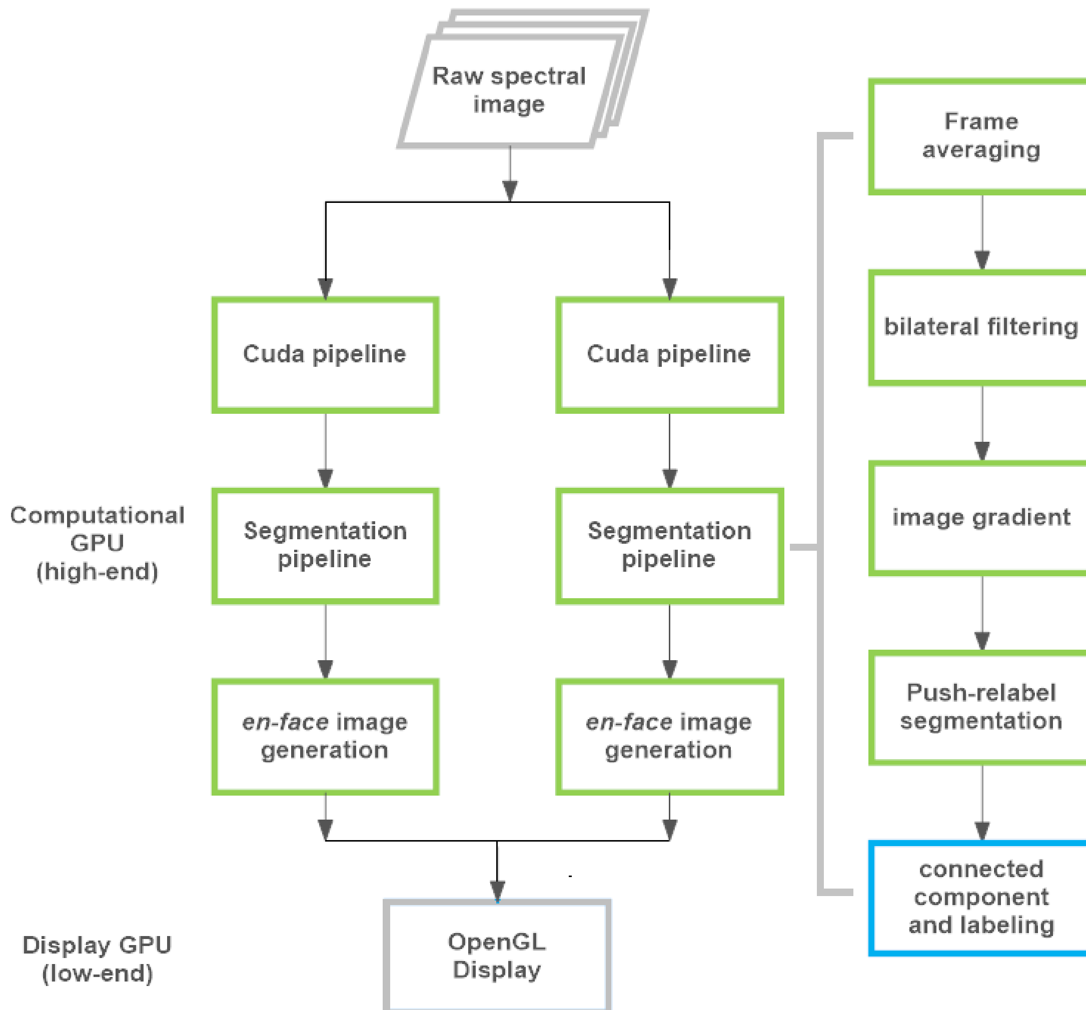


Figure 4-1 processing work flow for segmentation pipeline for dual GPU computation system

In the scenario where only one CUDA device is installed for data processing, if the data volume is partitioned into 30 batches, the CUDA device will read the data memory batch-

by-batch from the host memory space. In this case, the host is the CPU, and the device is the GPU. If the processing time of a batch is longer than the acquisition time, then the system will lag. This can be resolved by using a multi-GPU setup, in which each CUDA device will read the data from host memory with memory offsets. When there were two CUDA devices processing the data volume, device 0 will process every second batch (ie, the even batches) while device 1 will process the other ones (odd batches). Each CUDA device will treat the sub-volumes as an individual entity, which is acceptable because the data in batches are independent of each other. All sub-volumes will be processed simultaneously. After each CUDA device finishes the current cycle, the result of the sub-volume will be transferred back to the host memory. Since each individual GPU operates independently, the entire processing is asynchronous. Without considering data traffic control and device thermal throttling, the theoretical performance should be scaled by the number of GPUs running in the system. In this research, an additional GPU was installed for display tasks only. This way the CUDA devices would be able to process the computational tasks full time.

4.2. Batch processing and real-time requirement

The image acquisition and processing program was designed to have two counters: a frame counter for visualization and a batch counter for processing. A batch is an evenly distributed segment of the input volume. The frame counter is the frame index of the B-scan image within the volume;it indicates which frame from the data volume is currently being displayed. On the other hand, the batch counter is the starting frame index of each batch being processed by the GPU. When the program processes the acquired image, a regular B-scan intensity image will be processed for every single frame. On the other hand, the process of segmenting a single retinal intensity image could take up to 70ms using graph cuts. Thus, applying graph cut segmentation on every single frame wouldn't be feasible for real-time high-resolution imaging. Therefore the input image data in the batch needs to be down-sampled in order to provide fast enough volumetric processing speed.

There are two ways in down-sampling the input image: reducing the B-scan image resolution or reducing the number of frames to process. We found that during the image acquisition, the lateral movement of the patient within the acquisition time of one batch (roughly 100ms) is negligible [**27**]. Thus we decided to apply GC segmentation to the first frame of each batch, and the result can then be used to represent the entire batch.

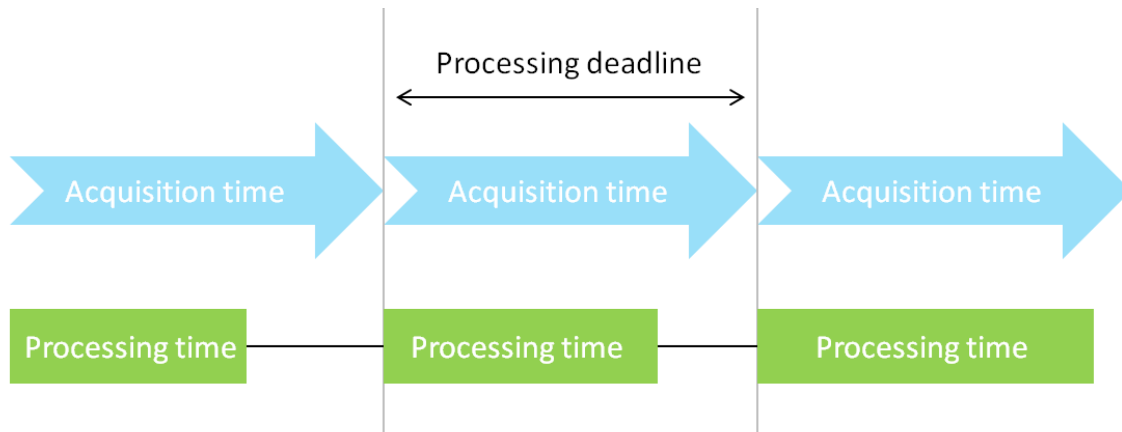


Figure 4-2 Real-time deadline

The definition of real-time processing states that the real-time program must guarantee a response with a specific given deadline. That means the processing of the current batch of data must be finished before the next input of data coming in. In the context of medical imaging, the real-time deadline is determined by the acquisition rate. That means, the retinal segmentation result needs to be finished and delivered for visualization before acquiring the next batch of data, as the result needs to reflect the change in real-time. Currently with the dual GPU setup, the processing deadline is around 105ms and the segmentation time for batch is around 70ms. For the acquisition size of 900 frames, a reasonable batch size for real-time processing would be around 30 frames per batch.

4.3. Segmentation Initialization

Before initializing the segmentation pipeline, we need to first eliminate the speckle noise presented in the image. The constructive/destructive interference corrupts the retinal image quality, ultimately making the process of segmentation pipeline slower and more

error prone. The speckle smoothing step shall not be computationally intensive, as additional processing complexity may jeopardize the overall processing time. Since the constructive/destructive interference is reflected on the image as white and black intensity fluctuations, we can simply average all of the frames in a batch (we will call this 'frame averaging') to bring down the fluctuation level; the results of frame averaging performed on a representative retinal image is qualitatively presented in Figure 4-3.

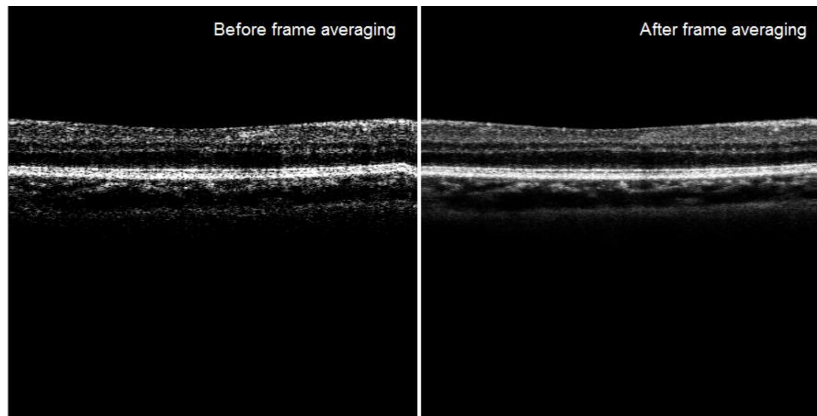


Figure 4-3 Before and after frame averaging for B-scan intensity image

The texture pattern of the retinal image will determine the run time of the segmentation processing. The more complex the image, the slower the segmentation process will be. Details on the benchmarking result will be discussed in the next chapter.

The bilateral filter is a nonlinear filter that combines the gray levels of the nearby pixels based on both of their geometric closeness and their photometric similarity [**28**]. It preserves the edge so that the retinal layers can be detected later during the segmentation step. The gray scale bilateral filter for GPU is implemented from a RGBA bilateral filter implementation in CUDA SDK sample application. The CUDA implementation is based on the same published articles [**28**]. Each pixel is weighted by considering both the Gaussian spatial distance and Euclidean color distance between its neighbours. Figure 4-4 shows the representative effects of a bilateral filter for imaging smoothing. As a result, the texture pattern of the filtered image has become simpler. The Gaussian and Euclidean parameter were set such that the image was further smoothed

while preserving the retinal edges. This step is required for PR algorithm to perform better in terms of correctness and speed.

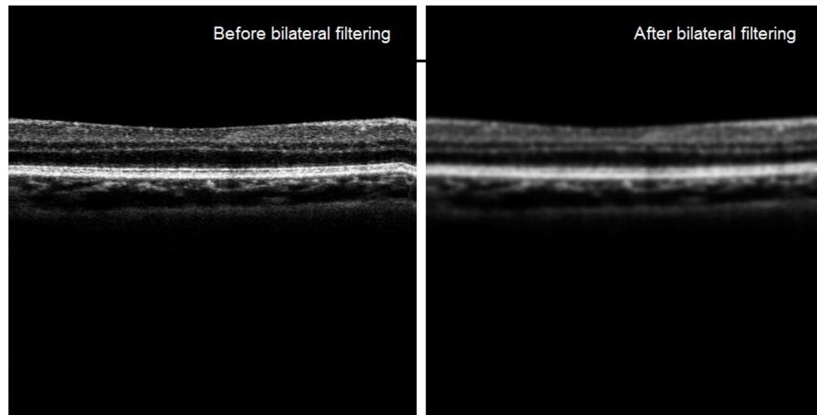


Figure 4-4 Bilateral filtered B-scan intensity image

After the filtering stage, we want to construct the graph for GC. As discussed in the previous chapter, a graph is a flow network that consists of node and edges. The goal is to cut the graph to yield a set of coordinates that represent the result of image segmentation. Details of the algorithm were described in Chapter 3. In the retinal image, we define the foreground as the retinal layers, and the background as the vitreous and posterior chamber. According to the PR algorithm, the residual edge that doesn't have the forward capacities will be removed from the residual graph. Thus, the edge capacity for which connections between objects and background needs to be smaller than the edge connecting within objects or background so that they can be easily saturated from the push operation. Since the transition in pixel values from background to objects are quite large, the easy way to construct such a graph can be simply done by calculating the gradient of the retinal image and assigning smooth-term (n-link) edge capacities with values inversely proportional to the absolute intensity values of the gradient image. Data-term (t-link) edge capacities are assigned based on the intensity levels of the original image. Figure 4-5 shows a gradient generated from bilateral filtered retinal image.

