SCALABLE PARALLEL OPTIMIZATION
OF DIGITAL SIGNAL PROCESSING
IN THE FOURIER DOMAIN

A THESIS SUBMITTED TO
THE UNIVERSITY OF KENT
IN THE SUBJECT OF COMPUTER SCIENCE
FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

By
Konstantin Ivanov Kapinchev
2016
Canterbury

Abstract

The aim of the research presented in this thesis is to study different approaches to the parallel optimization of digital signal processing algorithms and optical coherence tomography methods. The parallel approaches are based on multithreading for multi-core and many-core architectures. The thesis follows the process of designing and implementing the parallel algorithms and programs and their integration into optical coherence tomography systems. Evaluations of the performance in terms of latency and scalability of the proposed parallel solutions are presented.

The digital signal processing considered in this thesis is divided into two groups. The first one involves generally employed algorithms operating with digital signals. Those include forward and inverse discrete Fourier transform, cross-correlation, convolution and others.

The second group involves optical coherence tomography methods, which incorporate the aforementioned signal processing algorithms. These methods are used to generate cross-sectional, en-face and confocal images. Identifying the optimal parallel approaches to these methods allows improvements in the generated imagery in terms of performance and content. The proposed parallel accelerations lead to the generation of comprehensive imagery in real-time. Providing detailed visual information in real-time improves the utilization of the optical coherence tomography systems, especially in areas such as ophthalmology.

Acknowledgements

Firstly, I would like to thank my supervisor Dr Fred Barnes for his help and support during my studies. I would also like to thank Prof Adrian Podoleanu. His insights on optical coherence tomography contributed to the direction of this research.

Also, I would like to thank all the academic and administrative staff at the School of Computing. Their support throughout the years made my studies at Kent an unforgettable experience.

All this was made possible, thanks to the EPSRC scholarship. This support made all the difference.

I would also like to thank NVIDIA Corporation for supporting my research by providing the NVIDIA Tesla GPU, which is part of the NVIDIA GPU Research Center Grant.

Finally, I would like to thank my family, for their unconditional help, for all the patience and understanding.

Contents

Chapter 1

Introduction

The concept of parallel computing spans the architecture of the processing units and the design of the programming tools, which utilize them. These programming tools include operating system support, programming language specifications, and libraries. To extract optimal performance from a parallel architecture, computer algorithms and programs need to reflect the characteristics, capabilities, and limitations of the utilized hardware and software components. Parallel approaches need to consider the computational costs of launching parallel threads of control, the overheads of the thread management, and the underlying memory model.

1.1 Motivation

In 2004, Intel Corporation announced the cancelation of the release of two of its processors, namely Tejas and Jayhawk. They were supposed to replace Pentium 4 and Xeon processors. The constant increase of the processors' clock rate approached the physical limitations of the technology [1]. In a Communications of the ACM, Moshe Vardi described this phenomenon as unresolvable without major technological changes [2]. Later on, Intel introduced the multi-core architecture with the release of their first Core Duo processor.

Nevertheless, increased number of processing cores does not translate automatically into increased performance. A performance gain from a multi-core architecture requires significant redesign of the algorithms and the programs. In [3], Sutter’s widely cited article, the author could not stress that enough. Based on this report, the following key points can be identified:

1. Continuation of the performance growth depends on concurrent and multithreaded applications.

2. One of the primary reasons for employing concurrency is the logical separation of portions of the code.

3. The other one is the performance achieved by taking advantage of multiple processors.

4. Some applications are naturally parallelizable and others are not.

Considering the scope of the digital signal processing (DSP), these key points can be extended as follows:

- There is a constant demand for increased performance. For example, an improved quality of imagery leads to increased size of data, or signal, and increased amount of computations, but still within the same real-time requirement.
- In almost all cases in practice, DSP is not performed independently, for its own sake. Usually, it is part of a larger system with multiple components, each one with its own specific task. An optical coherence tomography (OCT) system for example, during operation acquires data, generates samples, reads user inputs, applies DSP algorithms, and displays images. The logical separation of the DSP from the rest of the system, which otherwise would be part of the data acquisition software, contributes to the stability of the overall system and the readability of the source code.
- The parallel architectures of the multi-core central processing unit (CPU) and many-core graphics processing unit (GPU) allow multithreaded programs to take advantage of them and to improve the performance.
- DSP algorithms perform the same computations on all data points from the digital signals. In most cases, the processing of each data point is completely independent from the rest. In the scope of this thesis, the integration is the only exception. This makes DSP a good example of naturally parallelizable algorithms.

Considering these points, DSP algorithms appear as good candidates to take advantage of parallel optimizations and as a result, to maximize their performances.

Motivated by the aforementioned points, this research studies the parallel optimization of DSP algorithms and OCT methods. Considering the current parallel computing technology, this research investigates different parallel approaches and identifies those, which lead to optimal performance and improved imagery in the OCT systems.

1.2 Scope

The scope of the research presented in this thesis is parallel optimization based on multithreading for the multi-core (CPU) and many-core (GPU) architectures. Although different in their nature, both the CPU-based and the GPU-based parallel threads deliver accelerated computations, due to their simultaneous execution. This forms a basis for comparisons between these two approaches.

The multithreading solutions implement general DSP algorithms, such as Fourier transform, cross-correlation and convolution. These algorithms are used in OCT methods, such as the conventional Fourier domain (CFD) method and the newly introduced Master-Slave Interferometry (MSI) method [4].

The study presented in this thesis reflects the current state of the parallel computing technology and the performance it offers. Aspects of this technology are taken into account, such as the multi-core and many-core architectures, the coarse-grained and fine-grained parallelism, the programming language support, and the overheads caused by the thread management. The integration of the parallel solutions into real-time systems, such as OCT, is also considered.

1.3 Application Area

The practical application area of this research is optical coherence tomography. The purpose of the parallel optimization is to enable OCT systems to process digital signals and visualize OCT images in real-time. OCT systems, along other medical imaging technics such as radiology and magnetic resonance imaging (MRI), are able to image semitransparent objects below the surface. These technologies are employed to visualize the internal structures of tissues. They are used to assist in diagnosis and surgery [5].

OCT systems are employed to generate two types of images, namely cross-sectional and en-face. The cross-sectional images display multiple depths, while the en-face images visualize a single depth, as seen in Figure 1.1.

Chapter 3 presents a closer look at the OCT systems, their main components, the employed DSP algorithms and OCT methods, and the generated images.

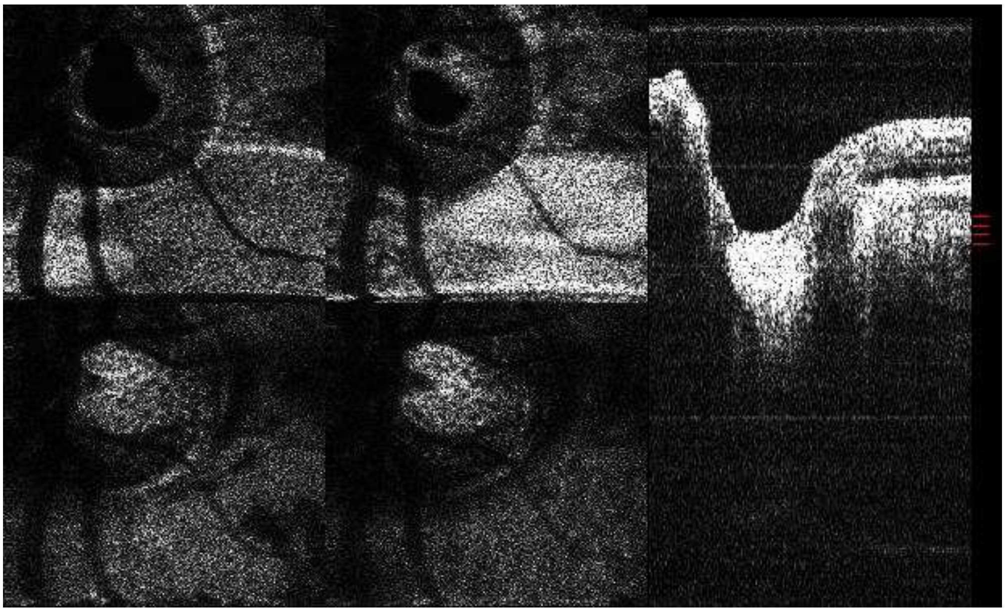


Figure 1.1: One cross-sectional (right) and four en-face (left) OCT images of a human eye. The four red lines on the right indicate the position in depth of the en-face images

1.4 Parallel Programming Model

The algorithms and methods presented in this thesis are designed for OCT systems operating in real-time, which is processing the signals and visualizing the images while the system is imaging the object. In these systems, the data is generated by data acquisition software and after processing is handled by a visualization tool. These systems have a single source of data and a single destination for the result. Therefore, a shared memory parallel programming model, implemented on the same computer system as the data acquisition software and the visualization tool, is more viable than a distributed solution communicating over a network.

1.5 Fast Fourier Transform Libraries

Forward and inverse discrete Fourier transforms are crucial building blocks in the digital signal processing, as they provide the transition from time domain to frequency (Fourier) domain. By definition, the complexity of the discrete Fourier transform is $O(N^2)$. This indicates a computationally intensive algorithm. Significant efforts to optimize the Fourier transform resulted in a collection of algorithms, denoted as Fast Fourier Transform (FFT). A popular example of FFT algorithm is the Cooley-Tukey FFT [6]. It reduces the number of operations to O(N×Log(N)), here N is the size of the input data. The mathematical foundations of this algorithm can be traced back to Carl Friedrich Gauss (1777-1855) [7]. This algorithm is implemented in the Fast Fourier Transform in the West (FFTW) library, introduced in [8], and in the cuFFT library [9], developed at NVIDIA Corporation.

Two approaches to parallelization, denoted as coarse-grained and fine-grained, are developed and presented in Chapters 4 and 5.

The coarse-grained approach launches parallel threads created by the application programming interface (API) provided by the operating system. In this approach, the digital signal is equally divided among the parallel threads. Each parallel thread performs all processing steps on its corresponding part of the signal, including multiple FFTs. The parallelization process equally divides these multiple FFTs among the parallel threads. As a result, the FFT is parallelized by the API, rather than the multithreading support provided by the employed FFT implementation, which is FFTW [10]. Therefore, this research does not evaluate the parallel capabilities of the FFTW library. The FFT is included into a parallelization, which is designed for the studied DSP algorithms and OCT methods.

The fine-grained approach targets the many-core architecture of the GPU. It is developed as an NVIDIA CUDA C application and uses the NVIDIA FFT library, cuFFT, which is designed for the CUDA programming model. This library utilizes the parallel capabilities of CUDA-enabled GPUs [9].

1.6 Original Contribution

A number of CPU-based and GPU-based multithreaded approaches to widely adopted DSP algorithms are presented in Chapter 4 of the thesis. A comparison between these two parallel approaches is presented. Also, the parallel approaches are compared with their corresponding sequential implementations. The optimal parallel approach is identified for every case.

Parallel optimizations of two OCT methods, based on the aforementioned approaches, are presented in Chapter 5. To the best of the author's knowledge, the first report of real-time generation of MSI-based cross-sectional images implemented on the GPU was published in [11]. Subsequently, one of the first reports of the generation of up to 40 en-face images in real-time based on the same OCT method were published in [12] and [13].

A comprehensive GPU-enabled OCT imagery is presented in Chapter 6 of the thesis. It consists of multiple en-face images, accompanied by one confocal and two cross-sectional images. All images are generated simultaneously on the same volumetric data. These images, delivered in real-time, provide detailed information about the studied objects. This improves significantly the utilization of the OCT systems, especially in areas such as ophthalmology. In this area, comprehensive OCT imaging can improve the process of diagnosis and assist in real-time imaging during surgery and ablation [5].

The contributions of this thesis can be summarized as follows:

- Demonstration of the computational capabilities of the multi-core and many-core architectures to perform DSP algorithms and OCT methods
- Identification of the optimal parallel approaches to these algorithms and methods
- Improvements of the operation of OCT systems in terms of:
    - Performance
        - Achieved real-time performance, presented in Chapter 5
    - Content
        - Comprehensive OCT imagery, presented in Chapter 6

1.7 Terminology

A large amount of books, textbooks, journal articles and conference papers involving parallel computing and signal processing are published. Some variations of the terminology used in these areas can be observed.

In [14] and [15], the term multithreading is used to describe a programming paradigm, in which a single process is divided into multiple parallel threads. Concurrency is based on the ability of these threads to advance independently, but not necessarily simultaneously. Parallel optimization, based on multithreading for multi-core and many-core architectures, presumes simultaneous execution of the threads. Some form of serialization may occur in the case of a larger number of parallel threads, which depends of the parallel capabilities of the targeted architecture.

This thesis follows the aforementioned terminology. It considers parallel optimization as the transformation of an existing sequential algorithm or program into a parallel one. In most cases in practice including this research, the transformation targets a specific parallel environment, such as specific parallel architectures and programming language support. An expected and desired result of the parallel optimization is an improved performance, as discussed in numerous published books and articles, including [3], [16] and [17].

The terms "sequential" or "serial" are used interchangeably in [18] to describe programming paradigm.

In this thesis, tread of control generalizes the notion of logically separated instructions organized as functions or kernels. They have the ability to advance independently. Examples for specific implementations are:

- Processes launched by executable binaries or routines, such as fork and spawn
- Threads running on host processors (multi-core CPU)
- Threads launched on the GPU by kernels
- Fibers, lightweight threads

The computations presented in this research are applied on digital signals. Although most of the presented algorithms can be applied on more general type of data, the scope considered in this thesis is digital signals generated synthetically or by an OCT system.

The presented computations are divided into DSP algorithms and OCT methods. The DSP algorithms perform mathematical operations on signals, such as Fourier transform, cross-correlation and others. The OCT methods consist of a number of DSP algorithms applied on digital signals generated by OCT systems.

The GPU-based fine-grained approaches discussed in this thesis are designed for NVIDIA GPUs. Therefore, the terminology concerning GPU computing used in this thesis is influenced by the terminology introduced by the NVIDIA Corporation.

1.8 Structure of the Thesis

Chapter 2 discusses the foundations of multithreading. It is presented from the point of view of parallel acceleration of DSP algorithms and OCT methods.

Chapter 3 presents generalized view of the signal processing. The chapter also describes the principles of the optical coherence tomography. Two OCT methods, based on the discussed signal processing algorithm, are presented.

Chapter 4 follows the development of different approaches to parallel optimizations of the aforementioned DSP algorithms. Their performances are reported and compared.

Similarly, Chapter 5 proposes parallel optimizations of the discussed OCT methods. These methods are employed to generate cross-sectional, en-face and confocal images. The goal of the parallel approaches is to reach image generation in real-time.

Chapter 6 presents a comprehensive OCT imaging solution, which is based on the outcomes from the previous chapters. The comprehensive imagery combines the previously presented OCT images. This solution is successfully integrated into a working OCT system and operates in real-time.

Chapter 7 concludes the results from the previous chapters. It also provides some key directions for future development in the area of parallel computing and optical coherence tomography.

1.9 Publications

The following publications are based on the research presented in this thesis.

1. Kapinchev, Barnes, Rivet, Bradu, and Podoleanu, "Parallel Approaches to Integration with Applications in Optical Coherence Tomography", 10th International Conference on Signal Processing and Communication Systems, Gold Coast, 2016

2. Kapinchev, Bradu, Barnes, and Podoleanu, "Coarse-Grained and Fine-Grained Parallel Optimization for Real-Time En-Face OCT Imaging", Photonics West, San Francisco, 2016

3. Bradu, Kapinchev, Barnes, Podoleanu, "In-Vivo, Real-Time Cross-Sectional Images of Retina Using a GPU Enhanced Master Slave Optical Coherence Tomography System", Photonics West, San Francisco, 2016

4. Kapinchev, Bradu, Barnes, and Podoleanu, "GPU Implementation of Cross-Correlation for Image Generation in Real Time", 9th International Conference on Signal Processing and Communication Systems, Cairns, 2015

5. Bradu, Kapinchev, Barnes, Podoleanu, "Master Slave En-Face OCT/SLO", Biomedical Optics Express Journal, 2015

6. Bradu, Kapinchev, Barnes, Podoleanu, "On the Possibility of Producing True Real-Time Retinal Cross-Sectional Images Using a Graphics Processing Unit Enhanced Master-Slave Optical Coherence Tomography System", Journal of Biomedical Optics (open access), 2015

7. Bradu, Kapinchev, Barnes, Garway-Heath, Rajendram, Keane, Podoleanu, "Real-Time Calibration-Free C-scan Images of the Eye Fundus Using Master Slave Swept Source Optical Coherence Tomography", Photonics West, San Francisco, 2015

8. Bradu, Kapinchev, Barnes, Podoleanu, "Master-Slave Optical Coherence Tomography for Parallel Processing, Calibration Free and Dispersion Tolerance Operation", Photonics West, San Francisco, 2015

9. Kapinchev, Barnes, Bradu and Podoleanu, "Approaches to General Purpose GPU Acceleration of Digital Signal Processing in Optical Coherence Tomography Systems", IEEE International Conference on Systems, Man, and Cybernetics, Manchester, 2013

Chapter 2

Background on Multithreading

2.1 Introduction

This chapter considers the body of work built around multithreading as a technique for parallel implementations, designed for multi-core and many-core architectures. The comparison between these two parallel approaches, although different in their nature, is based on an accelerated performance, result of a simultaneous execution of multiple threads.

Until mid-2000, the larger part of the computing community expected improved performance delivered from every new line of processing units. As expected, this power growth became unsustainable, [1], [3], [17]. Further improvements of the performance of the single-core processors came with a significant increase in the consumed power, as noted in [19].

The utilization of multi-core and many-core processing architectures by parallel solutions is currently the primary route to achieve improved performance, with multithreading being one of the predominant techniques to achieve this [20]. An improved performance based on parallel threads requires the following four key elements:

1. Parallel architecture. Single-core architecture would improve a performance of a multithreaded application, if the parallel threads overlap different activities, such as I/O and compute intensive operations. In all other cases, improved performance requires multi-processing architecture [20].

2. Operating system support. In most cases, this support comes in the form of a preemptive scheduling of processes and threads. In addition, the operating system may

provide API support, such as the Microsoft Windows API [21].

3. Programming language support. Apart from calling the aforementioned API functionality, if presented with any, the language support may be implemented as part of the language specification (Java), or as an additional library (POSIX Threads).

4. Parallelized code. This aspect is within the responsibilities of the design and the implementation of parallel algorithms and programs. Otherwise, the performance of a sequential program would not be affected by the aforementioned parallel capabilities, [3].

Currently, there are no generally applicable solutions, which automatically transform existing sequential computer programs into parallel ones. This is due to a number of obstacles, such as loops with unknown at compile time number of iterations and flow of control depending on the input. Attempts are made, but without significant success [22]. Therefore, if parallelism is needed, it has to be explicitly implemented as part of the algorithm.

There are two cases in the execution of a parallel computer program, depending on the number of threads of control and the number of processing units:

1. The number of threads of control is larger than the number of processing units. This case is referred as concurrency. A scheduling mechanism implemented by the operating system organizes the switching between the tasks. If the switching is fast enough, the execution of the multiple threads of control will resemble simultaneous execution [23].

2. The number of threads of control is smaller than or equal to the number of processing units. This case is referred as true simultaneous execution or true parallelism. In this case, the operating system ensures that each thread of control runs on a separate processing unit [14].

2.2 Parallel Architectures

The simultaneous utilization of a number of processing units by a number of independent sequences of instructions underpins the notion of parallel computing. The processing units are usually incorporated on a single chip and the sequences of instructions as processes or threads. The collection of these processing units, along with their ability to share resources and communicate comprises the parallel architecture.

A number of classifications for parallel architectures exist. These classifications can be based on the employed memory model and the instruction-data relation, among others. The following section presents the Flynn taxonomy and its relevance to the contemporary multi-core and many-core architectures.

2.2.1 Flynn Taxonomy

This classification is proposed in 1966 by Michael Flynn in [24] and further developed in [25]. It applies not only to parallel architectures, but also to computer architectures in general. It recognizes the following classes:

1. Single Instruction Single Data (SISD). The case describes the classical von Neumann architecture based on the sequential execution of instructions [26].
2. Single Instruction Multiple Data (SIMD). This case illustrates the simultaneous execution of the same instruction on a number of operands, or data points.
3. Multiple Instruction Single Data (MISD). In this case, a number of operations are performed on the same data. An argument exists, that at the completion of each stage the data are modified, hence cannot be treated as the same data.
4. Multiple Instruction Multiple Data (MIMD). This class describes the multi-core architectures, where different threads of control can advance independently and in parallel on multiple processing cores.

2.2.2 Single Instruction Multiple Threads

This classification was introduced by the NVIDIA Corporation in correspondence with the Flynn taxonomy. It describes an architecture which underpins the utilization of the GPU for general purpose computing [27]. It is employed in NVIDIA Tesla, first GPU dedicated solely to general purpose computations, designed to act like a parallel co-processor, rather than graphics adapter.

In this architecture, the parallel threads are grouped into warps. Each warp consists of 32 threads. The threads within a wrap are scheduled to run synchronously, unless some threads diverge from the common execution path [28]. This architecture corresponds to the aforementioned SIMD class, as discussed in [27] and [29].

This paradigm reaches optimal performance, if all threads within a single warp follow the same execution flow. Threads are not prohibited by the language specification from branching out of the warps. However, this could reduce the performance.

