Pseudo-real-time Retinal Layer Segmentation for OCT

by

Worawee Janpongsri

B.A.Sc, Simon Fraser University, 2017

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

SIMON FRASER UNIVERSITY

Fall 2019

Approval

Name:	**Worawee Janpongsri**
Degree:	**Master of Applied Science**
Title:	**Pseudo-real-time retinal layer segmentation for OCT**
Examining Committee:	**Chair:** Dr. Ash M. Parameswaran Professor

Dr. Marinko V. Sarunic
Senior Supervisor
Professor

Dr. Yifan Jian
Co. Senior Supervisor
Assistant Professor

Dr. Pierre Lane
Internal Examiner
Associate Professor of Professional Practice

Date Defended/Approved: Dec 19, 2019

Abstract

In this thesis, we present a pseudo-real-time retinal layer segmentation for high-resolution Sensorless Adaptive Optics-Optical Coherence Tomography (SAO-OCT). Our pseudo-real-time segmentation method is based on Dijkstra’s algorithm that uses the intensity of pixels and the vertical gradient of the image to find the minimum cost in a geometric graph formulation in a limited search region. It segments six retinal layer boundaries in an iterative process according to their order of prominence. The segmentation time is strongly related to the number of retinal layers to be segmented. Our program permits *en face* images to be extracted during data acquisition to guide the depth specific focus control and depth dependent aberration correction for high-resolution SAO-OCT systems. The average processing times for our entire pipeline for segmenting six layers in a retinal B-scan of 992x400 pixels, 496x400 pixels and 240x400 pixels are around 23 ms, 26 ms and 14 ms, respectively.

Acknowledgements

I would like to express my deepest thanks Dr. Marinko Sarunic, my senior supervisor for providing me an opportunity to learn and do research in the field of medical image segmentation and guiding me throughout my master project. This opportunity allowed me to experience image processing applications in both Central Processing Unit (CPU) and Graphic Processing Unit (GPU). This reminds me with the advantages and disadvantages of CPU and GPU while programming to implement the best efficient and successful programs.

Also, I would also like to thank Dr. Yifan Jian for teaching and guiding me from the beginning of my wonderful graduate program. He has invested his full effort in guiding me with the effective algorithms advices used in this project.

Last but not least, I would like to thank the rest of BORG members for supporting me throughout my degree.

Table of Contents

List of Tables

List of Figures

List of Acronyms

SLO	Scanning Laser Ophthalmoscopy
OCT	Optical Coherence Tomography
GPU	Graphics Processing Unit
svOCT	speckle variance Optical Coherence Tomography
AO	Adaptive Optics
HS-WFS	Hartmann-Shack Wavefront Sensor
SAO	Sensorless Adaptive Optics
OCT-A	Optical Coherence Tomography Angiography
ILM	Inner Limiting Membrane
RPE	Retinal Pigment Epithelium
TD-OCT	Time Domain Optical Coherence Tomography
FD-OCT	Fourier Domain Optical Coherence Tomography
SD-OCT	Spectral Domain Optical Coherence Tomography
SS-OCT	Swept-Source Optical Coherence Tomography
FT	Fourier Transform
PCIe	Peripheral Component Interconnect express
CUDA	Compute Unified Device Architecture
CPU	Central Processing Unit
ALU	Arithmetic Logic Unit
CU	Control Unit
IF	Instruction Fetch

PC	Program Counter
ID	Instruction Decode
EX	Execute
MEM	Memory Access
WB	Write Back
SM	Stream Multiprocessor
SP	Stream Processor
SPU	Special Purpose Unit
API	Application Program Interface
ILP	Instruction-level Parallelism
TLP	Task-level parallelism
DLP	Data-level parallelism
SISD	Single Instruction Single Data
SIMD	Single Instruction Multiple Data
MISD	Multiple Instruction Single Data
MIMD	Multiple Instruction Multiple Data
RAM	Random Memory Access
NPP	NVIDIA Performance Primitive
CCL	Connected Component and Labeling
PR GC	Push-Relabel Graph-Cut
ROI	region of interest
NFL/GCL	Nerve Fiber Layer/Ganglion Cell Layer
INL/OPL	Inner Nuclear Layer/Outer Plexiform Layer

IPL/INL	Inner Plexiform Layer/Inner Nuclear Layer
OPL/ONL	Outer Plexiform Layer/Outer Nuclear Layer
BORG	Biomedical Optics Research Group
MKL	Math Kernel Library
MIP	Maximum Intensity Projection
GPGPU	General-Purpose Graphics Processing Unit
FPGA	Field Programmable Gate Arrays
VLSI	Very-Large-Scale Integration
HDL	hardware description language

Chapter 1

Introduction

Our eyesight is one of the most important senses in which about 80 percent of our senses come through our sense of sight [1]. Most people have eye problems at one time during their lives. Some are minor problems and can be treated easily while some may need special care from ophthalmologists. In some serious situations, the surgical procedures may be required during the clinical ophthalmic treatments. As our eye is a small and fragile organ and is heavily supplied by nerves and blood vessels, the extreme care during the surgical procedures must be taken; otherwise, it could cause negative surgical outcomes. Thus, using an imaging and processing modality that capable of yielding rapid high-resolution ocular images and evaluating immediately anatomical changes to help assist during surgery could potentially increase the rate of success.

Since 1886, the first living human fundus image is photographed by Jackman and Webster [2], there have been critically advances in the knowledge and technique to acquire the retinal images. In ophthalmology, there are several imaging methods used in clinics such as color fundus photography, scanning laser ophthalmoscopy (SLO) and optical coherence tomography (OCT). Color fundus photography uses a fundus camera which is a camera with a specialized low power microscope to capture color images of the interior surface of the eyes known as fundus. Fundus image includes retina, retinal vasculature, optic dish and posterior pole. The advantage of color fundus photography is that it is a quick and simple technique providing a large retinal field of view. However, this imaging method gives two-dimensional image which is difficult to observe abnormalities due to lacking the depth information. Also, it is uncomfortable to the patients being imaged due to the exposure of bright visible light. SLO is a non-invasive method that uses coherent light source and a confocal raster scanning technique to generate the retina and the optic nerve head images. It uses horizontal and vertical scanning mirrors to scan a particular region of the retina and form reflectance images. Although SLO can generate high lateral resolution, its axial resolution is poor. On other hand, OCT is a non-invasive, and painless imaging method that uses light waves to create high resolution cross-sectional retinal images. Since OCT has been introduced in early 1990s, the algorithm and modalities of OCT imaging have dramatically improved in both resolution and acquisition speed. The evolution from Time-Domain (TD) OCT to Fourier-Domain (FD) OCT dramatically increased the imaging speed, resulting in shorting the time to acquire a scan and reducing the artefacts related to the eye movements. The video rate imaging speed of FD-OCT enabled real-time retinal tracking and calibration of OCT modality.

1.1 Research Motivations

OCT is an imaging modality that has been widely used in the field of biomedical imaging such as retinal imaging. It provides cross-sectional three-dimensional images of tissues that can be analyzed for tissue identification and properties. Ocular imaging with OCT allows ophthalmic clinicians to view and measure the distinctive retinal layer structure to diagnose retinal diseases such as glaucoma and age-related macular degeneration. OCT technology has continuously improved its acquisition speed; however, due to the complexity of OCT data processing from interferometric fringe data into images, the signal processing is computationally burdensome. Thus, powerful computational resources such as Graphics Processing Units (GPUs) were used to perform the parallelizable aspects of processing interferometric fringes into A-scans, and rendering the resulting volumes [3]–[10]. Our custom GPU pipeline could perform OCT processing at 2.24 MHz axial scan rate [3] and was also demonstrated for displaying flow contrast *en face* images extracted from the selected depth region on speckle variance OCT (svOCT) angiography [4] in real-time.

With the OCT images generated during acquisition, there is an opportunity to process the B-scan images to extract additional information. For example, segmentation of the retinal layer boundaries provides an opportunity to perform thickness measurements, which are an important part of clinical retinal imaging. Alternatively, *en face* images (taken in the C-scan directions) can be extracted from the OCT volumes. The *en face* images can be used for adjusting the focus to a particular depth plane, or for the application such as improving the lateral resolution using adaptive optics (AO) [11].

Optical aberrations caused by imperfections in the cornea and the intraocular lens reduce the image resolution. Adaptive optics have been integrated with optical retinal imaging to correct aberrations and allow diffraction limited imaging [12]; in a particular, our interest is in the combination of AO with OCT [11]–[18]. The conventional approach to AO uses a Hartmann-Shack Wavefront Sensor (HS-WFS) to detect wavefont distortions and compensate them using a deformable mirror. However, the HS-WFS is sensitive to back-reflections, causing most of the conventional AO to use curved mirrors instead of lenses. We are developing a lens-based sensorless adaptive optics (SAO) approach to correct optical aberrations up to 21st order Zernike polynomials, starting from a defocus [19]. The SAO methods directly evaluate an image quality metric to drive the AO correction. Hence, OCT systems can be used for applications where there are multiple reflecting surfaces from the sample (which could confound a wavefront sensor measurement) or requiring depth resolved aberration correction.

To extract the thickness measurements and *en face* images from OCT volumes, many segmentation methods have been introduced. The most reliable way is manual segmentation which is not suitable for large data sets because it is very time-consuming. Active contours segmentation [20]–[22] uses an energy formulation; however, it requires a good initialization, and the constraints on the boundaries can cause errors when the retinal layers are in irregular shapes. Besides, the active contours approach also has a high computational cost which is not suitable for real-time applications. Machine learning approaches [23], have been recently introduced, and they perform retinal segmentation based on learning data representations. They give accurate results; however, they require substantial amount of labelled learning data sets supplied to the networks and it is computationally expensive to train the networks. Automated segmentation methods, based on graph theory, use pixel intensity and the gradient of the image to find the minimum cost in a geometric graph information for each retinal layer boundary [24]–[27]. The accuracy and the speed of segmentation, based on graph theory, depend on the algorithmic implementation used.

To integrate segmentation in real-time OCT imaging applications, a robust segmentation algorithm with low computational cost and low complexity is required. Utilizing real-time retinal segmentation has several benefits. It can be used to calculate the thickness of the retinal layers, and permit investigation of the major diseases causing blindness. Also, the diseases related to retinal vessels, such as diabetic retinopathy, can be diagnosed through the retinal vasculature and its hierarchical structure from *en face* OCT-Angiography (OCT-A) images. In clinical applications, a system such as computer-assisted surgery [28]–[34] that can yield rapid high-resolution ocular images and evaluate anatomical changes immediately to help assist during surgery could potentially increase the rate of success. Moreover, real-time retinal layer segmentation can provide effective focus control and direct feedback of aberration correction performance with image-guide AO techniques [35]. This last application is the main focus of this report.

There are several fast retinal layer segmentation attempts for OCT images. Fabritius *et al.* reduced the processing time of segmenting Inner Limiting Membrane (ILM) and Retinal Pigment Epithelium (RPE) by heavily down-sampling each B-scan [36]. The total processing time using a computer with 2.4 GHz CPU to segment a healthy macula volume (1024x320x140) was 6.7 seconds: the average time needed for the ILM was 4.0 seconds and for the RPE was 1.9 seconds. Tian *et al.* developed a faster (~10x) automatic segmentation program called OCTRIMA 3D [27] based on the previous work by Chiu *et al.* [24]. In a follow up paper [37], Tian *et al.* evaluated different segmentation softwares on a computer with an Intel® Core^{TM} i7-2600 @3.4 GHz CPU. The result showed that OCTRIMA 3D could perform the segmentation of an OCT volume (768 x 496 x 610) in 28 seconds.

In this thesis, the main goal is to minimize the processing time of the automated retinal layer program while maintaining the reliability of the segmentation results. We present a pseudo-real-time retinal layer boundaries segmentation program in mice modified from the *Caserel* software [25]. The retinal segmentation was integrated in our custom GPU pipeline [3] with SAO algorithm [19] to perform a pseudo-real-time retinal layer segmentation in mice for high-resolution real-time visualization of vascular network, which provides focus control and aberration correction. The organization of the rest of this thesis is as follows. Chapter 2 provides the overview of OCT system and acquisition. The background of heterogeneous computing are provided in Chapter 3. Chapter 4 introduces high-speed retinal layer segmentation algorithm. Chapter 5 explains the retinal segmentation pipeline. Chapter 6 demonstrates the retinal layer segmentation results and its speed along with the analysis and Chapter 7 covers the conclusion and the future work.

Chapter 2

Optical Coherence Tomography: Overview of system and acquisition

This chapter provides an overview of OCT, our SAO-OCT imaging system for small animals (LIVMAOS system) [19], and our processing and display program for the OCT retinal images (OCTViewer) [3]. They are integrated to form a complete real-time imaging and processing system. The LIVMAOS system is a custom multi-modal imaging system that includes OCT, OCT-A, SLO, and fluorescence detection. The OCTViewer, is a heterogeneous system that processes interferometric fringe data to OCT cross-sectional (B-scans) images and the pseudo-real-time retinal layer segmentation program (will be discussed in Chapter 5) generates retinal segmentation for investigating and evaluation retinal structure.

2.1 Optical Coherence Tomography

OCT is a non-invasive medical imaging technique based on low coherence interferometry, typically using near-infrared light to capture high-resolution cross-sectional views of biological tissues. OCT is based on the principle of low coherence interferometry in which low coherence light coupled to a fiber-optic splitter travels through a beam splitter, and one arm is guided through the ocular media to the retina and the other to a reference mirror. The interference pattern is produced by the interaction of reflected light from the sample and the reference mirror when the distance between the light source and the retinal tissue is equal to the distance between the light source and the reference mirror. Then the interference fringes are detected using a photodetector and processed into an axial scan. The light is scanned along the retina to create a two-dimensional image.

There are two general categories of OCT systems, Time Domain (TD-OCT) and Fourier Domain (FD-OCT). In TD-OCT systems, the mirror in the reference arm is adjusted to match with the delay in the various layers of the sample. The interference signal produced is processed to create the axial scan waveform. Figure 2-1 shows the setup of TD-OCT. This TD-OCT method has low speed of image acquisition due to the movement of the reference mirror which moves one cycle for each axial scan.

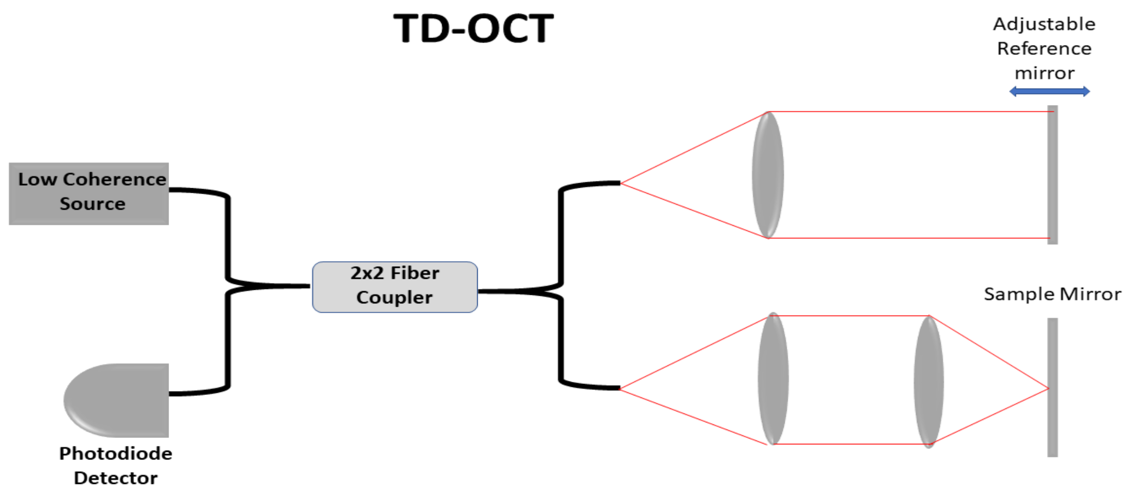
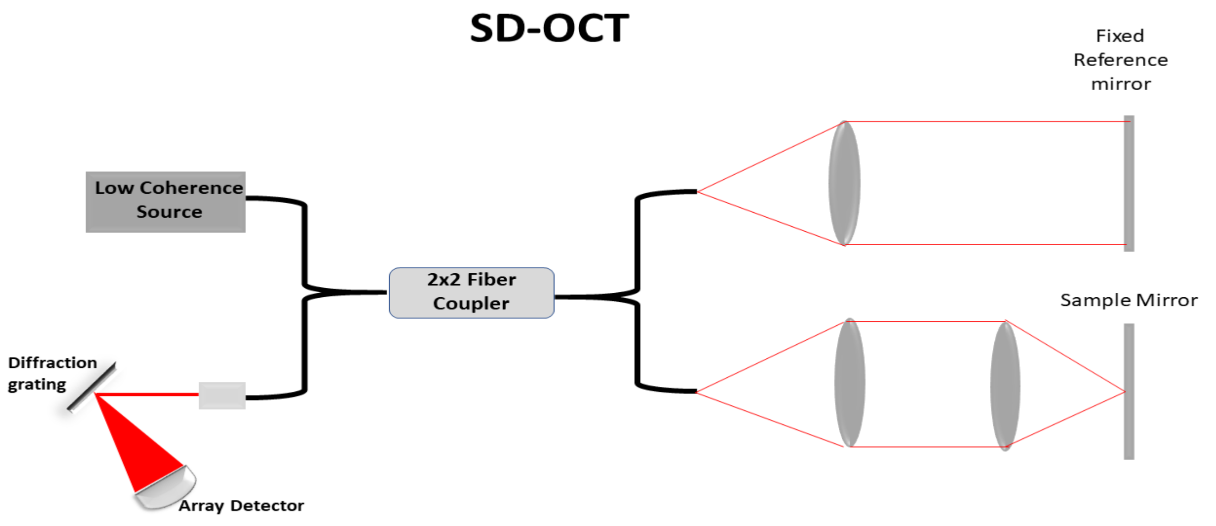


Figure 2-1 Time Domain Optical Coherence Tomography.

On other hand, in FD-OCT, the reference mirror is at stationary position and the interference between the sample and the reference reflection is measured as a function of wavelength. The absence of moving the reference mirror and the reflections from all layers in the sample are detected simultaneously, resulting in higher speed and higher sensitivity.

FD-OCT can be implemented in either Spectral Domain OCT (SD-OCT) or Swept-Source OCT (SS-OCT). SD-OCT uses a broadband light source, and a high-resolution spectrometer is employed in the detector arm of the interferometer to collects the interferogram as a function of wavelength. A Fourier Transform (FT) is performed on the interferogram which produces a spatial representation of the sample tissue in depth. The spectrometer uses a grating (or a prism) to spread the light into a spectrum which is detected by a high-speed line camera. On other hand, SS-OCT uses a narrowband light source that sweeps through the spectrum while a single-element photodetector collects the signal. The wavelength of the narrowband light source is encoded as a function of time. Figure 2-2 demonstrates the setup of SD-OCT and SS-OCT.



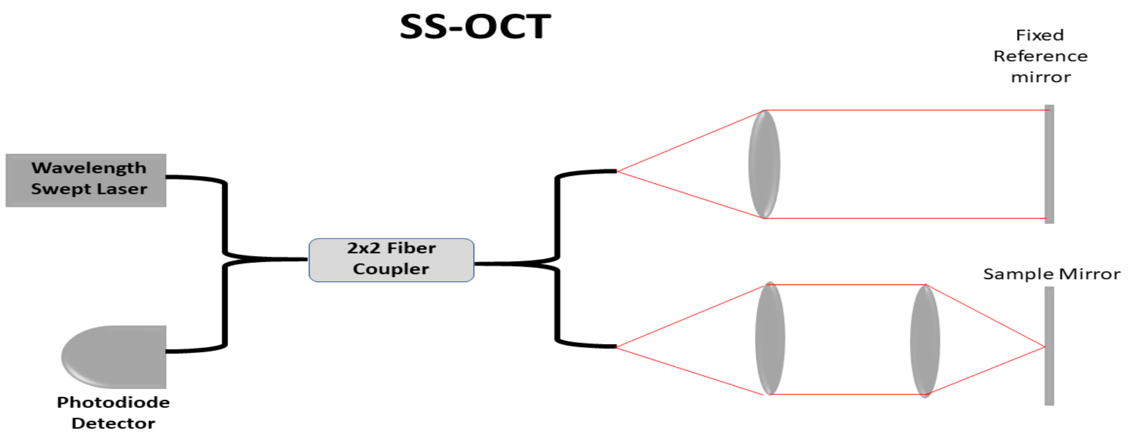


Figure 2-2 Spectral Domain and Swept Source Optical Coherence Tomography.