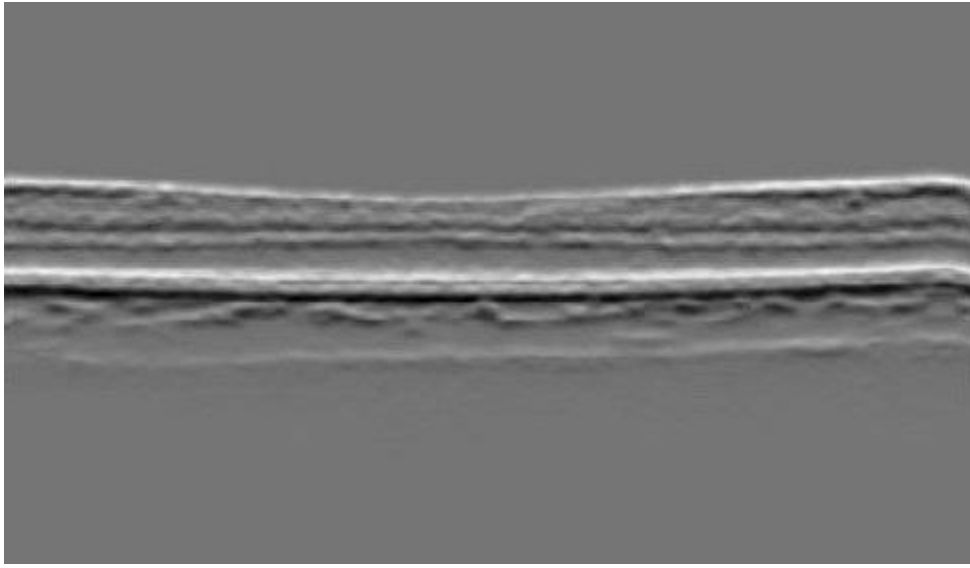


Figure 4-5 Gradient image for graph construction

4.4. Binary image segmentation with GPU

The NVIDIA Performance Primitives Image (NPPI) library is a collection of GPU-accelerated functions that was pre-developed by NVIDIA for their GPUs. The NPPI library contains the GC segmentation function that is based on PR introduced in Chapter 3. Although PR doesn't have the best theoretical time complexity, in all practicality, PR performs the best in terms of average running time [**11**]. The NPPI GC segmentation function takes the input constructed from the previous section and outputs a binary image with the object region highlighted as white. Figure 4-6 shows a representative retinal segmentation result for the macular region.

Figure 4-6 Binary image segmented by NPPI GC

Note that there are additional artifacts beneath the segmented layers. The GC function outputs what the algorithm recognizes as objects, which could contain undesired artifacts that may corrupt the result. The undesired artifact can be eliminated by adjusting the image histogram. However, a manual approach to tuning the intensity and contrast would defeat the purpose of minimizing the human effort during the clinical use of this software tool. The image contrast adjustment could be automated, but since every image volume has different histogram distributions and requires precise histogram

adjustment to eliminate the artifacts, this may not be a reliable solution in a clinical environment. Therefore, additional processing steps to automatically eliminate the undesired artifacts are required such that the clinical practicality can be achieved.

4.5. Layer extractions through CCL

The NPPI library provides morphological functions such as image erosion and dilation. Usually, the undesired artifacts from the GC segmentation can be eliminated by such morphological functions if the artifacts are small enough. However, in OCT imaging, artifacts are often created by beam echoes and other noise, of which are in large and connected forms that cannot be removed by erosion and dilation.

An algorithm named Connected-Component Labeling (CCL) is often used in companion with the segmentation algorithm; the details of CCL were presented in Chapter 3. Rather than smoothing the artifacts as with erosion and dilation, the CCL algorithm will label each connected region so that we can output specific regions. In this case, our regions of interest are the two largest connected components in the binary image, which represent the ILM layer and the BM / Retinal Pigment Epithelium (RPE) complex, respectively. For details of CCL, please refer to chapter 3. Figure 4-7 demonstrates the result after applying CCL.

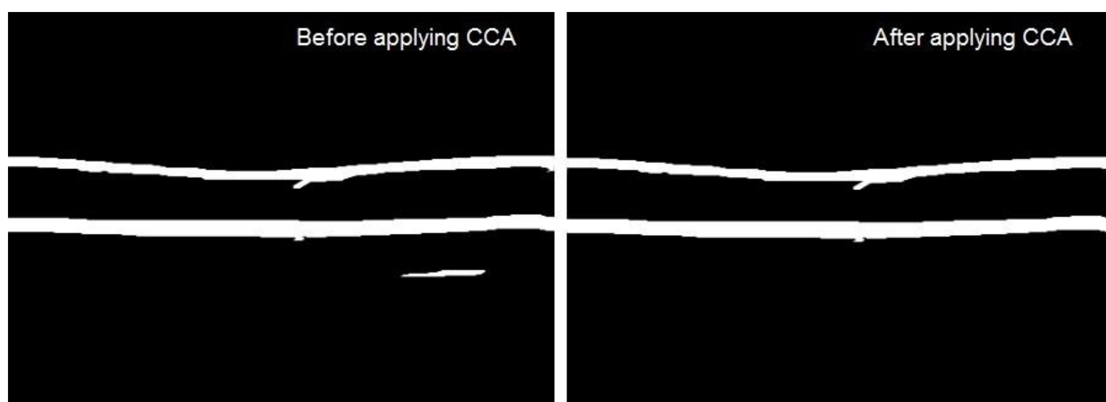


Figure 4-7 Before and after applying CCL on the binary image

Note that the CCL algorithm was implemented using CPU. This step alone takes up to nearly 20% of the entire run time when implemented in serial operation. Due to the sequential nature of CCL, we found it rather difficult to implement CCL properly in CUDA. Certain CCL algorithms can be parallelized, but it has been reported that the sequential algorithms often outperform the parallel ones in real applications [**29**]. There have been reports on implementations of CCL algorithms for other applications in Computer Vision using CUDA [**25**]. However, this implementation was not adapted to our existing program due to time constraints. After GC segmentation and CCL, the final step is to locate the coordinate of the segmented layers in the image. Figure 4-8 is a representative result for GPU retinal GC segmentation.

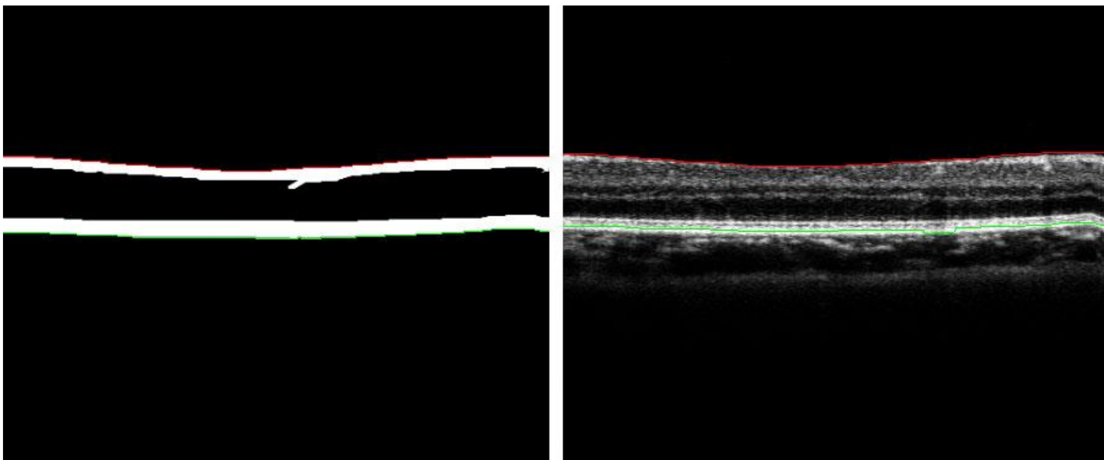


Figure 4-8 Line overlay for GPU retinal segmentation result

To this point, we have successfully extracted the targeted retinal layers from the B-scan intensity image. Recall that in Section 4.2 we downsized the total number of frames by processing only one frame in each batch. .Therefore, before sending the segmented layer to the next stage, which is *en-face* image display, the resulting coordinate set needs to be re-interpolated back to full frame size. As explained in previous sections, since the movement within one batch imaging is negligible, one set of segmented coordinates is sufficient for the clinician to evaluate the result in real-time. After segmentation and interpolation operations are complete, the program will then proceed to generate a thickness map and the SV angiography for the final *en-face* views for each of the sub-volumes they were assigned. These results from each batch will then be sent back to host memory space and combined for OpenGL display. The detailed analysis of

qualitative and quantitative result for GPU Retinal layers Segmentation (GRS) will be discussed in the next chapter.

4.6. Summary

In this chapter, we described the processing pipeline of retinal segmentation using GPU. The segmentation algorithm uses a graph input that was interpreted from a cross-sectional B-scan image of the retina. The retinal image is required to be pre-processed using frame averaging and bilateral filtering before proceeding to set up the graph. The GPU graph segmentation is executed by using a pre-built NPPI library. The GPU segmented results could contain undesired artifacts. An additional processing step of CCL is applied to reliably extract the targeted objects from the segmented result. After the segmentation result is acquired, the coordinate of targeted retinal layer will be recorded and sent back to the host for *en-face* view visualization.

Chapter 5.

GPU-Based Retinal Layer Segmentation Result and Discussion

In this chapter, we will be reviewing the GPU retinal segmentation (GRS) results both qualitatively and quantitatively. At the same time, we will be investigating the environmental factors that may affect the result and the optimization of the algorithm based on the tested cases.

5.1. Environmental setup

The retinal image acquisition system in VGH is a SS-svOCT system with a 1060nm laser source that operates at a 200kHz A-scan rate, meaning the acquisition time for a SV volume at the size of 1024 x 300 x 900 is roughly 1.575 seconds. All data volumes used in this chapter are all acquired in VGH but tested on the code development machine in the BORG lab at SFU. The hardware specification for the development machine is listed in the following table.

Table 5-1 Hardware specification of testing bench

Components:	Model	Clock Speed	Description
CPUs	2 x E5-2620	2.3GHz	2 x 6 cores, on chip PCIE controller provides x40 PCIE3.0 lanes
RAM	8 x 8 GB DDR3	1866MHz	Overclocked DDR3 RAM with CL9 latency
Display GPU	GTX 460 1G VRAM	778MHz	Fermi architecture device, x8 PCIE lanes were in use when running the program
Compute GPU 0	Quadro K6000 12G VRAM	900MHz (797MHz)	Kepler architecture device, x16 PCIE lanes were in use when running the program, Thermal throttled to 797MHz due to spacing
Compute GPU 1	Quadro K6000 12G VRAM	900MHz	Kepler architecture device, x16 PCIE lanes were in use when running the program

Note that the development machine is a server based entry-level workstation. The single core computational capability of the CPU in the development machine is not on par with the single core capability of the CPU in the acquisition machine. Our imaging software relies more on the CPU single-core capability because of the heavy use of algorithms and OpenGL rendering. As a result, the low clock frequency of CPU on the development machine bottle-necks the performance by a considerable amount. This setup was used because this is the only machine available with the full x40 PCIE lanes at the time of writing this thesis. On the good side, if the un-optimized hardware setup could achieve the targeted performance, then real-time performance is guaranteed in a proper hardware environment. Moreover, both computer hardware (CPU and GPU) are two generations behind the state-of-the-art technology, which means the code performance can be boosted by using up-to-date hardware. More details on overall system optimization will be discussed in the discussion and future work section.

5.2. Qualitative result of GPU retinal layer segmentation

In this section, we will be only focusing on the quality aspect of the GPU Retinal Segmentation (GRS). Both of the results for thickness measurement and Segmented Speckle-Variance Angiography (SSVA) will be reviewed. In each section, the review will start from successful results for both healthy and pathological volumes, followed by cases where the GRS could fail. All data volumes used in this chapter were acquired from the ECC at VGH.

5.2.1. Thickness measurement

The macular region of a representative healthy retina is shown in Figure 5-1. GRS accurately detected the ILM layer and BM layer as shown in Figure 5-1 (A). Panel 5-1 (B) and (C) are the *en-face* view and the thickness map of the retina respectively. In comparison to a gold standard human delineation of these layers, the automated segmentation result follows the target layer correctly in most cases with location mismatch less than 3 pixels distance. The SS-svOCT image acquisition system was equipped with a 1060nm laser source, which provided an axial resolution of roughly ~2μm per pixel after Fourier transform [**30**]. Given that the segmented result tolerates up to 3 pixels difference, which translates to roughly 6~7um tolerance in actual scale, the segmentation result is on par with the axial resolution and able to provide accurate information for retinal thickness measurement. Figure 5-1 shows a representative result of retinal thickness map visualization.