2.3 Scalability in Parallel Computing

The notion of scalability applied in this thesis, characterizes the improvements in the performances of the proposed parallel solutions, as discussed in [26 p. 63]. In the scope of this research, the goal of the parallel optimization is to deliver real-time operation of the OCT systems. The real-time criterion of an OCT system is defined as the time window, within which the processing and the image generation need to complete. Therefore, the performances are measured in terms of latency and speed-up.

Two cases of scalability are identified in [22 pp. 56-57], namely strong and weak scalability:

1. A number of parallel optimizations are applied on a task processing a fixed size of data. A scalable parallel solution is expected to improve its performance when the number of parallel threads of control increases, Figure 2.1.A.

2. An increase of the data is followed by a proportional increase of the parallel threads of control. A scalable solution is expected to absorb the increased amount of data and to increase its latency at a slower rate, compared with the sequential implementation of the task, Figure 2.1.B.

These two cases of scalability are observed in cross-sectional and en-face imaging in OCT, presented in Chapter 5. In Figures 5.2 and 5.3 the size of the processed data is fixed. The data is divided equally among various numbers of parallel threads. In Figures 5.9 and 5.10 the size of the processed data is connected with the number of parallel threads.

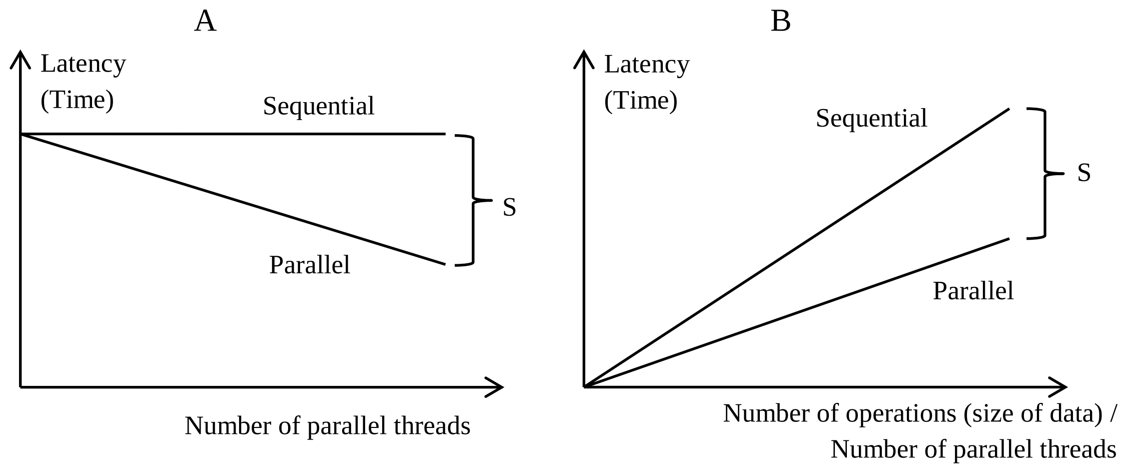

A
Latency
(Time)
Sequential
Parallel
S
Number of parallel threads
B
Latency
(Time)
Sequential
Parallel
S
Number of operations (size of data) /
Number of parallel threads

Figure 2.1: Two cases of scalability (S) of parallel solutions, strong scalability (A) and weak scalability (B), introduced by John Gustafson in [30]

2.4 Granularity in Parallel Computing

An important aspect of the parallelization process is how the computations are distributed among the parallel threads of control. This concept is denoted as granularity. It distinguishes two cases: coarse-grained and fine-grained parallelism [16]. These cases have more recognizable meaning in a comparative sense, rather than being used independently. Considering a job that is subject to parallelization, the coarse-grained approach will divide it into a smaller number of larger tasks, while the fine-grained approach will divide it into a larger number of smaller tasks. The question of granularity stays in almost all cases of parallel optimization.

On a parallel architecture, a task improves its performance by its division among a number of threads of controls. On the other hand, the creation of these threads consumes resources needed by the thread management. The choice of granularity is a trade-off between these two factors.

Additional factor is the communication between the threads. This communication is directly related to the nature of the job. Cases with intensive communications between the threads would achieve optimal performance with a smaller number of threads.

Chapters 4 and 5 of the thesis propose coarse-grained and fine-grained approaches to DSP algorithms and OCT methods. The granularity in these parallel optimizations observes the nature of the digital signals, especial the ones generated by OCT systems. Those signals consist of sub-signals, called channeled spectra, and data points. The coarse-grained and fine-grained parallel approaches reflect this division of the signals into sub-signals.

2.5 Processes and Threads

The two primary techniques for parallelism are processes and threads. Different in their nature, these two techniques find different applications.

In general, running a computer program results in the creation of a process. In multitasking operating systems, multiple processes can run concurrently, sharing the system resources. These multiple processes are initiated by different programs. Also, a single program can fork or spawn into more than one process. Every process has at least one thread.

The key components of the process are virtual address space, an image, and the context of the process, which holds a number of attributes describing the process, such as process ID, current state, priority and ownership. Switching between processes involves switching between contexts.

Multiple threads of execution can exist within a single process. These threads share the same virtual address space, context and global variables. Therefore, creating a new thread does not involve creating new virtual address space and context. As a result, a multithreaded solution would consume less resources and less time in switching between concurrent threads, as noted in [23].

On the other hand, threads do not provide the same level of independence and isolation, as processes do. Unstable threads can affect other threads from the same process, which makes parallelizing via processes in some cases the preferable choice. Table 2.1 presents a comparison between processes and threads.

Table 2.1: Process and thread comparison

	Processes	Threads
Virtual Address Space	Separate	Shared
Data Exchange	IPC	Global variables
Synchronization	IPC	Global variables/API
Priority	Yes	Yes

In most cases, a given DSP algorithm processes a specific type of data generated by a single data source. In this case, the level of independence and isolation provided by the parallel processes is unnecessary. It would require more complex tools to cooperate and increase the overheads without improving the processing itself. Threads, on the other hand, are less isolated and easier to cooperate. This makes the threads the choice for the CPU-based parallelization of the DSP algorithms presented in this thesis.

2.6 Types of Multithreading Support

The implementation of a parallel algorithm as a multithreaded program requires support from the programming language. This support may be implemented in one of the three following fashions:

1. Multithreading support provided by the language specification. Examples for this case are Java, C# and Ada.

2. Library. This case usually applies to a languages initially developed without multithreading support. A prominent example for this case is the standard C language and the POSIX Thread (pthread) library [31].

3. API. In some cases multithreading is supported by the operating system via API. For example, Microsoft API provides multithreading which can be used from programming languages part of the Visual Studio integrated development environment [21].

In [32], possible issues with the library-based multithreading are identified and discussed. These issues may occur when the compiler reorders some operations for optimization purposes, without considering the parallel execution of the threads. In this case, unexpected or incorrect results may arise, if two or more threads access the same variable, or memory location, concurrently.

The same issue is observed in NVIDIA CUDA, which assumes weekly-ordered memory model. If necessary, this can be avoided by adding calls to __threadfence_block for threads from the same thread block and __threadfence for threads from different thread blocks. These calls act as memory fencing and ensure the ordering of the memory operations, as described in [28].

This case does not occur in multithreaded implementations of the DSP algorithms studied in this research for two main reasons:

1. Each parallel thread processes one or more data points. All data points are independent from each other. The processing steps performed by one parallel thread do not interfere with the processing steps performed by the rest of the threads. This case is known as embarrassingly parallel, as discussed in [33 p. 14]. In this research, the only exception is the Integration step, where multiple parallel threads contribute to the same result. In this case, synchronization is ensured by introducing barriers.

2. In the employed DSP algorithms, each step f in Figure 2.2 uses the result from the previous step. In other words, the argument of the current function is the result of the previous one. This prevents the compiler from reordering the operations.

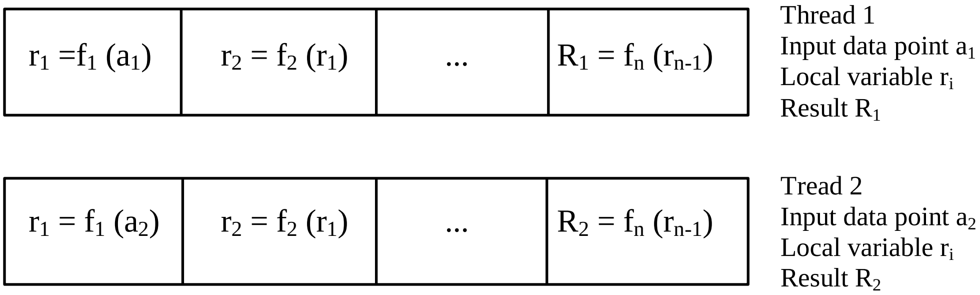


Figure 2.2: Two threads processing data concurrently

These two points do not take into account the parallelized implementation of the discrete Fourier transform provided by NVIDIA. It is assumed that this implementation took into account the problem discussed in [32]. Extensive tests conducted by a wide community using CUDA, including this research, acknowledge the correctness of this FFT implementation.

2.7 Multithreading Support for Shared Memory Model

This research considers parallel optimization designed for DSP algorithms. In most cases in signal processing, there is one source of data, which is usually the data acquisition software. Also, the output signal generated as a result of the signal processing is usually handled by one module, which purpose is to visualize or store the result. Therefore, a parallelized signal processing algorithm expects all parallel threads to have equal access to the input and output data. This requires parallel optimization based on a shared memory model, Figure 2.3. A shared-memory SIMD architecture was presented in [34] as a sub-class of the abovementioned Flynn taxonomy.

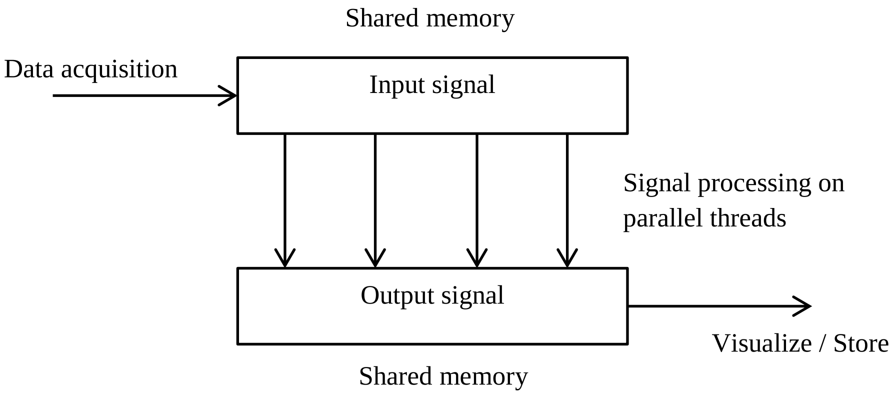


Figure 2.3: Multithreading based on shared memory

The following section lists some of the most popular languages and libraries, available for multithreaded applications that support the shared memory model. This list is by no mean exhaustive. The discussed languages and libraries target the multi-core architecture of the CPU and the many-core architecture of the GPU. In the cases of POSIX and Microsoft API, the shared memory is the collection of the global variables of the process launching the parallel threads. In the CUDA case, the shared memory is the global GPU memory.

The GPU-based fine-grained approaches to the DSP algorithms and OCT methods presented in Chapters 4 and 5 yields optimal performances. These approaches are implemented as solutions integrated into an OCT system, presented in Chapter 6.The result is a comprehensive imagery delivered in real-time. Therefore, this chapter pays special attention to the employed NVIDIA CUDA programming model.

2.7.1 POSIX Threads and gcc

A prominent example of a library-based support for multithreading is the POSIX Threads library. This library extends the capabilities of the standard C language, provided by the gcc (GNU Compiler Collection) [31]. This library can be linked with C/C++ programs, compiled with gcc. In a multithreaded program, all threads have access to the global memory. A number of semaphore functions can be used to implement critical sections.

In a POSIX multithreaded solution, new thread is created by pthread_create function. Thread attributed, such as user-defined stack size and scheduling priority, can be passed to the newly created thread. If not specified, the newly created thread loads default values for these attributes.

The pthread_join function suspends the calling thread, which is normally the main thread of the process, until the referred thread terminates. The two functions, pthread_create and pthread_join can be used to implement the fork-join multithreaded paradigm.

2.7.2 Microsoft API

A set of API functions provided by Microsoft extend the capabilities of the Visual C++ language to allow multithreaded applications [21]. The two main routines supporting the multithreading are CreateThread and WaitForMultipleObject. These API correspond to the aforementioned POSIX functions pthread_create and pthread_join.

2.7.3 OpenMP

OpenMP (Open Multi-Processing) provides multithreading support based on API [35]. It is designed primarily for C/C++ programming languages, but also supports FORTRAN programs. It supports various platforms, including POSIX compliant operating systems and Microsoft Windows. The utilization of OpenMP divides the program into two sections, sequential and parallel. Its parallelism is based on the fork-join paradigm, Figure 2.4.

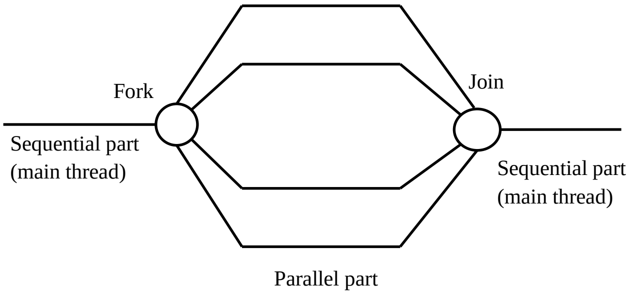


Figure 2.4: Parallel programming with OpenMP

In both POSIX and Windows environment, OpenMP allows control over the number of parallel threads in two different ways: by the environment variable OMP_NUM_THREADS and by the function omp_set_num_threads (int N).

Besides data-based parallelism, OpenMP can be used to implement task-based parallel solutions. This feature allows the implementations of consumer-producer types of problems.

The OpenMP includes parallel features, such as critical sections and reduction, as discussed in [18 pp. 32-36]. In a C/C++ program, these features are accessible by using the #pragma omp parallel clause, with keywords critical and reduction.

2.7.4 OpenCL

OpenCL (Open Computing Language) is an API-based language extension, which provides the C/C++ language with multithreading capabilities [36]. This framework, developed by the Khronos Group [37], supports variety of parallel architectures, including multi-core CPU, GPU, field programmable gate arrays (FPGA), and digital signal processors. Its performance is often compared with NVIDIA CUDA, [38], [39]. In [40], the comparison includes FPGA and Advanced Micro Devices (AMD) graphics accelerators. Unlike CUDA, OpenCL is not vendor locked-in, which makes it more portable. The portability is implemented by an abstract layer, SYCL [41]. In the scope of the general purpose GPU (GPGPU) computing, this abstract layer hides some hardware features, which are otherwise exposed by the NVIDIA CUDA programming model. As noted in [42], the increased portability affects the performance unless specifically tuned kernels, which are functions designed to run on a parallel architecture, are developed.

OpenCL allows kernels to be compiled in offline and in online mode. When compiled in online mode, also denoted as runtime, the compiler has the ability to take advantage of the features of the targeted architecture, as discussed in [43]. This increases the portability of the programs developed using OpenCL. On the other hand, this approach may lead to the distribution of the source code to other computer systems, which in some cases is unwanted effect.

2.7.5 NVIDIA CUDA

The CUDA programming model, introduced by NVIDIA, is designed to allow C/C++ and FORTRAN style solutions to extract improved performance from the NVIDIA GPUs. With the added GPU-based parallel capabilities, CUDA can also be considered as an extension or thin film to these standardized programming languages. With its multiple libraries, such as cuFFT, CUDA Math Library, cuSparse and others, CUDA finds applications in many areas of science and engineering.

The CUDA programming model is designed to utilize the NVIDIA GPU architecture, which is based on multiple streaming multiprocessors (SM). Each SM has local memory and a number of single-precision and double-precision processing units, denoted as CUDA cores. Successive architectures increase the overall number of processing units incorporated on a single GPU, and thus provide scalability to existing software solutions.

2.7.5.1 Historical Overview

This section presents some of the key events that mark the evolution of the GPU architecture and its impact on the high performance computing. These events span more than a decade, which saw the spread of GPU computing from the 3D rendering industry to almost every area of parallel computing.

The term graphics processing unit (GPU) was coined by the NVIDIA Corporation in 1999 with the release of the NVIDIA GeForce 256 graphic card [44]. Designed primarily for fast 3D rendering, the GPU targeted the gaming industry.

In 2007, the NVIDIA Corporation introduced the CUDA programming model. Its purpose was to enable programs to run parallelized code on the many-core architecture of CUDA-capable GPUs. The programming model was built around the CUDA C, an extension, or a thin film, of the standard C language. Besides C, CUDA provides extensions to FORTRAN and Python.

With the release of Tesla GPU in 2007, NVIDIA introduced a new line of GPU-based parallel co-processors. In 2008, the TSUBAME Supercomputer, the first high performance computer based on Tesla GPU, was announced. A performance of 4.8 TFLOPS the GPU-based TSUBAME 2.0 was reported in [45].

The idea of a parallel co-processor offloading computationally intensive tasks from the CPU was embraced by Intel with the release of the Xeon Phi co-processor in 2012. It was supported by Intel Parallel Studio, a development environment built around the Intel C++ Compiler, which further solidified Intel position in the massively parallel solutions. Currently, NVIDIA Tesla and Intel Xeon Phi remain primary competitors in the desktop-based high performance computing market.

The Kepler architecture, incorporated in the GK110 GPU, was introduced in 2012. It kept the same number of parallel threads per warp, which is 32, but increased the number of warps per multiprocessor from 48 to 64 [46]. Also, the number of thread blocks in one-dimensional GPU grid was increased from 2^{16}-1 to 2^{32}-1.

In 2016, NVIDIA released the Pascal architecture implemented in the Tesla P100 GPU [47]. Among other improvements, this architecture incorporated streaming multiprocessors with 32 double precision CUDA cores. It increased the number of streaming multiprocessors from 24, in Tesla GM200, to 56.

2.7.5.2 Host Code and Device Code

The execution of a program written in CUDA C is carried out by a host and a device. The host, being the CPU, loads the entry point of the program, which is the main function in C/C++. The device, being the GPU, executes the parallel threads specified (defined) by specialized functions, called kernels. A device can be any CUDA enabled GPU. The NVIDIA Corporation supports a list of these GPUs. The separation between the host code and the device code is specified by the following three qualifiers of the function types:

__global__ specifies a kernel execute by the device. It can be called both from the host and the device.

__device__ specifies a kernel execute by the device. It can be called only from another kernel.

__host__ specifies a function execute by the host. It can be called only by another host function, including the main function.

2.7.5.3 Kernels, Threads and the GPU Grid

The multithreading on the GPU is implemented by calling a specialized type of functions, called kernels. The calls to these kernels specify the number of parallel threads to be launched on the GPU.

The kernel call has two parameter lists: the first one defines the GPU grid and the second one defines the actual parameters of the kernel:

CUDAKernel <<< $\mathrm{ParameterList}_1$ >>> ($\mathrm{ParameterList}_2$) ;

The first parameter list consists of maximum four elements:

1. Dimension of the GPU grid. The GPU grid consists of one, two or three dimensional array of thread blocks. Its dimension is specified by a variable of the type dim3. A one-dimensional GPU grid can also be specified by a constant.

2. The number of parallel threads per thread block.

3. The size of the shared memory. This is per block shared memory accessed from all threads from the same thread block. This parameter is optional.

4. The identifier of the stream. This parameter is also optional and not used in the case of a single stream application.

The second parameter list consists of the actual parameters of the kernel. Normally, these parameters are pointers to variables (addresses) on the global GPU memory. These addresses are obtained from cudaMalloc functions. These functions, called by a host function, allocate memory for the variables on the global GPU memory.

The kernel calls in CUDA are asynchronous, that is, the control returns to the main thread running on the host before the threads launched by the kernel complete their execution. As a result, the overall latency t of a number of kernels K_i is usually smaller than the sum of the latencies of the individual kernels, Equation 2.1.

$$t(K_1) + t(K_2) + \cdots + t(K_n) > t(K_1; K_2; \cdots; K_n) \tag{2.1}$$

The maximum number of GPU-based parallel threads is equal to the number of threads per thread block multiplied with the maximum number of thread blocks in each dimension of the three dimensional GPU grid. The actual number of parallel threads per grid depends on the micro-architecture of the GPU. Currently, the maximum number of threads per thread block is 1024, and the maximum number of thread blocks per GPU grid dimension is 2^{32}-1 in Kepler and Maxwell architectures.

2.7.5.4 CUDA Memory Hierarchy

Computations performed by CUDA on the GPU require the data to be in the GPU memory. The only exception is the zero-copy paradigm. This requires data copy over the PCI Express bus, which becomes the most significant bottleneck in the GPU Computing, which is discussed in [48]. The CUDA memory hierarchy consists of GPU global memory, per-block shared memory, local thread memory, and register memory. These types of memory are significantly different in terms of capacity and access time. The correct utilization of the memory hierarchy is crucial for achieving optimal performance.

Register Memory

Variables residing in register memory are private to the threads. This type of memory has the fastest access time and the smallest capacity. Scalar variables are normally stored in the register memory. Arrays which size is known at compile time may also be stored in the register memory, subject to array size and availability.

Local Memory

This memory is private to the threads. Variables in the local memory have the lifetime of the thread. A variable declared in a kernel, which launches multiple parallel threads, has a local copy in each thread.