In both TD-OCT and FD-OCT, an A-scan is obtained by measuring the the distance from the beam splitter to a reflector in the sample relative to the reference. The "echoes" of the deeper surfaces take longer to return to the beam splitter for interference with the reference signal, which makes them have a higher interference frequency at the OCT detector, and consequently a larger distance on the A-scan display. A B-scan is formed by combining a series of laterally adjacent A-scans. This type of scan is useful to generate an image of retinal layers, the application area in which this work is interested. The axial resolution of the OCT systems is determined by the central wavelength (λ) and the spectral width of the light source (Δλ) whereas the lateral resolution depends on the system numerical aperture (NA).

$$Axial\ resolution\ (l_c) = \frac{2ln2\lambda^2}{n\pi\Delta\lambda}. \qquad \text{Eq. 2-1}$$

$$Lateral\ resolution\ (\Delta x) = 1.22\frac{\lambda}{2NA}. \qquad \text{Eq. 2-2}$$

2.2 OCT processing and display program (OCTViewer)

Real-time application of the OCT images requires high throughput and low overhead (latency). In this research, we used the parallelization strategies introduced by Jian *et al.* [3] to accelerate OCT processing. To fully utilize the Peripheral Component Interconnect express (PCIe) bandwidth, we transferred the interferometric data from the host to the device as a batch rather than a single frame. In order to hide memory transfer latency, the memory transfer from the host to the device and the data processing on the device was implemented using two Compute Unified Device Architecture (CUDA) streams concurrently; one to transfer the data processing on the device and another to process the interferometric fringe data on the device. While the small batch of the interferometric fringe data was being transferred from the host to the device by the transfer stream, the previous batch that is already in the device is simultaneously being processed by the kernel stream. These two CUDA streams, which are executed simultaneously, are synchronized after processing each batch. The original implementation by Jian *et al*. [3] demonstrated a high throughput; however, it suffered from a large latency as its processing pipeline was completely asynchronous with the acquisition. In this thesis, we improved the program latency by synchronizing the data acquisition and processing at the batch level, achieving a minimum latency of one batch which was configured as 20 B-scans. Figure 2-3 shows the flowchart of the processing tasks.

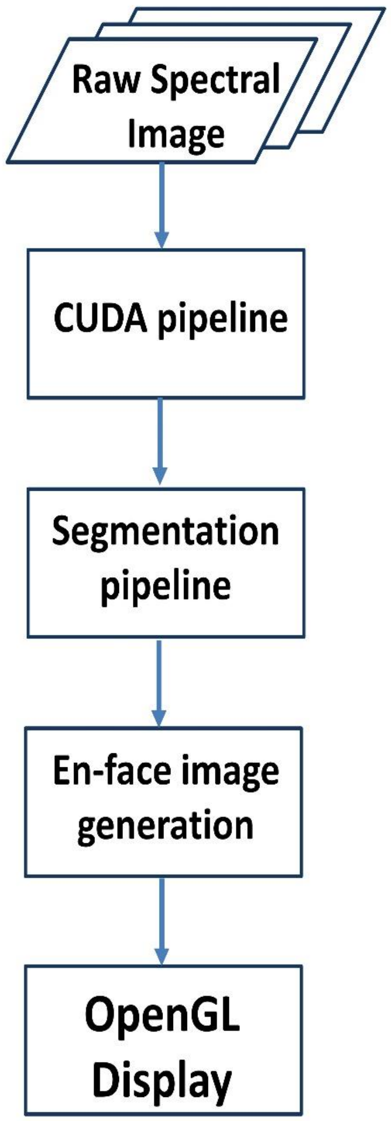


Figure 2-3 Flowchart of the processing pipeline.

2.3 SAO-OCT imaging system for small animals (LIVMAOS system)

LIVMAOS System is a multi-modal mouse retina imaging system that includes OCT, OCT-A, confocal scanning laser ophthalmoscopy (SLO), and fluorescence detection [19]. It is a compact lens-based system incorporating the SAO technique to correct the optical aberrations instead of using a Wavefront Sensor to measure the aberrations. For this thesis, we only used the modified OCT subsystem for mouse imaging. The OCT subsystem in this thesis used a central wavelength of 810 nm. We also integrated our retinal layer segmentation program in our OCT processing and display program along with the SAO–OCT imaging system. Our segmentation program segments retinal layers on the cross sectional images and uses these results to project *en face* images from the selected retinal layers. The SAO uses the *en face* projection of the OCT volume as the input of the merit function; the sum of the intensity squared of each pixel of the *en face* image, to perform the optimization. The OCT acquisition modality provided a 100 kHz line scan rate for retina imaging and the OCT volumes are acquired user selected dimensions. Two B-scans are acquired at the same lateral location to generate an OCT-A B-scan image [4].

2.4 Summary

In this chapter, we covered the overview of OCT, OCT processing program and OCT acquisition system. Due to the complexity of OCT data processing and high speed of the OCT acquisition system, we implemented the A-scan processing program using CUDA on the GPU [3]. We used the parallelization strategies for data transferring from the host to the device by using processing kernels in order to hide memory transfer latency and also rendering the images using the GPU without the need of transferring data back to the CPU. However, the image processing step (i.e., the retinal layer segmentation) is needed to determine if it should be done on either GPU or CPU and it will be discussed in Chapter 3 and 4.

Chapter 3

Background on Heterogeneous Computing

Heterogeneous computing refers to the systems that utilize more than one kind of processors or cores in order to gain a better performance. These systems select the type of processors or cores corresponds to their task specialization. The Central Processing Units (CPUs) are designed to accurately perform sequentially complex tasks at high speed while the GPUs are a better option in terms of the speed to handle the operations that can be divided to run as the smaller tasks simultaneously. Many applications including medical image processing, which involve complex computational tasks and massively parallelizable operations, require a heterogeneous computing system that consists of both CPUs and GPUs. This chapter will discuss the fundamentals of both CPUs and GPUs, their differences and the advantages of employing a heterogeneous computing system in our research.

3.1 Central Processing Unit

A CPU is the primary element or the 'brain' of the computer that executes instructions by processing the data received from both hardware and software to generate the output for the running application. Each CPU has at least one processor or core to process the instructions. CPUs with two, four, six and eight processors or cores are called dual-core, quad-core, hexa-core and octa-core CPUs, respectively. CPU consists of three primary components: the arithmetic logic unit (ALU) is a digital circuit which is responsible for integer arithmetic and bitwise logic operations, the control unit (CU) that directs the operation of the processor by serving the instruction for the ALU, and processor registers which are the temporary storage area for the instructions and the operands to be further processed by the ALU.

There are four basic functions for the CPU to process the data:

1) Fetch (IF) – The CPU fetches the instruction stored in the computer memory with each instruction has its own address. The program counter (PC) of the CPU is responsible for tracking the address of the instructions to be fetched next for the running program.
2) Decode (ID) – Different programs are written in different programming languages so there is a need for the complier to translate them into the assembly language in which the CPU understands. Then the assembler decodes assembly language into binary code that can be executed.
3) Execute (EX) – Depending on the instructions, the execution can be calculating arithmetic operations by ALU, moving the data from one memory location to another location or jumping to other instruction addresses in the program.
4) Store or Write Back (WB) – After executing data, the CPU stores the result or the output back to the computer memory.

3.2 Graphic Processing Unit

A GPU is a programmable electronic circuit chip or a processor that is specialized in enhancing the computing speed of rendering images or graphics in the memory and displaying them on the display devices. GPUs are commonly used in computers, mobile phones and game consoles to handle the graphics and image processing. GPUs are the processors which are designed for performing parallel operations on a large set of data. They are not used only for the graphics applications but also for the non-graphics applications as the vector processor to perform repetitively non-graphic calculations.

The term '*GPU'*, was popularized by the company named NVIDIA Inc. in 1999, who produced the first GPU called GeForce 256. It had 17 million transistors that could perform billions of calculations in a second and at least 10 million polygon manipulations per second [38], [39]. Later on, there are companies such as Intel, Matrox, S3 Graphics and AMD participate in producing the efficient GPUs in the market.

GPUs must be used with a CPU-base host as an accelerator or a co-processor. GPU is a massively parallel architecture which is different from CPU hardware architecture. Each GPU contains several Stream Multiprocessors (SMs) where each SM contains several

Stream Processors (SPs). The combination of SMs and SPs characterizes for processing multiple tasks at the same time or for processing a task with a higher speed. The overall throughput of a GPU is determined by the number of SP, the memory bandwidth, and the parallel architecture that the programmers implement. Each NVIDIA GPU has different identifying numbers that define its features, engines and number of registers it contains. Therefore, when designing a program, one needs to consider its support capability for the next generations of the GPUs which may have a greater number of SMs and/or SPs to increase the performance of the program on the newer GPUs.

Depending on the generation of the GPU, the number of the SPs that a SM contains can grow from eight SPs. Each SM has access to a register file which is a type of memory where the speed is same as Special Purpose Units (SPUs). A SPU is responsible for performing special hardware instructions. Again, the size of the register file differs among the GPU generations. A register file consists of thousands of 32-bit registers that are allocated to threads specified during the kernel launch. These registers are the fastest memory and they can store both integer and floating-point data. The scope for each register is per thread running on an SP.

Local memory is a part of the main memory of a GPU and it is used when the registers run out or they cannot be used. This phenomenon is called *register spilling*. It happens if there are many variables per thread using the registers and the dynamically indexed arrays are used in the kernel because the register files are not dynamically addressable. Again, the scope of the local memory is per thread. This memory is cached into L1 then L2 cache.

Another important memory within a SM is a shared memory. Threads use this memory to communicate within the thread block. The size of the shared memory depends on the compute levels. For example, the compute levels from 2.0 to 3.5 have 48 KB of the shared memory per SM whereas the compute 3.7 contains 112 KB of the shared memory per SM [40].

Texture memory is a memory specialized in indexing and interpolating pixels in two-dimensional images since it has many extra addressing tricks. It has its own cache that is not L1 and L2 cache. This memory is read-only by the GPU but read and write to the CPU. Also, there is a constant memory which is a part of the main memory of the GPUs that has its own cache. It permits the GPU to read-only which is same as the texture memory. On other hand, the global memory allows both the GPU and the CPU to read or write to it, but it has the slowest speed to access.

Compute Unified Device Architecture (CUDA) is the extension to the C language which is used to write the GPU programming in regular C, C++ and Fortran. It is created by NVIDIA and its applications can only be used on the CUDA-enabled GPUs. CUDA is an application program interface (API) that is ideal for embarrassingly parallel problems such as matrix operations and the problems where that is less or no communication required between threads or blocks. Communication between blocks happens with invoking multiple kernels in series using a global memory. CUDA divides a program into grids which each grid contains a set of blocks with X and Y axes. A block can be assigned to any SMs whichever have free slot(s) and a set of blocks is run at a time with ordering in implementation of the blocks can be in any order.

Concurrency is a program or an algorithm property that divides a problem into smaller sub-problems which can be executed independently in any order. The results of each sub-problem are combined to form a final result that must be the same in any execution order. Concurrency property helps to enhance overall execution speed. However, concurrency definition differs from the term *parallelism* even though they are both commonly be used in multithreading. Parallelism is the property of the program to require at least two processors or cores in order to perform multi-tasks at the same time. Concurrency is the algorithm in which two tasks are executed in overlapping time period. This may cause one task to pause for a short period of time to execute another task. Concurrency can happen when there is only one processor or core.

An application is concurrent but not parallel when it executes more than one task with no two or more tasks being executed at the same time. Whereas, an application is parallel but not concurrent is when multiprocessors or cores perform their individual task at a particular time. An application can be neither both when a single processor or a core executes the tasks sequentially. Finally, an application can also be both concurrent and parallel when it executes multiple tasks at a particular time and divides those tasks to execute individually.

There are three common parallel patterns which a program can be transformed into a parallel program; loop-based pattern, fork/join patterns, and divide and conquer pattern. Loop-based pattern is the most basic pattern that occurs in many programs and it is the easiest pattern which can be transformed into parallel programming. In order to implement loop-based pattern program into parallel program, the dependencies within the loop (for, while and do..while) must be removed. The dependencies are when the calculation depends on the previous iteration(s). After removing the dependencies of the loop, the program can be split into sub-parts and they are calculated by the processors. However, when splitting the tasks, one needs to minimize the communication between the processors and maximizing the use of on-chip resources. Fork/Join pattern commonly appears on the serial programming where at a particular point of time, it needs to fork a number of threads or processes to execute each section of the code independently in parallel. After each thread or process completely executes its own part through communication among them, they join their works together. This approach is the best when all the threads or the processes have equal works to performs; otherwise, they would wait for others to complete their tasks before they can converge their outputs. Divide and conquer pattern decomposes a problem into smaller sub-programs of the same type as the original problem. Then it solves them recursively and combines the results to be as the result of the original problem. This type of algorithm is used in quick sort that gives out the best performance among other sorting algorithms. Quick sort divides the data into two groups by comparing with the pivot; the data where the values are larger than the pivot's value and the data where the values are lower than the value of the pivot. It then recursively divides the data till each dataset consists of two items where they are compared and swapped their positions.

3.3 Parallel Computing

Parallel Computing is the method in which the instructions of the program or the processes are executed simultaneously. A large problem is divided into smaller independently parts that can be executed instantaneously and the results of those broken parts are combined after all the parts are executed successfully. Parallelism becomes an important factor to determine the computing performance of the computer due to the physical constraints regarding a clock speed and the power consumption of the computer. Fast computers require higher clock speed rate hence consume more power to perform the computations faster. This makes them generate more heat and it is a must for them to require a special and expensive cooling machine to draw out the heat. Power consumption is dissipated in form of thermal heat. Rising in temperature leads to power consumption increased and in turn lead to rising the temperature. This is an iterative process till the temperature reaches a runaway point after which the circuit burns which is called *thermal runaway*. In order to avoid thermal runaway, parallel computing has come into play for the computer architecture by having multi-core processors.

3.3.1 Instruction-level Parallelism

Instruction-level parallelism (ILP) is the simplest method for a computer to increase parallelism. ILP method allows the next instruction to be executed before the prior instructions completely implemented by breaking down the instruction into five stages: instruction fetch (IF), instruction decode (ID), execute (EX), memory access (MEM) and register write back (WB). This technique is called *instruction pipelining*. Pipelining permits multiple instructions from the same instruction stream to be executed concurrently by the hardware or by the complier. Without pipelining, two instructions are executed completely in ten cycles which one instruction starts being executing after the prior instruction is completed as shown in Table. 3-1. On other hand, if the pipelining is used, three instructions are executed successfully in seven clock cycles shown in Table. 3-2. Absolutely, pipelining enhances the speed of execution of instructions. However, the pipeline hazard may occur in pipeline architecture when the execution of the instruction must be delayed due to unavailable of operands. This is solvable by delaying the execution of the instruction by inserting a pipeline stall.

	Phase									
INST	1	2	3	4	5	6	7	8	9	10
1	IF	ID	EX	MEM	WB					
2						IF	ID	EX	MEM	WB

Table 3-1 Execution of instructions without pipelining.

	Phase									
INST	1	2	3	4	5	6	7	8	9	10
1	IF	ID	EX	MEM	WB					
2		IF	ID	EX	MEM	WB				
3			IF	ID	EX	MEM	WB			

Table 3-2 Execution of instructions without pipelining.

3.3.2 Task-level Parallelism

Task-level parallelism (TLP) allows multiple threads or processes to be manipulated instantaneously. It decomposes a problem into sub-problems and assigns each sub-problem to a processor or a core to execute. The processors or cores which execute these sub-problems communicate among themselves in order to update the data in the memory. For instance, there are four independent arrays of the same size are needed to be transformed. With TLP, each array is assigned to one processor or core to execute it. Nevertheless, the limitations of this parallelism are synchronization and communication overheads between the processors or the cores.

3.3.3 Data-level Parallelism

Data-level parallelism (DLP) is the method that focuses on the data to be processed rather the tasks. Again, the four independent arrays of the same size are needed to be transformed. DLP would instead of assigning an entire array to each core, divides an array into multiple sub-arrays and assigns each sub-array to each processor or core. Then after completing transforming the first array, it moves to the second array and so on. One array at a time rather four arrays at a time. Yet, this method is limited by bandwidth and unusual data manipulation designs.

3.4 Flynn’s Taxonomy

Flynn’s Taxonomy is the classification of different computer architectures. These are:

- Single Instruction Single Data (SISD) - the standard uniprocessor in which there is only one instruction executes on a single data set at a time. This category behaves as multi-tasking by doing a fast switching between tasks called time-slicing.
- Single Instruction Multiple Data (SIMD) – a method in which there is a single instruction to be performed on multiple data set. Different data sets are assigned to different processors and are executed with the same instruction.
- Multiple Instruction Single Data (MISD) - a method which allows different instructions to be performed on the same data set. It performs a single instruction on the data set at a time.
- Multiple Instruction Multiple Data (MIMD) - the most common method in today’s computer architecture; dual-, quad-, hex- and octa-core devices. Each core has its own data and instructions to be executed independently.

3.5 Differences between CPU and GPU

Both CPUs and GPUs are processor units that are used to process information however they are different in term of architecture. A CPU is a general-purpose processor which can perform any computations, but a GPU is a special purpose processor specialized in graphics processing. Since the CPU contains a small number of more powerful devices or cores with high caches which are designed for running a number of complex tasks. While the GPU contains a huge number of less powerful devices or cores which are designed for performing the tasks that can be broken down into smaller independently parts. Due to small number of cores of the CPU, it suffers from dividing the workload between the cores evenly. On the other hand, the GPU splits the workload between the cores better but undergoes difficulty in synchronization and coordination among a large number of cores.

CPUs and GPUs have different ways of handling multi-tasking. When there are large number of tasks to perform concurrently, the CPU does scheduling based on time-slicing by giving equal time to each task to ensure that every task gets to be processed by the processor equally. By switching between tasks to be performed by the core, the data or information of the current task is saved to the Random Access Memory (RAM) from the registers per core and the data of the next task or information is retrieved from the RAM to the registers per core. As the CPU has small number of registers per core that are used to execute a task, it is expensive in term of context switching as the number of tasks increases and it leads to lower efficiency of the CPU. While, the GPU contains multiple banks of registers and when the GPU does context switching, it sets a bank selector to switch in and switch out the current set of registers. This makes context switching in the GPU faster than that in the CPU because of the cost of transferring data between the registers and the RAM is higher.

The CPU can be dual-, quad-, hex-, or octa-core devices and it often runs a single thread program and calculates only single data point per iteration. Unlike the CPU, the GPU, for example, Fermi devices have 16 SMs which makes the GPU calculate to 32 data point per SM giving 32 times benefits in term of throughput for the GPU. The CPU is designed with higher frequency and is good for handling branching programming such as if-statement and switch-statement. Conversely, the GPU is bad for executing the branching statements because the number of running threads are divided into different branches and a branch is executed one after another.

In conclusion, the CPU is a brain of the computer while the GPU is a co-processor which helps performing special functions in parallel. The performance of the computers can be efficient by assigning tasks to the CPU and the GPU according to their capabilities.

3.6 Needs for Heterogeneous Computing

The CPUs are proficient in performing complex algorithm operations while the GPUs excel in executing larger simple tasks which can be broken down into smaller parts. These smaller parts need less or no communication among them and can be executed independently. Since the medical imaging applications are dealing with large amount of data as well as complex calculations, the heterogeneous computing, which is the combination of the CPU and the GPU, is required to optimize the processing time. The OCT imaging software, which performs OCT data acquisition, signal and image processing, and data interpretation, requires heterogeneous computing. The raw data acquisition is a small but complicated algorithm that requires low operation latency, best performing by the CPU. On the other hand, image processing that transforms the raw interferometric data into the B-scan images dealing with large amount of data and requiring high data throughput. Hence, it is better performing on the GPU. The data interpretation, which is in this project, is the retina layer segmentation requiring both the CPU and the GPU to perform the complex sequential parts and the massively intensive parts. Therefore, a heterogeneous computing is essential to gain high performance and optimize the processing rate.