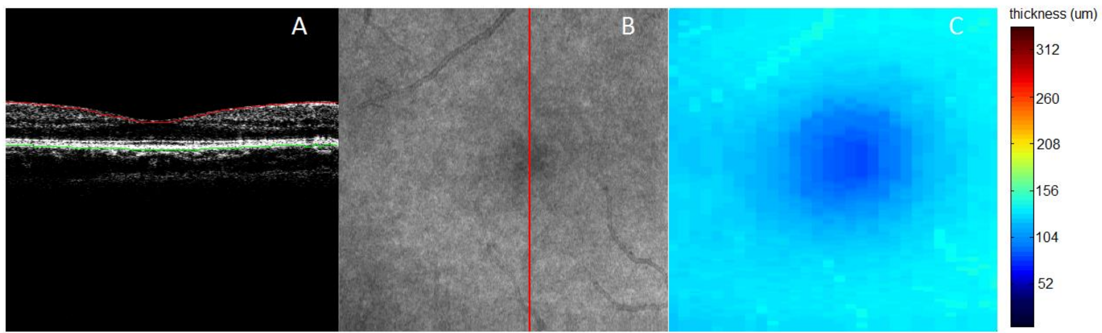


Figure 5-1 Thickness map of the representative healthy volume (SV05)

The red line in Figure 5-1 (A) represents the segmented ILM layer, and the green line represents the segmented RPE layer. With the coordinates of the ILM and BM layers segmented in real-time, the thickness map can be generated based on the difference between the two coordinates. The thickness map of a healthy retina is expected to have a nominally rotationally symmetric thickness from the foveal dip to the surrounding macular tissue, as demonstrated by 5-1(B). In the case of pathological retinas, the rotational symmetry no longer holds when the retinal features are distorted. Such distortion will be reflected on the thickness map. Figure 5-2 is a case of pathological data with Age-related Macular Degeneration (AMD) and Non-Proliferative Diabetic Retinopathy (NPDR).

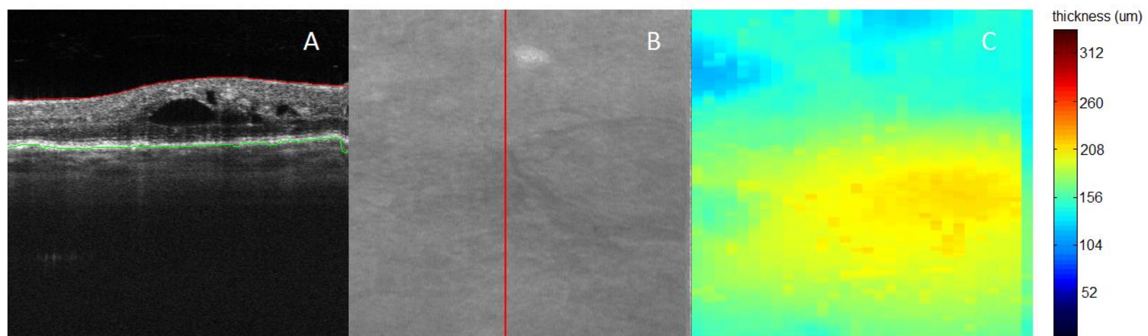


Figure 5-2 Thickness map for a pathological volume with AMD and NPDR (H468)

In Figure 5-2 (A), the pathological distortion underneath the Outer Nuclear Layer (ONL) has created a structural bump. The location of this particular B-scan is near the lower portion of the *en-face* view, which was indicated by the red line in Figure 5-2 (B). As we can see, the thickness map has a corresponding geographical distribution of the thickness matching to the *en-face* view. Please note that this thesis does not particularly target any specific pathology, but rather the structural features that the GRS can work with. The pathological case listed above is for demonstration purposes.

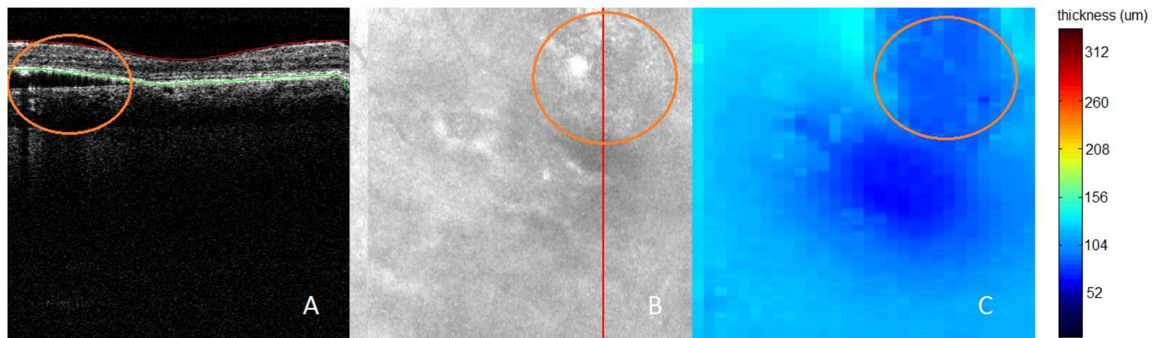


Figure 5-3 Thickness map for a pathological volume with central serous chorioretinopathy (H435)

Figure 5-3 shows us a pathological data with Central Serous Chorioretinopathy. The thickness symmetry only exists in the center portion of the macular region. In the upper-right region, the RPE layer is detached from BM layer due to the presence of the sub-retinal fluid. As a result, the thickness of the region near the sub-retinal fluid is thinned. The ILM layer was detected accurately. On the contrary, the RPE/BM complex are hard to separate since they are the brightest region in the image and mixed together. Thus, what GRS can extract is often the brightest transition from RPE/BM complex to choroid rather than a specific location in the image. In Chapter 4, we discussed how the graph was constructed inversely based the gradient value of the image. The bigger in intensity difference, the easier for the push-relabel algorithm to segment the target layers. As we may observe, the RPE detachment can create an even clearer transition from the bright pixels to the dark ones. As a result, the algorithm chose to follow deformed RPE layers. These results demonstrate that the retinal layer segmentation can unveil pathology through calculating the thickness map. Before we start the section where the

segmentation might fail, let's first review a case for which GRS succeeded and resulted in a false negative. A pathological retina with a lamellar hole is presented in Figure 5-4.

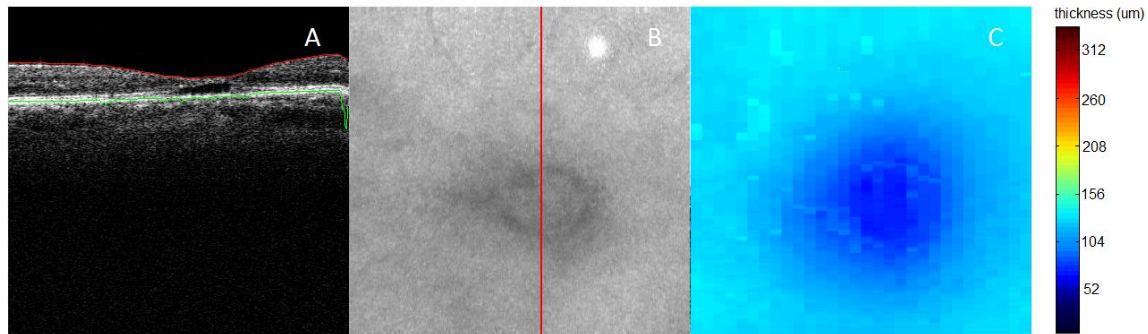


Figure 5-5-4 Thickness map for a pathological volume with lamellar hole (H505)

Despite the fovea detaching from the RPE layer, GRS wasn't affected by the dark region created by the lamellar hole and still successfully detected the BM layer as shown in Figure 5-4 (A). The overall thickness was unaffected. If we were merely judging from the thickness map in Figure 5-4 (C), we might have the false impression that the retina could be healthy if we only look at the thickness map itself, as it shows desired healthy retinal features such that the retina has a rotationally symmetric thickness increase around the fovea dip. The segmentation is functioning as designed because the RPE layer is still in place. Thickness changes caused by pathological features such as lamellar holes cannot be detected by measuring the layer thickness between BM and ILM along. Current measurement to this issue is reliant on the display of the B-scan image, which requires monitoring from clinicians.

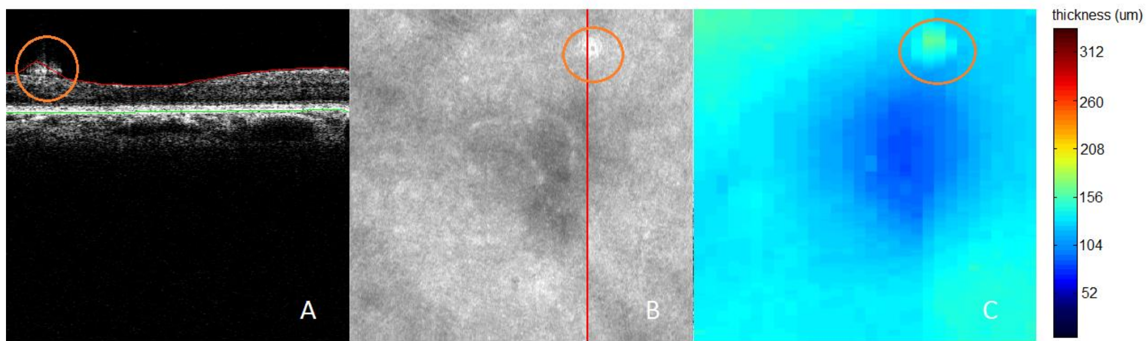


Figure 5-5 Segmentation result corrupted by bright artifacts

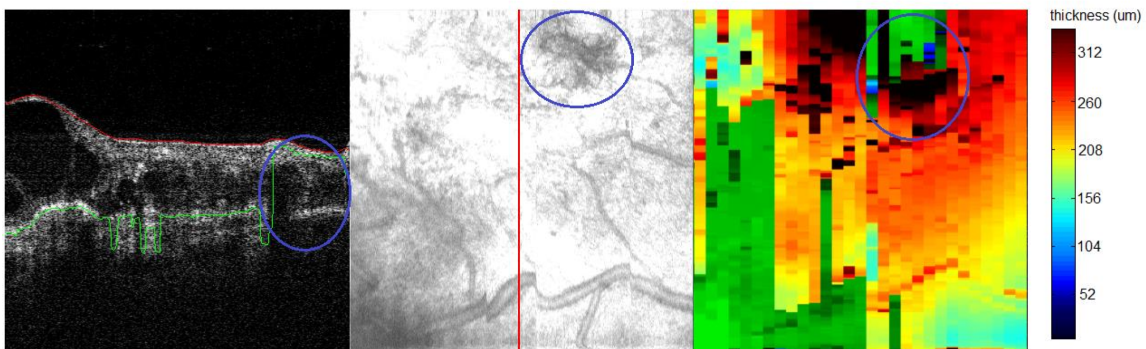


Figure 5-6 Extreme case

Figures 5-5 and 5-6 show scenarios where the segmentation can fail. A healthy retina is presented in Figure 5-5 with additional bright artifacts. The push-relabel algorithm mistakenly recognizes the bright artifacts as foreground. Consequently, GRS delivered an incorrect layer extraction as a result. To conclude, whenever there is unexpected bright artifact presented near the target layer, the segmentation will incorrectly include the bright artifact as part of the feature. Figure 5-6 shows us a retinal tissue volume with significant motion artifact. Technically speaking it is beyond reasonable to expect any fast segmentation algorithm would work with such corrupted anatomical structures, but this particular volume demonstrates the other case where the segmentation can fail. For the region marked by the region denoted by the blue annotations, the GC detected Nerve Fibre Layer (NFL) rather than the RPE layer. This is because the RPE layer was “discontinuous” due to the shadowing artifacts in the OCT B-scan image. Please recall in chapter 4 that a CCL step was introduced to remove the unexpected foreground by noise and artifacts, the discontinued RPE layer was thereupon recognized as artifacts by

CCL and consequently removed. Thus, the final segmentation result was corrupted as a result.

5.2.2. Segmented SV angiography

Based on the previous section, we learned that GRS could reasonably extract the ILM and BM layers for thickness measurement. There is more than one area where GRS can be useful. The segmented coordinate of ILM and BM layers can be further used for enhancing the view of SV angiography. Figure 5-7 (A) is a non-segmented SV angiography result listed as a reference for representative image quality. The same volume but using the segmentation results to extract only the layers of interest is shown in Figure 5-8 against Figure 5-7 for direct comparison.