The following set of special variables, residing in the local memory, can be used to construct a global index to identify all threads from the GPU grid:

threadIdx identifies the thread within a thread block. The maximum number of threads is 1024, therefore this variable can hold integer values from 0 to 1023.

blockIdx identifies the thread block within the grid. More specifically, this variable identifies the thread block according to each dimension of the thread block. The maximum number of dimensions is three. The corresponding X, Y and Z dimensions are identified with blockIdx.x, blockIdx.y and blockIdx.z.

blockDim denotes the number of thread blocks on each dimension. Like the blockIdx, the corresponding identifiers are blockDim.x, blockDim.y and blockDim.z.

A thread can be fully identified by the following constructions:

- One-dimensional grid:

$$\text{ThreadGlobalIndex} = \text{blockDim.x} * \text{blockIdx.x} + \text{threadIdx.x};$$

- Two-dimensional grid:

ThreadGlobalIndex =

gridDim.x * blockDim.x * blockIdx.y + blockDim.x * blockIdx.x + threadIdx.x ;

Identifying threads within the GPU grid allows the partition of the data and the assignment of each partition to a particular thread. This technique implements data parallelism, as noted in [49].

Shared Memory

Variables declared with the __shared__ specifier are stored in per-block shared memory. The shared memory is incorporated on the streaming multiprocessor. All threads belonging to the same thread block have equal access to the shared memory. To ensure this, no thread block can be divided between multiple streaming multiprocessors. The lifetime of the shared variables is the same as the lifetime of the thread block [50], [46].

Global Memory

The global GPU memory corresponds to the main memory in the CPU-bound computations. It can be accessed only by the tools dedicated for graphics processing, such as OpenGL, Direct3D, OpenCL, and CUDA. In CUDA, variables are allocated in the global GPU memory using the cudaMalloc and freed using cudaFree functions. The second function prevents some possible memory leakage. Both functions are called from the host. In most cases in practice, the size of the data processed by GPU threads exceeds the capacity of the aforementioned levels of memory hierarchy. These data normally reside in the global GPU memory. Therefore, its capacity and access speed is crucial for the overall performance of the GPU computations. The memory characteristics, in terms of capacity and access time, vary in different GPU generations and are described in the corresponding documentations, including [46] and [50].

2.7.5.5 CUDA FFT Library

The NVIDIA CUDA programming environment includes a number of libraries aiding scientific and engineering solutions, such as nvGRAPH, cuSPARSE, CUDA Math Library, and others. One of the most popular CUDA libraries with impact in numerous scientific areas is the CUDA FFT library, cuFFT [9]. This library is extensively used in this research. Its performance, due to the utilization of the parallel resources of the GPU, contributes to the performances of the GPU-based optimizations, presented in this thesis.

The cuFFT library follows an interface similar to the FFTW library. It implements the Cooley-Tukey FFT algorithm, which has complexity of O(N×Log(N)). Table 2.2 presents the transforms available in the cuFFT library.

Table 2.2: NVIDIA cuFFT transforms

Transform	Type	Dimension	Data type
Forward	Real to complex	1D, 2D, 3D	Single precision, double precision
	Complex to complex		
Inverse	Complex to real		
	Complex to complex		

Reflecting the scope of this research, the input and output data involved in the FFT is equivalent to the input and output digital signals. The FFT can be applied once on a signal, or multiple times (batch transform) on a number of signals.

The following are two examples of single and multiple (N transforms) forward 1D real to complex transform using single precision data type. The implementation is based on the following steps:

1. Allocate GPU memory for the input and output digital signals.

```
cufftReal *InputSignal; cufftComplex *OutputSignal;
cufftReal *InputSignalMultiple; cufftComplex *OutputSignalMultiple;

cudaMalloc(&InputSignal,SignalSize*sizeof(cufftReal));
cudaMalloc(&OutputSignal,(SignalSize/2+1)*sizeof(cufftComplex));

cudaMalloc(&InputSignalMultiple,N*SignalSize*sizeof(cufftReal));
cudaMalloc(&OutputSignalMultiple,N*(SignalSize/2+1)*sizeof(cufftComplex));
```

Due to the non-redundant transform, the size of the output signal is equal to SignalSize/2+1. The next Chapter 3 discusses this feature of the FFT implementation.

2. Initialize the FFT plan

Generally, there are two types of FFT plans, for single and for multiple transform. The multiple transform applied the same type of transform repeatedly over multiple digital signals. This type of transform is more used in practice, than the single transform.

```
        cufftHandle SingleFFTPlan;
        cufftHandle MultipleFFTPlan;

        cufftPlan1d(&SingleFFTPlan,SignalSize,CUFFT_R2C,1);
        Arr[0]=(int)SignalSize;
        cufftPlanMany(&MultipleFFTPlan,1,Arr,NULL,1,0,NULL,1,0,CUFFT_R2C,N);
```

Additionally, the cufftPlanMany call can specify the distance between two successive input and output data points (stride) and the distance between two successive signals. In this case, stride equal to 1 and distance between signals equal to 0 indicate that all data points will be processed.

3. Initialize the digital signal

The FFT operates with data residing in the GPU memory. Therefore, the initialization is implemented by the following kernels, which initialize two sinusoidal signals:

```
__global__ void InitSignal(cufftReal *InputSignal)
{
        int i;
        i=threadIdx.x;
        InputSignal[i]=(cufftReal)(sin((2*3.14*i)/SignalSize));
}
__global__ void InitMultipleSignals(cufftReal *InputSignalMultiple)
{
        int i;
        i=blockIdx.x*blockDim.x+threadIdx.x;
        InputSignalMultiple[i]=(cufftReal)(sin(N*((2*3.14*i)/SignalSize)));
}
InitSignal<<<1,SignalSize>>>(InputSignal);
InitMultipleSignals<<<N, SignalSize>>>(InputSignalMultiple);
```

4. The actual transform can be performed after the initialization of the FFT plan and the input data:

```
        cufftExecR2C(SingleFFTPlan, InputSignal, OutputSignal);
        cufftExecR2C(MultipleFFTPlan, InputSignalMultiple, OutputSignalMultiple);
```

After the completion of the transform, the variables OutputSignal and OutputSignalMultiple will hold the result, which is the discrete Fourier transform of the digital signals InputSignal and InputSignalMultiple, presented with complex numbers. In other words, the result can be treated as the input signal in the Fourier domain.

5. Release the resources allocated for the transforms.

The call cudaFree releases the memory allocated on the GPU for the input and output signals. Also, resources allocated for the transforms are freed by the cufftDestroy call. Calling these functions after the transform and when the variables are no longer necessary prevents memory leak.

```
        cufftDestroy(SingleFFTPlan);
        cufftDestroy(MultipleFFTPlan);
        cudaFree(InputSignal); cudaFree(OutputSignal);
        cudaFree(InputSignalMultiple); cudaFree(OutputSignalMultiple);
```

A complete specification of the cuFFT functionality is published in [9]. This FFT implementation is employed in various research fields and compared with other FFT libraries. In [45], the CUDA FFT improved the performance of the multi-GPU TSUBAME system. In [51], a reduced computation time on a Tyan-based barebone server is reported. A comparison between FFTW and cuFFT used in OCT is presented in [52], where the observed performance gain from the GPU utilization depends on the size of the processed signal.

An FFT algorithm designed for the GPU is proposed in [53]. It demonstrates further improvements in the performance, which is compared with the NVIDIA cuFFT and the FFT developed as part of the Intel Math Kernel Library (MKL). This implementation of FFT is optimized for the Intel multi-core and many integrated core (MIC) architectures.

2.8 Summary

This chapter presented multithreading as programming model for parallel optimization. The multithreading was presented with the premise of parallel acceleration of the DSP algorithms and OCT methods. These algorithms and methods are presented in the next Chapter 3. Due to the nature of the operation of the OCT system, the parallelization expects a shared memory model, where each parallel thread has equal access to the data, that is the input and output signals.

Chapter 3

Background on Digital Signal Processing

3.1. Introduction

Signals surround people. They are different in their nature, travel through different media, and have different characteristics. Signals carry information, which can be extracted by mathematical manipulations. These mathematical manipulations are the foundations of the signal processing algorithms. They bring meaning and usefulness to the signals.

There are three major groups of signals, analog (continuous), discrete and digital. Analog signals either occur naturally, or are generated by analog equipment. They can be divided into the following types:

- Electromagnetic radiation, including radio waves and visible light
- Electrical current
- Mechanical waves
    - Surface waves, such as ocean waves and seismic waves
    - Longitudinal waves, such as sound and vibration
- Spatial signal, such as images

These signals can be subject to mathematical manipulations, called analogue signal processing. They can be processed by analog equipment, such as analog electronic boards.

Computer systems (digital processors), due to their digital nature, cannot process analog signals. In this case, the signals need to be digitized first. This is done by sampling and quantization.

A signal can be represented as a function of one or more independent variables [54].

A light ray can be presented as sinusoid electromagnetic wave which changes over time t. In this case t parameterizes the signal. In image processing, the signal represents the intensities of the points from a two or three dimensional space. These points are parameterize by their x, y, and z coordinates.

This research is focused on the single-variable signals generated by OCT systems and processed by OCT methods. These OCT methods consist of signal processing algorithms, which are presented and discussed in this chapter.

3.2 Sampling

Analog signal f parameterized by a real variable t is defined for every value of t within an interval, $t \in [a, b]$. If the analog signal is considered as a continuous function f(t), this interval can be treated as its domain.

The sampling process converts the analog signal defined for every value of its parameter t into a sequence of individual samples of this signal, collected at a certain values of t. The resulting signal, called discrete signal, is defined for a certain t equal to t_0, t_1, ... , t_{n-1}. The difference between successive values of t (t_k - t_{k-1}) is denoted as a sampling interval and the frequency at which these values are collected is denoted as sampling frequency or sampling rate. The resulting discrete signal is not defined outside these values. This process results in a significant loss of information.

The Nyquist-Shannon theorem, also called the sampling principle, sets conditions in the sampling process in order to minimize this loss of information, as presented in [55] and [56]. This theorem states that a signal with a limited bandwidth, which is the difference between the highest and the lowest frequencies of the signal, can be reconstructed from its discrete-time domain, if the sampling frequency is greater than or equal to two times the bandwidth, Equation 3.1. A widespread sampling technique is sample and hold, as discussed in [57].

$$Freq_{sf} \geq 2 \times Freq_{bw} \tag{3.1}$$

Where:

$Freq_{bw}$ is the bandwidth of the signal

$Freq_{sf}$ is the sampling frequency

3.3 Quantization

The sampling process results in a sequence of values extracted from the continuous signal at certain frequencies. These values remained unchanged during the sampling process. The quantization step converts these samples into the allowed values of a certain data type, chosen to represent the analog signal. This data type is usually integer of floating point. Figure 3.1 illustrates the sampling and quantization process.

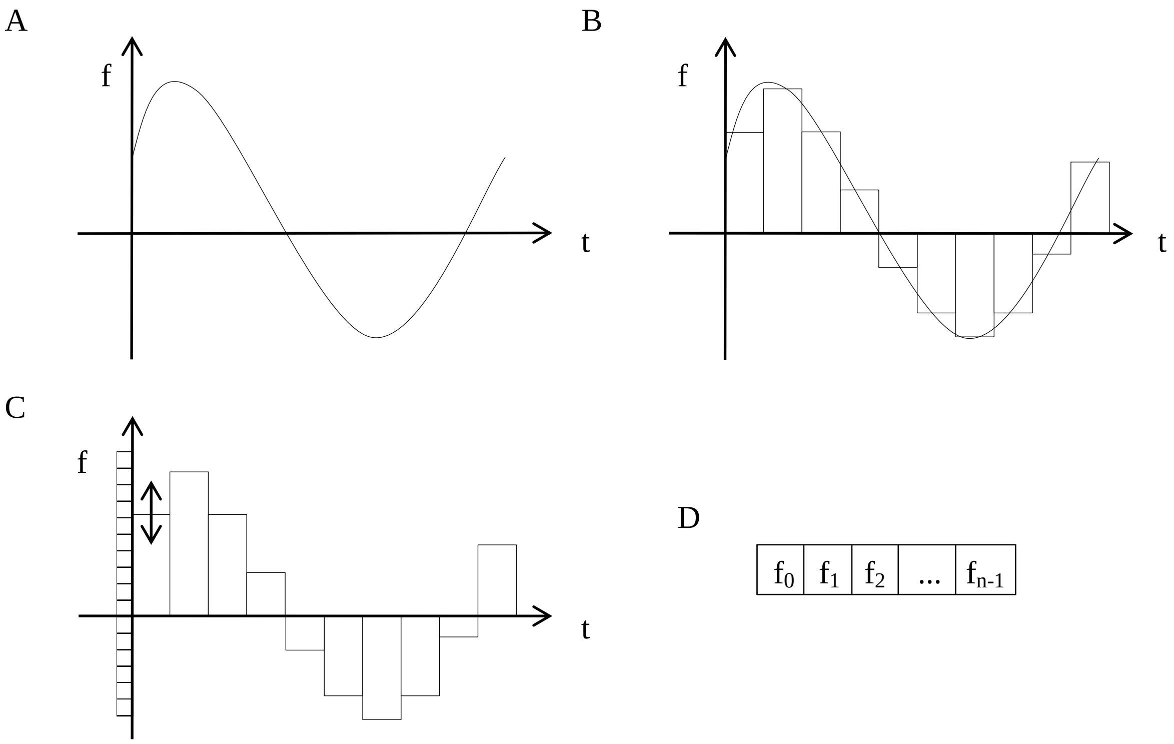


Figure 3.1: Digitizing an analog continuous signal (A) by sampling (B) and quantizing (C). The result is a sequence of values from a certain data type (D)

The analog-to-digital converter board in the OCT system used in this research generates digital signals presented as 12-bit integers. As both OCT methods employed in this research operate in Fourier domain, the 12-bit integer digital signals are converted by the data acquisition software into the data types required by the utilized FFT libraries.

3.4 Time Domain and Fourier (Frequency) Domain

Time domain and frequency domain are the two ways to represent signals. Time domain represents the change of the value of the signal in respect to time. In image processing, the corresponding domain is spatial domain, which represents the signal (intensity) with respect to position (coordinates). The frequency domain represents the signal with respect to frequency. A signal F in the frequency domain is represented with complex numbers F_k. Each complex number contains the amplitude and phase of the corresponding frequency, Equations 3.2 and 3.3 [55 p. 112].

$$Amplitude_k = AbsoluteValue(F_k) = \sqrt{\left(Re(F_k)\right)^2 + \left(Im(F_k)\right)^2} \qquad (3.2)$$

$$Phase_k = Tan^{-1}\left(\frac{Im(F_k)}{Re(F_k)}\right) \qquad (3.3)$$

The mathematical tools to transform analog continuous-time signals from time domain to frequency domain are Fourier transform and Laplace transform. The corresponding transforms for discrete-time signals are discrete Fourier transform (DFT) and Z transform.

The utilization of DFT in OCT methods, such as the CFD and MSI, made discrete Fourier transform a central DSP algorithm in OCT. A number of programming environments provide tools to calculate the DFT of digital signals, making the Fourier transform accessible to implement and integrate into various systems.

3.5 Fourier Transform

The Fourier transform, named after Jean-Baptiste Joseph Fourier (1768-1830), is the mathematics tool to transform a signal from time domain to frequency domain. Equations 3.4 and 3.5 present the forward and inverse Fourier transform of a continuous signal f(t). In the case of a time-domain signal, the argument t of the function f represents time and the argument k of its Fourier transform F represents frequency [58 pp. 37-38].

$$F(k) = \int_{-\infty}^{\infty} f(t)e^{-2\pi ikt}dt \tag{3.4}$$

$$f(t) = \int_{-\infty}^{\infty} F(k)e^{2\pi ikt}dk \tag{3.5}$$

The discrete Fourier transform (DFT), presented in [58 p. 42] and illustrated by Equations 3.6 and 3.7, is the corresponding transform applied on discrete signals. It converts a finite set of samples x_n, into a set of complex values X_k, which represent the amplitudes and phases of the corresponding frequencies, as illustrated in Equations 3.2 and 3.3.

$$X_k = \sum_{n=0}^{N-1}\left[x_n\left(\cos\left(\frac{2\pi kn}{N}\right) - i\sin\left(\frac{2\pi kn}{N}\right)\right)\right] = Re(X_k) + Im(X_k) \tag{3.6}$$

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi ikn}{N}} \tag{3.7}$$

The inverse discrete Fourier transform is presented in Equation 3.8. It differs from the forward DFT by the coefficient 1/N and the sign of the exponent.

$$x_n = \frac{1}{N}\sum_{k=0}^{N-1} X_k e^{\frac{2\pi ikn}{N}} \tag{3.8}$$

The complexity of the DFT is $O(N^2)$. A direct implementation of the DFT is reported in [59], where its slow performance, due to the large number of operations and the utilized platform, is noted.

A class of DFT algorithms, denoted as Fast Fourier Transform, are developed with the aim to reduce the complexity of the DFT. One of the most wide-spread DFT algorithms is proposed by Cooley and Tuckey in [6].

A number of FFT implementations, such as NVIDIA cuFFT, FFTW, National Instruments’ LabVIEW and MathWorks’ MATLAB FFT, employ this algorithm in their FFT libraries.

3.5.1 Redundancy and Normalization in FFT

The software implementations of the FFT can generate non-redundant, redundant, normalized and un-normalized transform.

In non-redundant FFT, the relation between the size of the signal in time domain, M_{TD}, and the size of the signal in Fourier domain, M_{FD}, is presented by Equation 3.9.

$$M_{FD} = \frac{M_{TD}}{2} + 1 \tag{3.9}$$

As seen, the size of the signal in Fourier domain is approximately half the size in time domain.

In the case of redundant FFT, the signal has the same size in time domain and in Fourier domain. In redundant FFT, the seconds half of the signal in Fourier domain mirrors its first half, as shown in Equation 3.8, where H is the mid index of the signal in both time domain and Fourier domain.

$$SFD_{H+i} = SFD_{H-i}; where\ i = H, \cdots, M-1; H = \frac{M}{2}; M = M_{TD} = M_{FD} \tag{3.10}$$

In normalized FFT, applying forward FFT and then inverse FFT (IFFT) on a signal results into the original signal, Equation 3.11.

$$x_k = IFFT\big(FFT(x_k)\big); k = 0, \cdots, M-1 \tag{3.11}$$

In un-normalized FFT the result after applying first forward and then inverse FFT is multiplied, or scaled, by the number of data points M in the digital signal in time domain, Equation 3.12.

$$M \times x_k = IFFT\big(FFT(x_k)\big); k = 0, \cdots M-1 \tag{3.12}$$

Both the FFTW and CUFFT libraries, used in this chapter, perform non-redundant and un-normalized transforms. MATLAB and LabVIEW implementations, on the other hand, perform redundant and normalized FFT. Figures 3.2, 3.3 and 3.4 illustrate time domain signal and its corresponding redundant and non-redundant FFT.

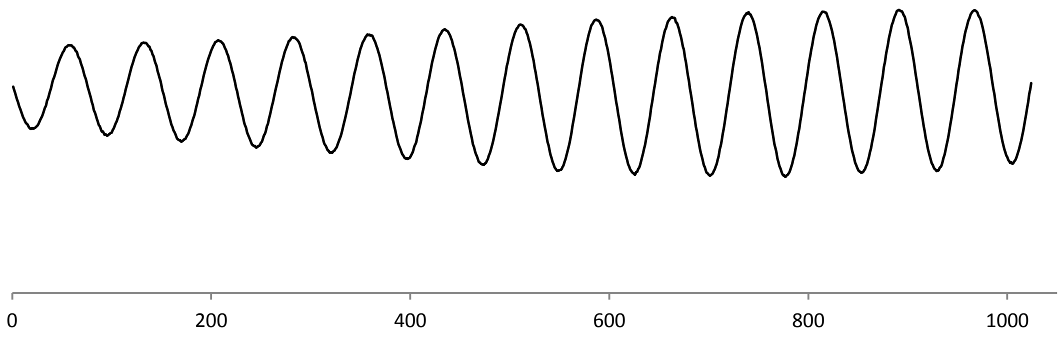


Figure 3.2: Sinusoidal signal in time domain with 1024 data points

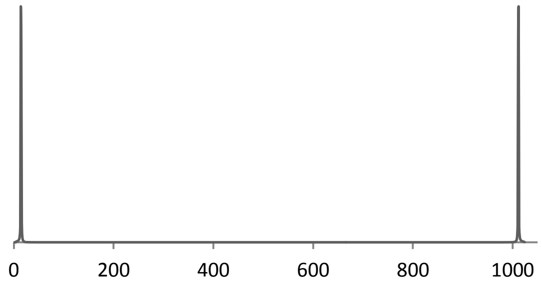


Figure 3.3: Signal in Fourier domain after redundant Fourier transform (1024 data points)

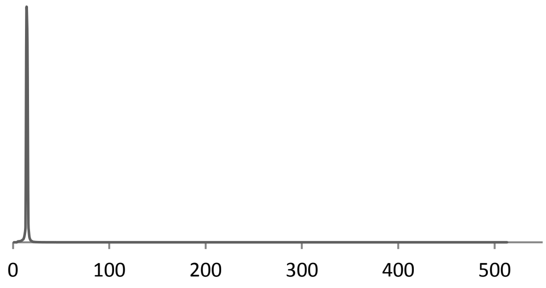


Figure 3.4: Signal in Fourier domain after non-redundant Fourier transform (513 data points)

3.6 Cross-Correlation

Cross-correlation is a fundamental signal processing algorithm. It can be applied both on analog and digital signals. Equation 3.13 presents the definition of cross-correlation denoted with · between two continuous functions, or analog signals, f and g, where $\bar{f}$ is the complex conjugate of f [55 p. 56].

$$(f \cdot g)(L) = \int_{-\infty}^{\infty} \bar{f}(t) g(t+L) dt \tag{3.13}$$

The cross-correlation between two functions or signals can be considered as a function of the lag, denoted with L in Equation 3.13. The lag is a shift, or delay, applied on one of the signals. If these signals are defined in time domain, the lag signifies a delay in time.