3.7 Summary

In this chapter, we covered the basics of CPUs and GPUs along with their advantages and disadvantages. We also covered the parallel computing and the needs for heterogeneous computing in order to increase the performance of the system. Next chapter, we will discuss about the high-speed retinal layer segmentation algorithms on both CPU and GPU.

Chapter 4

High-speed Retinal Layer Segmentation Algorithm

Segmentation has played an important role in graphics and computer vision problems such as object detection, object recognition, image compression and image edition. The main idea of segmentation is to partition the elements of the graph into multiple disjoints groups by assigning a label to each element such that the elements with the same label share the same characteristics such as color, intensity or texture. The goal of segmentation is to simplify and/or change the representation of the image into something more meaningful and easier to analyze. This chapter also discusses the implementation of pseudo-real-time retinal layer segmentation algorithms on CPU versus GPU for OCT.

4.1 Graph Theory

Graph theory is the study of graphs. A graph is an abstract representation of a set of objects connected by links. A graph, ***G = (V, E),*** in a mathematical context consists of vertices, nodes or points, denoted by ***V,*** which are linked by edges, arcs or lines, denoted by ***E***. A weighted graph has weight, ***W***, assigned to each edge connecting between two vertices. A graph may be undirected or directed graph. An undirected graph has edges that do not have direction and its edges indicate two-way relationship or its edges are unordered pairs whereas a directed graph has edges with direction and its edges indicate one-way relationship or its edges are ordered pairs. A graph normally contains special vertices called terminals and these terminals are called the source ***s*** and the sink ***t***. An ***s-t graph*** is a weighted directed graph with two identified vertices, the source ***s*** and the sink ***t***. An ***s-t cut, C(s, t)*** is a set of edges, $\boldsymbol{E_{cut}}$, such that there is no path from ***s*** to ***t*** after $\boldsymbol{E_{cut}}$ is removed from the graph, ***E***. The capacity or the cost of the cut is the total weight of its edges from ***s*** to ***t***.

4.2 Graph-cut background

One of the classic problems in graph theory is to find the shortest path between two vertices in a graph. A path is a sequence of vertices, $<v_0, v_1, v_2, \ldots, v_k>$, in a graph, ***G = (V, E)***, such that each vertex is connected to the next vertex in the sequence. The weight of a path, ***w(p)***, is the sum of all the weight on the edges along the path. The shortest path weight from vertex u to v is:

$$delta(u,v) = \min\{\, w(p) : u \to v\}\; if\ there\ is\ a\ path\ from\ u\ to\ v,$$

$$delta(u,v) = infinity\ otherwise. \qquad \text{Eq. 4-1}$$

In computer vision, a graph cut can efficiently perform the image segmentation by solving the energy minimization problem. The general energy-based function can be represented as:

$$E(L) = \sum_{p \in P} D_p(L_p) + \sum_{(p,q) \in N} V_{p,q}(L_p, L_q). \qquad \text{Eq. 4-2}$$

where $L = \{L_p \mid p \in P\}$ is a labeling of image P, $D_p(.)$ is a data penalty function which indicates individual label-preferences of pixels based on observed intensities, $V_{p,q}$ is an interaction potential which encourages spatial coherence by penalizing discontinuities between the neighboring pixels and N is a set of all pairs of the neighboring pixels [41].The penalty in energy function is the cut cost in which the cost is calculated as the weight of the edges along the cut. The main idea of graph-cuts is to partition the elements of the graph, which is a series of the nodes separated by the weighted edges, into two disjoint groups using different algorithms such as maximum flow/minimum cut [41] and Dijkstra's algorithm [42]. The maximum flow/minimum cut theorem uses the concept of a flow network where the maximum amount of flow from the source to the sink is defined as the total weight of the edges in the minimum cut. An alternative method of solving the graph is using Dijkstra's algorithm with an adjacency matrix, which is the matrix containing the weights of the edges between two nodes, to find the minimum path or the shortest path. The functions for determining the weight value are important in terms of the results providing an accurately graph-cut. These can be calculated as the distance and/or the intensity value difference between the pixels of the image or nodes of the graph. Graph-cuts can be differentiated into GPU methods and CPU methods, and both methods will be discussed in details in the next section of this report.

4.3 Graph-cut on GPU

Image segmentation using Graph-cut can be done using the GPU since the parallelization decreases the segmentation time. The NVIDIA Performance Primitives (NPP) graph-cut APIs provides the graph-cut segmentation function based on the push-relabel algorithm.

4.3.1 Push-Relabel Algorithm

The push-relabel algorithm is an algorithm for computing maximum flows of a flow network or a graph. A network is a directed graph, ***G***, with vertices, ***V***, and edges, ***E***, combined with the cost function, ***c***, which assigned on each edge with non-negative values. A flow network is a network that flows the source, *s*, to the sink, *t*. A flow in a flow network is a function ***f*** that assigns each edge a non-negative value and follows two conditions:

- The flow of each edge cannot exceed its capacity.

$$0 \le f(e) \le c(e). \qquad \text{Eq. 4-3}$$

- The sum of the incoming flow of vertex *u* must be at least equal to the sum of outgoing flow of *u*

$$\sum_{(v,u)\in E} f((v,u) \ \ge \sum_{(u,v)\in E} f\big((u,v)\big). \qquad \text{Eq. 4-4}$$

Therefore, it is possible for some vertex to have an incoming flow more than an outgoing flow, and it results in having some excess flow and the excess function is defined as:

$$x(u) = \sum_{(v,u)\in E} f((v,u) - \sum_{(u,v)\in E} f\big((u,v)\big). \qquad \text{Eq. 4-5}$$

The residual capacity of an edge c_f is defined as:

$$c_f(v,u) = c(v,u) - f(v,u). \qquad \text{Eq. 4-6}$$

And the graph consists of residual edges is called a residual graph, G_f.

The height function or the labeling function, ***h***, is a function to ensure the regulation of push operation and the termination of the algorithm and it is valid if the three following conditions are held:

$$\begin{gathered} h(s) = |V|, \\ h(t) = 0, \\ h(u) \leq h(v) + 1 \; if\ there\ is\ an\ edge\ (u,v)\ in\ the\ residual\ graph. \end{gathered} \qquad \text{Eq. 4-7}$$

The algorithm will start with initializing each edge outgoing from the source with its maximal capacity ***f(s,u) = c(s,u)*** while other edges with zero and the height of the source is the number of vertices in the graph ***h(s)*** = |***V***| while the height of other vertices are zero. During execution, with the push operation, we try to push as much excess flow, ***min(x(u), c(u,v) – f(u,v))***, from one vertex ***u*** to a neighboring vertex ***v,*** where we are allowed to push the flow from ***u*** to ***v*** only if ***h(u) = h(v)+1***. In the case where the vertex ***u*** has excess but it is not possible to push it to its neighboring vertices, we use the rebel operation to increase the height of ***u*** by as much as possible if the labeling is still valid. The algorithm will continue till all vertices except the source and the sink are free from the excess, and this indicates the flow is a maximum flow.

4.3.2 The complexity of Push-Relabel Algorithm

The complexity or the efficiency is the computational complexity that describes the amount of the time the algorithm uses to run. Since the running time may differ depending on the inputs, the worst-case time complexity should be considered. The time complexity is commonly expressed in terms of Big-O notation because it gives the upper bound of the running time.

The relabel operation is performed at most $2|V|^2 -1$ times which results in a bound of $O(V^2)$ for the relabel operation. While the push operation is performed at most O(VE) for saturating pushes and at most $O(V^2E)$ for non-saturating pushes. The total complexity of Push-Relabel algorithm is $O(V^2E)$ [43].

4.3.3 Connected Component and Labeling

Sometimes, the Push-Relabel algorithm segments the unwanted regions along with the segmentation results, resulting in a corrupted segmentation result. Connected Component and Labeling (CCL) [44] is an algorithm to remove the artifacts from the segmentation results produced by the Push-Relabel algorithm. The CCL detects two largest connected groups and removes all unwanted artifacts and it is done on the CPU because it is a sequential instruction algorithm which is not easy to be implemented in parallel. This results in slowing down the segmentation pipeline because of transferring data between the CPU and the GPU.

4.3.4 Discontinuity of NPP graph-cut APIs

Unfortunately, the pre-built NPP graph-cut APIs is removed after CUDA 7.5 version [45]. For this reason, other segmenting methods are needed to be investigated to replace the NPP graph-cut APIs and they should have better performance than that of NPP graph-cut APIs. It should not produce the unwanted artifacts along with the segmentation results, so the CCL will not be required to perform after Graph-cut segmentation of the retinal layers.

4.4 Graph-cut on CPU

Since the NPP graph-cut APIs is removed and we do not want to use other algorithms that creates artifact results which requires another algorithm such as CCL to remove them, we found that Dijkstra's algorithm is suitable to be used in this project. Dijkstra's algorithm is a segmenting method that performed on the CPU and does not need CCL because it segments a retinal layer by finding one shortest path which extends across the entire width of the image.

4.4.1 Dijkstra's Algorithm for Graph-cut

One algorithm for segmenting a weighted graph from the starting node or the source **s,** to the target node, the sink **t,** is Dijkstra's algorithm. Dijkstra's algorithm is an algorithm for finding the single-source shortest path where it computes the length of the shortest path in the graph from the source to all other points in the graph. One condition to use Dijkstra's algorithm is that all the edges in the graph need to have non-negative weight ranging from 0 to 1. If the edges are negative, then the actual path cannot be obtained.

The graph has:

- Vertices denoted by ***v*** or ***u***
- Weighted edges connecting two nodes, ***(u, v)***, denoted an edge connecting the vertex ***u*** and the vertex ***v*** and ***w(u, v)*** denotes its weight

Before the algorithm starts:

- ***dist***, a distance array from the source ***s*** to each vertex in the graph is initialized as ***dist(s) = 0*** and for other vertices ***v***, ***dist(v)*** = ∞. As the algorithm proceeds, the ***dist*** from the source to each vertex in the graph will be recomputed and finalized when the shortest path is found
- ***Q*** is a queue of all vertices in the graph. At the end of the algorithm, ***Q*** will be empty
- ***S***, an empty set indicating which vertices has been visited. At the end of the algorithm, ***S*** will contain all the vertices in the graph

Procedures of the algorithm:

- While ***Q*** is not empty, select the vertex ***v*** in ***Q*** that has minimum distance and is not in ***S***. At the beginning, the source vertex ***s*** will be selected as ***dist(s) = 0***. Later, the vertex with the smallest ***dist*** will be selected
- Add ***v*** to ***S*** to indicate that ***v*** has been visited
- Update ***dist*** values of the adjacent vertices of the current node ***v*** as follows: for new adjacent vertex ***u***,
    - If ***dist(v) + w(u, v) < dist(u)***, update ***dist(u)*** to the new minimal distance value as the new minimal distance for ***u*** is found
    - Otherwise, no updates are made to ***dist(u)***

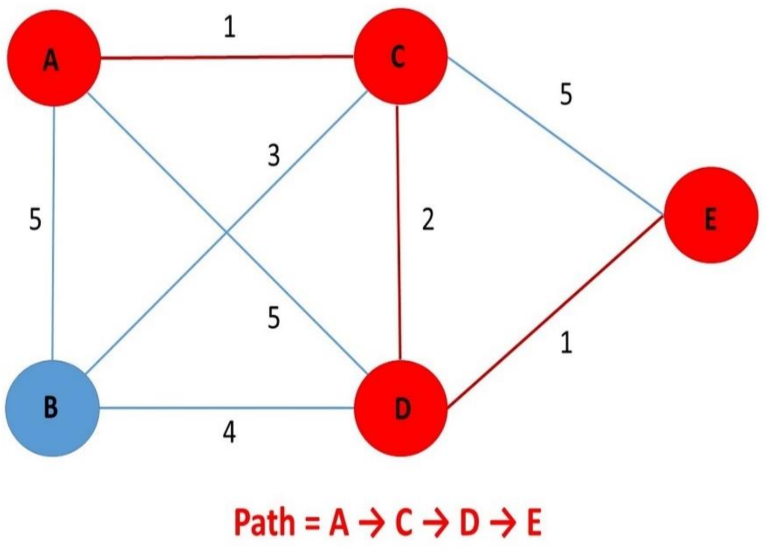
After the algorithm has visited all vertices in the graph and the smallest distance of each vertex is found, the shortest path from the source ***s*** and ***dist*** contains the shortest distance from the source ***s*** are found.

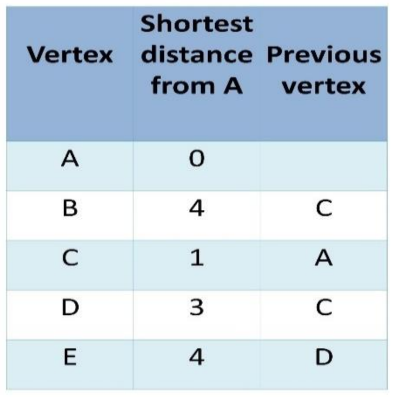
In summary, the algorithm in pseudo code is as follows:

```
Dijkstra (Graph, source)
{
    dist(s) = 0;
    S = Ø;
    Add s to S;
    for (each vertex v in Graph):
        if v is not source
            dist(v) = infinity
        add v to Q

    while (Q is not empty):
        v = vertex in Q not in S with minimum dist(v)
```

```
            remove v from Q and add v to S

            for (each neighbor u of v):
                if dist(v) + w(v, u) < dist (u)
                    dist(u) = dist(v) + w(v, u)
        return dist []
    }
```

Figure 4-1 shows the example of how to find the shortest path using Dijkstra’s algorithm.





Vertex	Shortest distance from A	Previous vertex
A	0	
B	4	C
C	1	A
D	3	C
E	4	D

Figure 4-1 Shortest Path using Dijkstra’s algorithm.

4.4.2 Efficiency of Dijkstra’s algorithm

The time complexity of Dijkstra’s algorithm depends on the number of the vertices **|*V*|** and number of the edges **|*E*|** in the graph. If the algorithm uses Fibonacci heap which is a min-priority queue, its time complexity is ***O(|E| + |V|log|V|)***. Whereas, if the algorithm uses a standard binary heap then its time complexity is ***O(|E|log|E|)***.

4.5 Dijkstra’s Algorithm from Boost Graph Library

Dijkstra’s algorithm from Boost Graph Library is used in this project because using Boost Graph Library is as simple as using other header files that requires no separate compilation. In order to use it, we set the location of the additional include directories in the working environment of Microsoft Visual Studio.

Dijkstra’s algorithm from Boost Graph Library differs from the Dijkstra’s algorithm mentioned earlier since it finds all the shortest paths from the source ***s*** to every other vertex by iteratively growing the set of vertices ***S*** to which it knows the shortest path. At each step, the next vertex is added to ***S*** decided by a priority queue, ***Q***. ***Q*** prioritizes vertices by their distance label or the weight on their edges. The vertex ***v*** at the top of the priority queue is added to ***S***. The algorithm loops till the priority queue is empty indicating the shortest path is found. Moreover, Dijkstra’s algorithm in Boost Graph Library uses color markers (white, gray and black) to keep track of each vertex. Vertices are colored as black are in ***S*** and white or gray in ***V-S***. White vertices are the vertices that have not been visited and gray vertices are those vertices in queue ***Q***. Also, there is an option to record the shortest paths tree in a predecessor map, for each vertex **u** in ***V***, ***p[u]*** is the processor if ***u*** in the shortest paths tree.

The time complexity of Dijkstra’s algorithm from Boost Graph Library is ***O((|V|+|E|) log |V|)*** or ***O(|E|log|V|)*** if all vertices are reachable from the source.

The pseudo code of Dijkstra’s algorithm from Boost Graph Library is as follows:

```
Dijkstra (Graph, source)
{
        dist(s) = 0; color(s) = GRAY; add s to Q;
        S = Ø;
```

```
    for (each vertex v in Graph:
        if v is not source
            dist(v) = infinity; p(v) = v; color(v) = WHITE;

    while (Q is not empty):
        v = vertex in Q with minimum dist(v)
        remove v from Q and add to S

        for (each neighbor u of v):
            if dist(v) + w(v, u) < dist (u)
                dist(u) = dist(v) + w(v, u)
                p(u) = v;
                if (color(u) = WHITE)
                    color(u) = GRAY;
                    add u to Q
                else if (color(u) = GRAY)
                    decrease key of u in Q
        color(v) = BLACK;
    return dist, p;
}
```

4.6 Summary

In this chapter, we discussed the details of graph theory, graph-cut method in the GPU which is the push-relabel segmentation provided by NPP graph-cut APIs. Nevertheless, sometimes the NPP push-relabel generates artifacts along with the retinal layers which later requires CCL to remove those artifacts on the CPU. This results in the overhead of transferring data between the device and the host causing the speed performance of the algorithm to decrease. Furthermore, graph-cuts in NPP APIs are removed. Consequently, we perform Dijkstra's algorithm from Boost Graph Library in the CPU to segment the retinal layers on the retinal OCT cross-sectional images. The complexity of Dijkstra's algorithm from Boost Graph Library is $O((|V|+|E|) \log |V|)$ whereas the push-relabel takes $O(|V|^2|E|)$. In conclusion, due to the complexity of the Graph-cut segmentation, it is more suitable to perform using the CPU as a processor. In next chapter, we will describe about the retinal segmentation pipeline using Dijkstra's algorithm to find the retinal layers.

Chapter 5

Retinal Segmentation Pipeline

This chapter describes the implementation of the pseudo-real-time retinal layer segmentation in mice in C/C++. The Method description is divided into six subsections: image cropping to contain only area of interest, logarithmic scaling and noise reduction, layer endpoint initialization, weights calculation, ILM and RPE layers segmentation, and limiting the search region for the other layers.

5.1 Retinal Segmentation

In our previous work where we chose to perform a Graph-cut on the GPU to segment ILM and RPE of human retina using Push-Relabel Graph-Cut (PR GC) algorithm [44], we found that sometimes the PR GC generates an unwanted region along with the two retinal layers. As a result, we performed Connected Component Labeling (CCL), which is a computational expensive method, after segmentation to identify the two largest connected groups and remove smaller artifacts. According to M. Miao [44], the segmentation pipeline with multiple GPU could only segment ILM and RPE layers on a single OCT B-scan (1024x300x900 pixels) in 57.26 ms which was largely affected by the CCL (12.45 ms). Nevertheless, due to the discontinuity of the pre-built NVIDIA Performance Primitives (NPP) graph-cut APIs after Compute Unified Device Architecture (CUDA) 7.5 version [45], other segmentation methods are needed to replace NPP graph-cut APIs and they should have a better speed performance and do not generate artifacts with retinal layers. Although there are several alternatives with both CPU and GPU implementations available for the NPP graph-cut APIs [41], [46]–[49], these max-flow/min-cut algorithm requires the CCL to remove artifacts which is quite slow.

Conversely, Dijkstra’s algorithm performs a Graph-cut by finding the shortest path or the minimum cost path between two nodes. It produces only a single path without any artifacts. Besides, Dijkstra’s algorithm from the Boost Graph Library [50], implemented on the CPU, has a generic interface and can be utilized easily using a header file in C/C++. Therefore, we implemented our retinal layer segmentation program using Dijkstra’s algorithm from the Boost Graph Library.

Figure 5-1 shows the flowchart outline of the steps for our retinal layer segmentation which was based on the *Caserel* software that was implemented in MATAB [25]. However, we changed the values of some parameters in the *Caserel* software for better segmentation results on our mice retinal images. We perform image cropping on the GPU following the generation of the B-scan image, and then the rest of the program is performed on the CPU in order to gain performance in heterogeneous computing according to the capability of each type of the processor.