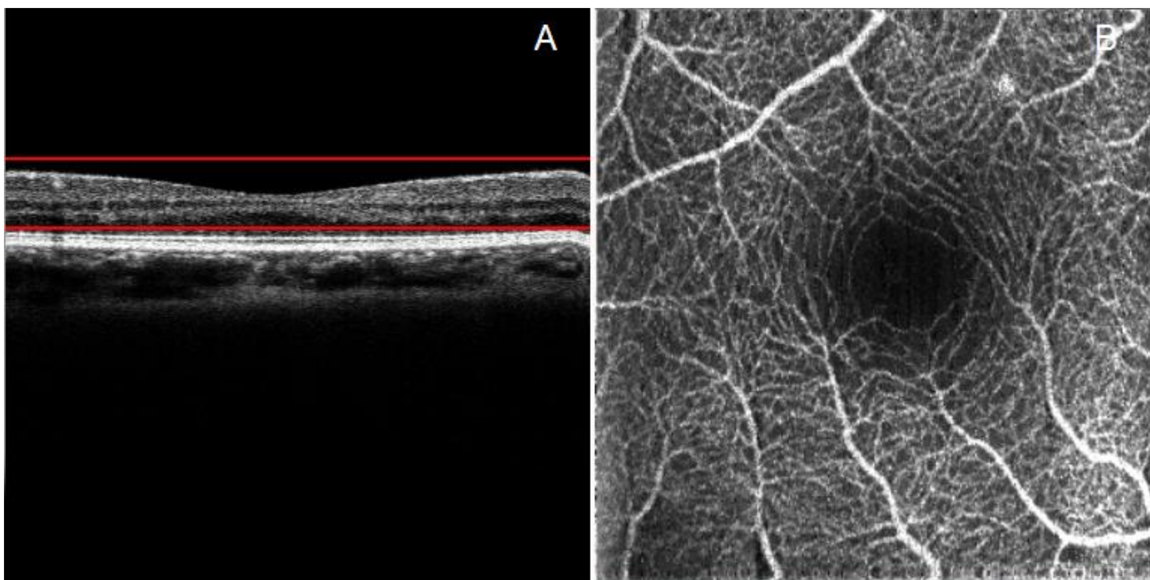


Figure 5-7 Visualization of retinal vasculature network by naive region selection (SV05)

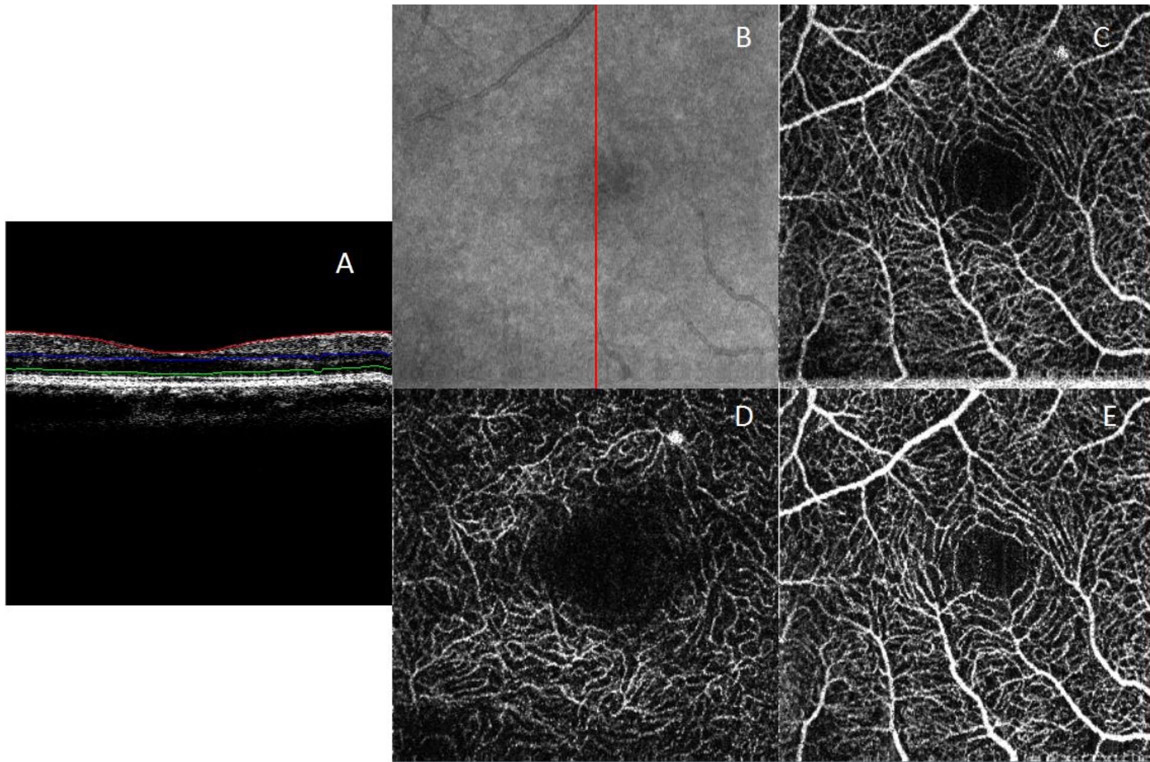


Figure 5-8 Visualization of vasculature network for a representative healthy volume. (SV05)

In Figure 5-7(B), the middle fovea region is relatively dark because of the included extra dark region in the Figure 5-7(A). On the contrary, the result in Figure 5-8(C) does not have this issue. In Figure 5-8, the red line is the extracted ILM layer, and the green line and the blue line that are covering Inner Nuclear Layer (INL) and Outer Nuclear Layer (ONL) are generated by shifting the segmented RPE layer. Figure 5-8(B) is the regular fundus intensity images for reference, 5-8(C) is the primary result that corresponds to the region between red line (ILM) and green line (ONL). 5-8(D) corresponds to the vasculature network of the region between the blue line (INL) and the green line (ONL) whereas 5-8(E) corresponds to the vasculature network of the region between the red line (ILM) and the blue line (INL). The images in panels D and E are additional results to the main blood vessel visualization 5-8(C). They were displayed for quick evaluation of the retinal structure during the real-time imaging session. Figure 5-9 shows the vasculature network of the previously presented retinal volume with a lamellar hole.

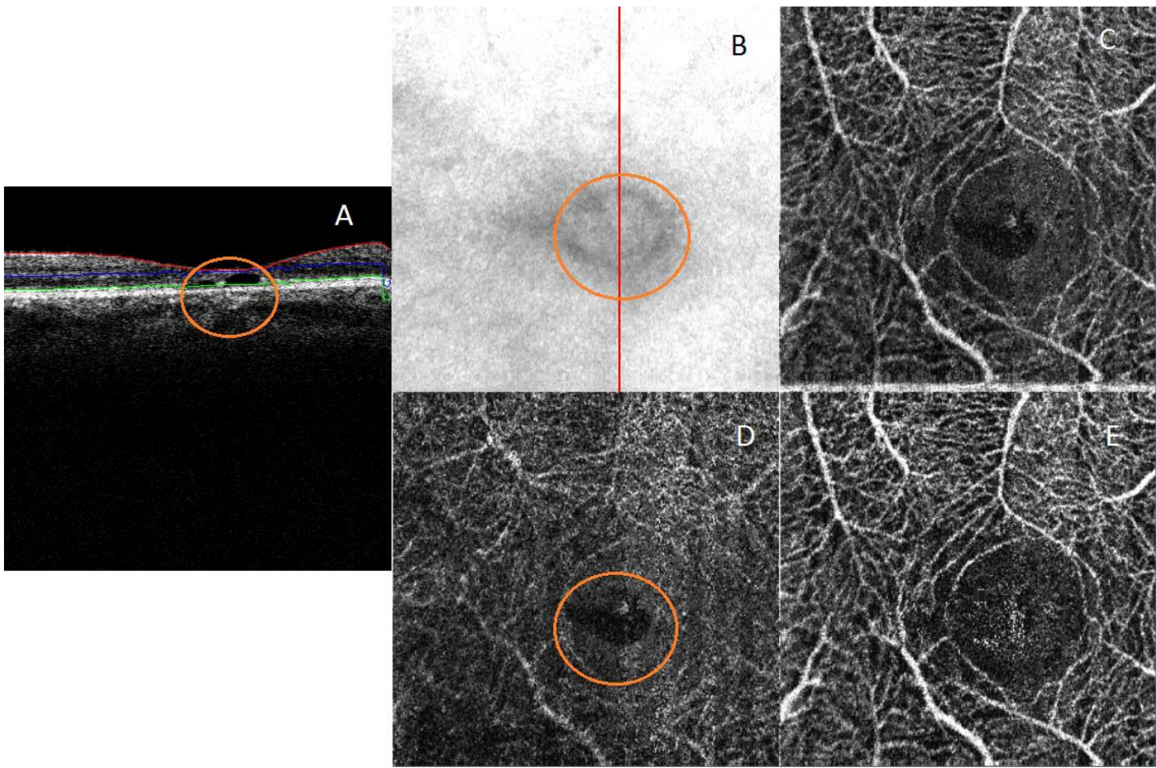


Figure 5-9 Visualization of vasculature network for a pathological volume with lamellar hole (H505)

The RPE layer detection works the same as before, but speckle variance imaging changes the story entirely. In Figure 5-9 (C) and (D) we can clearly observe a black region in the center fovea region. This is because the lamellar hole was created by the detachment of the fovea from the RPE. Hence the dark region would appear around the place where the fovea would be. Figure 5-10 and 5-11 shows the other two pathological volumes we reviewed in the thickness measurement section.

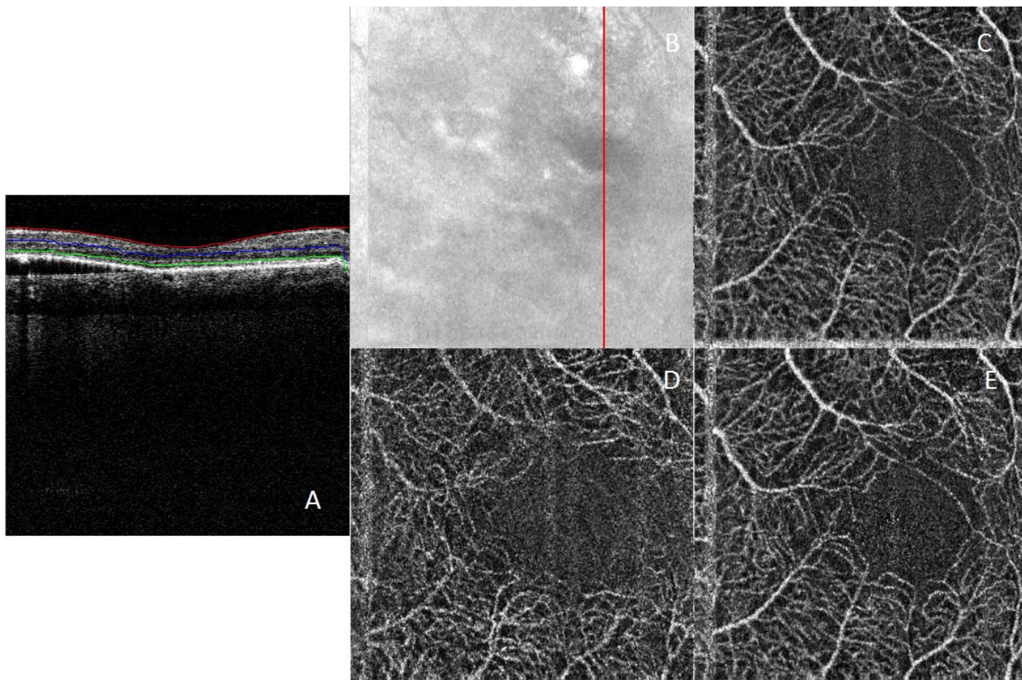


Figure 5-10 Visualization of vasculature network for a pathological volume with central serous chorioretinopathy (H435)