3.6.1 Cross-Correlation Theorem

The cross-correlation theorem, presented in Equation 3.14, allows the calculation of the cross-correlation between two signals f and g using forward and inverse Fourier transforms [55 p. 359]. This allows the implementation of cross-correlation by using FFT libraries such as the NVIDIA CUFFT and FFTW.

$$f \cdot g = IFT\left(\overline{FT(f)}\big(FT(g)\big)\right) \tag{3.14}$$

As a signal processing algorithm, the cross-correlation has applications in many areas. Those areas include adaptive optics problems in optical telescopes, as discussed in [60], where cross-correlation between a reference image and an acquired "raw" image is employed.

Another application of the cross-correlation is the MSI OCT method, which is based on cross-correlation between two signals, a Mask signal representing a particular depth and a signal generated by the OCT system in real-time. Sequential and parallel implementations of this cross-correlation based method are presented in Chapter 5.

Real-time generation of multiple en-face OCT images based on the cross-correlation based MSI method is reported in [13]. Each point from the en-face images is a result of one cross-correlation. Up to 40 en-face images of 200×192 points are reported, which amounts to 1536000 cross-correlations. Similar approach is presented in [12], where the MSI method is used to generate cross-sectional images. Both cases reach the real-time criterion by implementing the cross-correlation on the GPU.

3.7 Convolution

The convolution is a central signal processing algorithm. Like the aforementioned cross-correlation, it is applied between two functions or signals. The convolution, denoted with *, between to signals, or functions, f and g is defined in Equation 3.15. In a mathematical sense, the convolution between two real-valued functions measures the amount of overlapping of the areas closed between the functions and the X axis. Like the cross-correlation, the convolution is a function of the lag L.

$$(f * g)(L) = \int_{-\infty}^{\infty} f(t)g(L-t)dt \tag{3.15}$$

Convolution has a specific meaning when applied in signal processing, where it is used to calculate the output of a linear system based on the input and the impulse response of that system, as illustrated in Equation 3.16. In analog systems, the impulse response in question is the response of the linear system to a delta function. Parameterized by an independent variable t, the delta function has infinite value at $t=t_0$ (usually $t_0=0$) and zero at all other values for t. The corresponding function in digital signals is Kronecker delta. In this case, the function has value of 1 at point t_0 and 0 for all other t.

$$Y_k = X_k * I_k \tag{3.16}$$

Where:
X_k is the input signal,
Y_k is the output signal,
I_k is the impulse response of the linear system.

The impulse response fully characterizes any linear system [57]. The convolution allows the calculation of the output signal of a linear system based on any input signal and known impulse response.

3.7.1 Convolution Theorem

The Convolution Theorem, defined in Equation 3.17, allows the calculation of convolution between two signals or functions by using forward and inverse Fourier transforms [61 pp. 951-952].

$$f * g = IFT\big(FT(f)FT(g)\big) \tag{3.17}$$

Considering Equations 3.14 and 3.17, the computation of cross-correlation and convolution follow a similar route, despite their theoretical differences.

Convolution as a signal processing tool has applications in many areas. In image processing it is frequently used for edge detection and image segmentation, as reported in [62].

In audio signal processing, convolution is used to reconstruct the reverberation of a physical space, such as room or hall, caused by a sound. The reverberation is the result of the convolution between the sound and the impulse response, which is a prerecorded sound characterizing the physical space.

3.8 Window Functions

Signals generated by real-time systems over a long period of time require some form of partitioning, in order to be processed, [58]. This partitioning divides the input signal into a number of sub-signals. In many cases in practice, the partitioned signals need to have some properties, such as retaining the periodicity. This is done by applying mathematical transformations on the partitioned signals, called window functions. In addition, a key purpose of the window functions is to improve the signal-to-noise ratio.

3.8.1 Rectangular Window

A basic example of a window function is the rectangular window. In this function, window boundaries W_1 and W_2 are defined. The data points within the window retain their values, while the data points outside the window are set to zero, as shown in Figure 3.5.

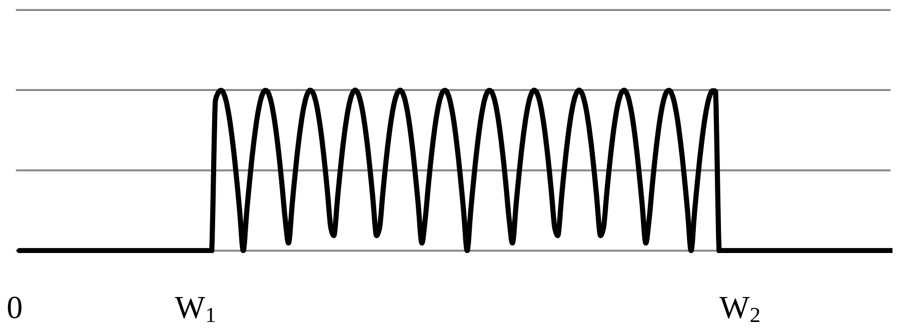


Figure 3.5: Rectangular window

Applied on a signal in Fourier domain, this window function is used to avoid lower and higher frequency samples, depending on the case and how W$_1$ and W$_2$ are defined.

3.8.2 Hann Window and Hamming Window

Fourier transforms, especially the ones employed in OCT, assume periodicity of the signals. A discontinuity of this periodicity results in an increased noise in the signals. Hann and Hamming window functions provide solution to this problem [63].

The Hann Window, named after Julius Ferdinand von Hann (1839-1921) [64], is illustrated by Equation 3.18.

$$y(n) = 0.5 \times x(n) \times \left(1 - cos\left(\frac{2\pi n}{N-1}\right)\right) \tag{3.18}$$

Where:

x is a partition of the input signal

y is the windowed signal

N is the number of data points (samples) in the partition, n = 0, ..., N-1

Hamming Window, proposed by Richard Wesley Hamming (1915-1998), is a modified version of the Hann Window. It is expressed by Equation 3.19, using the same notation as in Equation 3.18.

$$y(n) = x(n)\left(0.54 - 0.46 \times cos\left(\frac{2\pi n}{N-1}\right)\right) \tag{3.19}$$

Both Hann and Hamming windows gradually reduce the amplitudes of the signal at the end points. Doing so ensures continuity of the periodicity of the signals. Otherwise, a spectral leakage can be observed in the Fourier domain of the signal, where energy from one frequency leaks to the neighboring frequencies, which results in increased level of noise in the signal, as discussed in [58] and [65 pp. 540-541]. Figure 3.6 illustrates the effect of Hann window on a sinusoid signal. A corresponding Hamming window applied on the same signal would result in a signal with similar shape.

In OCT methods, the resolution, or sharpness, of the images depends on the peaks of the signals. In CFD OCT method, a peak without leakage results in images with better axial resolution [4].

The difference between the Hann Window and the Hamming Window is the behavior of the windowed signal at the end points, when n=0 and n=N-1. In Hann windows, the endpoints are equal to zero. In Hamming window, on the other side, the endpoints are close to zero, but do not reach it, as seen in Equations 3.18 and 3.19.

The aforementioned window functions increase the signal-to-noise ratio when used in OCT methods and improve the quality of the generated images. The choice of the window function depends on the signals generated by the OCT system and is based on the qualities of the resulting images. This choice can be made after direct observation of the cross-sectional and en-face images.

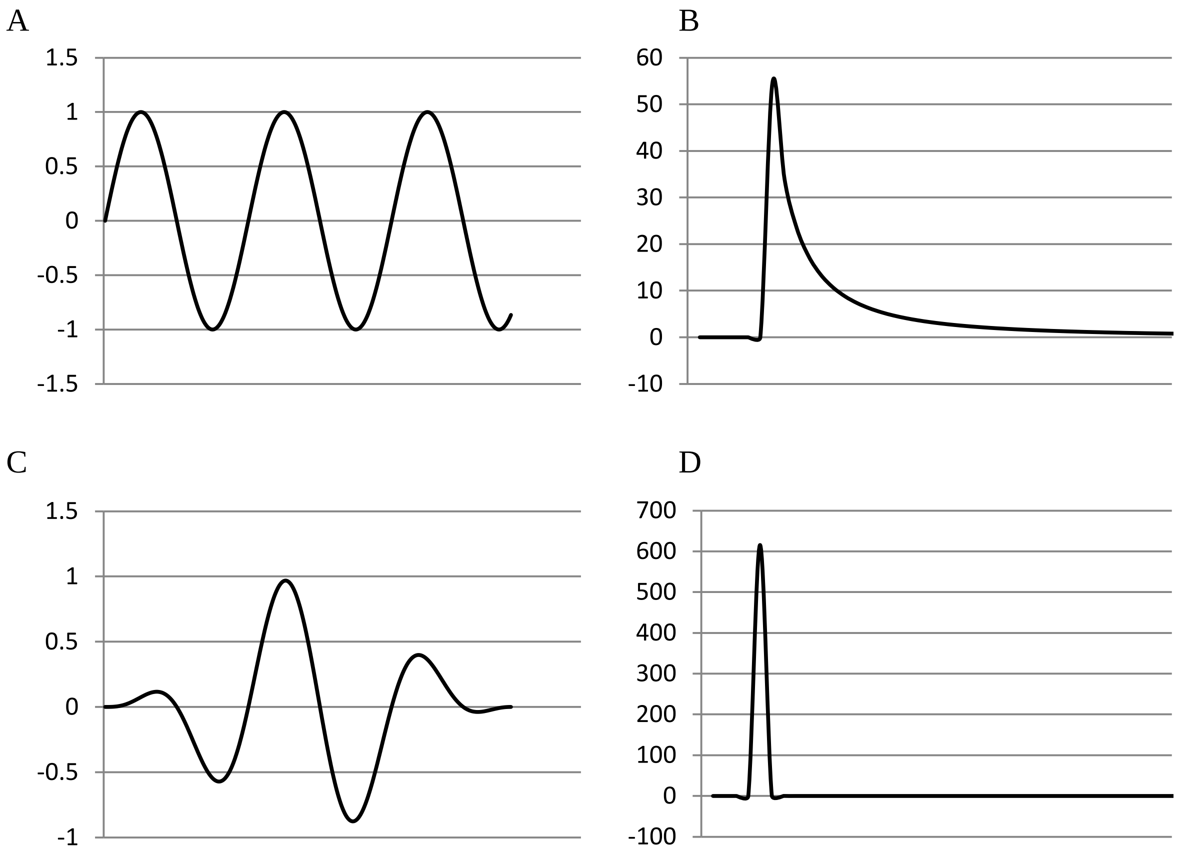


Figure 3.6: Sinusoid signal with discontinuity in the periodicity (A), its corresponding Fourier transform (B), the sinusoid signal after Hamming Window (C) and the corresponding signal after Fourier transform (D). The width of the peak in the signal in Fourier domain (B) displays leakage to the neighboring frequencies which is eliminated in the signal in (D)

3.9 Digital Signal Processing in OCT

Optical coherence tomography has the ability to image semitransparent objects, such as biological tissues, layers below the surface. It is based on the properties of the electromagnetic radiation at specific frequencies to penetrate semitransparent materials and to reflect visual information about the material [66]. It has application in two main areas: ophthalmology [67] and art investigation [68]. Currently, there are two major types of OCT: time domain OCT and Fourier domain OCT. Fourier domain OCT can be spectral domain and swept source.

The system employed in this research is a swept source Fourier domain OCT system with Alazartech digitizer and a National Instruments' LabVIEW Virtual Instrument (VI) project, acting as data acquisition software and user interface.

3.9.1 Operation of OCT Systems

The operating of a typical OCT system is based on the Michelson Interferometer [69 pp. 16-18], as illustrated in Figure 3.7. Two scanning mirrors, horizontal (fast) and vertical (slow) guide low energy near-infrared electromagnetic ray along the surface of the object. Normally, a triangular waveform is employed to drive the scanning mirrors while they traverse the surface of the object [11].

Semitransparent objects reflect the light from the surface and layers below the surface. The reflected light is captured by a photo detector, which is transformed into electrical signal. The electrical signal is digitized by analog-to-digital converter, or digitizer, which is usually implemented as an extension board, connected to a computer system via an expansion slot, such as the PCI Express.

Data Acquisition software, such as LabVIEW VI project, captures the digital signal and makes it available for further processing, storing and visualizing. The LabVIEW project also controls the scanning mirrors and synchronizes them with the data acquisition.

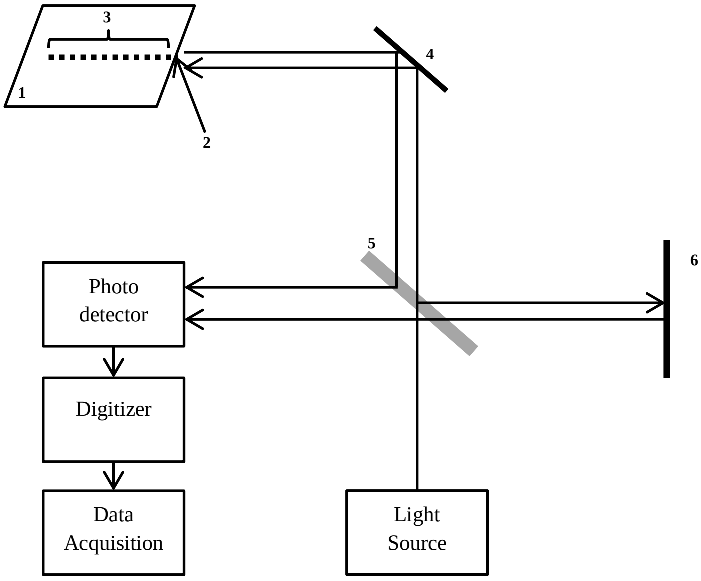


Figure 3.7: Main components in OCT, where 1 denotes the object, 2 the scanned point, 3 the points below the surface, 4 scanning mirrors, 5 beam splitter, 6 reference mirror

3.9.2 Imaging in OCT

For every scanned point from the surface of the object, OCT systems acquire an array of data points, denoted as channeled spectrum. This channeled spectrum has sinusoidal shape, as shown in Figure 4.1.B. It holds the reflectivity, or intensity, of the points which are located below the scanned point. All channeled spectra, acquired during a single scan of a line or an area along the surface of the object, are denoted in this research as one OCT frame and thus extending the notation of frame employed in [58] for the partitioning of a larger, theoretically infinite signal into frames. Cross-sectional and en-face images can be obtained by applying mathematical transformations on the OCT frames, known as OCT methods, as shown in Figure 3.8.

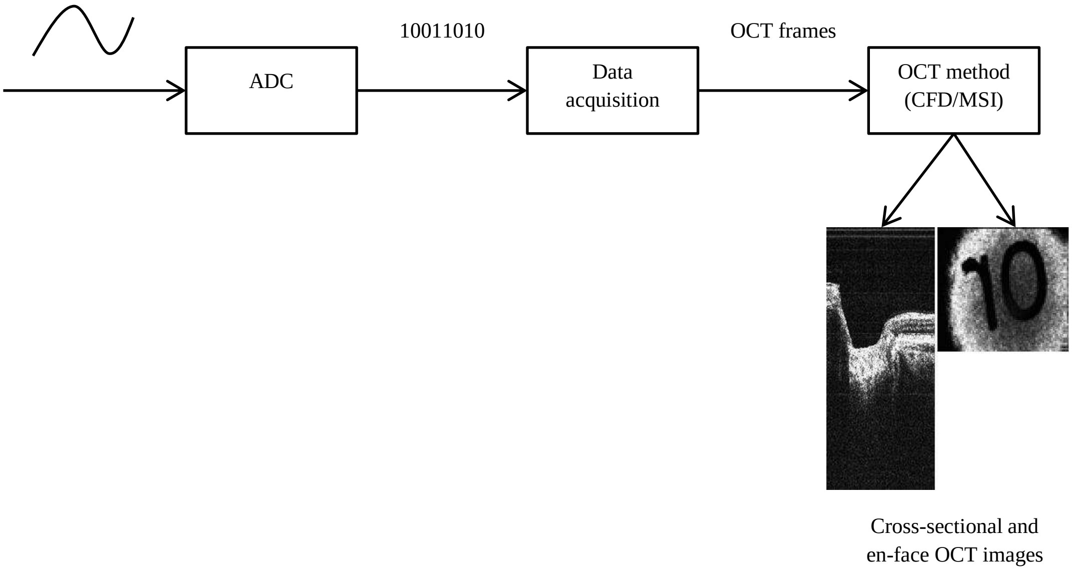


Figure 3.8: Imaging in OCT systems

3.9.2.1 Cross-Sectional OCT Imaging

During cross-sectional imaging, only the horizontal (fast) scanning mirror operates. The vertical (slow) mirror is in a fixed position. As a result, the electromagnetic ray traverses a line on the surface of the object and collects one channeled spectrum for every scanned point along that line. The horizontal scanning mirror in the OCT system used in this research operates at a frequency of 100 Hz. The system acquires new cross-sectional OCT frame every 10 milliseconds.

3.9.2.2 En-Face OCT Imaging

During en-face imaging both scanning mirrors operate. In this case, the electromagnetic ray traverses an area from the surface of the object. As a result, multiple lines, each one consisting of multiple channeled spectra, are collected. In the OCT system studied in this research the vertical scanning mirror operates at 1.25 Hz, or new en-face OCT frame is acquired every 800 milliseconds.

Multiple cross-sectional images can be extracted from the en-face OCT frame. They can be horizontal and vertical, as presented in Chapter 6 and in [12].

3.9.3 OCT Methods

The aforementioned OCT methods transform the OCT frames into cross-sectional and en-face images. Two OCT methods are studied in this research, namely the CFD and the MSI. Both of them employ discrete Fourier transform. These methods calculate the intensities of points from particular depths. These intensities can be scaled to the gray-scale interval and visualized.

The aforementioned partitioning of the digital signal into sub-signals is applied in OCT, where these sub-signals are denoted as channeled spectra. The channeled spectra are signals with fixed size M before DFT and with size M' after DFT. In a non-redundant FFT M'=M/2+1. The OCT system studied in this research generates channeled spectra with M=1024 data points before DFT and M'=513 data points after DFT.

3.9.3.1 CFD OCT Method

This method, presented in Equation 3.20, is widely used in Fourier domain OCT systems. It consists of the following steps:

1. Apply discrete Fourier transform on the channeled spectrum
2. Apply absolute value on result
3. Multiply with a gray-scale coefficient (GSC) to obtain a value from the gray-scale interval

$$Point_i = GSC \times |DFT(CS)|; i = 0, \cdots, M' - 1 \tag{3.20}$$

Theoretically, in Fourier domain OCT the depth is resolved by inverse discrete Fourier transform [70], [71]. However, due to the similarities between the forward and inverse DFT, both transforms generate identical images. This allows both CFD and MSI OCT methods to start with the same signal processing steps, as illustrated in Figure 5.7.

Applied on one channeled spectrum with M data points, this method generates M'=M/2+1 intensities due to the none-redundancy of the employed FFT. Figure 3.9 presents a channeled spectrum with 1024 data points, as acquired by an OCT system. Figure 3.10 presents the result of the CFD OCT method, applied on this channeled spectrum. It has 513 data points from the gray scale interval [0..255], which corresponds to the intensities of 513 points below the scanned point, as seen in Figure 3.7.

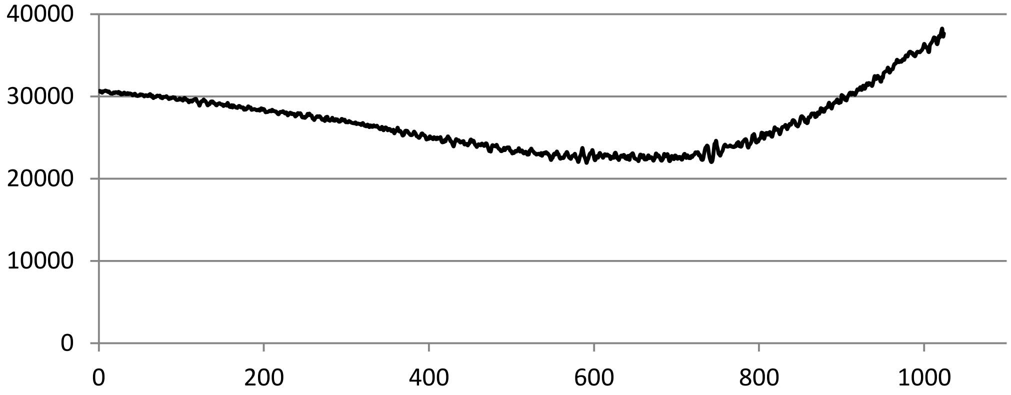


Figure 3.9: Channeled spectrum with 1024 data points acquired from an OCT system while imaging semitransparent object with reflectivity from multiple depths

In Figure 3.10, the first values are set to zero. These values correspond to low-frequency noise, denoted in some literature as DC noise.