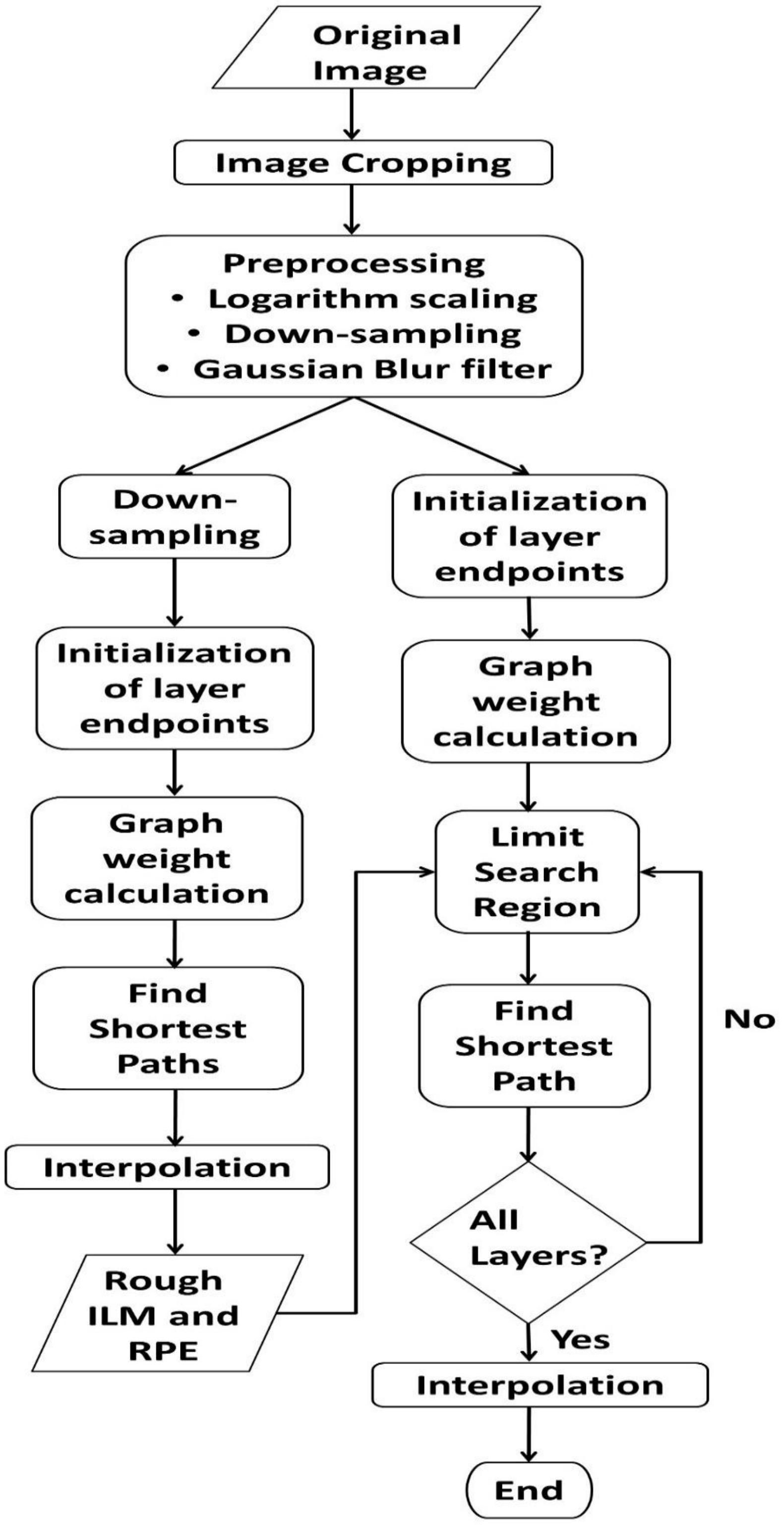


Figure 5-1: Flowchart of the real-time retinal layer segmentation.

5.1.1 Image Cropping for the region of interest

In OCT images, particularly when imaging the retina, the data in the axial direction (e.g, the retina) are contained in only a relatively small number of pixels. The pixels that do not contain image information affect the speed performance of the segmentation. In order to decrease the computational cost and make the delineation more reliable, cropping the image to contain only the region of interest (ROI) is necessary. Image cropping is performed on the GPU, immediately after the B-scan image has been generated from the interference signal, and is performed as follows:

Firstly, the average of pixels in each row of the image is computed along the lateral direction:

$$avg(y_j) = \frac{1}{N}\sum_{i=0}^{N-1} f(x_i, y_j). \qquad \text{Eq. 5-1}$$

where *f(x_i,y_j)* is the grayscale intensity of pixel *(x_i,y_j)*, *i* and *j* are the horizontal index and vertical index respectively, and *N* is the width of the image.

Secondly, we generate a histogram on the row average where the number of bins is chosen to be ten and the size of the bins is calculated as:

$$bin\ size = \frac{\max(row\ avg) - \min(row\ avg) + 1}{number\ of\ bins}. \qquad \text{Eq. 5-2}$$

After the histogram is computed, the value of the bin that contains the most common elements is selected to be the threshold. Empirically, the first and the last indices of the rows for which the average values are greater than the threshold correspond to the position where the retinal structure begins (ILM layer) and ends (RPE layer), respectively. In order to contain all the retinal characteristics in the ROI, the indices where the ROI begins and ends are calculated as:

$$ROI_{begin\ index} = i - offset * height_{image},$$

Eq. 5-3

$$ROI_{end\ index} = j + offset * height_{image.}$$

where *i* and *j* are the first and the last indices where the value of the row average are greater than the threshold. In this study, we set the *offset* = 0.1.

We only applied the cropping step on images with height greater than 200 pixels; otherwise, we skip this step because limiting the ROI may lead to cutting out some important retinal characteristics for segmentation. Figure 5-2 shows the original B-scan and its cropped image. In addition, we implemented this step on the GPU for faster parallel operations performance on a large set of data.

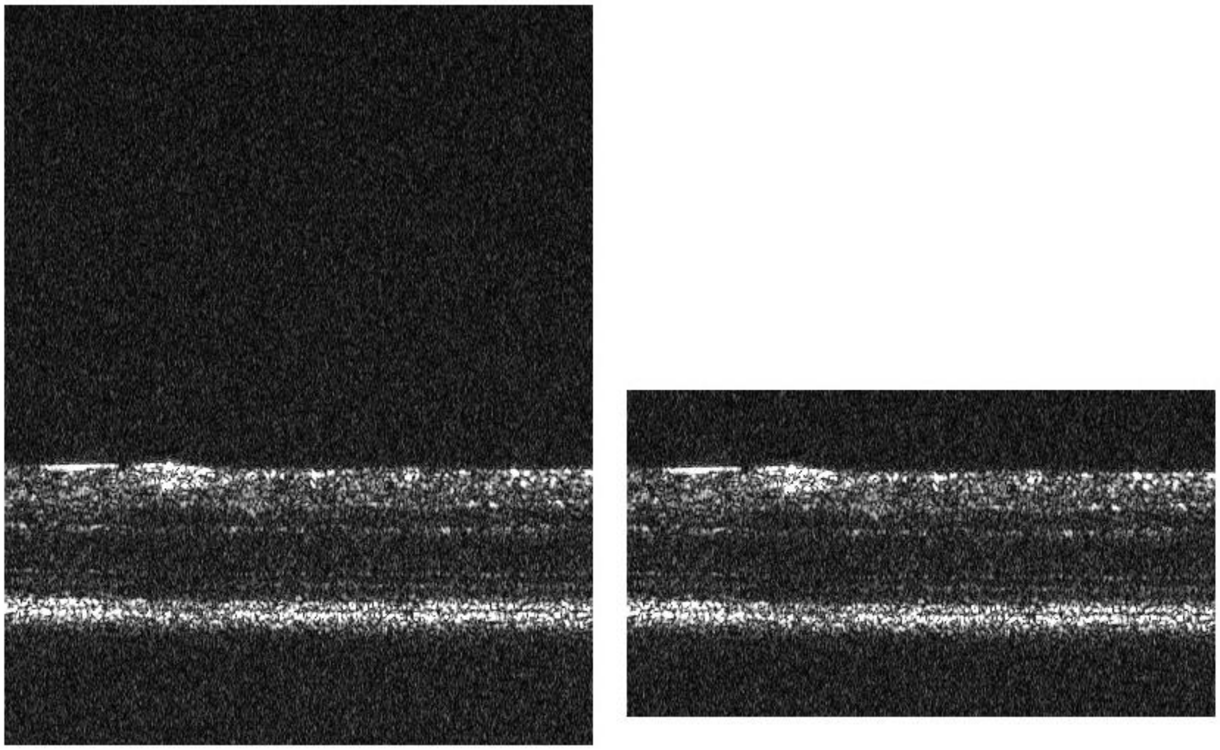


Figure 5-2 Original B-scan (496 x 400) and its cropped image (220x 400).

5.1.2 Logarithmic Scaling and Noise Reduction

Before segmenting the OCT retinal images, we employed a logarithm operation on each pixel on the image such that the dynamic range of the image is compressed to enhance low intensity pixel values. It expands the values of the dark pixels while compressing the higher-level values. Then we down-sampled the cropped image by a factor of two and referred to this as a *resized image*. Removing the noise from the images before further processing the images is essential as the noise could affect the quality of the automated segmentation. The most common noise in OCT images is the speckle which is produced by constructive/destructive interference. Speckle noise appears as white and black intensity fluctuations and can be reduced in appearance by applying a Gaussian blur filter. The formula of a Gaussian function in one dimension is:

$$Gaussian(x) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2}{2\sigma^2}}. \qquad \text{Eq. 5-4}$$

The formula of a Gaussian function in two dimensions is:

$$Gaussian(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}. \qquad \text{Eq. 5-5}$$

where, x is the distance from the origin in the horizontal axis, y is the distance from the origin in the vertical axis and σ is the standard deviation of the Gaussian distribution. Mathematically, the Gaussian blur effect is generated by convoluting the image with the Gaussian function. Figure 5-3 shows the Gaussian blurred image after a down-sampling by a factor of two of the cropped image.

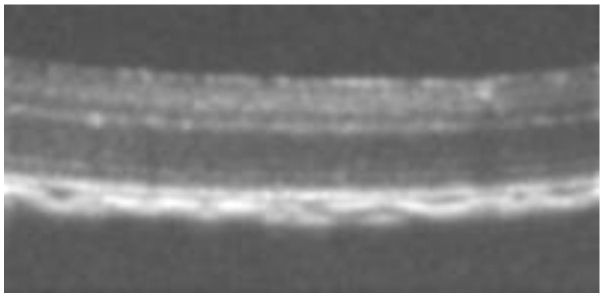


Figure 5-3 Gaussian blurred image.

5.1.3 Layer Endpoint Initialization

Each image is considered as a graph of nodes, in which a node equates to a pixel on the image having edges connecting to other nodes [41]. A graph may consist of multiple layered structures, and segmenting a particular layer requires the selection of the start and the end nodes. The start and the end nodes are automatically initialized by assuming that the retinal layers to be segmented extend across the entire width of the image. One vertical column is added to each side of the Gaussian blurred image, and they are assigned with zero values. The start node is the left top corner pixel and the end node is the right bottom pixel. These additional columns are removed after the segmentation is completed. Figure 5-4 shows an example of an image with one additional column on each side.

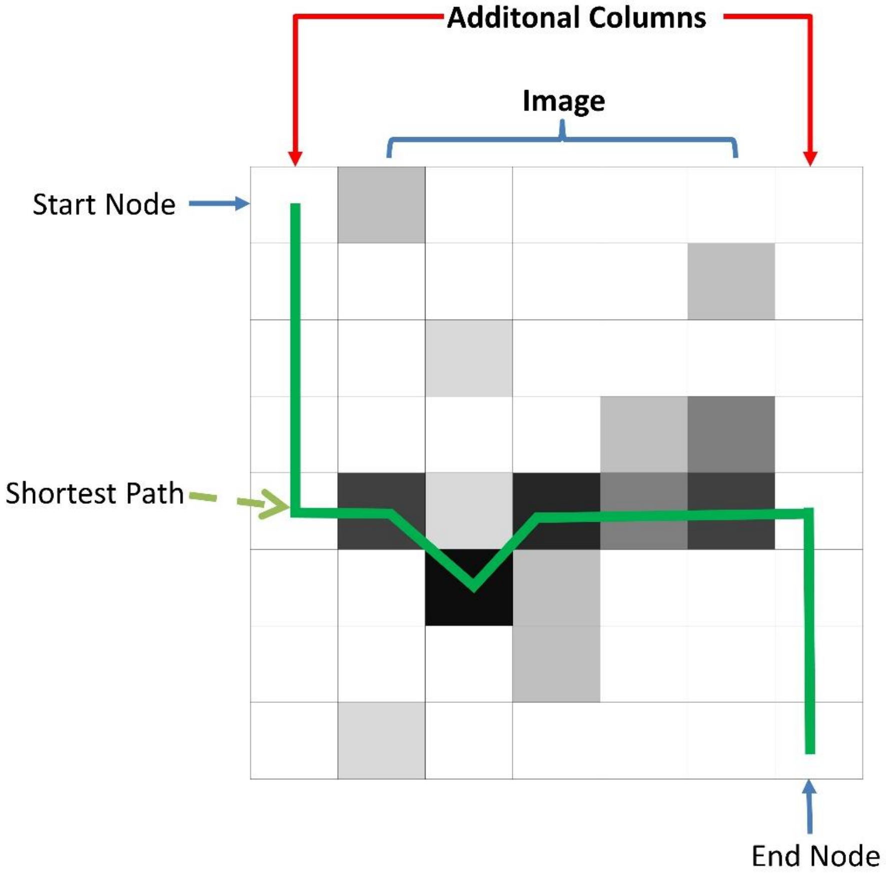


Figure 5-4 Example of an image with one additional column on each side.

In the retinal images, the foreground is defined as the retinal layers and the background as the vitreous and posterior chamber. The transition in pixel intensity from the background to the foreground is large so a graph can be simply contributed based on calculating the vertical gradient of the image. Figure 5-5 shows the two gradient images (the gradient and the negative gradient) of size 110x202 pixels. Both of them are generated because some retinal layer boundaries, such as the vitreous/ILM appears to have a darker layer above a brighter layer, whereas other boundaries, such as the Nerve Fiber Layer/Ganglion Cell Layer (NFL/GCL), have a lighter layer above a darker layer.

The gradient of the image is constructed as follows:

```
Gradient (image)
{
   for (each pixel y along the height of image)
      for (each pixel x along the width of image)
              if (y is starting pixel of the height)
                      G(x,y) = image(x,y+1) – image(x,y)
              else if (y is ending pixel of the width)
                      G(x,y) = image(x,y) – image(x,y-1)
              else
                      G(x,y) = image(x,y+1) – image(x,y-1)
}
```

Then the gradient is normalized by the following formula:

$$G(x,y) = 1 - \frac{-G(x,y) - \min(-G(x,y))}{\max(-G(x,y)) - \min(-G(x,y))} \qquad \text{Eq. 5-6}$$

The negative gradient image is generated as:

$$NG(x,y) = 1 - G(x,y) \qquad \text{Eq. 5-7}$$

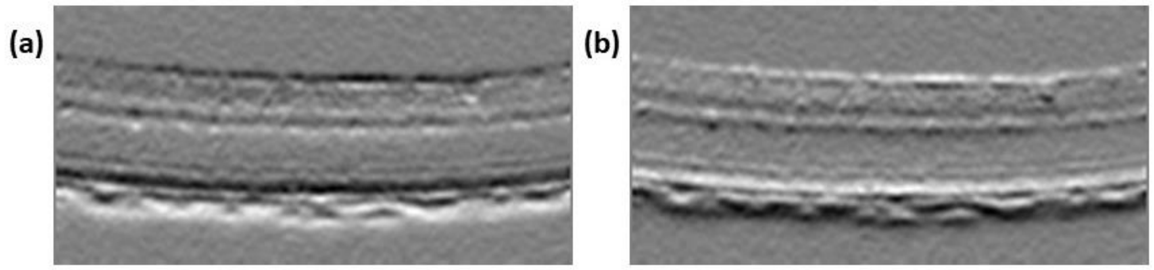


Figure 5-5 (a) The gradient and (b) the negative gradient images.

5.1.4 Weight Calculation

The weight of the edges usually represents the geometric distance and/or the intensity difference between the neighboring pixels. However, in SD-OCT retinal images, the features of interest have a smooth transition between neighboring pixels and each pixel is only connected with its eight nearest neighboring pixels and disconnected with other nodes. Hence, the weight of the edges is a function of the intensity difference between the neighboring pixels.

Because the retinal layers in OCT images are horizontal structures distinguishable by a vertical change in pixel intensity, the weights are calculated based on the vertical gradient. The formula used in this method for calculating the weights is:

$$w_{ab} = 2 - (g_a - g_b) + w_{min} \qquad \text{Eq. 5-8}$$

where w_{ab} is the weight assigned to the edge connecting nodes a and b, g_a is the vertical gradient of the image at node a, g_b is the vertical gradient of the image at node b and w_{min} is the minimum weight in the graph (1E-5).

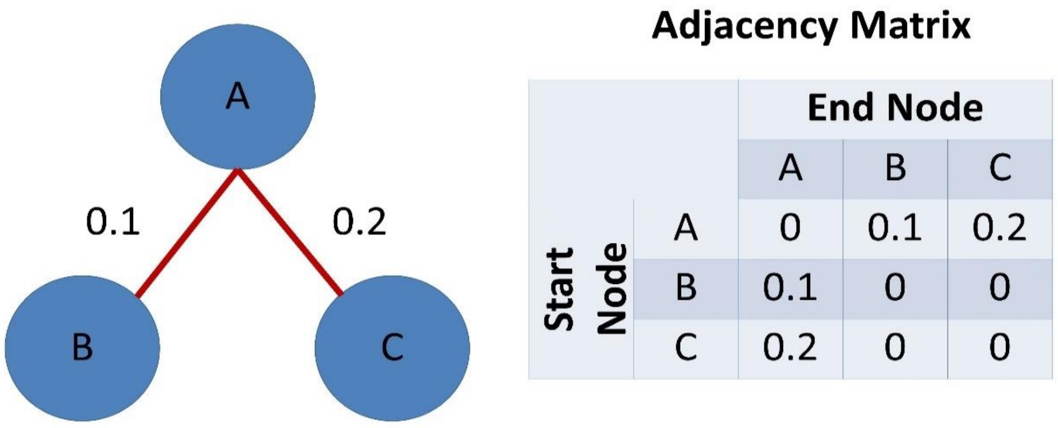
The weights are also calculated based on the directionality of the gradient. As the result, we have two sparse adjacency matrices of intensity difference graph weights of size [MN x MN] with MNC filled entries where [M x N] is the image size and C is the number of the nearest neighbors (in this case is eight). One axis of the sparse adjacency matrices of intensity difference graph weights represents the start node and the other axis represents the end node. The value of the element indicates the weight of the edge connecting the two nodes. For the retinal layer boundaries that exhibit a lighter layer above a darker layer use light-to-dark sparse adjacency matrix of graph weight calculated as:

$$w_{dl} = 2 - (G(x,y)_a - G(x,y)_b) + w_{min} \qquad \text{Eq. 5-9}$$

For the retinal layer boundaries which are dark to light boundaries uses dark-to-light sparse adjacency matrix of graph weight calculated as:

$$w_{ld} = 2 - (NG(x,y)_a - NG(x,y)_b) + w_{min} \qquad \text{Eq. 5-10}$$

As mentioned above, we added one additional column on each side of the image, we assign the weight values in those additional columns to be w_{min} so the shortest path calculation would not be affected by the additional columns. The edge weight of zero indicates that the two nodes are unconnected. Figure 5-6 shows the example of graph weight for three connected nodes and its table of the weight values called adjacency matrix of graph weight.



Adjacency Matrix

		End Node		
		A	B	C
Start Node	A	0	0.1	0.2
	B	0.1	0	0
	C	0.2	0	0

Figure 5-6 Example of graph weight for three connected nodes and its adjacency matrix.