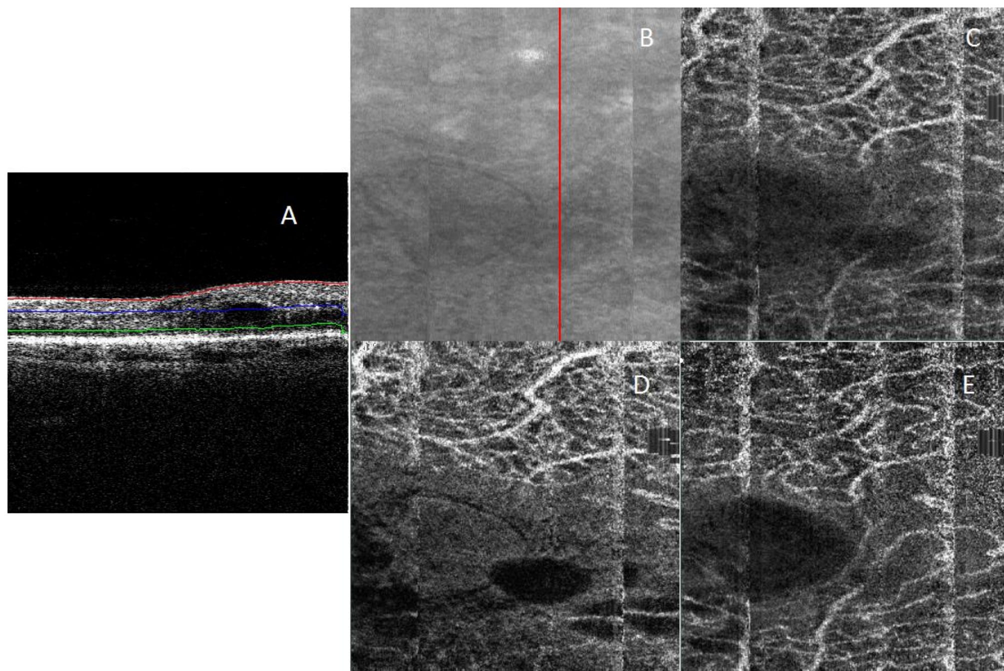


Figure 5-11 Visualization of vasculature network for a pathological volume with AMD and NPDR (H468)

Figure 5-10 (D) displays the vasculature network between the OPL and INL without being affected by the RPE detachment, thanks to the segmented result following RPE layers. The white stripes observed in 5-10 (B) and (C) are motion artifacts that happen to be more pronounced in the lower portion of the vasculature network; these artifacts are common in current state-of-the-art OCT Angiography. Figure 5-11 shows a vasculature network *en-face* images for the AMD-NPDR volume reviewed in the thickness section. Note that in case where a retinal hole was created above the OPL layer, shifting ILM layer for corresponding vasculature visualization would be a better approach since the shape of INL is more close to ILM in comparison.

5.3. Limitations

Currently, the PR GC implementation by NVIDIA can be only applied on the macular region. The PR GC does not yield correct results when a discontinuity of the target layer is present, as is the case with images of the optic nerve head. The current GPU implementation is not fast enough for intra-retinal layer segmentation. Since the real-time constraint only permits us running segmentation pipeline once for each image, intra-retinal layer segmentation is consequently not achievable as it would require iterating the segmentation pipeline multiple times for multi-level of retinal layers.

5.4. Speed of the GPU retinal layer segmentation

The usability of the real-time GC retinal layer segmentation application for both thickness measurement and the segmented vasculature network visualization is subject to the run time of the pipeline. Figure 5-12 shows a representative timeline for the entire segmented SV angiography process on a single GPU. Recall that the SS-svOCT system in VGH takes roughly 1.575 seconds for a data volume with 30 batches. The average acquisition time for one batch would be around 52.5 ms. For a single GPU implementation, the retina would be required to have ideal SNR and a certain shape that suits the segmentation algorithm in order to run in real-time.

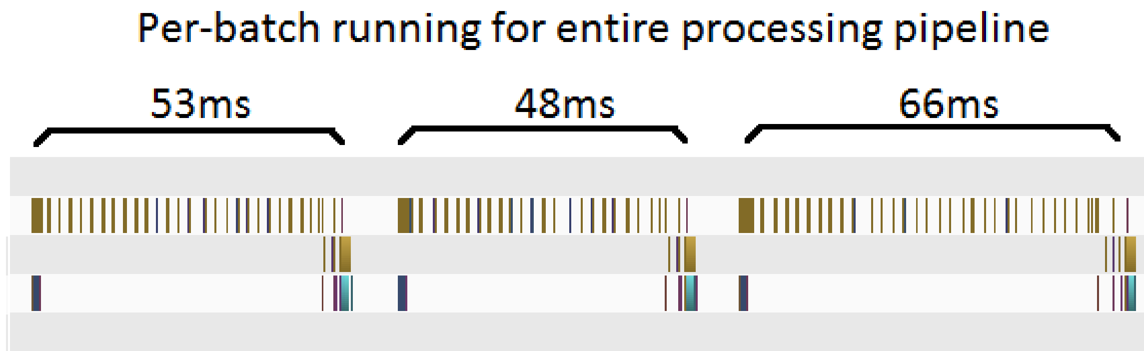


Figure 5-12 Representative per-batch timeline for entire processing pipeline captured by NVIDIA Visual Profiler

As we can see, although one batch successfully achieved under 52.5ms requirement, the rest of batches couldn't meet the required timeline. Please note that the hardware used for conducting this research were outdated and couldn't deliver the up-to-date hardware raw performance. This 52.5ms requirement would most likely be fulfilled if a state-of-the-art machinery was used. Regardless, our goal was to meet the real-time requirement even if using the out-dated machine we currently have. Please recall the parallel approach we discussed in chapter 4, where more than one GPU was used to ease the burden of real-time constraint. In this case, we have a PC with two dedicated GPUs for computation; this setup effectively relaxes the time requirement from 52.5ms to 105ms since two GPUs will concurrently be working together to process the data volume. As a result, the 105 ms easily becomes an achievable number. The healthy volume SV05 is used as the representative dataset for benchmarking the processing timeline. Figure 5-13 shows the batch-level processing timeline for GPU graph-cut vs segmentation pipeline vs entire processing pipeline for thickness map calculation.

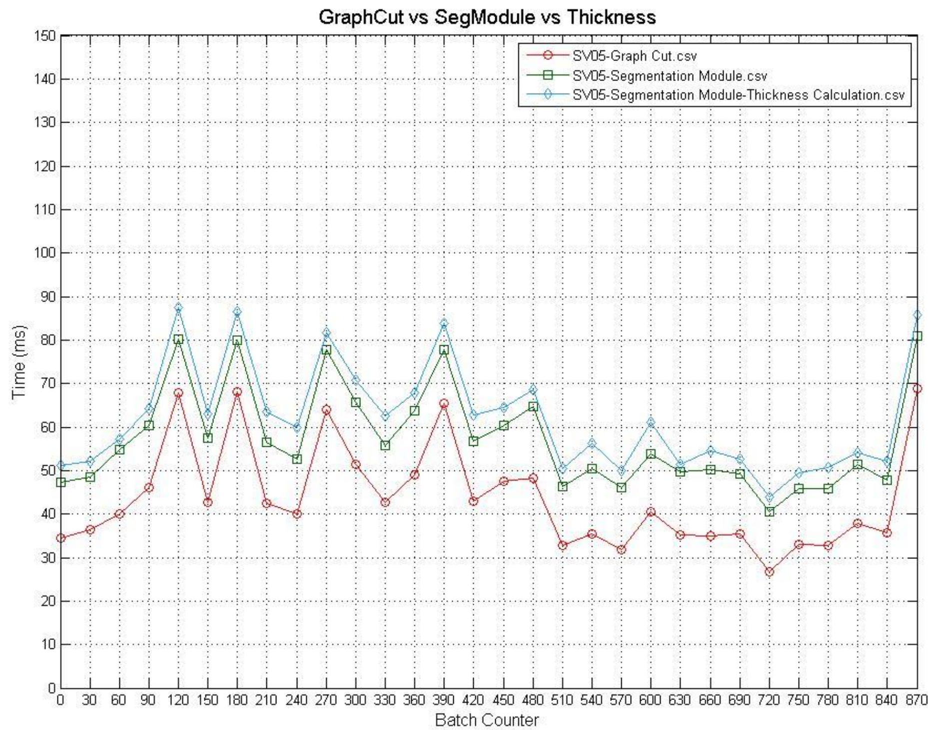


Figure 5-13 GC vs Segmentation pipeline vs SSVA (SV05)

All numerical benchmarking results presented in this chapter are obtained through averaging 20 distinct measurements. This way we can minimize the fluctuation caused by regulation of the data traffic. As indicated in the graph, the processing timeline for both the segmentation pipeline (named as segmentation module in the above plot) and pipeline for thickness map calculation are almost vertically offset versions of the timeline for GPU GC. This is because the additional calculation for thickness map generation is simply parallelized matrix subtractions and geographical pixel mapping. Most of the time consumption was taken by memory transfers. The computational cost of the subtraction and thickness map generation are actually quite low by themselves. On the other hand, the performance gap between GPU GC and entire segmentation pipeline were mostly taken by CCL (marked as Connected-Component and Analysis in the following figures) and partially memory transfer. Figure 5-14 shows the processing timeline for CCL vs Segmentation pipeline vs Segmented SV angiography pipeline (SSVA). Table 5-2 shows the numerical result of the processing timeline for each individual stage of CCL (named

as CCA in the plot), segmentation pipeline and SSVA (named as Segmentation Module Speckle-Variance).

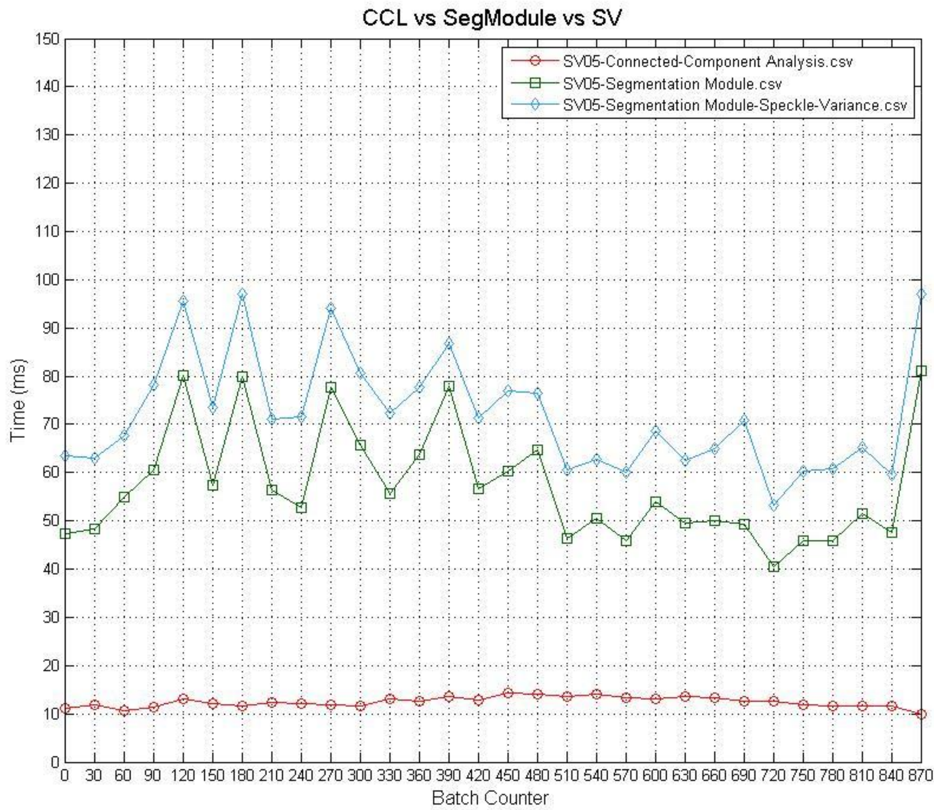


Figure 5-14 CCL vs Segmentation pipeline vs SSVA (SV05)

Table 5-2 Profiling result (SV05)

	Graph-Cut (ms)	CCL (ms)	Segmentation pipeline (ms)	Thickness map (ms)	SSVA (ms)
Volume-level mean	655.24	186.74	858.99	929.98	1081.76
Batch-level mean	43.68	12.45	57.26	62.00	72.81

In Figure 5-14, we may observe that the processing time for GC and CCL does not add up to the entire processing time of the segmentation pipeline. The time difference was caused by the additional memory transfer between host and device. The SSVA takes the longest processing time due to the additional SV processing steps. Note that the overall plot trend of SSVA and the segmentation pipeline are largely the same. This is because the time complexity for SV processing is constant. The time fluctuations were caused by the GPU data traffic. For the healthy volume SV05, the processing time for both thickness map calculation and SSVA are quite satisfyingly below the required real-time constraint of 105ms. However, retinas with complex features such as volumes with RPE detachment (H435) could oppose extra obstacles for the algorithm to process. By intuition, the pathological volume H435 should take longer time to process than the healthy volume SV05. Figure 5-15 shows a batch level result for CCL vs Segmentation pipeline vs SSVA for H435.