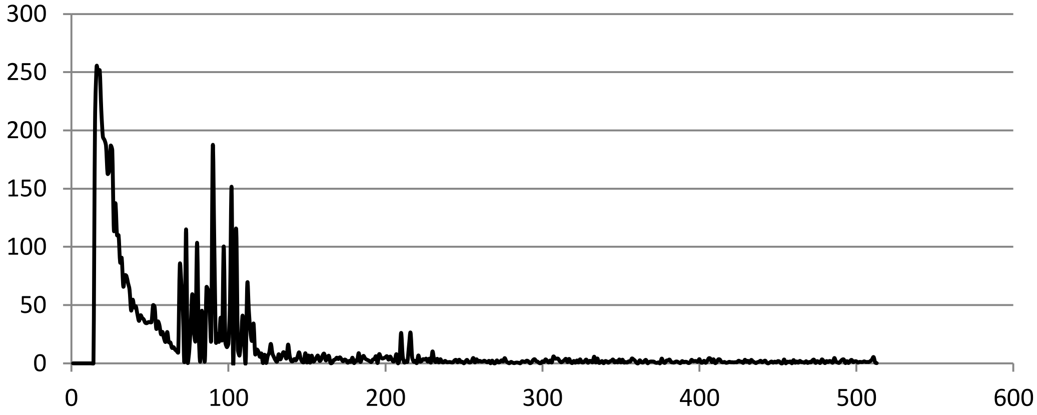


Figure 3.10: Output signal presenting the gray-scale values generated by applying CFD OCT method on the channeled spectrum from Figure 3.9

3.9.3.2 MSI OCT Method

This method is proposed in [4]. Improvements brought by this method are presented and discussed in [72] and [73]. The application of this method in ophthalmology is discussed in [74] and [75]. Its GPU-based parallel implementation is proposed in [13]. The method is based on cross-correlation between two signals, the channeled spectrum and a previously recorded digital signal, called Mask. These two signals need to be equal in size. Equation 3.21 illustrates the method.

$$Point_D = GSC \times \sum \left| IDFT\left(DFT(CS) \times \overline{DFT(MS_D)}\right) \right| \tag{3.21}$$

Mask signals are obtained by using a flat mirror as an object. They are presented in arbitrary units. Each Mask signal represents a particular depth D. This depth is determined by the difference in the optical path lengths between the beam splitter and the reference mirror on one side and the beam splitter and the object on the other [4]. Obtaining images from multiple depths requires multiple Mask signals. Figure 3.11 presents two mask signals corresponding to two different depths.

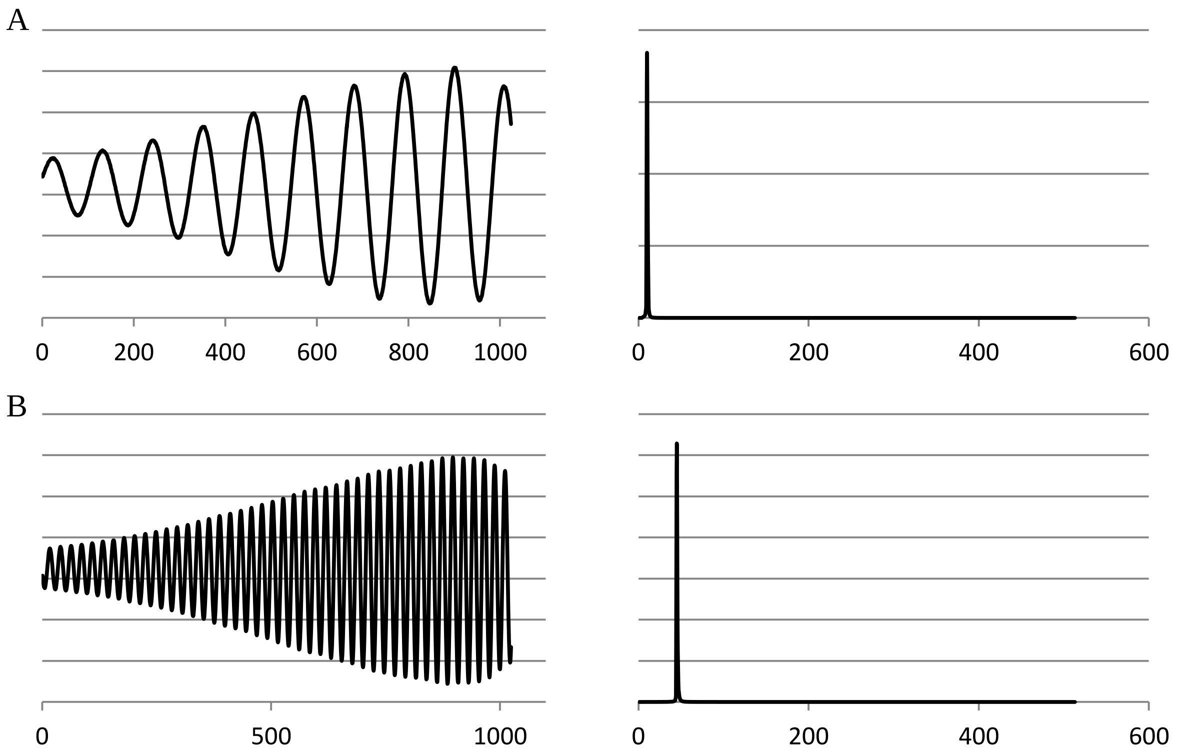


Figure 3.11: Two Mask signals (A) and (B) before and after DFT, representing two different depths

The Mask signals presented in Figure 3.11 can be considered as channeled spectra obtained while imaging a flat mirror with two different optical paths lengths. In these channeled spectra the reflectivity is from one depth, the surface of the mirror at its current position. An OCT system employing this method requires a set of Mask signals permanently stored in the memory. The peaks in Figure 3.11 directly correspond to the intensity, or the brightness, of the surface of the imaging mirror. The thicknesses of these peaks represent the axial resolution of the OCT system, or the resolution in depth. This thickness, and consequently the resolution, can be altered by applying a window function.

The cross-correlation algorithm used in the MSI OCT method measures the level of similarity between the channeled spectra generated by the OCT system while imaging and object and the Mask signal. The Mask has maximum intensity from one particular depth. The resulting reflectivity, or the intensity, from that particular depth will be quantified by the cross-correlation, e.g. higher cross-correlation corresponds to higher intensity and vice-versa. The level of similarity has a numerical value, which after scaling represents a value on the gray-scale interval. This value corresponds to the reflectivity from a particular depth below the scanned point.

In the CFD OCT method, as seen in Equations 3.20, one Fourier transform of the channeled spectrum of M data points results in M' intensities, or points, which can be visualized after calculating the absolute value and the gray-scale value. In the MSI OCT method, on the other hand, the intensity of one point from a particular depth is generated by one cross-correlation between a channeled spectrum and a Mask signal. This cross-correlation consists of two forward and one inverse discrete Fourier transforms, as seen in Equation 3.21. This is a significant increase in the amount of computations in the MSI OCT method.

3.9.4 Parallel Acceleration in OCT

Significant efforts are made to improve the images generated by OCT systems. These efforts can be divided into two groups: hardware solutions and numerical solutions. The numerical solutions come in the form of new algorithms and methods. The MSI is an example for such method. These numerical solutions are applied on considerable amount of data generated by OCT systems. On the other hand, the introduction of additional computations does not change the real-time criteria of the OCT systems.

Therefore, the OCT systems are expected to benefit from the parallelization of these numerical solutions. A significant improvement of the performance is expected particularly from the parallel computations on the GPU. But the introduction of the GPU Computing in OCT brings two main challenges in the form of performance overheads:

1. A GPU application, for example one written in CUDA C, is not a data acquisition software. It relies on other software to provide the data. These two software solutions need to be able to communicate with each other and exchange considerable amount of data.

2. The GPU processes data which is residing on the global GPU memory. These data need to be transferred over the PCI Express bus. Even the newly introduced unified address space does not eliminate the need for this copy.

The aim is to implement these numerical solutions on the GPU with minimum latency, which will not prevent the real-time operation of the OCT system. This is a significant challenge, both to the GPU architectural design and the design and implementation of the parallel algorithms and programs.

A number of efforts in this direction are made. For example, in [71], an image of 1024×512 is processed with the rate of 27.9 frames per second. The processing is accelerated by the CUDA FFT library.

A GPU-implemented interpolation was reported in [76], where three approaches to interpolation are compared, namely nearest neighbor, linear and spline. The interpolation was followed by FFT and absolute value, which are the steps of the CFD OCT method. The proposed solution allowed improvements to cross-sectional images of size 1024×1024 pixels and resulted in 25 Hz rate of processing and display.

A performance improvement by the factor of 39 is reported in [77]. In this case, the GPU performed linearization and FFT, which resulted in real-time generation of cross-sectional images with size of 1024×512 pixels.

The aforementioned cases of GPU utilization in OCT improve the generation of a single cross-sectional image generated by signal processing steps equivalent to the CFD OCT method. The improvement of the performance depends on the processing steps, the size of the processed data, and the architecture of the GPU. Figure 3.12.A illustrates the generalized concept of these approaches.

Chapter 5 and Chapter 6 of the thesis present coarse-grained CPU-based and fine-grained GPU-based parallel implementations of both CFD and MSI methods. Figure 3.12.B illustrates the GPU-based parallel optimization of the MSI method, which leads to comprehensive imagery, presented in Chapter 6, consisting of 40 en-face, one confocal and two cross-sectional images, which amounts to 1775104 processed points in total (40×200×192+200×192+200×512+192×512). Besides the parallelization efforts, this result is also contributed by the lower real-time requirement for en-face OCT imaging, which is 800 milliseconds. This result also depends on the utilized GPU architecture.

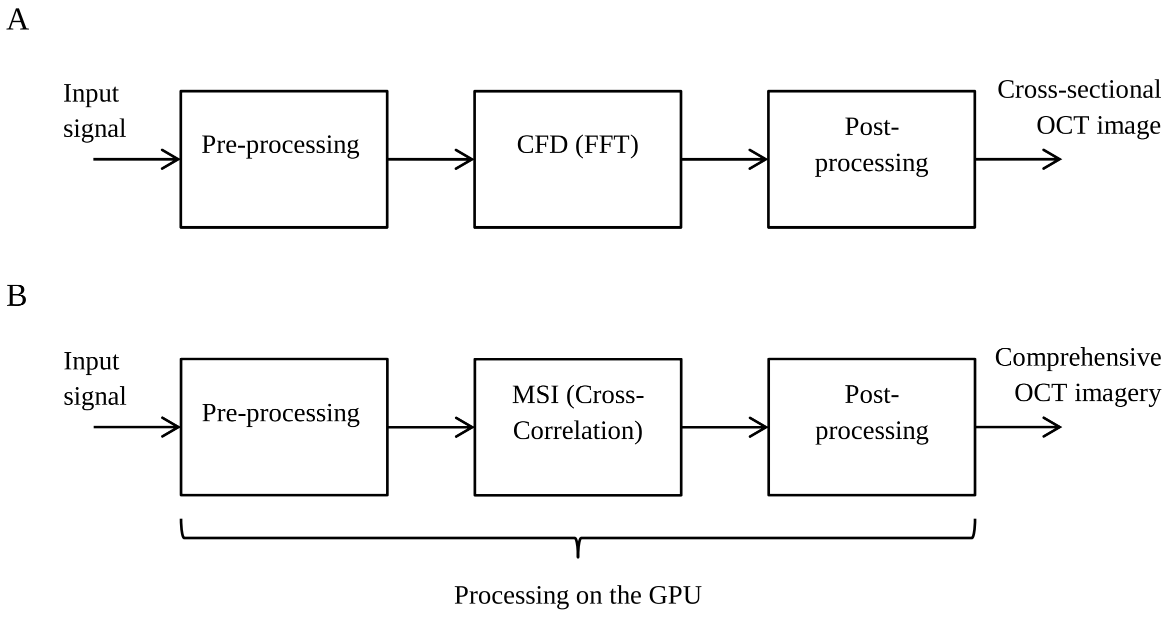


Figure 3.12: GPU utilization in OCT

3.10 Summary

This chapter presented some of the most popular digital signal processing algorithms and their applications in OCT. Two OCT methods, the CFD and the MSI, employing the DSP algorithms are presented. Both of them operate in the Fourier domain. The CFD method is well established and widely used in Fourier domain OCT systems. The newly introduced MSI method brings some improvements in the operation and the output of the OCT system, but increases significantly the amount of the computations.

Regardless of the employed OCT method, the real-time criterion of the OCT system remains the same. Sequential processing of MSI OCT results in up to four en-face images generated in real-time, as reported in [72]. This amount of en-face images in the case of ophthalmology is not sufficient. Increased number of images in sequential implementation will result in skipped frames and delays. It will prevent the OCT system from generating images in real-time. This reduces the efficiency of the OCT system if used in ophthalmology, where the patient voluntarily and involuntarily blinks and changes sight direction.

Various parallel approaches to DSP algorithms and OCT methods are presented in the next Chapters 4 and 5. Their performances, in terms of latency and speed-up, are measured and reported. These optimizations are expected to absorb the increased computations of the more intensive MSI OCT method and to improve the overall performance of the OCT system.

Chapter 4

Parallel Optimization of Digital Signal Processing Algorithms

4.1 Introduction

This chapter explores how multithreading for multi-core and many-core architectures improve the performance of a number of DSP algorithms. The selected algorithms are widely used in many different areas, such as communications, image, audio and video processing. The parallelized implementations are compared with sequential implementations, carried out by a single thread of execution.

The parallelism, described in this chapter, is achieved by multithreading. The basic principle of the multithreading programming model is the division of a larger and more complex task into a number of smaller and simpler tasks, called threads, and the concurrent, and where possible simultaneous, execution of these threads.

The parallel optimizations proposed in this chapter derive from sequential algorithms. These algorithms are based on well-known and widely used digital signal processing theorems.

The multi-core approach, based on a multi-core CPU, is developed as Microsoft Visual C++ multithreaded application utilizing the Windows API multithreading support. Its performance is measured on Intel Core i7 5820K CPU with 6 cores, at 3.30 GHz and 4 GB of RAM. The many-core approach, based on the GPU, is implemented as an NVIDIA CUDA C application. Its performance is measured on NVIDIA Tesla K40 GPU with 2880 CUDA cores and 12 GB of onboard memory.

The aim of this chapter is to identify optimal approaches to the DSP algorithms. These DSP algorithms form the basis in the OCT methods, presented in the next Chapter 5. The overall aim is to integrate these OCT methods into working OCT system. This integration determined the choice of the aforementioned programming environments.

4.2 Techniques for Measuring the Performance

The performances of the parallel approaches are evaluated by measuring the elapsed time. The time needed by a single thread of execution to complete the signal processing tasks is considered as a baseline. This single-threaded performance is utilized in presenting the speed-up introduced by the parallel optimizations.

In order to confirm the obtained performance data, the elapsed time is measured using two different techniques, the Performance Counter and GetSystemTime function. Both approaches reported virtually identical processing times.

4.2.1 Performance Counter

This counter can be used to monitor the performance of the computations, as shown in Algorithm 4.1. It can be accessed by using the following Windows Application Programming Interface (API) functions:

- QueryPerformanceFrequency, retrieves the number of cycles per seconds of the performance counter

- QueryPerformanceCounter, provides the current value of the performance cycle counter

On Intel Core i7 running Microsoft Windows 7, the frequency of the counter is 3,220,839 cycles per second. This allows a maximum resolution of approximately 3 microseconds. This frequency allows sub-millisecond measurement of the performance, as demonstrated in Algorithm 4.1.

Algorithm 4.1: Utilization of performance counter

```
QueryPerformanceFrequency ( &Frequency )
CyclesPerMillisecond = Frequency.QuadPart / 1000
CyclesPerMicroseconds = Frequency QuadPart / 1000000
QueryPerformanceCounter ( &StartTime )
        CodeToMeasure
QueryPerformanceCounter ( &EndTime )
ExecutionTimeInCycles = EndTime.QuadPart - StartTime.QuadPart
ExecutionTimeInMilliseconds = ExecutionTimeInCycles /CyclesPerMillisecond
ExecutionTimeInMicroseconds = ExecutionTimeInCycles / CyclesPerMicroseconds
```

4.2.2 GetSystemTime

The GetSystemTime is a Microsoft Windows API function, which retrieves the current system time and stores it in a predefined structure SYSTEMTIME with the following two-byte fields: wYear, wMonth, wDayOfWeek, wDay, wHour, wMinute, wSecond, wMilliseconds. The highest possible resolution in this approach is milliseconds. Algorithm 4.2 illustrates its usage.

Algorithm 4.2: Utilization of GetSystemTime

```
GetSystemTime ( &StartTime )
        CodeToMeasure
GetSystemTime ( &EndTime )
StartTimeInMilliseconds = StartTime.wMilliseconds + 1000×StartTime.wSeconds
                + 1000×60×StartTime.wMinutes + 1000×60×60×StartTime.wHour
EndTimeInMilliseconds = EndTime.wMilliseconds + 1000×EndTime.wSeconds
                + 1000×60×EndTime.wMinutes + 1000×60×60×EndTime.wHour
ProcessingTimeInMilliseconds = EndTimeInMilliseconds - StartTimeInMilliseconds
```

4.2.3 Measuring the Performance of the GPU

In this chapter, the reported time of the GPU-based computations does not include the time to transfer the data to and from the global GPU memory over the PCI Express bus. This chapter compared the computations on the CPU and the GPU with data already available for processing. Algorithm 4.3 illustrates the measurement of the latency of the GPU-based computations. In this case, both the aforementioned GetSystemTime and Performance Counter can be used.

Algorithm 4.3: Measurement of GPU Latency

```
cudaDeviceSynchronize ( )
SaveCurrentTime ( &StartTime )
    CodeToMeasure ( kernels )
    cudaDeviceSynchronize ( )
SaveCurrentTime ( &EndTime )
```

4.3 Format and Size of the Digital Signal

The DSP algorithms, studied in this chapter, find application in the area of OCT, among others. Therefore, the size and structure of the digital signals employed in this chapter are influenced by the signals in the OCT systems. As described in Chapter 3, the OCT systems generate OCT frames at a certain rate. These frames are divided into multiple channeled spectra. An OCT frame consisting of multiple channeled spectra corresponds to the more general notion of a digital signal consisting of multiple sub-signals.

The algorithms and methods studied in this research are applied per sub-signal in the more general case and per channeled spectrum in the case of OCT.

In the course of this research, a number of OCT frames with different sizes are collected during cross-sectional and en-face imaging. These frames are processed and visualized as cross-sectional and en-face images, as presented in Chapter 5. The sizes of these images and the number of points in these frames depend on the characteristics and the capabilities of the particular OCT system. The digital signals employed in this chapter reflect these sizes and include a wider range, as presented in Table 4.1.

Table 4.1: Sizes of digital signals and OCT frames

Size of digital signal (OCT Frame) in data points	Number of sub-signals (channeled spectra)
1024	1
2^4×1024	16
2^6×1024	64
100×1024[A]	100
200×1024[A]	200
2^8×1024	256
500×1024[A]	500
2^{10}×1024	1024
2^{12}×1024	4096
2^{14}×1024	16384
200×192×1024[B]	38400
(2^{16}-1)×1024[C]	65535

[A] Size of OCT frame collected during cross-sectional imaging, as presented in Chapter 5

[B] Size of OCT frame collected during en-face imaging, as presented in Chapter 5

[C] The largest digital signal employed in this chapter. In this case, the number of sub-signals (channeled spectra) is equal to the maximum number of blocks in the GPU grid of the NVIDIA Fermi architecture

The algorithms in this chapter are tested with two different signals, illustrated by Figure 4.1. The first one is a sinusoid generated by the proposed parallel implementations during initialization. The second one is the digital signal generated by the OCT system. In both cases, the performances of the parallel implementations yield equivalent results.

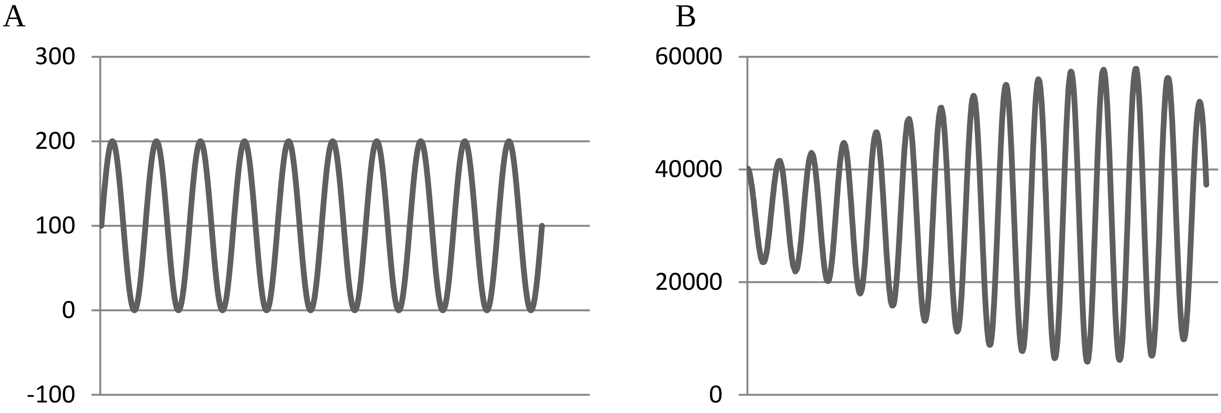


Figure 4.1: Synthetically generated sinusoidal signal (A) and signal, or channeled spectrum, generated by the OCT system (B)

4.4 CPU-Based and GPU-Based Multithreading

CPU-based multithreading is implemented by launching multiple parallel threads within a single process. This is achieved by the API functions CreateThread and WaitForMultipleObjects in Microsoft Windows. The corresponding POSIX (Portable Operating System Interface) functions are pthread_create and pthread_join, Figure 4.2. The combined utilizations of these functions can implement the fork-join parallel model, discussed in [26] and [49]. This model is followed by OpenMP.