5.1.5 ILM and RPE Layers Segmentation

The retinal layers are segmented in an iterative process according to their order of prominence. The ILM and RPE layers are segmented first due to their high contrast in pixel intensity relative to the background. To begin the retinal segmentation, the Gaussian blurred image is again resized by a factor of two to roughly segment the ILM and RPE layers. This twice down-sampled image is referred to as the *rough image*. Then we produce the negative gradient image and the dark-to-light sparse adjacency matrix of graph weights for the rough image. A ROI matrix of the same size as the rough image with two additional columns is generated and each pixel value of the ROI is set to 1 if the corresponding pixel on the rough image is greater than the mean value of the rough image, otherwise zero. We set the value of dark-to-light sparse adjacency matrix of graph weights to zero where the ROI is zero, otherwise the value is not changed.

The procedure described above helps with indicating the region to find the shortest path because zero edge weight means unconnected nodes. Next, we use Dijkstra’s algorithm and the dark-to-light sparse adjacency matrix of graph weights to find the shortest paths. The start and the end points are automatically initialized to be the upper left pixel and the bottom right pixel, respectively. After the first layer is segmented, we set the pixels of the first found layer on the ROI matrix to zero in order to segment the second rough layer. We iterate the process to find the second rough layer, setting the value of the dark-to-light sparse adjacency matrix of graph weights to zero where the ROI is zero; otherwise the same value is kept in order.

After the two rough layers are found, both of the rough layers are interpolated to have the same size as the resized image. Next, we set the layer where the mean value of the y-coordinate is smaller to be the ILM and the other layer to be the RPE. Then we use the ROI matrix of the same size as the resized image with the additional columns to find the precise ILM and RPE layers. First, we segment the ILM by setting the region of the ROI matrix near the rough ILM layer to one where the rest are zero. Also, we change the value of the dark-to-light sparse adjacency matrix of graph weights to zero where the ROI matrix is zero. Then again, we use Dijkstra’s algorithm and the dark-to-light sparse adjacency matrix of graph weights to find the precise ILM layer. The precise RPE is found in the same manner as the ILM. Figure 5-7shows the ROI images for finding the rough layers and the precise ILM and RPE layers on the resized image with two additional columns. The ROI images are 55x102 pixels whereas the resized image is 110x202 pixels.

Figure 5-7 (a) ROI for the first rough layer (b) ROI for the second rough layer and (c) precise ILM and RPE layers on the resized image with two additional columns.

5.1.6 Limiting the Search Region for the other layers

As mentioned earlier, due to the hyper-reflectivity of the ILM and RPE layers, they are easily segmented. In contrast, the remaining layers are not as prominent because their characteristics (relative intensity) are similar. In order to correctly segment the targeted layer, the search region is limited such that the irrelevant features are excluded. This exclusion is accomplished by setting the weight of the non-targeted features to zero before segmenting the graph using Dijkstra's algorithm. The search space of each layer is selected based on the previously segmented layers. The order of layer boundaries to be segmented is Inner Nuclear Layer/Outer Plexiform Layer (INL/OPL), NFL/GCL, Inner Plexiform Layer/Inner Nuclear Layer (IPL/INL), and Outer Plexiform Layer/Outer Nuclear Layer (OPL/ONL). Table. 5-1 shows the upper and lower boundaries and the sparse adjacency matrix of graph weights for segmentation along with the sparse adjacency matrix. Each boundary requires two previously segmented boundaries to be the upper and the lower bounds to limit the search region as indicate in Table. 5-1:

Retinal Layer	Upper Bound	Lower Bound	Sparse Adjacency Matrix
INL/OPL	ILM	RPE	Dark-to-light
NFL/GCL	ILM	INL/OPL	Light-to-dark
IPL/INL	NFL/GCL	INL/OPL	Light-to-dark
OPL/ONL	INL/OPL	RPE	Light-to-dark

Table 5-1 Upper and lower bound for each retinal layer along with its sparse adjacency matrix.

Figure 5-8 shows the ROIs for finding the shortest path for INL/OPL, NFL/GCL, IPL/INL and OPL/ONL.

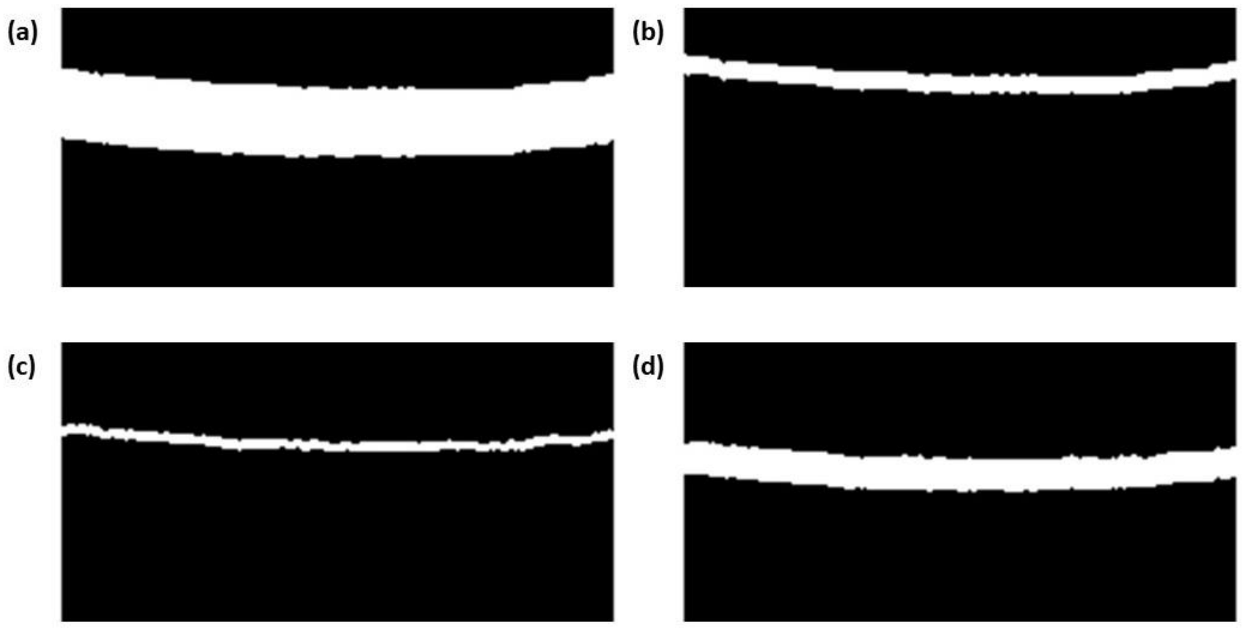


Figure 5-8 (a) ROI for segmenting the INL/OPL (b) ROI for segmenting the NFL/GCL (c) ROI for segmenting the IPL/INL and (d) ROI for segmenting the OPL/ONL.

5.1.7 Final Segmented Image

Once all of the retinal layer boundaries are segmented, the additional column on each side of the image is removed, leaving the accurate six retinal layer boundaries. Nonetheless, these retinal layers are not the same size as the original input image, and they need to be interpolated, smoothed and offset adjusted based on the cropped background to correctly delineate these features on the original uncropped image. Figure 5-9 shows representative results of the retinal layer boundaries delineated on mouse cross-sectional retinal images.

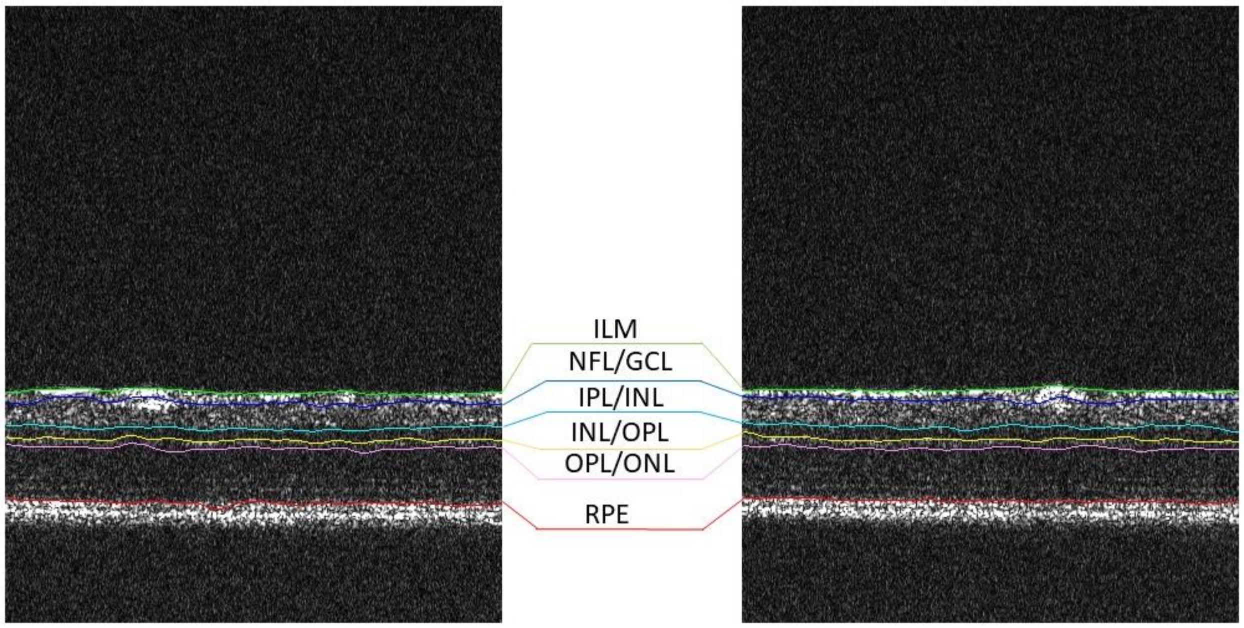


Figure 5-9 The original SD-OCT mouse B-scans with retinal layer boundaries segmented by our segmentation.

5.2 Summary

In this chapter, we described the processing pipeline of the retinal layer segmentation using heterogeneous computing (CPU and GPU). We performed image cropping on the GPU because of its parallel computing ability in calculating the row average and performing the histogram to find the threshold value. However, we performed the rest of the pipeline in the CPU due to the complexity of the graph construction and the shortest path search using Dijkstra’s algorithm. The results of the retinal layer segmentation and the discussion will be covered in the next chapter.

Chapter 6

Retinal Layer Segmentation Results and Discussion

This chapter describes the environmental setup for implementing our real-time retinal layer segmentation program, the speed performance for a real-time application and the segmentation results. Also, it will compare the Sensorless Adaptive Optics (SAO) optimization with and without our segmentation program using image quality or the sum of the intensity squared of each pixel of the *en face* image.

6.1 Environmental Setup

The mouse retinal image acquisition system called LIVMAOS *System* at Biomedical Optics Research Group (BORG) lab operates with a central wavelength of 810 nm. It provides a 100 kHz line rate for imaging *vivo* mouse retina. This system includes OCT, OCT-A, SLO and fluorescence detector but for this research we only use OCT and OCT-A subsystems. On the other hand, the software that controls the LIVMAOS *System* is called OCTViewer converts raw interferometric fringe into B-scan images and displays on the device. Our segmentation program is integrated in the OCTViewer such that the OCTViewer takes the segmentation results to generate *en face* OCT and OCT-A images. In order to correct the optics aberrations, the OCTViewer calculates the sum of the intensity squared of each pixel of the *en face* OCT and feeds it as the input for the merit function to optimize SAO.

6.2 Data for this project

The mouse imaging experiments were performed under protocols compliant to the Canadian Council on Animal Care, and with the approval of the University Animal Care Committee at Simon Fraser University. In this thesis, we used the mouse SD-OCT volume datasets of different dimensions (as indicated below), but each dataset contained 800 frames.

6.3 Speed Performance

This section compares the speed performance of the modified *Caserel* software using MATLAB R2019a with our real-time retinal layer segmentation written in C/C++ using Microsoft Visual Studio 2013. Both of the programs run on the CPU of Intel® Core^{TM} i9-9900K CPU @ 3.6 GHz (Turbo 5.0 GHz) with a Graphic Processing Unit of NVIDIA GeForce RTX 2060.

We modified the *Caserel* software as follows:

- Since our OCTViewer processes the interferometric fringe data into the processed floating-point data, we modified the *Caserel* software to read these types of input.
- We changed the resize scale for the rough image from 0.5 to be 0.2 according to our C++ algorithm because if the image is too small, the rough ROI for searching the second rough layer may generate the broken path resulting to the second rough layer not found.

Table. 6-1 shows the accumulated average speed performance of our segmentation up to the specified layer from 800 images in the same volume set using our C++ implementation and the modified version of *Caserel* software on SD-OCT data of different sizes. The order of retinal layers to be segmented is rough ILM and RPE, precise ILM, precise RPE, INL/OPL, NFL/GCL, IPL/INL and OPL/ONL.

Retinal Layer	992x400 pixels		496x400 pixels		240x300 pixels	
	Caserel	C++	Caserel	C++	Caserel	C++
Rough ILM and RPE	79.07 ± 21.10 ms	10.34 ± 0.82 ms	51.88 ± 8.20 ms	9.17 ± 0.37 ms	21.13 ± 7.50 ms	5.30 ± 0.47 ms
ILM	106.76 ± 9.80 ms	12.49 ± 0.86 ms	64.99 ± 8.40 ms	11.4 ± 0.50 ms	28.22 ± 7.70 ms	6.81 ± 0.47 ms
RPE	122.90 ± 9.80 ms	14.56 ± 0.96 ms	77.40 ± 8.70 ms	13.57 ± 0.51 ms	35.26 ± 7.80 ms	8.26 ± 0.47 ms
INL/OPL	140.45 ± 10.60 ms	17.78 ± 1.02 ms	91.46 ± 10.10 ms	18.88 ± 0.46 ms	43.91 ± 8.20 ms	10.60 ± 0.68 ms
NFL/GCL	156.08 ± 10.60 ms	19.29 ± 1.04 ms	104.63 ± 11.10 ms	20.9 ± 0.51 ms	51.46 ± 8.70 ms	11.74 ± 0.66 ms
IPL/INL	172.75 ± 10.70 ms	20.63 ± 1.13 ms	117.46 ± 11.70 ms	22.42 ± 0.57 ms	58.29 ± 8.90 ms	12.30 ± 0.81 ms
OPL/ONL	190.07 ± 11.30 ms	22.23 ± 1.07 ms	130.53 ± 13.20 ms	25.60 ± 0.61 ms	66.29 ± 9.30 ms	13.76 ± 0.72 ms

Table 6-1 Speed performance of our C++ segmentation and Caserel in milliseconds on different image sizes

The main objective of this project is to implement a pseudo-real-time retinal layer segmentation program that gives its result within a specific time constraint. We employed heterogeneous computing to gain the performance based on the nature of each task. CPUs are good at performing complex tasks, and in the context of Graph-cut search, are more reliable in terms of global convergence. In contrast, GPUs are optimized for performing tasks of lesser complexity that can be broken down into smaller independent parts that need less or no communication among tasks. We parallelized the code for cropping the image to contain only the retinal structure and to remove the redundant data by implementing it into CUDA for NVIDIA GPU. Removing redundant data helps Dijkstra's algorithm, which is performed on the CPU due to its capability to perform complex tasks, and to search for the shortest path faster because of the smaller (resized) image size. Moreover, our retinal layer segmentation program utilizes arrays and Intel® Math Kernel Library (MKL) rather than vectors and their operations. Arrays take less time to access their elements because of their contiguous property, and permits access to the elements efficiently with a constant time irrespective of the element location. Since the array is a fixed size data structure, and all elements must be of the same type, hence, it is type safe and the most efficient in terms of speed and performance.

The specifications of the segmentation time requirements (including the OCT signal processing time) were set by the acquisition system parameters. The image acquisition system used in this report provided a 100 kHz A-line scan rate, and completed a B-scan where the width is 400 A-lines in 4 ms and a B-scan where the width is 300 A-lines in 3 ms, corresponding to B-scan rates of ~250-333Hz. In order to maximize the data transfer across the PCIe bus, the B-scans are acquired in batches. Therefore, in order to ensure that the combination of our CUDA and segmentation pipelines (the processing pipeline) responds within the deadline of the acquisition system, the processing pipeline must guarantee the result of the current batch before the next batch is completely acquired. Our segmentation using C/C++ showed a significant improvement in the processing time of a B-scan by more than 74%, and is able to perform the rough segmentation on each (cropped) B-scan at the rate that it is acquired. We found that the lateral movement of the retinal layers within the acquisition time of a batch is negligible. Thus, we reduced the number of frames to be segmented by applying the full resolution segmentation using Graph-cuts to the first frame of each batch and its result can be applied on all frames in that batch. If we chose to segment only the ILM and RPE layers and chose a batch to contain 4 frames, the deadline is about 16 ms for the image where the width is 400 pixels and 12 ms for the image where the width is 300 pixels. As shown in Table 6-1, the execution time to segment the rough and the precise ILM and RPE on an image of size 992x400 and 496x400 takes about 14.56 ms and 13.57 ms, respectively, and on an image of size 240x300 takes about 8.26 ms. As a result, our segmentation pipeline can run at least at 62.5 Hz which is faster than a video rate of 60 Hz. However, we must consider the OCT acquisition and signal processing time along with the image segmentation time to respond within the deadline of the acquisition system. Thus, we chose a batch size to contain 6 frames which extends the deadline to 24 ms for the image where the width is 400 pixels and 18 ms for the image where the width is 300 pixels to include the OCT signal acquisition and processing time. This feature would provide axial tracking and extraction of the shape of the retina for visualization during acquisition. If we wanted to segment a specific retinal layer, we could offset the width between ILM and RPE layers to generate an *en face* image that is close to the anatomical shape of that layer. The segmentation of additional layers would require larger batch sizes. For example, we also could segment all six retinal layers and choose the batch size to be 10 frames to generate the results before the deadline of 40 ms and 30 ms for the images where the width is 400 pixels and 300 pixels, respectively. Indeed, in most cases, we do not need all six retinal layers delineated on the image while performing a real-time application. What we often need is to segment a particular layer by using two segmented retinal layer boundaries. For instance, the boundaries between INL/OPL and OPL/ONL are needed for segmenting the OPL layer. We could speed up our segmentation pipeline by segmenting only the rough ILM and RPE layers and use these rough layers to limit the search region to segment INL/OPL and again use INL/OPL and rough RPE to get OPL/ONL. Similarly for the NFL layer, we could segment the rough layers and use the rough layers to segment INL/OPL and again use rough ILM and INL/OPL to segment NFL/GCL.

6.4 Qualitative results of the Real-Time Retinal Layer Segmentation on SAO mouse datasets

This section shows the real-time retinal layer segmentation results of different retinal layers on different SAO mouse datasets. The top row shows the SAO cross-sectional mouse retina images with two segmented retinal layer boundaries. The middle and bottom rows on each column display the intensity and speckle variance *en face* images generated from the two segmented layer boundaries shown on the corresponding B-scan. Figure 6-1 shows the results from segmenting the ILM layer (green line) and the RPE layer (red line) along with their OCT and OCT-A *en face* images from different datasets. Figure 6-2 shows the segmentation results of the ILM layer (green line) and the NFL/GCL layer (red line) with their OCT and OCT-A *en face* images. Figure 6-3 displays the segmentation result of the NFL/GCL layer (green line) and the IPL/INL layer (red line) along with their OCT and OCT-A *en face* images from different datasets. Figure 6-4 shows the results of the INL/OPL layer (green line) and the OPL/ONL layer (red line) along with their OCT and OCT-A *en face* images from different datasets. Figure 6-5 shows the results from segmenting the IPL/INL layer (green line) and the OPL/ONL layer (red line) along with their OCT and OCT-A *en face* images from different datasets.