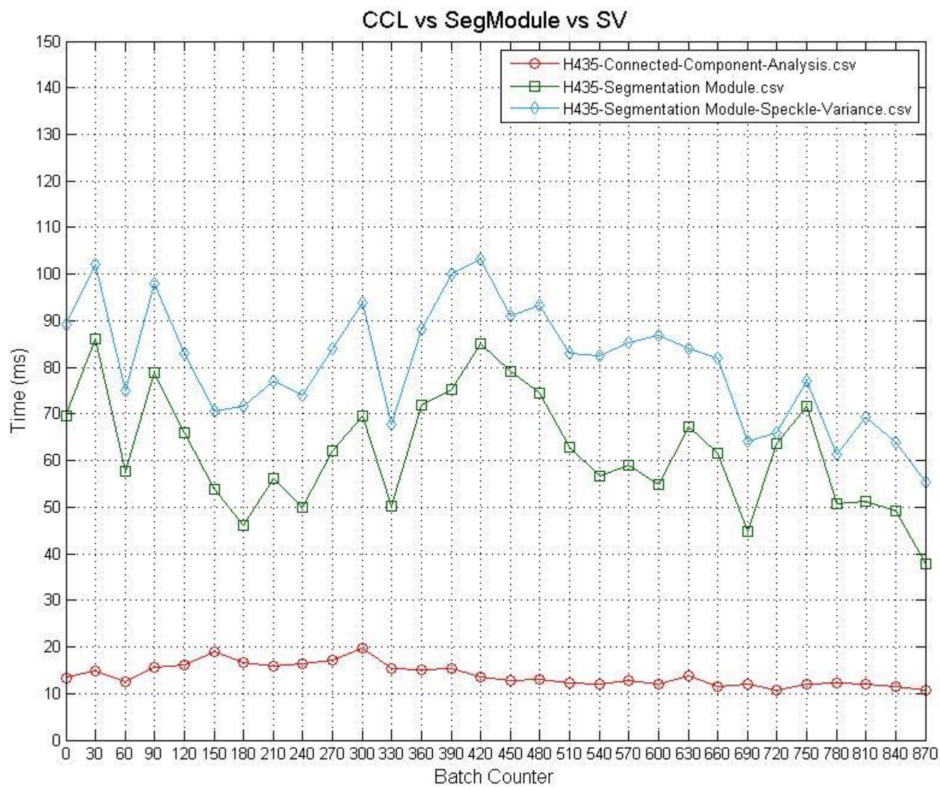


Figure 5-15 CCL vs Segmentation pipeline vs SSVA (H435)

Table 5-3 Profiling result (H435)

	Graph-Cut (ms)	CCL (ms)	Segmentation pipeline (ms)	Thickness map (ms)	SSVA (ms)
Volume-level mean	700.68	209.10	931.11	1026.45	1211.28
Batch-level mean	46.71	13.94	62.07	69.43	80.75
GPU0 total	701.86	211.26	935.82	1031.00	1213.32
GPU0 average	46.79	14.08	62.39	69.79	80.89

GPU1 total	699.49	206.94	926.41	1021.00	1209.25
GPU1 average	46.63	13.79	61.76	69.07	80.61

As anticipated, the pathological retina volume clearly takes longer time to process, with few entries that are very close to the processing deadline for an individual batch. The additional processing difficulties are also reflected by the increase in GC and CCL processing time in comparison to the healthy volume. Note that the total time is calculated as the mean value between the two GPUs since they are operating concurrently. However, it would be an incorrect assessment to use mean time between devices for the real-time deadline. In this volume we see few occurrences where the processing times are close to the deadline. Let's now assume that the mean value met the deadline and GPU1 also met the deadline, but GPU0 did not. This would still result in a failure in passing the real-time requirement. Therefore, taking the slower GPU would be a better assessment for real-time deadline. The performance discrepancy is caused by thermal throttling. There was limited space to install the three GPUs in the testing computer. One of the computational GPUs didn't have as much 'breathing space' (room for air flow and cooling) as the other one, resulting in thermal throttling and a drop in performance.

To conclude the performance of GRS, 9 retina volumes are randomly selected for SSVA benchmarking as listed in table 5-4. Note that for some specific volumes, the batches that were assigned to GPU1 could be more challenging to process than the batches that were assigned GPU0. Consequently, for volume H476, GPU1 took a longer time to process the assigned batches than GPU0, even though GPU0 was technically slower due to thermal throttling. The averaged processing time for SSVA per batch is 71.86 ms, which is 46.12% faster than the required deadline. Therefore, it is clear that GRS is fast enough for real-time retinal segmentation for both thickness map measurement and SSVA.

Table 5-4 Profilling summary

Volume Names	Mean total/per-batch (ms)		GPU0 total/per-batch (ms)		GPU1 total/per-batch (ms)	
Segmented SV Angiography timeline for x900 volume – 30 batches (deadline 1575 ms)						
H435-P	1211.28	80.75	1213.32	80.89	1209.25	80.61
H505-P	1148.49	76.56	1149.82	76.65	1147.15	76.47
SV01-H	1084.95	72.33	1110.32	74.02	1059.58	70.64
SV05-H	1081.76	72.11	1092.19	72.81	1071.33	71.42
SV42-P	969.78	64.65	976.11	65.07	963.43	64.22
Segmented SV Angiography timeline for x900 volume – 20 batches (deadline 1050 ms)						
H476-P	644.57	64.46	608.70	60.87	680.44	68.04
SV40-P	679.17	67.91	689.11	68.91	669.24	66.92
SV41-P	711.76	71.17	745.17	74.52	678.35	67.83
SV43-P	698.64	69.86	730.37	73.04	666.92	66.69
Average per batch timeline for Segmented SV Angiography (deadline 105 ms)						
Average		71.09		71.86		70.32

***With H stands for healthy, and P stands for Pathological**

5.5. Performance due to contrast to noise ratio

In the previous section, it was shown that the complexity of the retinal image can affect the processing time for GRS. Generally speaking, the Signal to Noise Ratio (SNR) would be the first thing to consider as the factor that could affect the performance. In terms of image processing, SNR could translate to image clarity or Contrast to Noise Ratio (CNR).

In this section, we will be investigating how the performance of the PR GC can be improved by adjusting the histogram to its optimal level. Two histogram range scaling factors are available in the OCT processing software for manually adjusting the histogram of the image, with one scaling factor responsible for adjusting the lower bound of the histogram and the other for higher bound. Again, we will be using the healthy retinal volume SV05 for benchmarking for CNR. Four instances of the retinal data and the GC segmentation with visually different CNRs were arranged as shown in Figure 5-16 by adjusting the values of the two scaling factors.

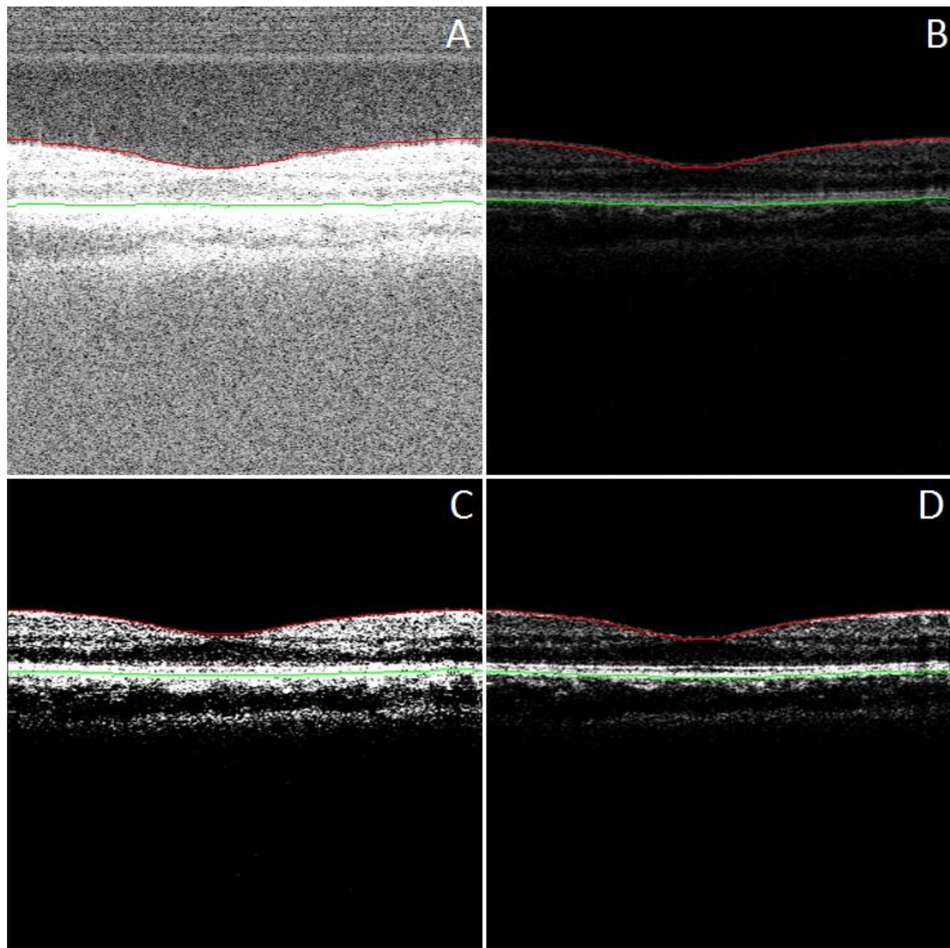


Figure 5-16 Images with different histogram scaling factors: (A) min: 7.5, max: 10.5 (B) min: 10, max: 16.5 (C) min: 10.5, max: 11 (D) min: 10.5, max: 12.5

Based on human observation, Figure 5-16 (A) is the noisiest with the lowest CNR, thus (A) is expected to have the slowest processing time. Figure 5-16 (C) visually has the highest contrast thus intuitively is expected to have the fastest processing time. However, recall that the graph was constructed from the gradient image, not the image itself, thus setting the contrast too high may not result in a more workable gradient image for constructing the graph. Figure 5-16 (D) visually has the most balanced contrast thus it is expected to have the best running time. Figure 5-16 (B) definitely has a better contrast than (A) and will most likely have a better workable gradient image for graph construction, but it is unclear whether it is better than (C) for GRS. Figure 5-17 is the volume-level running time comparison for segmentation pipeline between four different histogram scaling factor settings

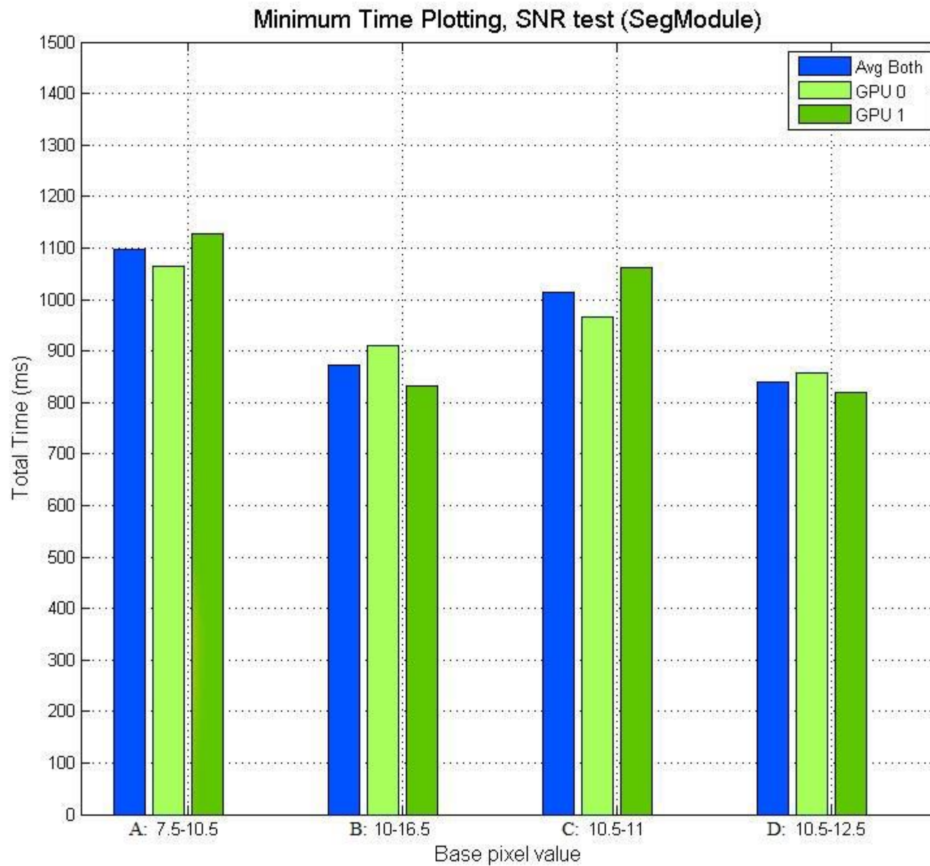


Figure 5-17 Running time comparison for segmentation pipeline between four different histogram range scaling factor settings

As anticipated, setting (A) has the worst run time whereas setting (D) has the best among the four. Apparently, setting (C) has too narrow dynamic range to generate a good gradient image for graph construction, ranking it the second worst. Lastly, setting (B) has the second best performance although it is visually dark to human eyes. Another two tests were conducted to find out the optimal intensity range of GRS for this particular volume: (i) different histogram scaling range with the same lower bound; (ii) different histogram scaling level but with the same range. Figure 6.18 shows the volume-level run time comparison (for segmentation pipeline) between 6 settings of different dynamic ranges with the same lower bound scaling factor.