Theoretically, the maximum number of parallel threads launched by a single process is defined by the size of the address space dedicated to the process and by the address spaces dedicated to each thread. These values vary in different operating systems [78].

A series of tests are made to evaluate the overhead latency introduced by launching multiple parallel threads on the CPU and on the GPU, using Microsoft Visual C++ and NVIDIA CUDA C respectively. Two groups of parallel threads are used to measure these overheads: empty threads and threads querying their identifier. The results are presented in Table 4.2 and Figure 4.3.

Parallel threads processing data generated by a single source need to partition the input data. This division of the input data is based on the identifiers of the parallel threads. These identifiers index both the parallel threads and the partitions of data. Therefore, Table 4.2 includes the latencies of threads obtaining their identifiers.

The CPU and the GPU threads obtain their identifiers differently. In the CPU case, an identifier is obtained by calling the Windows API function GetCurrentThreadId. In the GPU case, this is done by querying built-in variables. This difference further contributes to the differences in the latencies of the threads.

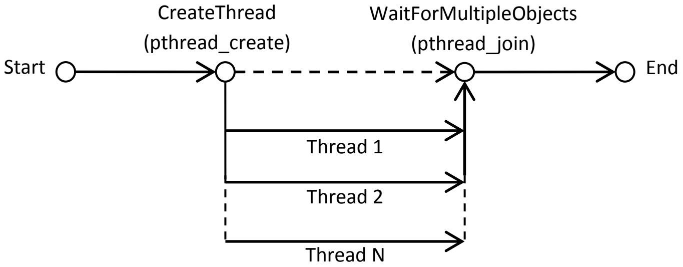


Figure 4.2: CPU-based multithreading

Table 4.2: Overheads in CPU and GPU multithreading in milliseconds

Number of parallel threads	Launch		ID	
	CPU	GPU	CPU[1]	GPU[2]
2	0.8	0.04	1.2	0.05
4	2.1	0.04	2.3	0.05
6	3.3	0.04	3.5	0.05
8	3.7	0.04	4.6	0.05
12	6.8	0.04	7.4	0.05
16	8.9	0.04	12.5	0.05
32	17.4	0.04	25.4	0.05
64	41.2	0.04	53.2	0.05
128	92.4	0.05	132.6	0.05

[1] Each CPU-based parallel thread obtains its own identifier using the Windows API function GetCurrentThreadId

[2] Each GPU-based parallel thread retrieves its own identifier using the expression:

int ThreadID = (blockDim.x) × (blockIdx.x) + (threadIdx.x)

In the GPU-based approach, the overheads caused by launching parallel threads are virtually the same for the listed number of threads. As seen from Table 4.2, the GPU-based multithreading has superior performance. It has the potential to deliver real-time performance in more cases.

In the CPU-based approach, the overheads from thread management are significant. Launching more than 16 parallel threads exceeds the real-time requirement in cross-sectional OCT imaging. These limitations reflect the optimal number of parallel threads in the different CPU-based approaches, presented in this chapter and in the next Chapter 5.

In the CPU-based approaches, instead of using the GetCurrentThreadId, the indexing of the data partitions is implemented by variables, passed as parameters to the functions launching the parallel threads. This is done for two reasons:

1. As shown in Table 4.2, the call to the GetCurrentThreadId has a significant latency, which will affect a possible real-time performance.

2. Calls to GetCurrentThreadId return system-wide identifiers. Normally, they do not start from 0 and there is no guarantee they will use successive numbers. Therefore, they are not suitable for indexing data partitions.

As seen in Table 4.1, the overheads introduced by the GPU-based approach are much smaller, compared to the CPU-based approach. On the other hand, GPU applications can only operate with data already in the GPU memory. This brings different types of overheads: the time to copy the data from the CPU memory to the GPU memory and back. These latencies affect the real-time performance of the GPU-based processing, which is discussed in Chapter 5.

This chapter reports the latencies of the computations on the CPU and on the GPU. It does not include the latencies of the data transfers over the PCI bus. These latencies are reported separately in Table 5.3 and 5.4.

4.4.1 Mapping the Digital Signal on the GPU Grid

As discussed in Chapter 2, the GPU organizes the parallel threads into thread blocks which form the GPU grid. A maximum parallelism and utilization of the GPU grid would be achieved if each data point is processed by one parallel thread. Also, some algorithms, such as the parallel reduction, require cooperation between the parallel threads processing one sub-signal or one channeled spectrum, which can be done by parallel threads within the same block. This cooperation is in the form of utilizing a shared per-block memory and the need for synchronization. This is done by processing each sub-signal by one thread block. Figure 4.3 illustrates the one-to-one mapping of the sub-signals onto the thread blocks and one-to-one mapping of the data points onto the parallel threads.

In the NVIDIA Fermi architecture, the largest number of threads per block is 1024 and the largest number of blocks per dimension is 2^{16}-1. The NVIDIA Kepler architecture has the same number of threads per thread block, but the number of blocks per dimension is increased to 2^{32}-1 [46]. As a result, the maximum amount of parallel threads in one-dimensional GPU grid is $(2^{16}-1)\times1024$ in Fermi architecture and $(2^{32}-1)\times1024$ in Kepler architecture.

A digital signal consisting of maximum 2^{16}-1 sub-signals with maximum 1024 data points within each sub-signal allows a one-to-one mapping onto the GPU grid in both architectures, as shown in Figure 4.3. In most cases, the OCT frames have significantly smaller number of sub-signals. However, a one-to-one mapping of larger digital signals is still possible, but it will require two or three dimensional GPU grid.

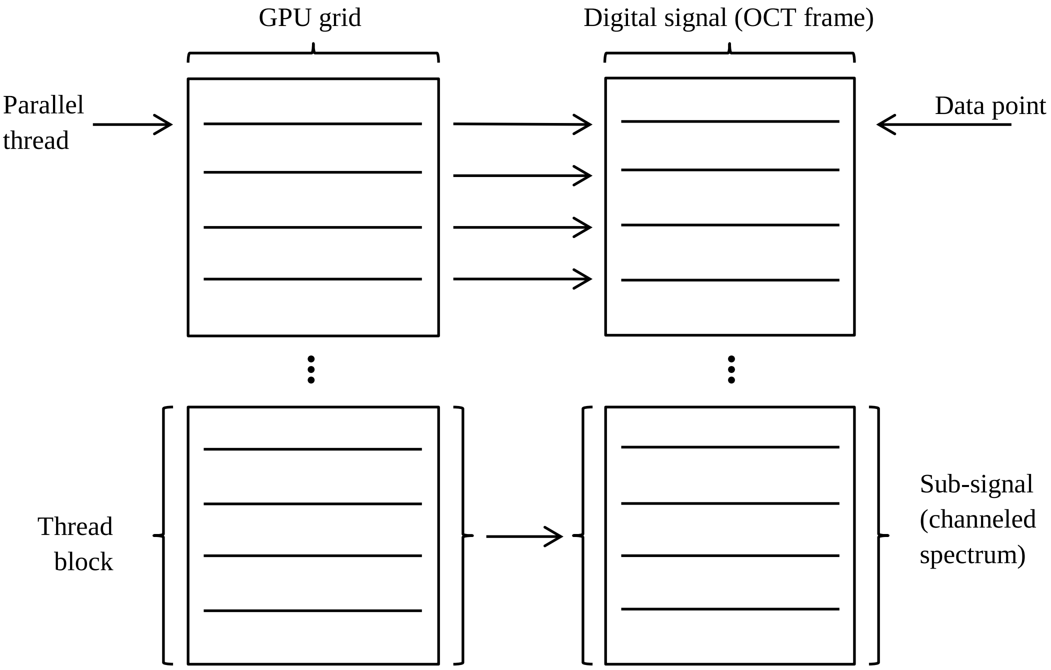


Figure 4.3: Mapping the data points and sub-signals on the GPU grid

4.4.1.1 Hamming Window Example

The following example of Hamming Window [63] , discussed in Chapter 3, applied on a digital signal f demonstrates the implementation of the kernel and the mapping of the digital signal onto the GPU grid. The CUDA C code in Listing 4.1 presents the parallel GPU-based implementation, based on Equation 4.1 and the sequential Algorithm 4.4. In this example, f can represent both digital signal consisting of N sub-signals where each sub-signal has M data points, or OCT frame consisting of N channeled spectra where each channeled spectrum has M data points.

Algorithm 4.4: Sequential approach to Hamming window

```
for j = 0 to N-1 do
    for i = 0 to M-1 do
        Angle = ( 2×π×i ) / M
        $f_{i,j}$ = 0.5×$f_{i,j}$×( 1 - Cos ( Angle ) )
    end for
end for
```

$$f_{i,j} = \frac{1}{2} f_{i,j}\left(1 - cos\left(\frac{2\pi i}{M}\right)\right); i = 0, \cdots, M-1; j = 0, \cdots, N-1 \tag{4.1}$$

Listing 4.1: NVIDIA CUDA C implementation of Hamming window

```
__global__ void HammingWindowKernel(float *f)
{
        int i,Index; float Angle;
        i=threadIdx.x;
        Index= (blockDim.x)*(blockIdx.x)+threadIdx.x; // Map the index of digital signal on the grid
        Angle=(float)((2.0*3.14159*i)/(blockDim.x)); // Calculate the angle for the cosine function
        f[Index]=(float)(0.5*f[Index]*(1.0-cos(Angle))); // Actual Hamming window
}
...
int M=1024; // Number of threads per block or size of sub-signal
int N=65535; // Number of blocks per 1D grid or number of sub-signals
cudaMalloc(&f,M*N*sizeof(float)); // Allocate GPU memory for signal f
HammingWindowKernel<<<N,M>>>(f); // kernel launching M*N parallel threads
```

In the GPU kernel performing the Hamming window, as shown in Listing 4.1, the number of threads per block is equal to the number of data points in the sub-signal and the number of blocks per GPU grid is equal to the number of sub-signals in the digital signal. All kernels presented in this thesis follow the same mapping of the data onto the GPU grid.

4.5 Fourier Transforms

Large part of the signal processing studied in this research involves transition between time domain and Fourier domain. This is accomplished by applying one-dimensional discrete Fourier transform on each sub-signal or channeled spectrum. In this research, this transform is considered an atomic operation. Therefore, in the CPU-based parallel approaches the number of possible parallel threads must divide the digital signal into integer number of sub-signals. In the GPU-based approaches, multiple FFT calls can be organized into a batch and performed by one CUDA FFT call, which automatically utilizes the parallel resources of the GPU, as discussed in Chapter 2.

Two libraries implementing fast Fourier transform (FFT) are used in this chapter, namely the Fast Fourier Transform in the West (FFTW) for the CPU-based approach and NVIDIA CUFFT for the GPU-based approach.

Both FFT libraries support the following types of FFT: 1D, 2D, 3D, forward real-to-complex and complex-to-complex, and inverse complex-to-complex and complex-to-real transforms.

4.5.1 CPU-Based Approach

In the CPU-based multithreading approach, each parallel thread needs to extract one or more sub-signals from the digital signal, perform FFT and copy back the result, as illustrated in Figure 4.4.

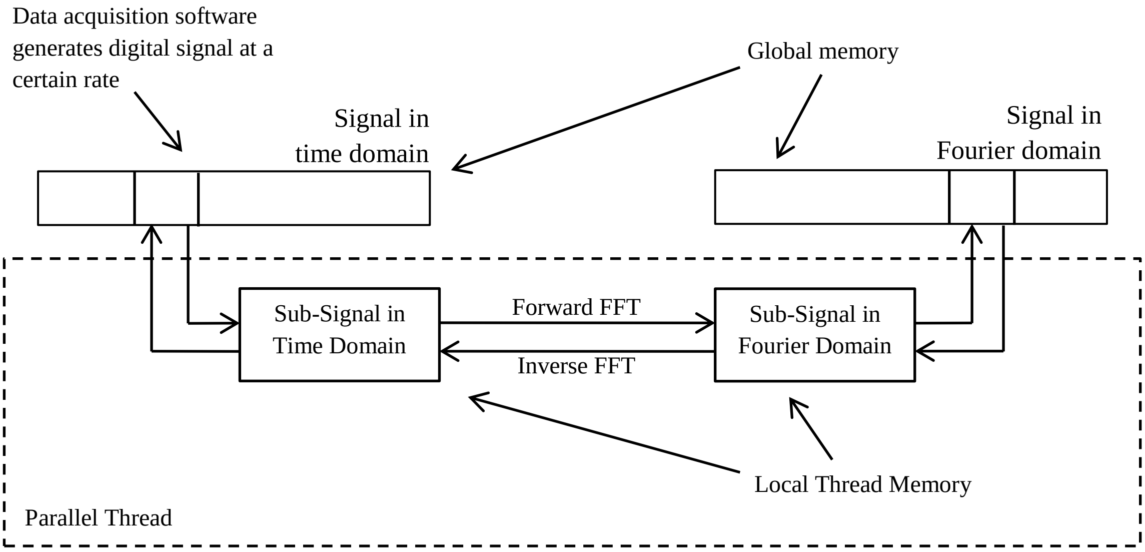

Data acquisition software generates digital signal at a certain rate
Global memory
Signal in time domain
Signal in Fourier domain
Sub-Signal in Time Domain
Forward FFT
Inverse FFT
Sub-Signal in Fourier Domain
Local Thread Memory
Parallel Thread

Figure 4.4: CPU-based parallel threads performing forward and inverse Fourier transforms

The FFTW library requires the following steps in order to complete forward and inverse FFT:

1. Create an FFT plan using the function fftw_plan_dft_r2c_1d()

2. Perform the forward or inverse FFT on the digital signal (data) by using fftw_execute_dft_r2c() or fftw_execute_dft_c2r() functions

3. Free the memory used by the FFT plan using fftw_destroy_plan() function

Algorithms 4.5 and 4.6 present the CPU-based approach.

Algorithm 4.5: CPU-based multithreading of FFT

```
Create FFT_Plan /fftw_plan_dft_r2c_1d/
Record Start Time
for t = 0 to NumberOfThreads-1 do
    Launch Parallel Thread$_t$ /CreateThread/
end for
Wait For Thread$_t$ /WaitForMultipleObjects/
Record End Time
Destroy FFT_Plan /fftw_destroy_plan/
```

In Algorithm 4.6, four signals are used: SignalTD, SignalFD, SubSignalTD and SubSignalFD. The SignalTD and SignalFD are declared in the global scope and all parallel threads can access them. SubSignalTD and SubSignalFD are declared locally in every thread. In the case of OCT data, SignalTD and SignalFD correspond to the OCT frame before and after discrete Fourier transform. In the same way, SubSignalTD and SubSignalFD correspond to the channeled spectra before and after the Fourier transform. Each parallel thread processes an equal number of sub-signals. All indices in Algorithm 4.5 start from 0.

Algorithm 4.6: Thread performing FFT on local sub-signal

```
Start Parallel Thread$_t$ ( ThreadID[1] )
    Start = ThreadID×NumberOfSubSignalsPerThread
    End = ( ThreadID + 1)×NumberOfSubSignalsPerThread
    for j = Start to End-1 do
        SubSignalTD$_i$[2] = SignalTD[3]$_{i,j}$ /memcpy/
        SubSignalFD$_i$[4] = FFT ( SubSignalTD$_i$ ) /fftw_execute/
        SignalFD$_{i,j}$[5] = SubSignalFD$_i$ /memcpy/
    end for
End Parallel Thread$_t$
```

[1] ThreadID is passed to the parallel thread from the calling (main) thread

[2] Local per-thread 1D array holding the sub-signal before DFT

[3] Global 2D array consisting of multiple sub-signals before DFT

[4] Local per-thread 1D array holding the sub-signal after DFT

[5] Global 2D array consisting of multiple sub-signals after DFT

Alternatively, the multithreaded FFT can operate directly with the global signals, as shown in Algorithm 4.7.

Algorithm 4.7: Thread performing FFT on global signal

```
Start Parallel $\text{Thread}_t$ ( ThreadID[1] )
    Start[2] = ThreadID×NumberOfSubSignals
    End[2] = ( ThreadID + 1 )×NumberOfSubSignals
    for j = Start to End-1 do
        $\text{SignalFD}_{i,j}$[3] = FFT ( $\text{SignalTD}_{i,j}$[3] ) /fftw_execute/
    end for
End Parallel $\text{Thread}_t$
```

[1] ThreadID is passed to the thread as a parameter

[2] Start and End are local per-thread variables. They define to range of sub-signals processed by each parallel thread

[3] SignalTD and SignalFD are global variables. All parallel threads have access to them

Some cases some implementations of the FFT, such as the FFTW, in some cases the input data is destroyed after the transform completes [79]. A solution in this case is performing forward and inverse FFT on a local copy of the sub-signal, as shown in Algorithm 4.6.

4.5.2 GPU-Based Approach

In the GPU approach, the FFT has the ability to perform a batch of multiple transforms over a digital signal consisting of multiple sub-signals. In this case, the digital signal is presented as two-dimensional array.

In the case of FFT employed in OCT methods, one-dimensional forward real-to-complex and one-dimensional inverse complex-to-real transforms are necessary. These transforms are applied on every channeled spectrum from the OCT frame.

The CUFFT library provided by NVIDIA takes advantage of the many-core architecture of the GPU [28]. This approach, similar to the coarse-grained implementation, uses the following functions:

1. cufftPlan1d - creates environment for the transforms, defines the type of transform
2. cufftPlanMany creates environment for multiple (batch) transforms
3. cufftExecR2C - performs the forward real to complex FFT
4. cufftExecC2R - performs the inverse complex to real FFT
5. cufftDestroy - frees the memory allocated by the FFT plan

The cufftPlanMany function can be used when multiple FFT are needed. In this case, the sub-signals do not need to be extracted from the time domain signal and processed separately. Listing 4.2 presents a CUDA C code applying forward FFT on a signal with the same size and format presented in Listing 4.1.

Listing 4.2: Batch execution of multiple 1D forward FFT on the GPU

```
int M=1024; // Size of sub-signal
int M_prime=M/2+1; // Size of sub-signal in Fourier domain
int N=65535; // Number of sub-signals and number of transforms
int DimSize[1]; // Size of sub-signal per dimension
DimSize[0]=M; // 1D FFT is employed
cudaMalloc(&SignalTDGPU,M*N*sizeof(cufftReal)); // Signal in time domain
cudaMalloc(&SignalFDGPU,M_prime*N*sizeof(cufftComplex)); // Signal in Fourier domain
cufftHandle ForwardPlan;
cufftPlanMany(&ForwardPlan,1,DimSize,NULL,1,0,NULL,1,0,CUFFT_R2C,N); // Initializing the plan
cufftExecR2C(ForwardPlan,SignalTDGPU,SignalFDGPU); // The actual transform
cufftDestroy(ForwardPlan); // Deletes the FFT plan
```

4.5.3 Performance of Forward and Inverse FFT

Forward and inverse FFT is applied on digital signal consisting of multiple sub-signals of 1024 data points each. One-dimensional forward and inverse FFT are applied per sub-signal. Tables 4.3 and 4.4 present the performance of CPU-based and GPU-based implementations of both Fourier transforms. The CPU-based implementation uses the FFTW and the GPU-based implementation employs NVIDIA CUFFT.