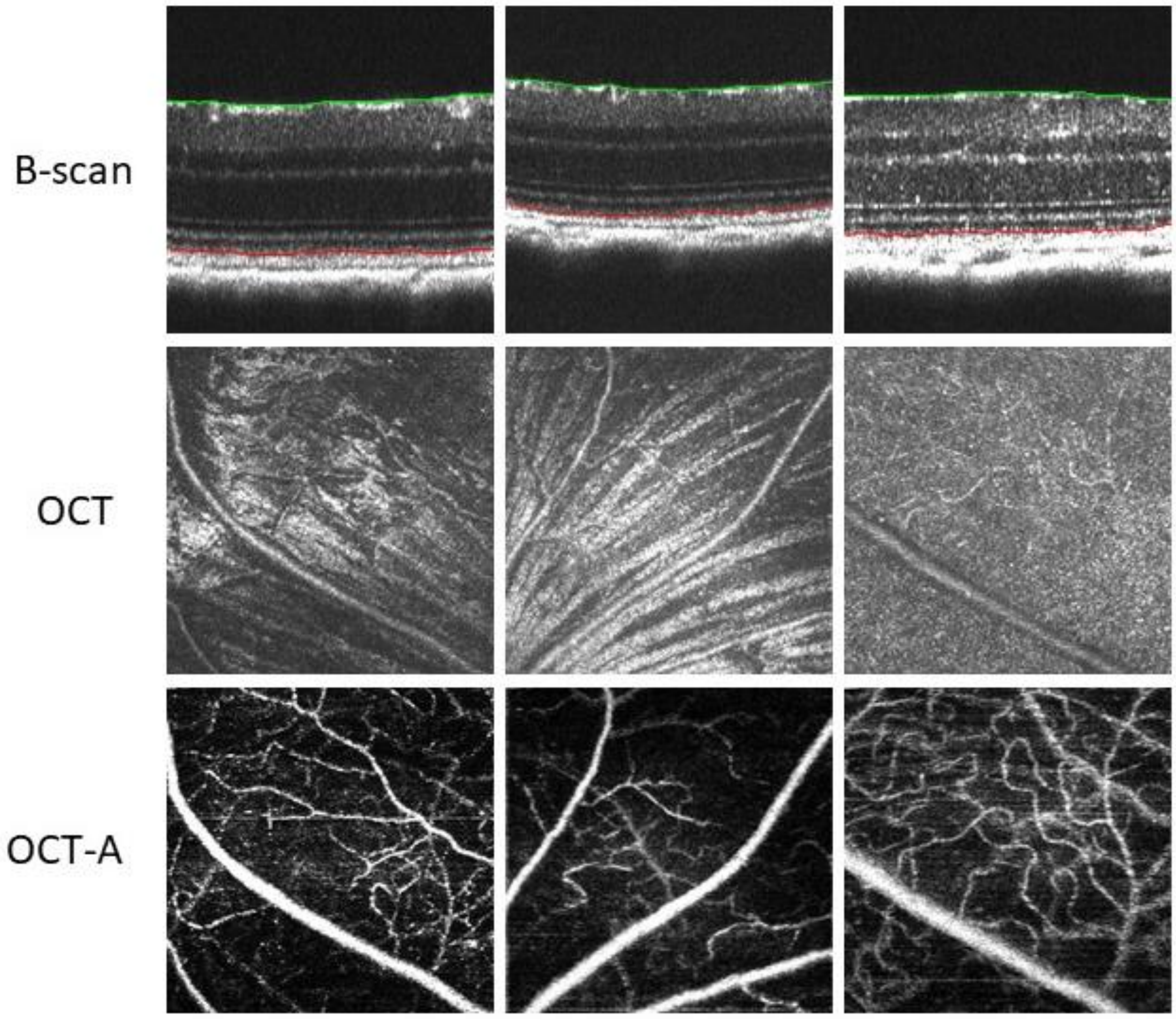


Figure 6-1 SAO-OCT B-scans with ILM (green layer) and RPE (red layer) and their corresponding SAO-OCT and SAO-OCTA en face images.

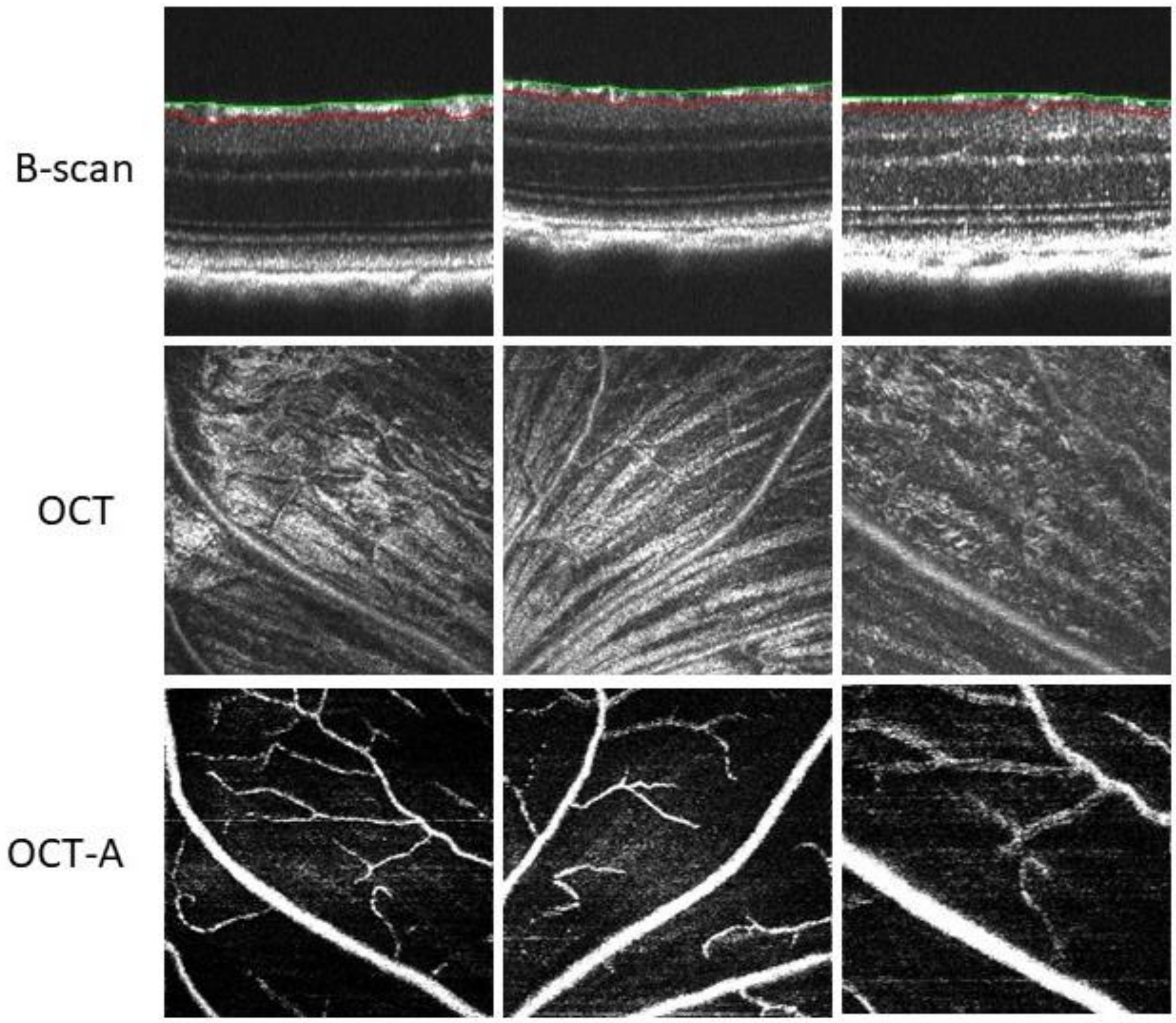


Figure 6-2 SAO-OCT B-scans with ILM (green layer) and NFL/GCL (red layer) and their corresponding SAO-OCT and SAO-OCTA en face images.

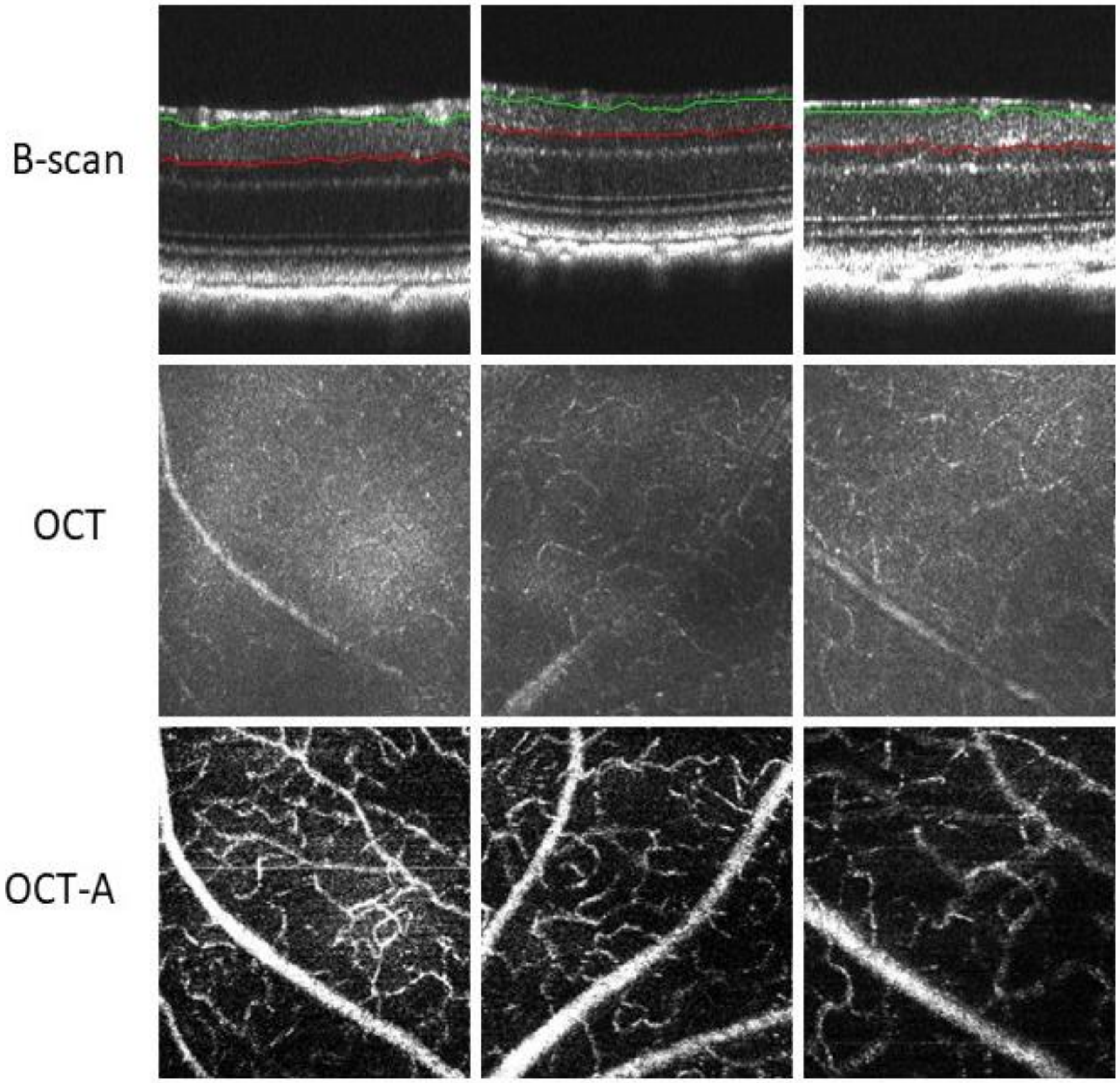


Figure 6-3 SAO-OCT B-scans with NFL/GCL (green layer) and IPL/INL (red layer) and their corresponding SAO-OCT and SAO-OCTA en face images.

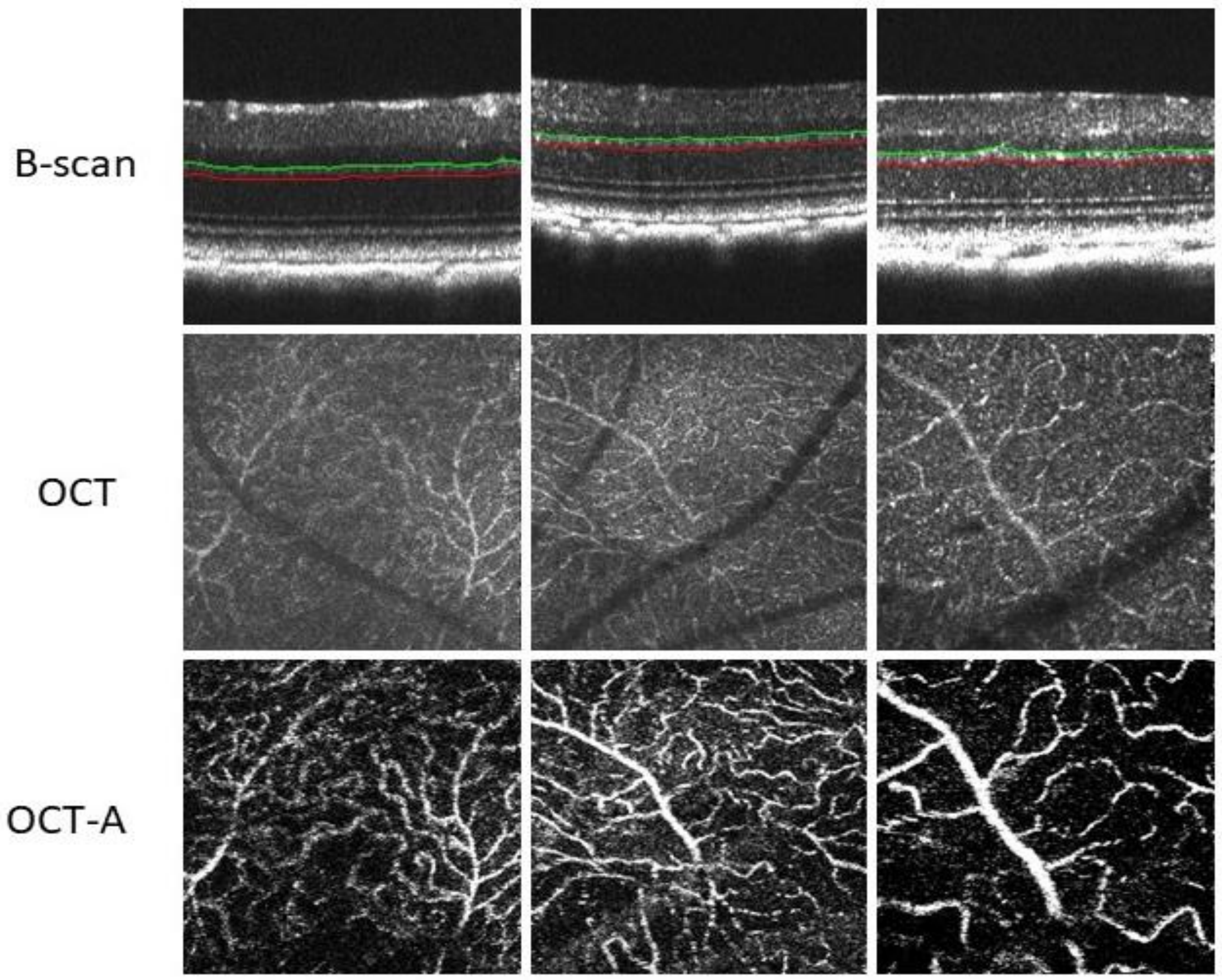


Figure 6-4 SAO-OCT B-scans with INL/OPL (green layer) and OPL/ONL (red layer) and their corresponding SAO-OCT and SAO-OCTA en face images.

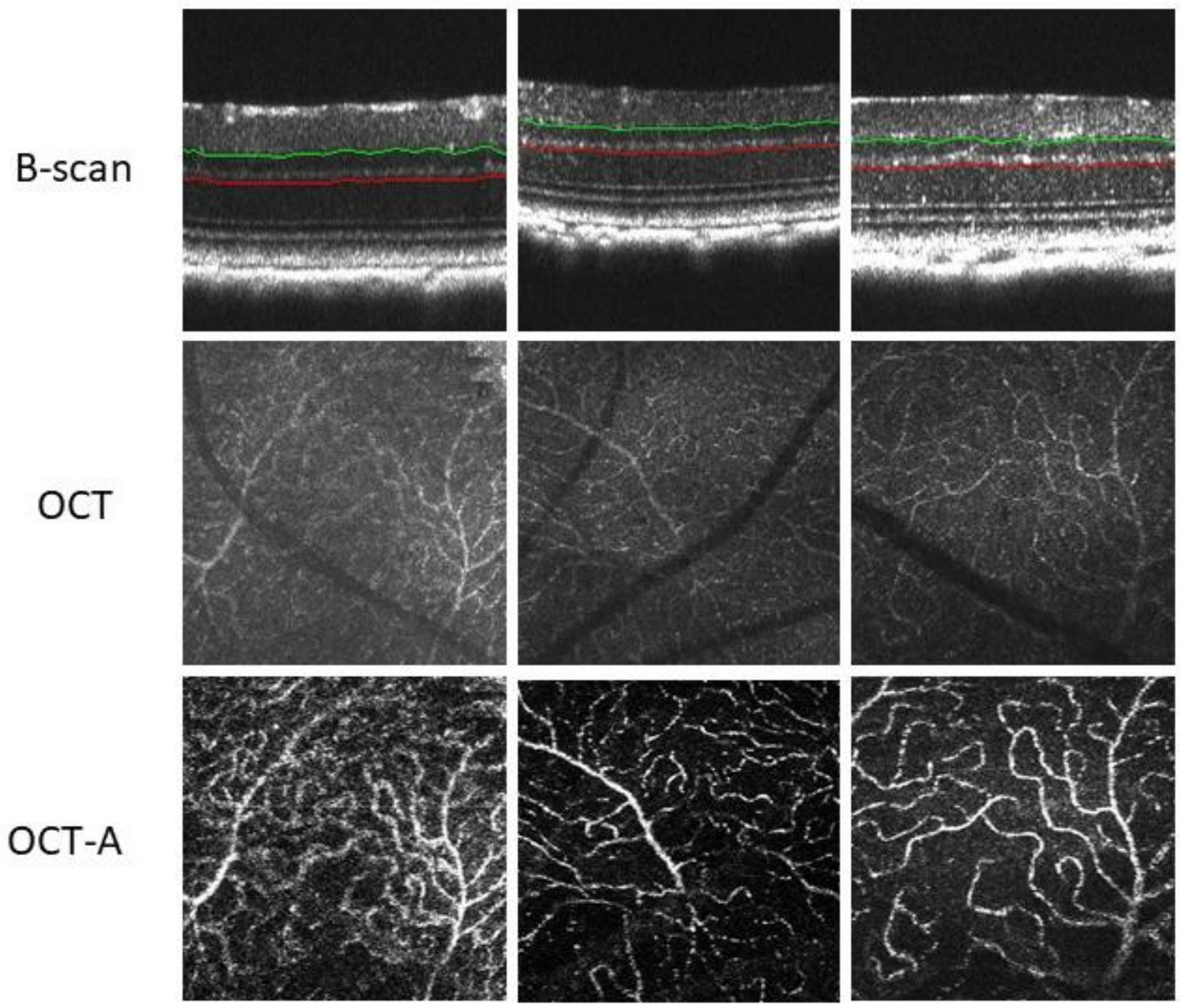


Figure 6-5 SAO-OCT B-scans with IPL/INL (green layer) and OPL/ONL (red layer) and their corresponding SAO-OCT and SAO-OCTA en face images.

6.5 Static (fixed) User-Selected Depth Retinal Layers versus Pseudo-Real-Time Retinal Layer Segmentation

This section compares the results of the intensity and speckle variance *en face* images when using static (fixed) depth locations to generate the *en face* images versus using the pseudo-real-time retinal layer segmentation. With the static user-selected depth option, the operator selects two horizontal lines on the B-scan image. Figure 6-6 (a) shows a representative B-scan with the static user-selected depth at NFL layer and its corresponding *en face* images.

Figure 6-6 (b) shows a representative B-scan with the pseudo-real-time retinal layer segmentation results of ILM and NFL/GCL layers with its OCT and OCT-A *en face* images. Although the image are similar, the top row is missing a vasculature feature near the top edge of the OCT-A image.