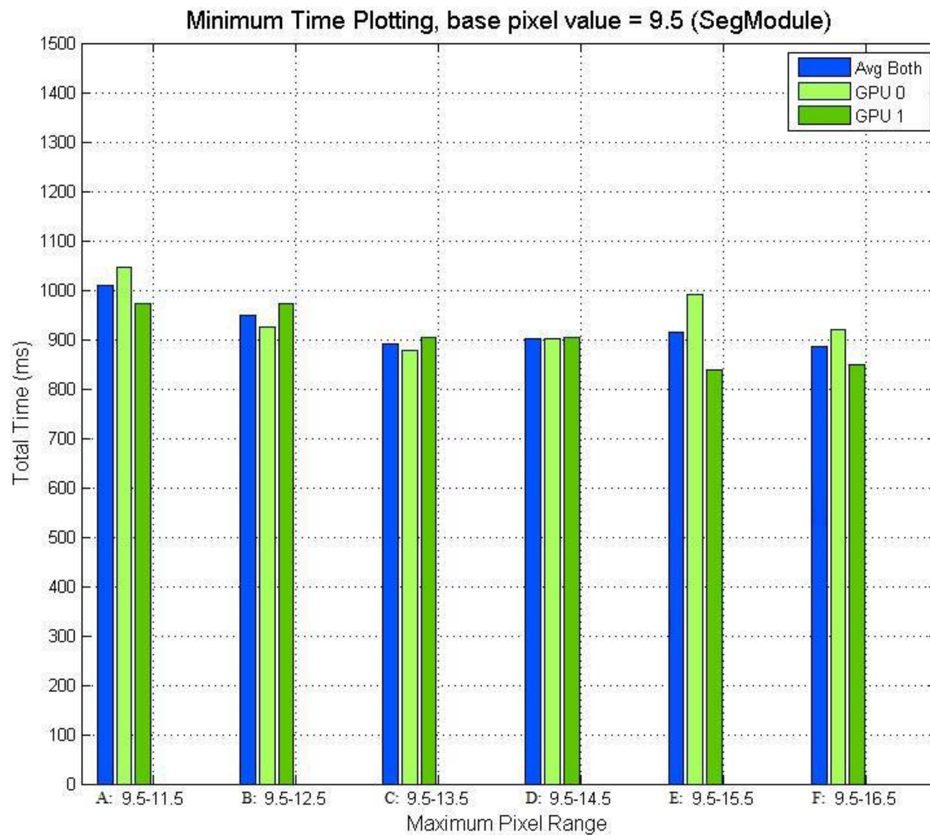


Figure 5-18 Running time comparison for segmentation pipeline between 6 settings of different histogram scaling range

Settings (C) and (D) provide relatively better run time with setting (C) slightly better. Since (F) yields relatively similar results to (C), it would seem the result could be better as we increase the dynamic range. However, the GRS starts delivering false results after the higher bound scaling factor goes beyond 16.5 as the dim image does not provide clear enough gradient image for constructing a graph. Figure 5-19 shows the volume-level running time comparison between 6 settings of different scaling factors but same dynamic range.

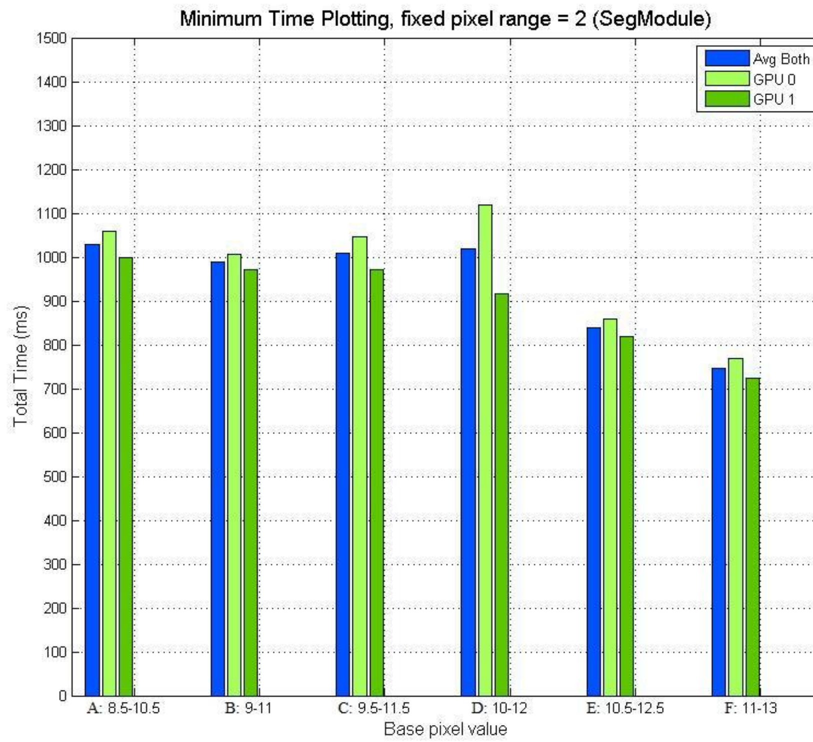


Figure 5-19 Running time comparison for segmentation pipeline between 6 settings of different scaling factor levels with the same range

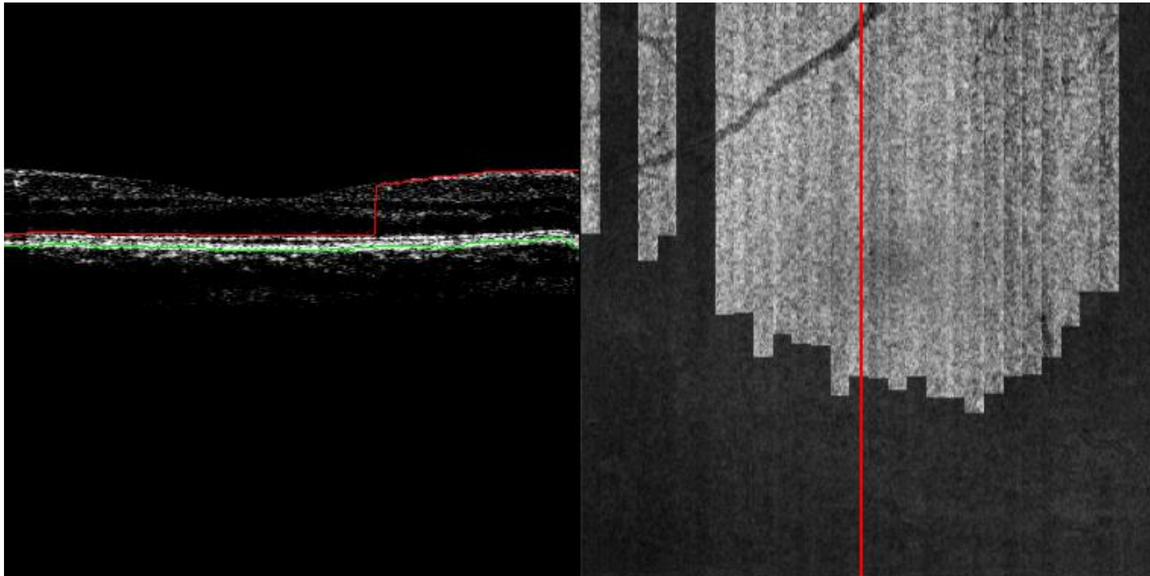


Figure 5-20 False segmentation result in the histogram scaling setting (F: 11-13)

Again, we observe a similar scenario that the running time seems to get better as the scaling factor increases. In spite of setting (F) has the best performance, the segmentation result is incorrect. Thus, the best performing setting is still (E:10.5-12.5). Based on figure 5-16 ~ 5-20, we can conclude that the optimal intensity/contrast range for GRS lies around the setting where the scaling factors are 10.5 to 12.5 for this particular volume. The scaling factors are created for adjusting the histogram of the image. Thus, applying a histogram adjustment technique could bring the time aspect of the performance to its most optimal level.

5.6. Final evaluation of GRS against the CPU implementation

The performance of the GRS was compared against a CPU implementation of the graph cut segmentation. The CPU MATLAB segmentation was implemented with shortest path and dynamic programming by Chiu *et al* [**31**]. However, Chiu's implementation requires a narrower band of the image intensity histogram in order to have the segmentation function correctly. The optimal range that was used for GRS will yield a false result for Chiu's implementation. The input image size was the same as presented in the previous sections, 512 width x 300 height. The test bench for Chiu's implementation was a different computer with 4.0 GHz i5-4670K processor. Since Chiu's implementation is purely done with CPU and MATLAB, the rest of the system specification are irrelevant. Figure 5-21 show the result of Chiu's implementation.

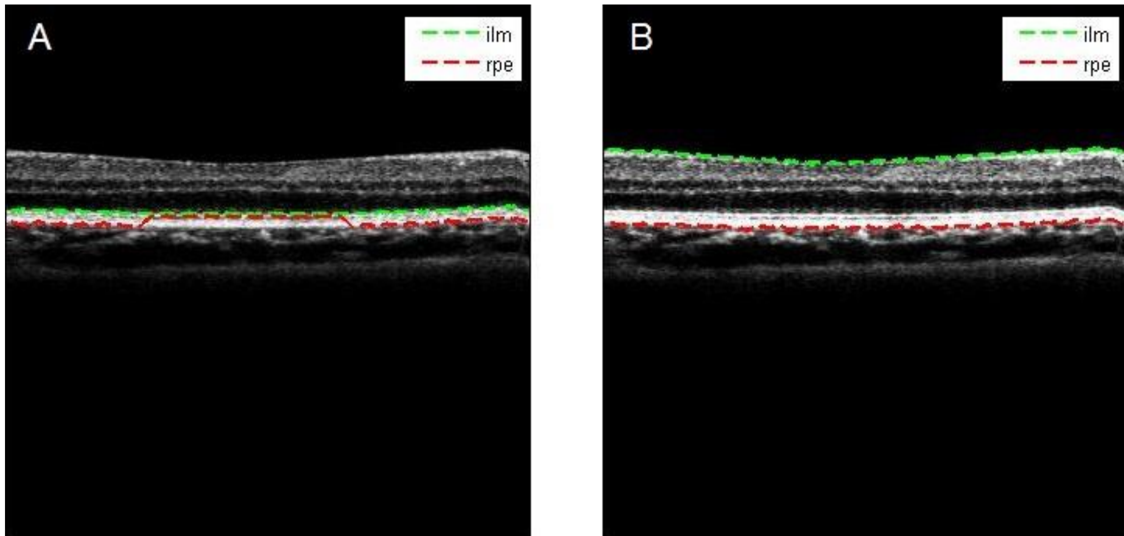


Figure 5-21 Result of Chiu's implementation with our input A) GRS optimal histogram B) Histogram adjusted for Chiu's algorithm

Table 5-5 Representative time comparison between GPU and CPU

	Chiu's implementation	GRS	GPU/CPU speed
Processing time for segmentation alone	60231.22 ms	43.68 ms	x1424.71 faster

Both of GPU and CPU timing are the average result measured over 20 repetitions. The quality of the GRS has been discussed in previous sections. The focus here is the speed improvement of GPU over CPU. The result is clear, the GPU implementation is more than a thousand times faster than the CPU implementation in MATLAB.. A more fair comparison of the speedup would be to compare the GPU performance in code developed and optimized in a low level language, such as C/C++. However, even in that case, a significant speedup is still expected based on reports comparing CPU to GPU implementations of graph cut segmentation [**25**].

There are a few things worth noting when it comes to using GPU for software implementation. The performance of the OCT imaging software relies on the following three main attributes of the GPU: core clock frequency, numbers of CUDA cores and VRAM capacity. For best performance, we want the GPU to have a highest possible clock frequency and high possible amount of CUDA cores. For VRAM, the OCT imaging software has a VRAM usage at a steady level of ~600MB. However, in current consumer GPU lineup , the high-end GPU that has high clock frequency and large number of CUDA cores often come with more than 6GB of VRAM. Although 6GB VRAM is a overkill for our application, the high-end GPUs such as GTX 1070 and GTX 1080 are still recommended due to their superior clock frequency and high numbers of CUDA cores.