Table 4.3: Performance of CPU-based and GPU-based multithreading of forward FFT in milliseconds

Size of digital signal	Number of FFT	CPU-based single thread	Number of CPU-based parallel threads					GPU-based multithreading
			2	4	8	16	32	
1024	1	0.01	-	-	-	-	-	0.06
2^4×1024	16	0.06	1.4	2.6	5.2	10.1	-	0.06
2^6×1024	64	0.63	2.2	3.0	5.3	10.4	19.1	0.08
2^8×1024	256	0.9	2.7	3.5	5.5	10.9	21.2	0.11
2^{10}×1024	1024	3.1	6.7	4.7	6.2	11.8	27.1	0.14
2^{12}×1024	2^{12}	17.6	18.3	6.4[A]	7.5	12.4	29.9	0.45
2^{14}×1024	2^{14}	63.3	54.8	24.8	18.1	17.8[A]	32.5	1.52
(2^{16}-1)×1024	2^{16}	212.2	144.3	76.8	47.1	46.4[A]	48.6	5.98

[A] Optimal performance of CPU based multithreading used to evaluate the speed-up

Table 4.4: Performance of CPU-based and GPU-based multithreading of inverse FFT in milliseconds

Size of digital signal	Number of IFFT	CPU-based single thread	Number of CPU-based parallel threads					GPU-based multithreading
			2	4	8	16	32	
1024	1	0.01	-	-	-	-	-	0.06
2^4×1024	16	0.07	1.5	2.6	4.6	10.5	-	0.08
2^6×1024	64	0.28	2.3	3.1	5.0	11.2	18.8	0.09
2^8×1024	256	0.9	2.8	3.4	5.1	11.6	19.8	0.13
2^{10}×1024	1024	3.4	7.3	5.2	6.6	12.7	22.9	0.15
2^{12}×1024	2^{12}	14.4	20.1	7.8[A]	8.9	14.8	23.3	0.49
2^{14}×1024	2^{14}	60.1	55.9	26.4	18.8	18.6[A]	28.7	1.53
(2^{16}-1)×1024	2^{16}	197.8	148.8	95.8	55.8	50.3[A]	52.1	6.06

[A] Optimal performance of CPU-based multithreading used to evaluate the speed-up

The evaluation of the performance of a variable number of threads processing variable sizes of digital signals requires an agreement between the number of parallel threads and the size of the signal, as each parallel thread processes equal amount of data, Equation 4.2. As seen from Table 4.3 and 4.4, the optimal number of parallel threads in the CPU-based approach depends, apart from the multi-core architecture of the CPU, also on the size of the processed digital signal. It varies between 2 and 16. Improved performance in both FFT and IFFT of the CPU-based parallel approach is observed for digital signals with sizes starting from 2^{12}×1024. Table 4.5 presents the speed-up of the forward and inverse FFT.

$$\text{TotalSizeOfSignal} = \text{NumberOfThreads} \times \text{SizeOfSignalPerThread} \tag{4.2}$$

Table 4.5: Speed-up of the CPU-based and GPU-based forward and inverse FFT

Number of FFT/IFFT	Forward FFT		Inverse FFT	
	Speed-up (CPU-based)	Speed-up (GPU-based)	Speed-up (CPU-based)	Speed-up (GPU-based)
64	-	7.88	-	3.11
256	-	8.18	-	6.92
1024	-	22.14	-	22.67
2^{12}	2.35	39.11	1.85	29.39
2^{14}	3.56	41.64	3.23	39.28
2^{16}	4.57	35.48	3.93	32.64

As seen from Table 4.5, the CPU-based optimization improves the performance for larger number of transforms. The overheads of the CPU-based thread management, presented in Table 4.2, prevent this approach to speed-up the processing of a smaller number of Fourier transforms. The GPU-based optimization, on the other hand, improves the performance when the number of FFT reaches 64. However, the GPU processing requires the data to be copied to the GPU global memory, which brings another kind of overheads.

4.6. Cross-Correlation

The cross-correlation is a key signal processing algorithm with applications in many areas, as discussed in Chapter 3. It measures the similarity between two signals, or functions. Equation 4.3 presents the cross-correlation theorem, as discussed in [55 p. 359]. This equation is the starting point for all cross-correlation approaches presented in this chapter. It is based on forward and inverse Fourier transforms. The cross-correlation, denoted with "$\cdot$", can be applied on analog or digital signals and functions, denoted with f and g.

$$f \cdot g = \int_{-\infty}^{\infty} \bar{f}(t) g(t+L) dt = IFT\left(\overline{FT(f)} FT(g)\right) \tag{4.3}$$

The cross-correlation theorem allows a Fourier-domain computation of this signal processing algorithm. This allows FFT libraries, optimized for parallel execution such as the NVIDIA CUFFT, to contribute to the performance of parallel optimizations of the cross-correlation.

The cross-correlation presented in this chapter is implemented between a smaller in size digital signal f, acting as a template, and a larger digital signal g, as presented in Figure 4.5. In the MSI method, f corresponds to the mask signal and g to the OCT frame. The larger signal g is split into a number of sub-signals. Each sub-signal has the same size, or the same number of data points, as the template signal f. The cross-correlation evaluates the similarity between each sub-signal and the template signal. The number of performed cross-correlations is equal to the number of sub-signals in signal g, denoted with N. The size of the template signal and each sub-signal is equal to M. The overall size of signal g is M×N. In the subsequent approaches, signal f is considered as a one-dimensional array SignalF_i and signal g is considered as two-dimensional array $\text{SignalG}_{i,j}$, where i=0,..,M-1 and j=0,..,N-1.

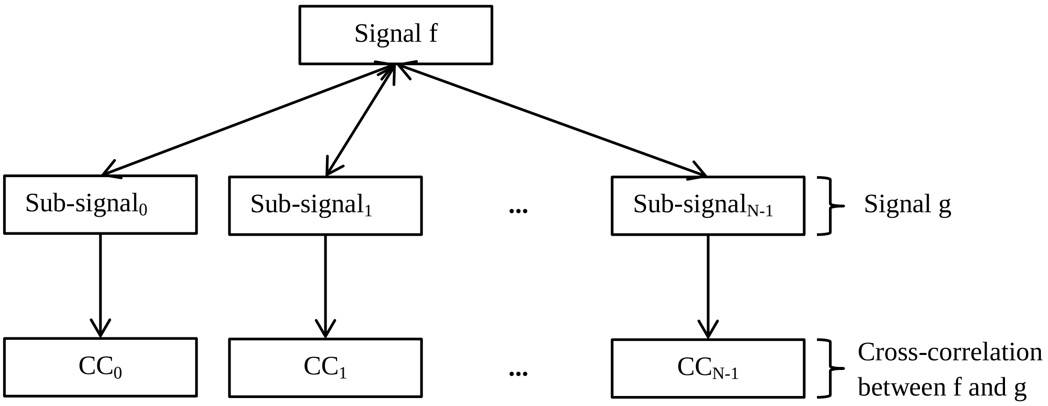


Figure 4.5: Cross-correlation between signal f and signal g

Figure 4.6 presents the cross-correlation between two sinusoid signals f and g with the same frequency and zero padding added to both of them, where:

f_n=sin(32×(2πn/N)) for n=0 to N/2-1; f_n=0 for n=N/2 to N-1

g_n=0 for n=0 to N/2-1; g_n=10×sin(32×(2πn)/N) for n=N/2 to N-1

N=1024

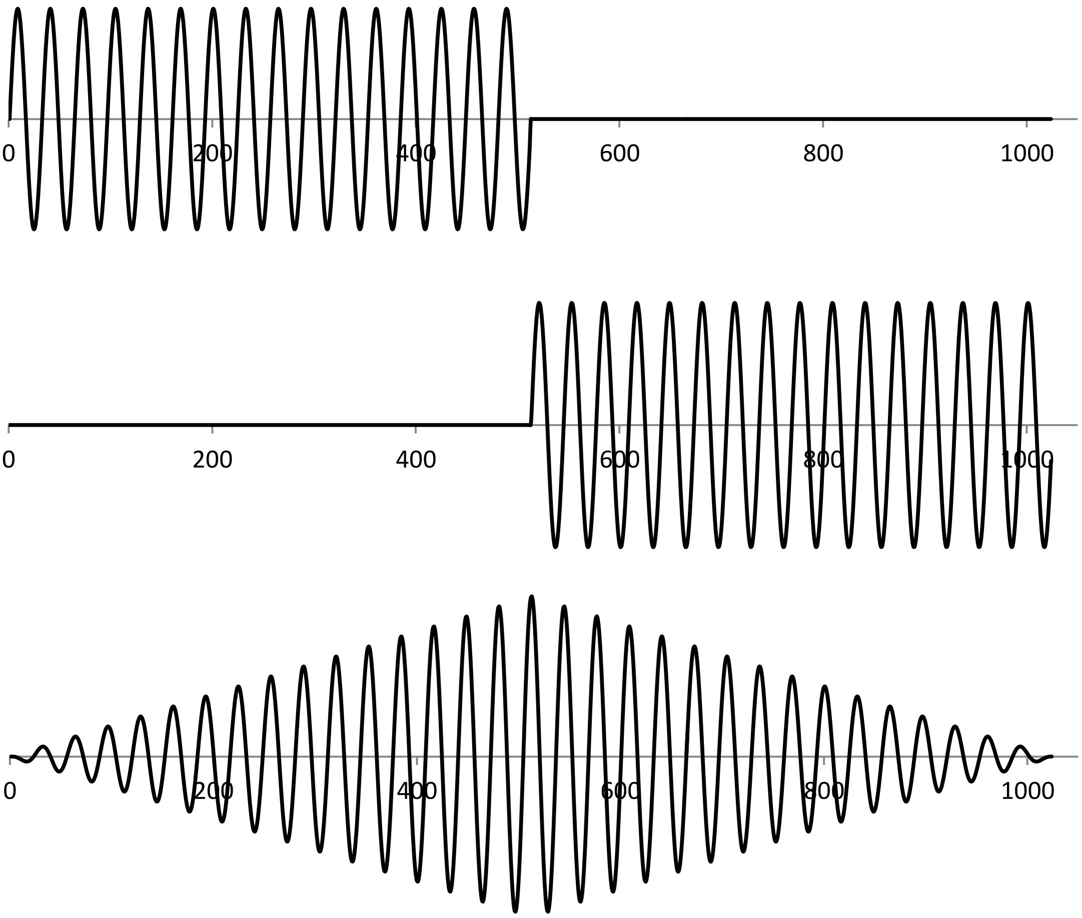


Figure 4.6: Cross-correlation between two signals (functions) f and g. A and B show signals f and g with zero padding applied on the opposite sides. C shows the cross-correlation of f and g obtained by using the cross-correlation theorem. The result is equivalent to sliding signal g shown in B from right to left and calculating the dot product with signal f shown in A at every point

Based on its definition, the cross-correlation applies the same set of computations over the data points of the digital signals. Therefore, its performance is expected to benefit significantly from parallel optimization implemented by dividing the computations among multiple parallel threads.

The implemented single-threaded, CPU-based multithreaded and GPU-based multithreaded approaches to cross-correlation are illustrated in Figure 4.7. The performance delivered by the sequential approach is considered as a baseline performance, which is compared against the subsequent parallel optimizations.

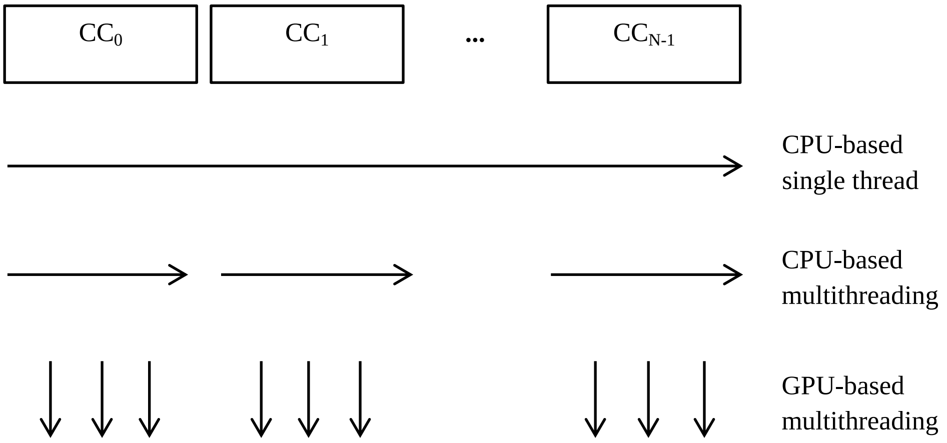


Figure 4.7: Sequential and parallel approaches to cross-correlation

4.6.1 Sequential Approach to Cross-Correlation

Algorithm 4.8 follows the steps of the cross-correlation theorem, as presented in [55] and in Equation 4.3. It processes one data point at a time. The signals SignalF', SignalG' and consequently CrossCorrelation' are in the Fourier domain. Therefore, the multiplication follows the rules of the complex multiplication. The sequential processing is organized into 'for' loops. The subsequent CPU-based parallel optimization aims at reducing the number of iterations in these loops. The GPU-based parallel approach completely eliminates the iterations from the implementation. However, due to the large number of parallel threads a partially sequential execution may be introduced by the scheduling policy of the GPU.

Signal g can be presented as one-dimensional array of sub-signals, or as a two-dimensional array of data points. In the second case, which is used in Algorithms 4.8, 4.9 and 4.10, i denotes the index of the data point within the sub-signal and j denotes the index of the sub-signal itself. The same applies for the resulting signal, the cross-correlation of f and g.

Algorithm 4.8: Sequential approach to cross-correlation

```
$M' = M/2 + 1$
$SignalF'_{i} = FFT ( SignalF_{i} )$
**for** $i = 0$ **to** $M'-1$ **do**
    $SignalF'_{i} = CC ( SignalF'_{i} )$
**end for**
**for** $j = 0$ **to** $N-1$ **do**
    $SignalG'_{i,j} = FFT ( SignalG_{i,j} )$
    **for** $i = 0$ **to** $M'-1$ **do**
        $CrossCorrelation'_{i,j} = SignalF'_{i} \times SignalG'_{i,j}$
    **end for**
    $CrossCorrelation_{i,j} = IFFT ( CrossCorrelation'_{i,j} )$
**end for**
```

4.6.2 CPU-Based Parallel Approach

The CPU-based parallel approach is based on the sequential algorithm presented in Algorithm 4.8. It launches multiple CPU-based parallel threads and thus reduces the number of iterations in the second, more computationally heavy, 'for' loop. The much smaller amount of computations in the first 'for' loop, consisting only of complex conjugate, would not benefit from parallel optimization.

The proposed CPU-based parallel optimization launches one parallel thread per number of cross-correlations. Controlling the number of the parallel threads allows a direct control over the size of the processing signal.

In Algorithm 4.9, the number of performed cross-correlations is equal to the difference End-Start, which is equal to the ratio N/NumberOfThreads. Designed in this way, the algorithm allows the higher number of parallel threads to reduce the number of cross-correlations per single thread.

Algorithm 4.9: Parallel CPU-based approach to cross-correlation

```
M' = M/2 + 1
$SignalF'_i$ = FFT ( $SignalF_i$ ) /fftw_execute_dft_r2c/
for i = 0 to M'-1 do
    $SignalF'_i$ = CC ( $SignalF'_i$ )
end for
Start Parallel $Thread_t$ (ThreadID[1])
    Start = ( N/NumberOfThreads )×ThreadID
    End = ( N/NumberOfThreads )×( ThreadID+1 )
    for j = Start to End-1 do
        $SignalG'_{i,j}$ = $FFT_j$ ( $SignalG_{i,j}$ ) /fftw_execute_dft_r2c/
        for i = 0 to M'-1 do
            $CrossCorrelation'_{i,j}$ = $SignalF'_i$ × $SignalG'_{i,j}$
        end for
        $CrossCorrelation_{i,j}$ = $IFFT_j$ ($CrossCorrelation'_{i,j}$ )
    end for
End Parallel $Thread_t$
```

[1] ThreadID is passed to the thread as a parameter

If the number of parallel threads is equal to the number of cross-correlation, the proposed Algorithm 4.9 performs one cross-correlation per parallel thread.

The size of the signal g in this approach is the same as the size of the signal employed in Algorithm 4.8.

4.6.3 GPU-Based Parallel Approach

Algorithm 4.9 presents the GPU-based approach to the cross-correlation. This implementation employs the NVIDIA CUFFT forward and inverse FFT transforms, which take advantage of the many-core architecture of the GPU.

Algorithm 4.10: Parallel GPU-based approach to cross-correlation

```
1: M' = M/2 + 1
2: SignalF'$_{i}$ = FFT ( SignalF$_{i}$ ) /cufftExecR2C, 1D transform/
3: SignalF'$_{i}$ = CC ( SignalF'$_{i}$ ) /kernel, M' GPU threads/
4: SignalG'$_{i,,j}$ = FFT ( SignalG$_{i,,j}$ ) /cufftExecR2C, batch of N 1D transforms/
5: CrossCorrelation'$_{i,,j}$ = SignalF'$_{i}$ × SignalG'$_{i,j}$ / Multiplication kernel with M'×N threads/
6: CrossCorrelation$_{i,,j}$ = IFFT ( CrossCorrelation'$_{i,,j}$ ) /cufftExecC2R, batch of N 1D transforms/
```

The multiplication kernel at line 5 in Algorithm 4.10 is presented in Listing 4.3. It follows the organization of the aforementioned implementation of the Hamming window and launches one parallel thread per each data point from signal g.

Listing 4.3: Multiplication Kernel

```
__global__ void MultiplicationKernel(cufftComplex *f,cufftComplex *g,cufftComplex *CrossCorrelation)
{
            int IndexF,IndexG;
            cufftReal fRe,fIm,gRe,gIm;
            IndexF=threadIdx.x; // Index of signal f
            IndexG=(blockDim.x)*(blockIdx.x)+threadIdx.x; // Index of signal g and cross-correlation
            fRe=f[IndexF].x;
            fIm=f[IndexF].y;
            gRe=g[IndexG].x;
            gIm=g[IndexG].y;
            CrossCorrelation[IndexG].x=fRe*gRe-fIm*gIm; // Complex multiplication, real part
            CrossCorrelation[IndexG].y=fRe*gIm+fIm*gRe; // Complex multiplication, imaginary part
}
```

4.6.4 Performance of the Cross-Correlation

The performances of the parallel approaches to the cross-correlation, along with the sequential one, are presented in Table 4.6. The size of signal f is 1024 data points in time domain and 513 data points in Fourier domain. The size of signal g varies. It is equal to the size of signal f multiplied by the number of cross-correlations. Multiple CPU-based parallel approaches are assessed, with the aim to identify the optimal number of CPU-based threads processing particular number of cross-correlations.

The MSI OCT method, discussed in Chapter 3 and implemented in Chapter 5, uses cross-correlation to generate the OCT imagery. Every point from a cross-sectional or en-face image is a result from a cross-correlation, e.g. a cross-sectional image of 500×512 points (pixels) is generated by 500×512=256000 cross-correlations. In this case, the cross-correlation is performed between two signals: OCT frame and Mask signal. The Mask signal in the MSI OCT method corresponds to the template signal in this implementation of the cross-correlation algorithm.

As a result, the performance of the cross-correlation, presented in Table 4.6, is crucial for the performance of this OCT method and its ability to operate in real-time.

Table 4.6: Performance of the cross-correlation in milliseconds

Size of digital signal g	Number of cross-correlations	CPU-based single thread	Number of CPU-based parallel threads					GPU-based multithreading
			2	4	8	16	32	
1024	1	0.04	-	-	-	-	-	0.2
$2^{4}\times 1024$	16	0.16	1.6	3.2	4.8	9.5	-	0.2
$2^{6}\times 1024$	64	0.9	2.6	3.8	5.6	10.1	19.1	0.3
$2^{8}\times 1024$	256	2.4	4.5	4.3	5.8	10.9	20.3	0.3
$2^{10}1024$	1024	8.8	13.7	6.2[A]	6.7	11.6	22.1	0.7
$2^{12}\times 1024$	2^{12}	34.4	32.7	12.9	11.5[A]	22.3	31.4	2.5
$2^{14}\times 1024$	2^{14}	130.6	99.1	63.4	29.3[A]	31.4	43.5	3.8
$2^{16}\times 1024$	2^{16}	523.5	325.9	172.2	114.3	101.2[A]	107.3	26.7

[A] Optimal parallel performance used to measure the speed-up

As seen in Table 4.6, the CPU-based multithreading improves the performance when the size of the signal reaches 2^{10}×1024 data points.

The GPU-based approach, on the other hand, improves the performance when the number of cross-correlations is 64 and larger. In this case, as seen in Table 4.6, the GPU-based cross-correlation outperforms the CPU-based one.

Table 4.7 illustrates the speed-up introduced by the parallelization of the cross-correlation algorithm.

Table 4.7: Speed-up of the CPU-based and the GPU-based cross-correlation

Number of cross-correlations	Speed-up (CPU-based)	Speed-up (GPU-based)
64	-	3
256	-	8
1024	1.42	12.57
2^{12}	2.99	13.76
2^{14}	4.46	34.37
2^{16}	5.17	19.61

4.7 Convolution

The convolution, as discussed in Chapter 3, can be calculated by using the convolution theorem discussed in [61] and illustrated by Equation 4.4.

$$y = f * g = IFFT\big(FFT(f)FFT(g)\big) \tag{4.4}$$

Expressed in this way, the only difference between convolution and cross-correlation is the lack of complex conjugate in the convolution case, as seen in Equations 4.3 and 4.4. This similarity is reflected in the implementations of both algorithms as sequential and parallel computer programs. Due to the low latency of the complex conjugate, which only operation is changing the sign of the imaginary part of the complex number, the performances of the cross-correlation and the convolution are virtually indistinguishable.