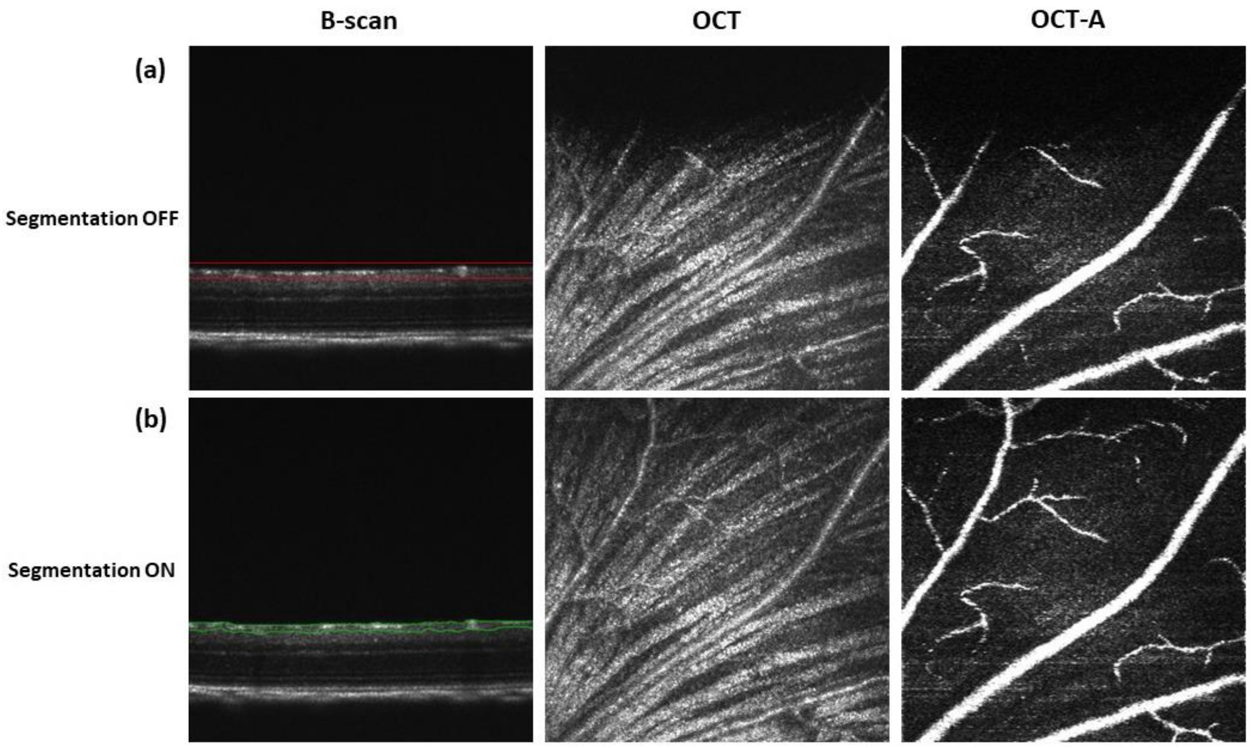


Figure 6-6 Intensity and Speckle variance NFL en face images using static user-selected depth of retinal layers and using the pseudo-real-time retinal layer segmentation.

Mouse retinal axial motion during imaging is typically on the order of 4.34 pixels during imaging due to breathing and related motion. Hence, without tracking of the retina position, the region of interest may shift in and out of the bounding box. Figure 6-7 shows a representative case of when the static user-selected depth option retinal layers option failed to detect the retinal layer of interest during data acquisition due to motion in *vivo* mouse.

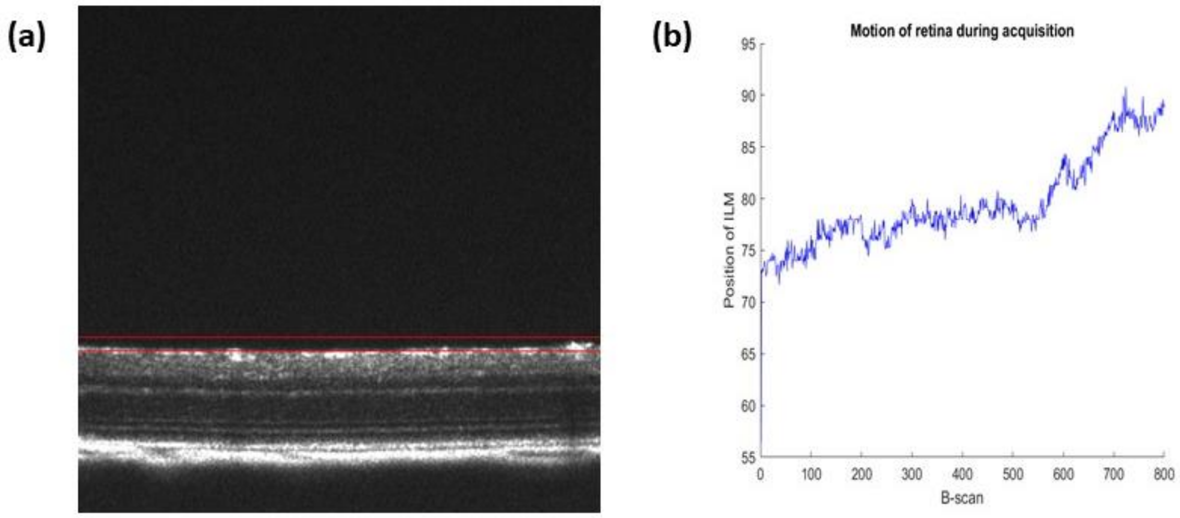


Figure 6-7 (a) Axial motion of the retina during acquisition leads to failure in detecting NFL layer using two fixed horizontal lines and (b) the graph showing the motion of the ILM position (in pixel) during acquisition.

Similarly, Figure 6-8 (a) shows the representative B-scan with the static user-selected depth at INL-OPL layer and its corresponding *en face* images. Figure 6-8 (b) shows the representative B-scan with the pseudo-real-time retinal layer segmentation results of IPL/INL and OPL/ONL layers with its OCT and OCT-A *en face* images.

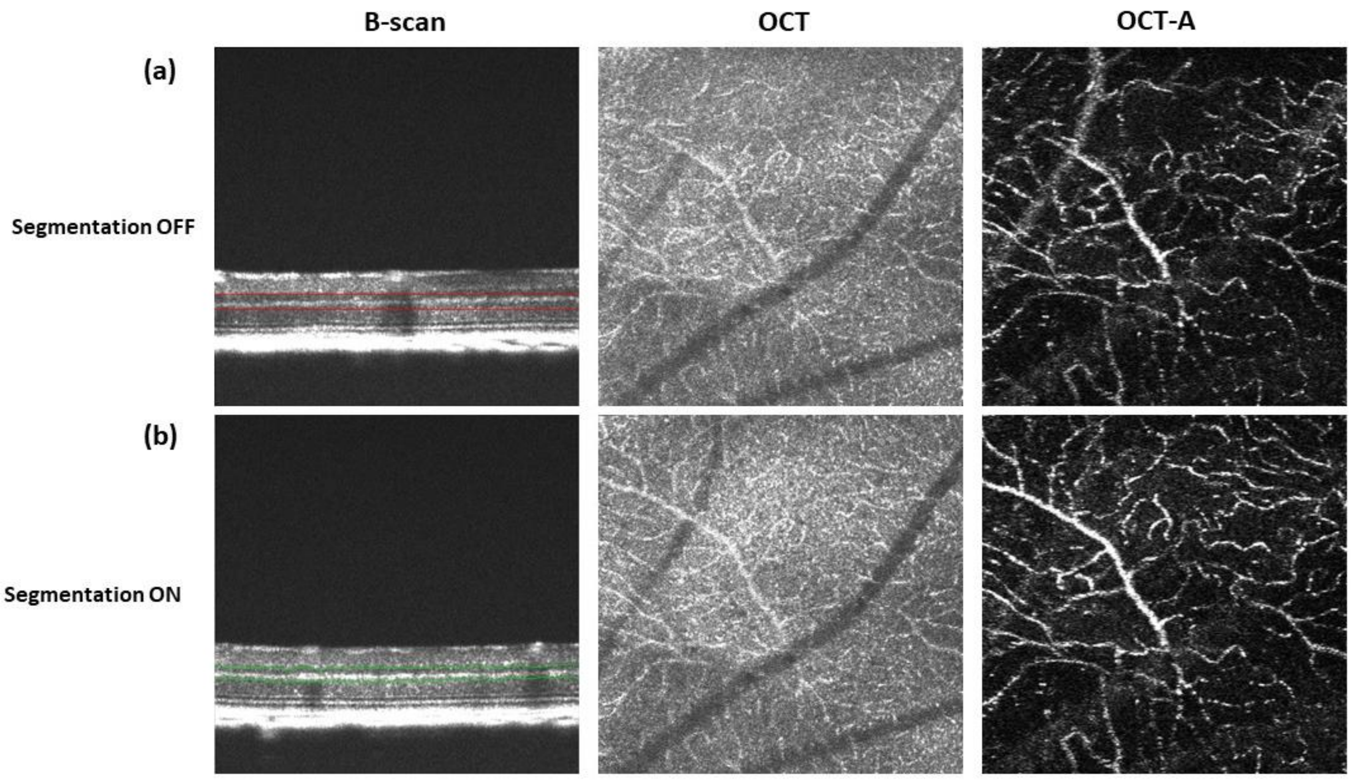


Figure 6-8 Intensity and Speckle variance INL-OPL en face images using static user-selected depth of retinal layers and using the pseudo-real-time retinal layer segmentation.

The results for NFL imaging SAO-OCT, and SAO-OCT-A visualization with and without our pseudo-real-time segmentation are shown in Figure 6-9. Without our segmentation, an operator selected two straight lines within the region of interest in order to obtain *en face* images. Figure 6-9 (a) shows *en face* SAO-OCT and SAO-OCT-A images with static user-selected depth at NFL layer. Figure 6-9 (b) shows *en face* SAO-OCT and SAO-OCT-A images with pseudo-real-time retinal layer segmentation between the ILM and NFL/GCL layers. Figure 6-9 (c) demonstrates the SAO optimization with and without the segmentation.

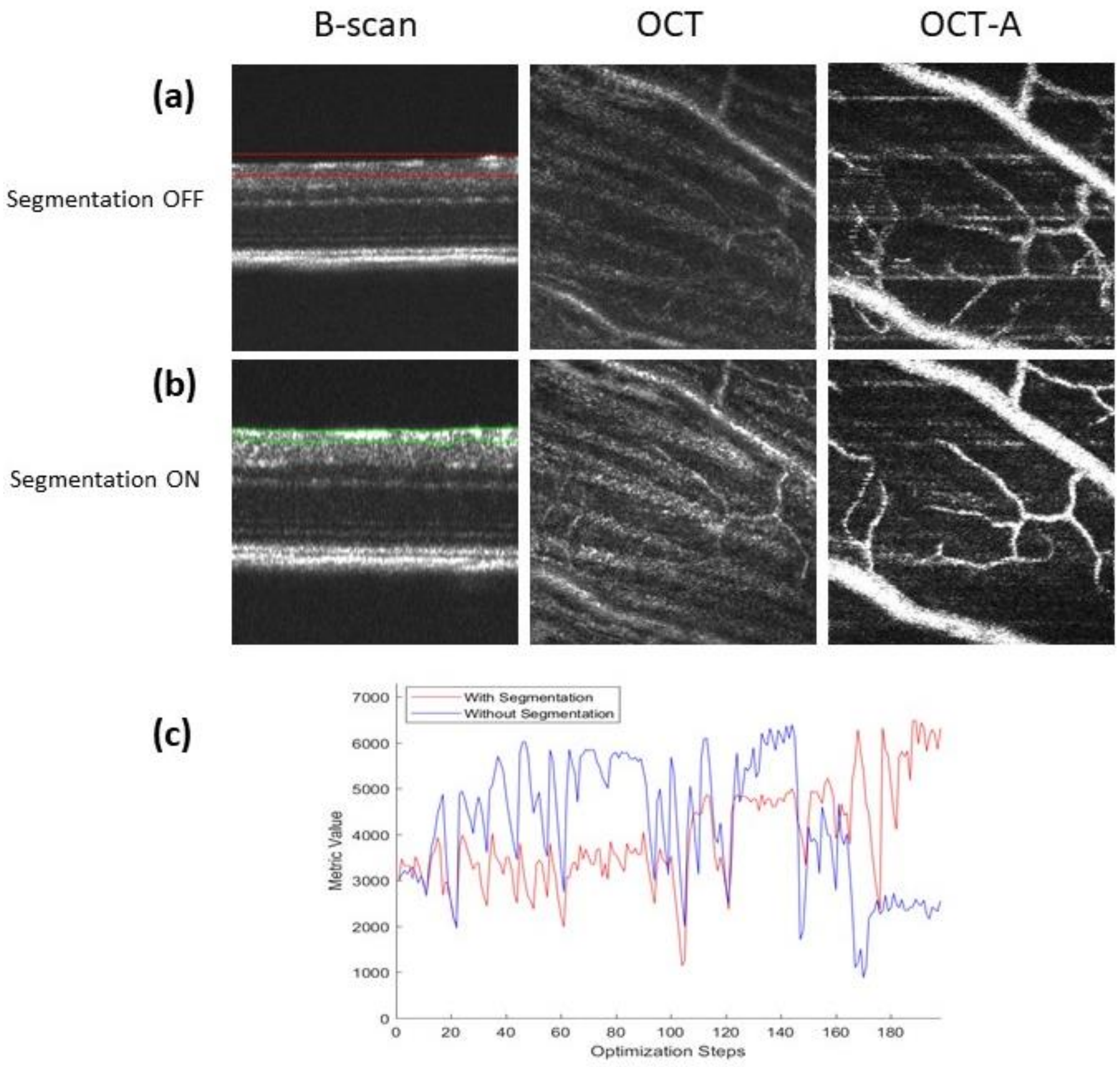


Figure 6-9 (a) B-scan with user selected retinal depth on NFL layer and its corresponding OCT and OCT-A en face images (b) B-scan with the segmentation of ILM and NFL/GCL layers and its corresponding OCT and OCT-A en face images (c) Image quality of each step in the SAO optimization with segmentation (the red plot) and without segmentation (the blue plot)

The results for OPL imaging with SAO-OCT, and SAO-OCT-A visualization with and without our pseudo-real-time segmentation are shown in Figure 6-10. Figure 6-10 (a) shows *en face* SAO-OCT and SAO-OCT-A images with static user-selected depth at OPL layer. Figure 6-10 (b) shows *en face* SAO-OCT and SAO-OCT-A images with pseudo-real-time retinal layer segmentation between the INL/OPL and OPL/ONL layers. Figure 6-9 (c) demonstrates the SAO optimization with and without the segmentation.

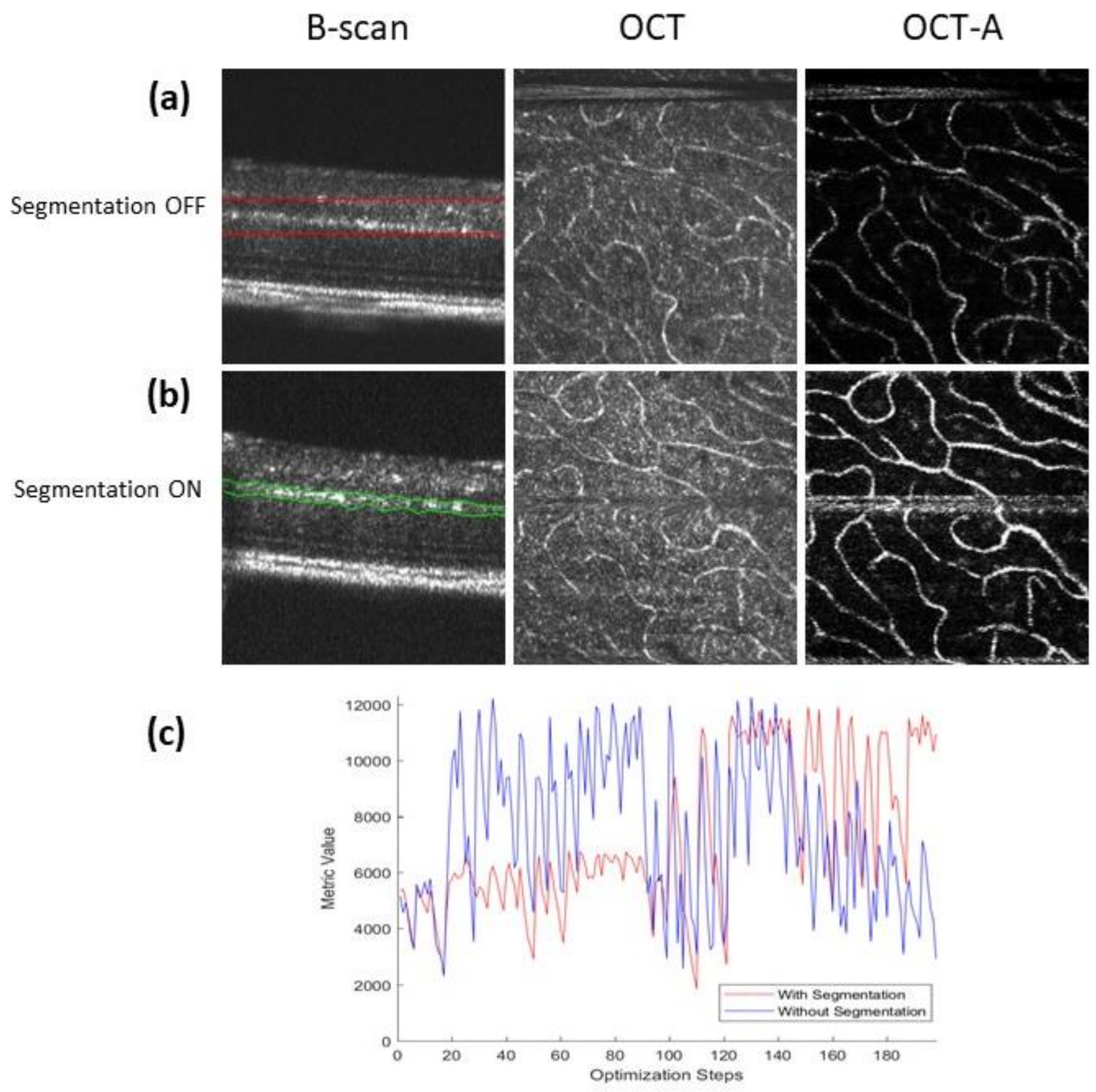


Figure 6-10 (a) B-scan with user selected retinal depth on OPL layer and its corresponding OCT and OCT-A en face images (b) B-scan with the segmentation of INL/OPL and OPL/ONLlayers and its corresponding OCT and OCT-A en face images (c) Image quality of each step in the SAO optimization with segmentation (the red plot) and without segmentation (the blue plot)

In this thesis, we implemented a pseudo-real-time 2D Graph-cut for retinal layer segmentation which was integrated into a complete OCT acquisition and processing software. We demonstrated the system performance for generating layer segmentations to guide an image-based SAO optimization. With the ability to segment the retinal layers, our processing program [3] created anatomically correct *en face* images by using a Maximum Intensity Projection (MIP) between two selected retinal layers for the input of the merit function for SAO optimization on the screen as the data was being acquired. The use of a static user-selected depth region of interest is not ideal to detect specific retinal layers because, for *in vivo* mouse imaging, the retinal position moves during the acquisition as shown in Figure 6-7. Figure 6-6 shows that the top of the NFL *en face* images are dark because during the acquisition, the retina moved down and out of the bounding region, causing the posterior chamber to be detected. In contrast, the pseudo-real-time retinal segmentation generated *en face* images with the correct retinal features because the segmented lines followed the motion of the retina *in vivo*. Similarly, in Figure 6-8, the *en face* images generated from the static user-selected depth retinal layers show fewer retinal features compared to the *en face* images generated from the retinal layer segmentation. The pseudo-real-time retinal layer segmentation tracks the layer of interest even though there is motion during acquisition which leads to a better *en face* image as the input for the SAO optimization as shown in Figure 6-9 and Figure 6-10. Figure 6-9 (d) and Figure 6-10 (d) shows the merit plots with and without our segmentation for SAO optimization of 18 modes in total. The merit plot with segmentation shows a better improvement in the image quality after each mode is optimized than the merit plot without the segmentation.