5.7. Summary

In this chapter, the results for GPU retinal layer segmentation on both thickness measurement and speckle variance angiography has been evaluated qualitatively and quantitatively. The GRS has proven to be a reliable tool for extracting ILM and RPE/BM complex under usual circumstances. The 71.86ms on average processing time is 46.12% faster comparing to the required processing deadline, which is determined by the acquisition speed of the OCT system. The resulting processing speed is sufficient to meet the requirement of real-time visualization. Currently, the PR GC implementation by NVIDIA can be only applied on the macular region. The PR GC does not yield correct results when a discontinuity of the target layer is present, such as in B-scan images of the optic nerve head. Also, automated intra-retinal layer segmentation is not achievable without running the algorithm multiple times, of which would then make the real-time aspect unachievable with the current configuration and hardware.

Chapter 6.

Conclusion and Future Work

In this thesis, we reviewed the GPU-based implementation for retinal layer segmentation. The result of the GRS is that it is capable of tracking and segmenting the ILM and BM within a reasonable tolerance of 6~7 microns yet also able to meet the real-time processing requirement. The running time of entire visualization cycle surpasses the target deadline for 200kHz swept laser source SS-svOCT system by 46% on average when running on outdated and non-optimized computer and GPUs, indicating that the performance can be boosted even higher with up-to-date and optimized hardware. Applying GRS along with intra-operative FD-OCT imaging could potentially provide the ability for immediate feedback of instrument-tissue interaction to surgeons. Successful integration of such technology to the FD-OCT systems at a hospital based ophthalmic center could have great impact on the success rate of membrane peeling and related procedures and improve the surgical outcomes for patients.

6.1. Future Work

While the performance of the GRS demonstrated in chapter 5 shows its great potential for intra-operative FD-OCT applications, there is also room for improvements in the processing steps. A parallel version of CCL is possible to implement, but difficult. Due to time constraints, parallel version CCL was not implemented for this project. Also, it has been mentioned several times that the hardware systems were not up-to-date and the hardware settings were improper. Such setting downgrades the performance of the GRS by a significant amount. In the following subsections, we will go over these areas for further improvement of the thesis implementation.

6.1.1. Software and Algorithms

Currently, the OCT processing pipeline for the testing performance lacks some functionalities such as software dispersion compensation. The dispersion compensated image has a better image clarity compared to non-compensated images. Having a higher image clarity equally translates to higher SNR when it comes to image processing. Thus, the performance of GRS can be further improved if the input image was dispersion compensated.

Although CCL is indeed difficult to parallelize due to its sequential nature, there has been a successful implementation of CCL documented in the book GPU GEMs [**25**]. However, the parallelized CCL was implemented for a different application and the source code was no longer available from the publisher. Due to the time constraint and implementation difficulty, such parallelized CCL was not implemented for this thesis. Given enough resources and time, CCL can be parallelized and used to eliminate the only CPU reliance of the GRS.

Lastly, the visualization of the display is implemented using OpenGL™ display. OpenGL is a low-level display API for computer graphics developers. There is one problem with the OpenGL library: it does not fully utilize the multi-core capabilities of CPUs. One core is responsible for most of the work done when running an OpenGL application [**32**]. Recently there has been a newly announced low-level graphics API called Vulkan™, which greatly improves the utilization of the multi-core CPU. Although using the Vulkan library would not contribute to the run time of image processing, it would greatly improve the display process so that the final visualization wouldn't possibly be bottlenecked by the front-end display.

6.1.2. Hardware environment

The computer hardware used for the performance testing of the GRS is two-to-three generations behind. Also, the imaging software, OCTViewer, was not optimized for heavy-multi-threading processing with the CPU. This was due to two reasons: low utilization of multi-core CPU capability of OpenGL and the use of sequential algorithms. The representative utilization rate of the CPU cores when running the OCTViewer was:

100% for only one core, and around 40-60% % for 4 cores with rest of the cores completely idle etc.

Thus, when integrating the GRS with a clinical system, the ideal CPU would be an up-to-date consumer grade CPU with faster single core performance, rather than the server grade CPU with many low clock-speed cores. Also, the GPU we used in this thesis is not designed for the medical imaging system that is running at ECC. The strength of the Quadro K6000 is the ability for double precision computation. However, the OCTViewer wasn't written in double precision format, and thus this capability was wasted. The reason why these GPUs were used was because of the exceptional VRAM capacity. But, with the recent announcement of NVIDIA Pascal GPUs [**23**], the 12GB VRAM can now be matched by consumer grade GPUs. Moreover, the Quadro devices were purposely down-clocked compared to their consumer counters parts for reliability reasons. Last but not the least, both Quadro devices and Xeon CPU are server grade products, which cost significantly more than their consumer counterparts, yet providing sub-par performance as they are not designed for these types of applications. An ideal hardware configuration is listed in Table 6-1.

Table 6-1 Testing bench vs ideal system

	Used in test bench			Optimized system		
	Model	Speed	Price(USD)	Model	Speed	Price(USD)
CPU	2 x E2-2620	2.3GHz	2 x 382	i7-6850K	3.8GHz	617
GPU	2x Quadro K6000	900MHz	2 x 5,999	2 x GTX 1080	~1700MHz	2 x 599
Price difference: 10,947 USD, rest of the system can be the same						

Note that not only are the price differences significant, but the performance gap between the two hardware configurations are significant as well. According to CPU benchmarking tool PassMark™, the single core capability of i7-6850K is 70% faster than that of the E2-2620 [**33**]. Also, according to NVIDIA, the theoretical performance of the latest Pascal architecture is 2.35x times than that of the Kepler architecture [**23**]. In all, even without the parallelized CCL and the use of Vulkan API, with proper up-to-date computer

hardware, the performance of GRS can be potentially improved by at least two fold, reaching the real-time speed that may even satisfy a 400 kHz laser source system.

With the hardware configuration used in this thesis, the intra-retinal layer segmentation was not achievable in a real-time constraint. If the hardware update could truly bring the performance boost as suggested above, the intra-retinal layer segmentation would be possible by running PR GC multiple times thanks to the performance improvements of the advanced hardware. With the rapid increase in computer performance, the implementation of GRS for novel applications in intra-surgical OCT is anticipated to grow.

References

[1] CNIB. Fast Facts about Vision Loss. [Online]. http://www.cnib.ca/en/about/media/vision-loss/Pages/default.aspx

[2] M.Ohr, P.Kaiser et al. J.Ehlers, "Novel microarchitectural dynamics in rhegmatogenous retinal detachments identified with intraoperative optical coherence tomography.," *Retina*, vol. 33, pp. 1428-1434, 2013.

[3] C.Baumal, C.Puliafito et al. M.Hee, "Optical coherence tomography of age-related macular degeneration and choroidal neovascularization," *Ophthalmology*, vol. 103, no. 8, pp. 1260-70, 1996.

[4] J.P. Ehlers, "Intraoperative optical coherence tomography: past, present and future," *Eye*, vol. 30, no. 2, pp. 193-201, 2016.

[5] Y.K. Tao, S. Farsiu et al. J.P. Ehlers, "Integration of a spectral domain optical coherence tomography system into a surgical microscope for intraoperative imaging," *Investigative Ophthalmology and Visual Science*, vol. 52, no. 6, pp. 3153-3159, 2011.

[6] E. Swanson, C. Lin et al. D. Huang, *Optical Coherence Tomography*.: Science, 1991, vol. 254.

[7] M. Sarunic, C. Yang et al. M. Choma, "Sensitivity advantage of swept source and Fourier domain optical coherence tomography.," *Optics express*, vol. 11, no. 18, pp. 2183-2189, 2003.

[8] K. Wong, M. Sarunic Y. Jian, "GPU Accelerated OCT Processing at MegaHertz Axial Scan Rate and High Resolution Video Rate Volumetric Rendering," *Journal of Biomedical Optics*, vol. 18, no. 2, p. 26002, 2013.

[9] S. Han, C. Balaratnasingam, Z. Mammo, K. Wong, S. Lee, M. Cua, M. Young, A. Kirler, D. Albiani, F. Forooghian, P. Mackenzie, A. Merkur, D. Yu, M.V. Sarunic J. Xu, "Retinal angiography with real-time speckle variance optical coherence tomography," *British Journal of Ophthalmology*, pp. 1-5, 2015.

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

[11] T. Falch, M. Bozorgi et al. E. Smistad, "Medical image segmentation on GPUs - A comprehensive review.," *Medical image analysis*, vol. 20, no. 1, pp. 1-18, 2014.

[12] NVIDIA. (2016) CUDA toolkit documentation. [Online]. http://docs.nvidia.com/cuda/cuda-samples/index.html#grabcut-with-npp

[13] InetDaemon. CPU - Central Processing Unit. [Online]. http://www.inetdaemon.com/tutorials/computers/hardware/cpu/

[14] Intel. (2016, May) 6th Generation Intel® Processor Family. [Online]. http://www.intel.com/content/www/us/en/processors/core/desktop-6th-gen-core-family-spec-update.html

[15] T. Jamil R. Stacpoole, "Cache memories," *IEE Potentials*, vol. 19, no. 2, pp. 24-29, 2000.

[16] Matt Bach. (2015, August) Haswell vs. Skylake-S: i7 4790K vs i7 6700K. [Online]. https://www.pugetsystems.com/labs/articles/Haswell-vs-Skylake-S-i7-4790K-vs-i7-6700K-641/#CPUPerformance-UnigineHeavenPro4_0

[17] Kevin Krewell. (2009, December) What's the Difference Between a CPU and a GPU? [Online]. https://blogs.nvidia.com/blog/2009/12/16/whats-the-difference-between-a-cpu-and-a-gpu/

[18] Nicholas Wilt, *THE CUDA HANDBOOK - A Comprehensive Guide to GPU Programming*.: Addison-Wesley, 2013.

[19] Y. Wang, Y. Ha, K.M.M. Aung Y. Chen, SAES: A high throughput and low latency secure cloud storage with pipelined DMA based PCIe interface, 2013, IEEE eXplore conference.

[20] pCI-sIG. (2010) PCI Express Base Specification Revision 3.0. [Online]. http://composter.com.ua/documents/PCI_Express_Base_Specification_Revision_3.0.pdf

[21] JEDEC. Main Memory: DDR3 & DD4 SDRAM. [Online]. http://www.jedec.org/category/technology-focus-area/main-memory-ddr3-ddr4-sdram

[22] NVIDIA, *CUDA C Programming Guide.*: Changes, 2014.

[23] NVIDIA. (2016) GP100 Pascal Whitepaper. [Online]. https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf

[24] W. Hwu D. Kirk, *Programming Massively Parallel Processors: A Hands-on Approach.*, 2010.

[25] NVIDIA, *GPU Gems.*: Pearson Education.

[26] E. Otoo, K. Suzuki K. Wu, "Optimizing two-pass connected-component labeling algorithms," *Pattern Analysis and Application*, vol. 12, no. 2, pp. 117-135, 2009.

[27] K, Wong, Y. Jian et al J.Xu, "Real-time acquisition and display of flow contrast using speckle variance optical coherence tomography in a graphics processing unit.," *Journel of Biomedical Optics*, vol. 19, no. 2, p. 026001, 2014.

[28] R. Manduchi C. Tomasi, "Bilateral Filtering for Gray and Color Images," *International Conference on Computer Vision*, pp. 839-846, 1998.

[29] A. Leistm D. Playne K. Hawick, "Parallel graph component labelling with GPUs and CUDA," *Parallel Computing*, vol. 36, no. 12, pp. 655-678, 2010.

[30] V. Wong, GPU Acceleration of Volume Segmentation for Retinal Thickness, 2013, Bachelor's Thesis.

[31] Xiao T. Li, Peter Nicholas, Cynthia A. Toth, Joseph A Izatt, Sina Farsiu Stephanie J. Chiu, "Automatic segmentation of seven retinal layers in SDOCT images congruent with expert manual segmentation," *Optics express*, vol. 18, no. 18, pp. 19413-28, 2010.

[32] Nathan Kirsch. (2016, Feburary) AMD and NVIDIA Release Vulkan 1.0 API Beta Drivers. [Online]. http://www.legitreviews.com/amd-nvidia-release-vulkan-1-0-api-beta-drivers_179095

[33] PassMarkSoftware. CPU Benchmarks. [Online]. http://cpubenchmark.net/