6.6 Summary

In this chapter, the speed performance to segment six retinal layers of our retinal layer segmentation was at least 74 percent faster than the speed of the modified *Caserel* software. In order to make our retinal layer segmentation a real-time application, we could choose to segment only the first two layers (i.e., ILM and RPE layers) and use the results for the entire batch where each batch consists of 6 frames. For segmenting the other layers rather than ILM and RPE layers, the user could offset the width between ILM and RPE layers to fit the layer of interest to produce the *en face* image. This option makes the segmentation pipeline to run at least at 62.5 Hz for the images of sizes of 992x400, 496 and 240x300 pixels. On other hand, we could segment only rough layers and layers that the layer of interest depends on to speed up the segmentation pipeline.

Furthermore, our real-time retinal layer segmentation provides the correct retina shape than using user-selected two fixed horizontal lines when there is motion during data acquisition or when the retina is not flat. As the result, the generated intensity *en face* images from our real-time retinal layer segmentation provide better image quality after the SAO optimization.

Chapter 7

Conclusion and Future Work

We have utilized consumer grade computer equipment to control the acquisition of real-time OCT signals and perform image processing. A General-Purpose Graphics Processing Unit (GPGPU) was used for OCT processing and for generating the B-scan images. The CPU was used for the processes such as shortest-path graph search to segment the retinal layers on the acquisition computer and extract a depth resolved layer from the volumetric data as it gets acquired. We employed down-sampling and parallel processing to improve the speed of the application and as a result our retinal segmentation program can be used as a pseudo-real-time application. However, the segmenting time strongly depends on the number of the retinal layers to be segmented. Segmentation of the retinal layers permits OCT and OCT-A *en face* images to be extracted during data acquisition and *en face* images are used to guide the depth specific focus control for high-solution OCT systems.

7.1 Future Work

In this project, we implemented the pseudo-real-time retinal layer segmentation in C/C++ using Visual Studio 2013 running on a CPU of Intel® Core^{TM} i9-9900K CPU @ 3.6 GHz (Turbo 5.0 GHz) with a Graphic Processing Unit of NVIDIA GeForce RTX 2060. In the future, we could consider an alternative computational hardware for OCT signal processing and retinal layer segmentation. Our current approach focused on the use of GPU and CPU because of the ease of implementation, ability for rapid changes to accommodate emerging applications, and for integration with relatively complex control of adaptive optics systems. Future work may involve the use of Field Programmable Gate Arrays (FPGAs). The OCT processing pipeline has been developed on FPGA [51]–[55]. With its advantage of the execution time and power consumption, an FPGA platform could also be a suitable option for speeding up the retinal layer segmentation. This approach would need to consider implementing a Graph-cut or some other efficient methods to segment the retinal layers on a FPGA in real-time. To the best of our knowledge, there is no work of retinal layer segmentation that uses FPGA; however, there are some works on image segmentation using Graph-cut on FPGA [56], [57]. There are reports on related applications, such as retinal vessel segmentation, using a hardware-based approach with high accuracy [58], [59]. Koukounis *et al*. [58] presented a very-large-scale integration (VLSI) retinal vessel segmentation system using a matched filter approach, introduced by Hoover *et al*. [60], on a Spartan 6 XC65LX150T FPGA running at 100 MHz. It claimed to segment with the overall accuracy of 92.4% and the execution time for image of 768x584 pixels was 48 ms. Similarly, Bendaoudi *et al*. [59] proposed two architectures for retinal blood vessel segmentation which one of them was completely hardwared on a Xilinx Kintex-7 (XC7K48oT-1FFG1156) FPGA. The algorithm was written in hardware description language (HDL) using a matched filter [60] and it could segment retinal vessel image of size 640 x 480 pixels in 2 ms and image size of 3504 x 2336 pixels in 54 ms. Due to the low power consumption, high speed performance and easy architecture integration, the FPGA platforms are proposed to be used as portable embedded biomedical systems. Nevertheless, the FPGA platforms do not support a floating-point operation which makes the systems lose accuracy [61].

Recently, deep learning based neutral networks approach have been introduced to successfully perform the retinal layer segmentation based on learning representative data [62]–[68]. Yufan *et al.* [64] claimed that their full segmentation operation could segment nine retinal layer boundaries in 10 seconds for a 3D volume (496x1024x49). Besides, Ngo et al. [67] proposed a deep neural network regression that used the intensity, gradient and adaptive normalized intensity score of an image to segment eight retinal boundaries in 10.596 seconds per image with approximate accuracy of 0.966. In addition, there are reports that combines neutral network with graph search methodology to automatically segment retinal layers [69], [70] in order to improve both segmentation accuracy and stability but with the higher cost of the computational burden. The deep learning based neutral network method could be our future alternative retinal layer algorithm that utilizes a large learning data produced by our pseudo-real-time retinal layer segmentation.

Reference

[1] “Why good vision is so important.” [Online]. Available: https://www.zeiss.com/vision-care/int/better-vision/health-prevention/why-good-vision-is-so-important-.html. [Accessed: 04-Nov-2019].

[2] “History of Ophthalmic Photography Blog - Ophthalmic Photographers’ Society.” [Online]. Available: https://www.opsweb.org/blogpost/1033503/History-of-Ophthalmic-Photography-Blog?tag=first+fundus+photograph. [Accessed: 04-Nov-2019].

[3] Y. Jian, K. Wong, and M. V. Sarunic, “Graphics processing unit accelerated optical coherence tomography processing at megahertz axial scan rate and high resolution video rate volumetric rendering,” *J. Biomed. Opt.*, 2013.

[4] J. Xu, K. Wong, Y. Jian, and M. V. Sarunic, “Real-time acquisition and display of flow contrast using speckle variance optical coherence tomography in a graphics processing unit,” *J. Biomed. Opt.*, 2014.

[5] D. Xu, Y. Huang, and J. U. Kang, “GPU-accelerated non-uniform fast Fourier transform-based compressive sensing spectral domain optical coherence tomography,” *Opt. Express*, 2014.

[6] N. H. Cho *et al.*, “High speed SD-OCT system using GPU accelerated mode for in vivo human eye imaging,” *J. Opt. Soc. Korea*, 2013.

[7] Y. Wang, C. M. Oh, M. C. Oliveira, M. S. Islam, A. Ortega, and B. H. Park, “GPU accelerated real-time multi-functional spectral-domain optical coherence tomography system at 1300nm,” *Opt. Express*, 2012.

[8] M. Sylwestrzak *et al.*, “Real time 3D structural and Doppler OCT imaging on graphics processing units,” in *Optical Coherence Tomography and Coherence Domain Optical Methods in Biomedicine XVII*, 2013.

[9] X. Wei *et al.*, “Real-time cross-sectional and en face OCT angiography guiding

high-quality scan acquisition," *Opt. Lett.*, 2019.

[10] J. Li, P. Bloch, J. Xu, M. V. Sarunic, and L. Shannon, "Performance and scalability of Fourier domain optical coherence tomography acceleration using," *Appl. Opt.*, 2011.

[11] M. Pircher and R. J. Zawadzki, "Review of adaptive optics OCT (AO-OCT): principles and applications for retinal imaging [Invited]," *Biomed. Opt. Express*, 2017.

[12] D. R. Williams, "Imaging single cells in the living retina," *Vision Research*. 2011.

[13] P. Godara, A. M. Dubis, A. Roorda, J. L. Duncan, and J. Carroll, "Adaptive optics retinal imaging: Emerging clinical applications," in *Optometry and Vision Science*, 2010.

[14] Y. Zhang, J. Rha, R. S. Jonnal, and D. T. Miller, "Adaptive optics parallel spectral domain optical coherence tomography for imaging the living retina," *Opt. Express*, 2005.

[15] R. J. Zawadzki *et al.*, "Adaptive-optics optical coherence tomography for high-resolution and high-speed 3D retinal in vivo imaging," *Opt. Express*, 2005.

[16] K. Kurokawa, Z. Liu, and D. T. Miller, "Adaptive optics optical coherence tomography angiography for morphometric analysis of choriocapillaris [Invited]," *Biomed. Opt. Express*, 2017.

[17] Z. Liu, J. Tam, O. Saeedi, and D. X. Hammer, "Trans-retinal cellular imaging with multimodal adaptive optics," *Biomed. Opt. Express*, 2018.

[18] R. S. Jonnal, O. P. Kocaoglu, R. J. Zawadzki, Z. Liu, D. T. Miller, and J. S. Werner, "A review of adaptive optics optical coherence tomography: Technical advances, scientific applications, and the future," *Investig. Ophthalmol. Vis. Sci.*, 2016.

[19] D. J. Wahl, R. Ng, M. J. Ju, Y. Jian, and M. V. Sarunic, “Sensorless adaptive optics multimodal en-face small animal retinal imaging,” *Biomed. Opt. Express*, 2019.

[20] M. Mujat *et al.*, “Retinal nerve fiber layer thickness map determined from optical coherence tomography images,” *Opt. Express,* 2005.

[21] A. Yazdanpanah, G. Hamarneh, B. R. Smith, and M. V. Sarunic, “Segmentation of intra-retinal layers from optical coherence tomography images using an active contour approach,” *IEEE Trans. Med. Imaging,* 2011.

[22] M. A. Mayer, J. Hornegger, C. Y. Mardin, and R. P. Tornow, “Retinal Nerve Fiber Layer Segmentation on FD-OCT Scans of Normal Subjects and Glaucoma Patients,” *Biomed. Opt. Express,* 2010.

[23] G. M. Somfai *et al.*, “Automated classifiers for early detection and diagnosis of retinopathy in diabetic eyes,” *BMC Bioinformatics*, 2014.

[24] S. J. Chiu, X. T. Li, P. Nicholas, C. A. Toth, J. A. Izatt, and S. Farsiu, “Automatic segmentation of seven retinal layers in SDOCT images congruent with expert manual segmentation,” *Opt. Express,* 2010.

[25] P. Teng, “Caserel - An Open Source Software for Computer-aided Segmentation of Retinal Layers in Optical Coherence Tomography Images.” [Online]. Available: https://www.researchgate.net/publication/308776805_Caserel_-_An_Open_Source_Software_for_Computer-aided_Segmentation_of_Retinal_Layers_in_Optical_Coherence_Tomography_Images. [Accessed: 03-Nov-2019].

[26] Q. Yang *et al.*, “Automated layer segmentation of macular OCT images using dual-scale gradient information,” *Opt. Express,* 2010.

[27] J. Tian, B. Varga, G. M. Somfai, W. H. Lee, W. E. Smiddy, and D. C. DeBuc, “Real-time automatic segmentation of optical coherence tomography volume data of the macular region,” *PLoS One,* 2015.

[28] J. P. Ehlers, S. K. Srivastava, D. Feiler, A. I. Noonan, A. M. Rollins, and Y. K. Tao, "Integrative advances for OCT-guided ophthalmic surgery and intraoperative OCT: Microscope integration, surgical instrumentation, and heads-up display surgeon feedback," *PLoS One*, 2014.

[29] J. P. Ehlers, Y. K. Tao, and S. K. Srivastava, "The value of intraoperative optical coherence tomography imaging in vitreoretinal surgery," *Current Opinion in Ophthalmology*. 2014.

[30] Y. K. Tao, J. P. Ehlers, C. A. Toth, and J. A. Izatt, "Intraoperative spectral domain optical coherence tomography for vitreoretinal surgery," *Opt. Lett.*, 2010.

[31] J. P. Ehlers, Y. K. Tao, S. Farsiu, R. Maldonado, J. A. Izatt, and C. A. Toth, "Integration of a spectral domain optical coherence tomography system into a surgical microscope for intraoperative imaging," *Investigative Ophthalmology and Visual Science*. 2011.

[32] J. Toth, CA; Carrasco-Zevallos, O; Keller, B; Shen, L; Viehland, C; Dong, HN; Hahn, P; Kuo, AN; Izatt, "Surgically integrated swept source optical coherence tomography (SSOCT) to guide vitreoretinal (VR) surgery | IOVS | ARVO Journals." [Online]. Available: https://iovs.arvojournals.org/article.aspx?articleID=2333387. [Accessed: 03-Nov-2019].

[33] O. M. Carrasco-Zevallos *et al.*, "Optical coherence tomography for retinal surgery: Perioperative analysis to real-time four-dimensional image-guided surgery," *Investig. Ophthalmol. Vis. Sci.*, 2016.

[34] Y. Huang, Z. Ibrahim, W. P. A. Lee, G. Brandacher, and J. U. Kang, "Real-time 3D Fourier-domain optical coherence tomography guided microvascular anastomosis," in *Medical Imaging 2013: Image-Guided Procedures, Robotic Interventions, and Modeling*, 2013.

[35] Y. Jian *et al.*, "Lens-based wavefront sensorless adaptive optics swept source

OCT," *Sci. Rep.*, 2016.

[36] T. Fabritius, S. Makita, M. Miura, Y. Yasuno, and R. Myllylä, "Fast retinal layer identification for optical coherence tomography images," in *Optical Coherence Tomography and Coherence Domain Optical Methods in Biomedicine XV*, 2011.

[37] J. Tian *et al.*, "Performance evaluation of automated segmentation software on optical coherence tomography volume data," *Journal of Biophotonics*. 2016.

[38] "Graphics processing unit | Computer Graphics | FANDOM powered by Wikia." [Online]. Available: https://graphics.fandom.com/wiki/Graphics_processing_unit. [Accessed: 05-Nov-2019].

[39] "NVIDIA GeForce 256 DDR Specs | TechPowerUp GPU Database." [Online]. Available: https://www.techpowerup.com/gpu-specs/geforce-256-ddr.c734. [Accessed: 05-Nov-2019].

[40] "CUDA - Wikipedia." [Online]. Available: https://en.wikipedia.org/wiki/CUDA. [Accessed: 04-Nov-2019].

[41] Y. Boykov and V. Kolmogorov, "An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. IEEE Transactions on," *Pattern Anal. Mach.*, 2004.

[42] "Dijkstra's algorithm - Wikipedia." [Online]. Available: https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm. [Accessed: 04-Nov-2019].

[43] "Push–relabel maximum flow algorithm - Wikipedia." [Online]. Available: https://en.wikipedia.org/wiki/Push–relabel_maximum_flow_algorithm. [Accessed: 04-Nov-2019].

[44] M. Miao and B. A. Sc, "Multi-GPU accelerated real-time retinal image segmentation," 2016.

[45] "CUDA Samples :: CUDA Toolkit Documentation." [Online]. Available:

https://docs.nvidia.com/cuda/cuda-samples/index.html#new-features-in-cuda-toolkit-6-5. [Accessed: 03-Nov-2019].

[46] L. Data, Y. Peng, L. Chen, W. Chen, and J. Yong, "JF-Cut : A Parallel Graph Cut Approach for Large-scale Data," *IEEE Trans. Image Process.*, 2013.

[47] V. Vineet and P. J. Narayanan, "CUDA Cuts : Fast Graph Cuts on the GPU by CUDA Cuts : Fast Graph Cuts on the GPU," *Compute*, 2008.

[48] W. M. W. Hwu, *GPU Computing Gems Emerald Edition*. 2011.

[49] O. Jamriska, D. Sykora, and A. Hornung, "Cache-efficient graph cuts on structured grids," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2012.

[50] "Boost Graph Library: Dijkstra's Shortest Paths - 1.65.0." [Online]. Available: https://www.boost.org/doc/libs/1_65_0/libs/graph/doc/dijkstra_shortest_paths.html . [Accessed: 03-Nov-2019].

[51] L. Shannon, J. Li, M. R. Mohammadnia, and M. V. Sarunic, "Evaluating the scalability of high-performance, fourier-domain optical coherence tomography on GPGPUs and FPGAs," in *Conference Record - Asilomar Conference on Signals, Systems and Computers*, 2011.

[52] J. Li, M. V. Sarunic, and L. Shannon, "Scalable, high performance fourier domain optical coherence tomography: Why FPGAs and not GPGPUs," in *Proceedings - IEEE International Symposium on Field-Programmable Custom Computing Machines, FCCM 2011*, 2011.

[53] T. E. Ustun, N. V. Iftimia, R. D. Ferguson, and D. X. Hammer, "Real-time processing for Fourier domain optical coherence tomography using a field programmable gate array," *Rev. Sci. Instrum.*, 2008.

[54] K. C. Jin, K. S. Lee, and G. H. Kim, "High-speed FPGA-GPU processing for 3D-OCT imaging," in *2017 3rd IEEE International Conference on Computer and*

Communications, ICCC 2017, 2018.

[55] V. Bandi, J. Goette, M. Jacomet, T. von Niederhäusern, A. H. Bachmann, and M. Duelk, "FPGA-based real-time swept-source OCT systems for B-scan live-streaming or volumetric imaging," in *Optical Coherence Tomography and Coherence Domain Optical Methods in Biomedicine XVII*, 2013.

[56] D. Kobori and T. Maruyama, "An acceleration of a graph cut segmentation with FPGA," in *Proceedings - 22nd International Conference on Field Programmable Logic and Applications, FPL 2012*, 2012.

[57] S. Saha, K. H. Uddin, M. S. Islam, M. Jahiruzzaman, and A. B. M. A. Hossain, "Implementation of simplified normalized cut graph partitioning algorithm on FPGA for image segmentation," in *SKIMA 2014 - 8th International Conference on Software, Knowledge, Information Management and Applications*, 2014.

[58] D. Koukounis, C. Ttofis, A. Papadopoulos, and T. Theocharides, "A high performance hardware architecture for portable, low-power retinal vessel segmentation," *Integr. VLSI J.*, 2014.

[59] H. Bendaoudi, F. Cheriet, A. Manraj, H. Ben Tahar, and J. M. P. Langlois, "Flexible architectures for retinal blood vessel segmentation in high-resolution fundus images," in *Journal of Real-Time Image Processing*, 2018.

[60] A. Hoover, "Locating blood vessels in retinal images by piecewise threshold probing of a matched filter response," *IEEE Trans. Med. Imaging*, 2000.

[61] Z. Jiang, J. Yepez, S. An, and S. Ko, "Fast, accurate and robust retinal vessel segmentation system," *Biocybern. Biomed. Eng.*, 2017.

[62] A. G. Roy *et al.*, "ReLayNet: retinal layer and fluid segmentation of macular optical coherence tomography using fully convolutional networks," *Biomed. Opt. Express*, vol. 8, no. 8, p. 3627, Aug. 2017.

[63] P. M. Maloca *et al.*, "Validation of automated artificial intelligence segmentation

of optical coherence tomography images," *PLoS One*, vol. 14, no. 8, p. e0220063, Aug. 2019.

[64] Y. He *et al.*, "Deep learning based topology guaranteed surface and MME segmentation of multiple sclerosis subjects from retinal OCT," *Biomed. Opt. Express*, vol. 10, no. 10, p. 5042, Oct. 2019.

[65] J. Kugelman *et al.*, "Automatic choroidal segmentation in OCT images using supervised deep learning methods," *Sci. Rep.*, vol. 9, no. 1, Dec. 2019.

[66] Y. Ruan *et al.*, "Multi-phase level set algorithm based on fully convolutional networks (FCN-MLS) for retinal layer segmentation in SD-OCT images with central serous chorioretinopathy (CSC)," *Biomed. Opt. Express*, vol. 10, no. 8, p. 3987, Aug. 2019.

[67] L. Ngo, J. Cha, and J.-H. Han, "Deep Neural Network Regression for Automated Retinal Layer Segmentation in Optical Coherence Tomography Images," *IEEE Trans. Image Process.*, vol. 29, pp. 303–312, Aug. 2019.

[68] F. G. Venhuizen *et al.*, "Robust total retina thickness segmentation in optical coherence tomography images using convolutional neural networks," *Biomed. Opt. Express*, vol. 8, no. 7, p. 3292, Jul. 2017.

[69] L. Fang, D. Cunefare, C. Wang, R. H. Guymer, S. Li, and S. Farsiu, "Automatic segmentation of nine retinal layer boundaries in OCT images of non-exudative AMD patients using deep learning and graph search," *Biomed. Opt. Express*, vol. 8, no. 5, p. 2732, May 2017.

[70] P. Zang, J. Wang, T. T. Hormel, L. Liu, D. Huang, and Y. Jia, "Automated segmentation of peripapillary retinal boundaries in OCT combining a convolutional neural network and a multi-weights graph search," *Biomed. Opt. Express*, vol. 10, no. 8, p. 4340, Aug. 2019.