4.8 Integration

In this thesis, integration denotes the repeated summation of terms. This operation is used in numerous cases in different areas. For example, if these terms are data points from a digital signal, the integration is applied to calculate the energy of the signal. Equations 4.5 and 4.6 illustrate the calculation of the energy of an analog and discrete (digital) signal, [55], [58]. If the terms are rectangular areas approximating area defined by a curve, the integration approximates this area. This case is used to calculate definite integral, illustrated in Figure 4.8 and Equations 4.7 and 4.8. This process is denoted as numerical integration.

$$Energy_f = \int_{-\infty}^{\infty} |f(t)|^2 dt \tag{4.5}$$

$$F_i = |f_i|^2; Energy_f = \sum_{i=-\infty}^{\infty} F_i \tag{4.6}$$

$$I = \int_{a}^{b} f(x)dx \tag{4.7}$$

$$I \sim \sum_{i=0}^{M-1} f(x_i)\Delta x \tag{4.8}$$

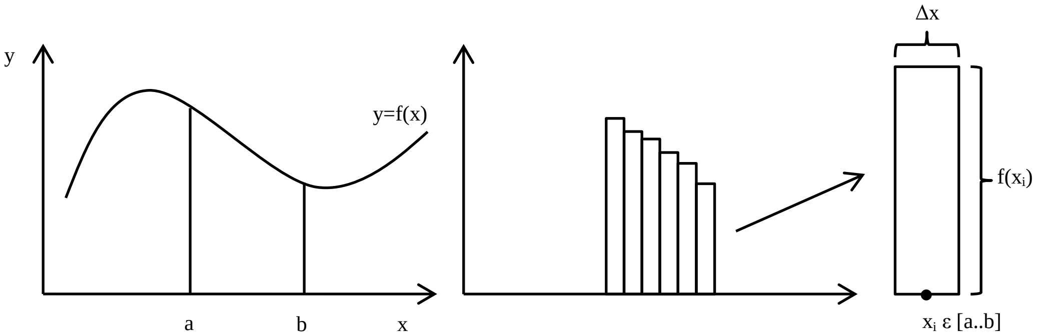


Figure 4.8: Approximating definite integral. Geometrically, the definite integral of a function of a single variable is equal to the area between the function and the horizontal axis along the segment [a..b]. A smaller Δx results into a larger number of summations and better approximation. Alternatively, larger Δx leads to smaller number of summations and less accurate approximation. This allows a trade-off between accuracy and performance

In most cases in signal processing, the integration is performed repeatedly within sub-signals, which are part of a larger signal **A**, as presented in Equation 4.9. The result of the overall process is a sequence of values Int_j representing the integration of each sub-signal. In

Equation 4.9 i is the index of a data point within a sub-signal and j is the index of the sub-signal. The total number of data points is equal to M×N, where M is the number of data points per sub-signal and N is the total number of sub-signals.

$$Int_j = A_{0,j} + A_{1,j} + \cdots + A_{M-1,j} = \sum_{i=0}^{M-1} A_{i,j}\,; j = 0, \cdots, N-1 \tag{4.9}$$

The integration can include all data points from a sub-signal, as in Equation 4.5, or it can be performed according to a window. Equations 4.6 and 4.7 demonstrate these two cases, where W_1 and W_2 define the borders of the window, where $0 \leq W_1 < W_2 \leq (M-1)$. Figure 4.9 illustrates the two ways to position the window. The need for a window and consequently the values of W_1 and W_2 depend on the particular case. If integration takes part in the image generation, as it is in MSI method implemented in Chapter 5, the need for a window and the values of the window borders can be determined by observing the images, generated from the signal and looking for lowest level of noise, for example in the form of speckles in the images.

$$Int_j \sum_{i=W_1}^{W_2} A_{i,j} \tag{4.10}$$

$$Int_j = \sum_{i=0}^{W_1} A_{i,j} + \sum_{i=W_2}^{M-1} A_{i,j} \tag{4.11}$$

In this thesis two cases require integration. The first one is the MSI OCT method after the cross-correlation, as discussed in Chapter 3 and implemented in Chapter 5. The second one is the integration stage during confocal imaging implemented in Chapter 5. In both cases the integration is within a window defined by the values W_1 and W_2. An integrated solution, presented in Chapter 6, allows the selection of W_1 and W_2 through the user interface of the OCT system in real-time.

Four approaches to integration are implemented and compared, namely Sequential Iterative, Partially Parallel Iterative, Parallel Reduction, and Zero-Frequency Component (ZFC).

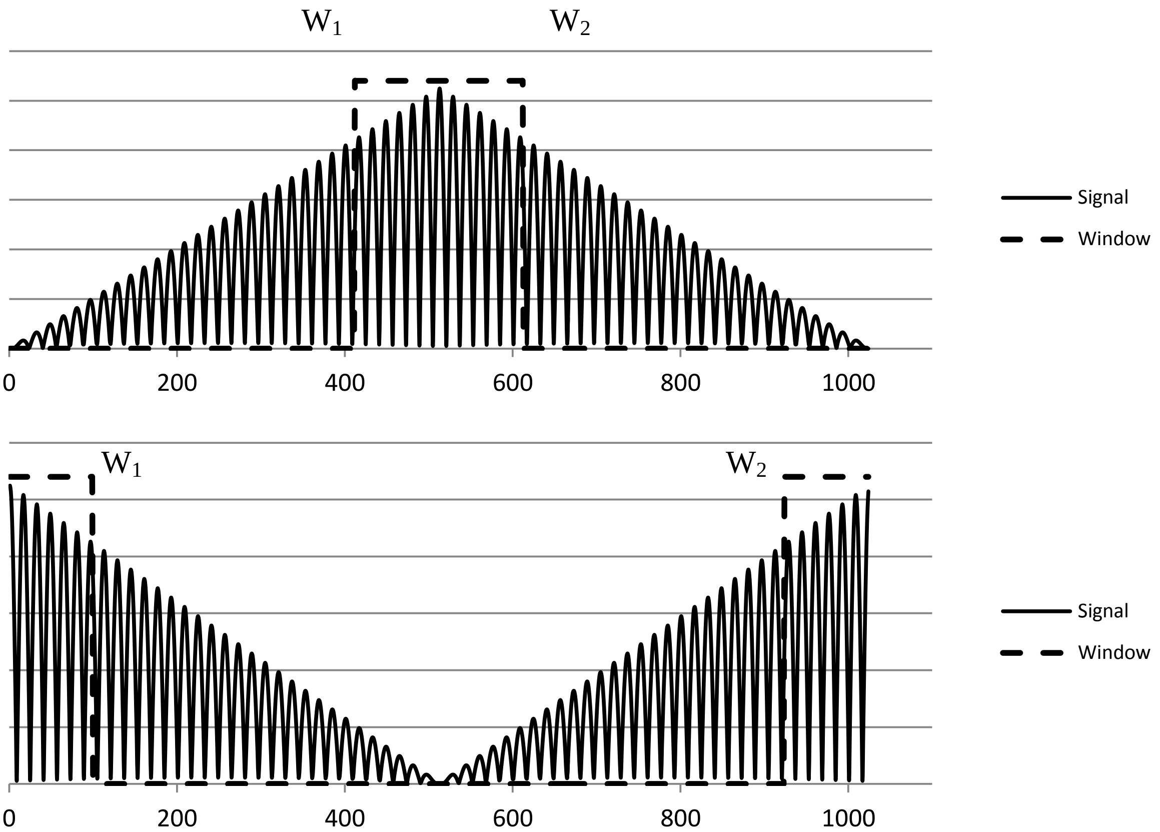


Figure 4.9: Two approaches to define an integration window

4.8.1 Sequential Iterative

This single-threaded approach, presented in Algorithm 4.11, can be implemented both on the CPU and the GPU. Its performance is used as a base-line performance in the comparisons with the rest of the approaches.

Algorithm 4.11: Sequential iterative integration (CPU and GPU)

```
for j = 0 to N-1 do
    Summation = 0
    for i = $W_1$ to $W_2$ do
        Summation = Summation + $A_{i,j}$
    end for
    $Int_j$ = Summation
end for
```

4.8.2 Partially Parallel Iterative

This approach, shown in Algorithm 4.12, partially parallelizes the aforementioned sequential approach. The approach launches one parallel thread per sub-signal. The maximum number of parallel threads within one block is 1024, a number much smaller than the number of sub-signals, N. Therefore, each parallel thread is launched from a different block. Each parallel thread calculates the corresponding integration value. The high number of sub-signals, up to N=2^{16}-1, makes this approach suitable for the GPU-based multithreading.

The following kernel call launches N blocks with 1 thread per block or N×1 parallel threads:

```
IntegrationKernel <<< N , 1 >>> ( A , Int ) ;
```

Algorithm 4.12: Partially parallel iterative integration (GPU)

```
Kernel PartiallyParallelIterative
    BI[1] = blockIdx.x
    Offset[2] = M[3] × BI
    Summation = 0
    for i[4] = $W_1$ to $W_2$ do
        Index = Offset + i
        Summation = Summation + $A_{Index}$
    end for
    $Int_{BI}$ = Summation
End Kernel
```

[1] Index of the current block

[2] The index of the first data point of each sub-signal

[3] Size of the sub-signal

[4] Index of a data point within a sub-signal

4.8.3 Parallel Reduction

This approach is based on the reduction algorithm, which GPU-based implementation is illustrated in [80]. Its suitability for GPU and FPGA is further discussed in [81]. It expects the size of the data to be a power of two. If this is not the case, the digital signal can be zeropadded to the next power of two. Figure 4.10 illustrates this approach with a digital signal with 8 data points.

In the Parallel Reduction algorithm, each summation is performed by a single parallel thread, T_1 to T_4 in Figure 4.10. This would require the number of parallel threads to be equal to half the size of the digital signal. This amounts to a significant number of parallel threads, making this approach suitable only for a GPU implementation.

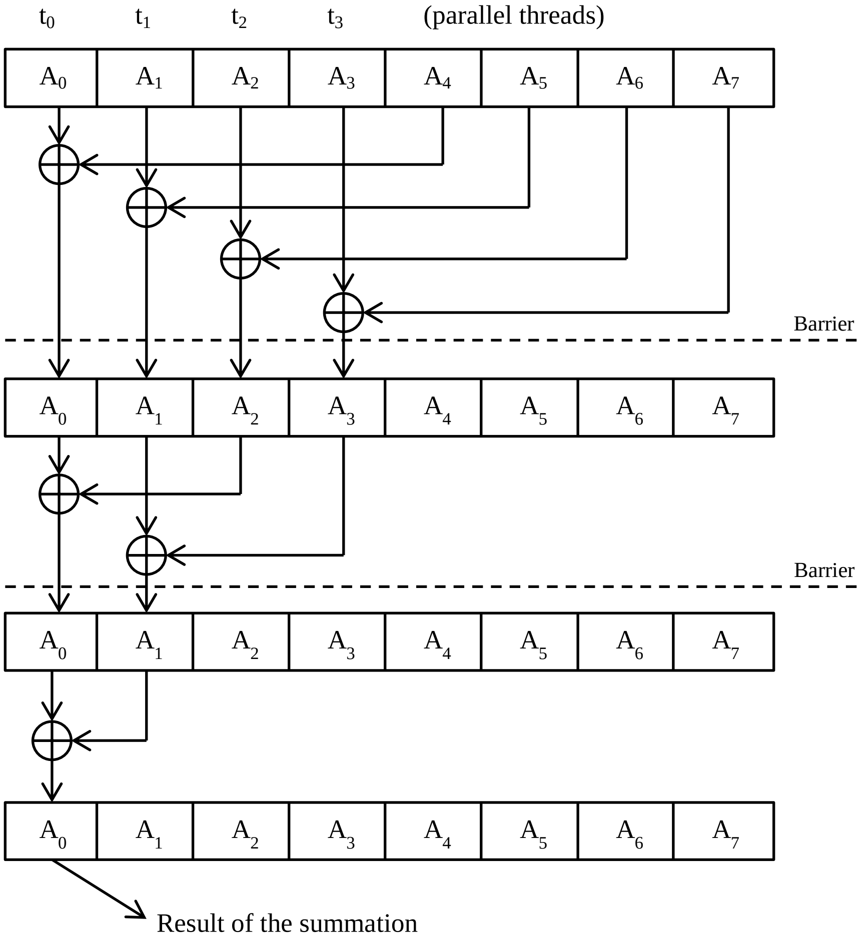


Figure 4.10: Parallel reduction

Algorithm 4.13 implements a GPU-based approach to the parallel reduction as illustrated in Figure 4.10. This algorithm maps each sub-signal to a thread block.

Algorithm 4.13: Integration by Parallel Reduction (GPU)

```
ParallelReduction Kernel
    TI = threadIdx.x
    BI = blockIdx.x
    Offset[1] = $M \times BI$
    SharedSum[2] $_{TI} = A_{Offset +TI} + A_{Offset+TI+M/2}$
    Barrier[3]
    if (TI<2 ) then
        LocalSum[4] = $SharedSum_{TI} + SharedSum_{TI+2}$
        $SharedSum_{TI}$ = LocalSum
    end if
    Barrier
    if ( TI = 0 ) then
        LocalSum = $SharedSum_0 + SharedSum_1$
        $Int_{BI}$ = LocalSum
    end if
End Kernel
```

[1] Offset is the index of the first data point of each sub-signal

[2] SharedSum is an array in the shared per-block memory

[3] Barrier implemented using __synchthreads ( )

[4] LocalSum is local per-thread variable used to store temporary the summation at each step

The barriers assure the completion of all previous summations before commencing the next ones. In NVIDIA CUDA C a block-wide barrier, e.g. involving all threads within a single block, is done by the function __syncthreads( ).

The proposed algorithm maps the digital signal consisting of sub-signals on the GPU grid in the following way:

1. Each sub-signal is processed by one thread block
2. Each data point within the sub-signal is processed by one GPU thread

The following kernel call illustrates the mapping:

```
ParallelReductionKernel
                    <<< NumberOfSubSignals, SubSignalHalfSize, SharedMemory >>>
                                        ( A, Int )
```

A signal with larger number of data points would change the algorithm by adding 'if' statements and adjusting the indices accordingly.

4.8.4 Zero-Frequency Component

The Zero-Frequency Component approach is based on the properties of the discrete Fourier transform, presented by Equation 4.12, where k is the frequency component.

$$F_k = DFT(f_n) = \sum_{n=0}^{N-1} f_n e^{-2i\pi kn/N} \tag{4.12}$$

Equation 4.13 shows the result after substituting k with 0. It represents the zero-frequency component of the digital signal in the Fourier domain.

$$F_0 = DFT(f_n) = \sum_{n=0}^{N-1} f_n \tag{4.13}$$

As presented in Equation 4.13, the zero component of a Fourier transform of a digital signal is the summation of all data points from that signal. Therefore, integration per sub-signal can be achieved by applying forward discrete Fourier transform on each sub-signal. In this case, the performance of the zero-frequency component is identical with the performance of the forward FFT presented in Table 4.3.

4.8.5 Performances of the Integration Approaches

Table 4.8 presents the performance of six approaches to integration. The integration is applied on a digital signal with the same format and size as used in the previous algorithms.

In general, the GPU based approaches offer better performance, compared to the CPU-based ones. The sequential approach is the only exception. In this case, both the CPU-based and the GPU-based computations are carried out by a single thread of execution.

Table 4.8: Performance of the integration approaches in milliseconds

Size of digital signal[A] (N×M)	Sequential Iterative		Sequential Partially Parallel (GPU)	Parallel Reduction (GPU)	ZFC[B]	
	CPU	GPU			CPU	GPU
1024	0.01	0.13	0.34	0.024	0.01	0.06
2^4×1024	0.02	5.56	0.64	0.026	0.06	0.06
2^6×1024	0.06	22.06	0.95	0.027	0.63	0.08
2^8×1024	0.21	87.94	1.03	0.029	0.91	0.11
2^{10}×1024	0.89	357.31	2.02	0.032	3.13	0.14
2^{12}×1024	3.41	1427.76	2.96	0.038	6.47	0.45
2^{14}×1024	14.55	5723.94	5.93	0.041	17.81	1.52
2^{16}×1024	55.06	22581.41	17.35	0.044	46.43	5.98

[A] In this table all data points from the digital signal are included in the integration, or W_1=0, W_2=M-1

[B] CPU-based and GPU-based ZFC performances derive from the corresponding optimal performances of the forward FFT taken from Table 4.3

Table 4.9 presents the speed-up of the aforementioned parallel approaches to integration.

Table 4.9: Speed-up of the parallel approaches to integration

Number of integrations	Sequential Iterative(GPU)	Partially Parallel Iterative(GPU)	Parallel Reduction (GPU)	ZFC (CPU)	ZFC (GPU)
1	0.077	0.029	0.417	1.000	0.167
16	0.004	0.031	0.769	0.333	0.333
64	0.003	0.063	2.222	0.095	0.750
256	0.002	0.204	7.241	0.231	1.909
1024	0.002	0.441	27.813	0.284	6.357
2^{12}	0.002	1.152	89.737	0.527	7.578
2^{14}	0.003	2.454	354.878	0.817	9.572
2^{16}	0.002	3.173	1251.364	1.186	9.207

The next chapter employs the integration on two occasions:

- Integration stage of the MSI OCT method. The CPU-based implementation of this method uses the Sequential Iterative approach, which offers the best CPU-based performance. Accordingly, the GPU-based implementation of the MSI OCT method uses the Parallel Reduction approach, which delivers the best GPU-based performance.

- Confocal OCT imaging. This type of OCT imaging is based on multiple en-face images. As seen in Chapter 5, en-face images generated on the GPU significantly outperform those generated on the CPU. Therefore, the proposed confocal imaging is based on the GPU. All three GPU-based approaches to integration are employed in the GPU-based implementation of the confocal OCT imaging.

4.9 Summary

This chapter presented DPS algorithms widely used in many areas including real-time systems, as discussed in Chapter 3.

A number of sequential approaches are presented, based on FFT library functions and signal processing theorems. Based on these sequential approaches, CPU-based and GPU-based parallel optimizations are developed and presented. Their aim is to deliver optimal performance of the employed DSP algorithms.

A comparison between the multi-core CPU-based and the many-core GPU-based parallel optimizations in terms of level of parallelization and performance are drawn.

In the GPU-base approaches, a one-to-one mapping of the data point onto the parallel threads allows each thread of execution to process one data point. This high level of parallelism yields better results when processing larger signals.

The CPU-based approach on the other hand, operates with smaller number of parallel threads. In this case, a decision must be made how many parallel threads to employ. The proposed CPU-based approaches identify the number of parallel threads necessary for optimal performance.

Chapter 5

Parallel Optimization in Optical Coherence Tomography Systems

5.1 Introduction

This chapter presents a number of sequential implementations and their corresponding parallel optimizations of two OCT methods, the CFD and the MSI. Both methods are capable of producing cross-sectional and en-face images. In addition, confocal images based on multiple en-face images are presented.

A series of single-threaded and multi-threaded algorithms are designed, implemented and studied. Both coarse-grained and fine-grained approaches to these methods are proposed.

In this chapter, a number of strategies aiming at improving the performance, such as dividing the OCT methods into stages and phases, are applied. The approaches, which provide optimal performance of the signal processing, are identified.

In the field of OCT, there are constant demands for improvements in two general directions: performance and imagery. Although different in their nature, these two directions can be both addressed by parallel optimizations.

In OCT, the most computationally intensive part is the DSP. In real-time operation, an OCT system generates certain amount of digital signals at a rate defined by its sweeping capabilities. The overall performance of the OCT system depends on the performance of the signal processing. A sequential implementation of the DSP is limited. This is especially the case with the more complex OCT methods, such as the MSI, a dual interferometry method which employs two digital signals denoted in this thesis as OCT frame and Mask signal. The parallel optimizations, presented in this chapter, are shown in Table 5.1.

Table 5.1: Sequential and parallel implementations

	Cross-Sectional		En-Face		Confocal	
	CFD	MSI	CFD	MSI	CFD	MSI
CPU-Based Single Thread	R[1]	N[2]	R	R/N[4]	-[5]	-
CPU- Based Multithreading	R	N	R	R/N[4]	-	-
GPU-Based Multithreading	R	R/N[3]	R	R	R	R/N[4]

[1] Real-time

[2] Not real-time

[3] Real-time operation depends on the size of the cross-sectional image

[4] Real-time operation depends on the number of en-face images

[5] Not implemented

In real-time operation, the OCT system is expected to process the digital signals and visualize the OCT images while imaging the object. Figure 6.6 illustrates the real-time criterion. It is defined by the speed of acquisition of one OCT frame. This speed depends on the sweeping capabilities of the OCT system.

5.2 Coarse-Grained and Fine-Grained Parallel Approaches

This chapter introduces the well-known coarse-grained and fine-grained parallelism, as discussed in [16], to the digital signal processing in the OCT systems. An approach utilizing a smaller number of parallel threads, each one processing larger amount of data, is considered a coarse-grained. The coarse-grained approaches allow varying amount of threads with the aim to identify the optimal number of employed parallel threads. On the other hand, an approach using a larger number of parallel threads each one processing smaller amount of data, e.g. one data point, is considered fine-grained.

The time needed to launch parallel threads on the CPU and on the GPU and the high number of parallel threads within a GPU grid determine the CPU as the architecture of choice for the coarse-grained approach and the GPU for the fine-grained approach.

The coarse-grained and fine-grained approaches are based on the way OCT systems group the digital signals into sub-signals. Those systems generate digital signals organized into OCT frames (OCTF). Each OCTF consists of multiple channeled spectra (CS) and each channeled spectrum consists of multiple data points (DP). The CPU-based coarse-grained approaches process one or more CS per single parallel thread. On the other hand, the GPU-based fine-grained approaches process one DP per thread, as shown in Table 5.2. An exception is the integration step of the processing, employed in the MSI OCT method and in the confocal imaging, where one parallel thread processes more than one data point.

Table 5.2: Digital signal processed by one parallel thread

Approach	Data Point	One Channeled Spectrum	Multiple Channeled Spectra	One OCT Frame
Single Thread (CPU)	×	×	×	√
Coarse-Grained (CPU)	×	√	√	×
Fine-Grained (GPU)	√	√	×	×

The GPU-based fine-grained approaches introduce another type of overheads (latency), which is the time to copy the data to and from the global GPU memory. This latency affects the requirements for the GPU-based performances. These requirements are dictated by the real-time criterion of the OCT system. The data transfer between the CPU and the GPU memory is done by the following calls to the cudaMemcpy function:

```
cudaMemcpy (GPU_Var,CPU_Var,SizeOfVar*sizeof(DataType),cudaMemcpyHostToDevice)
cudaMemcpy (CPU_Var,GPU_Var,SizeOfVar*sizeof(DataType),cudaMemcpyDeviceToHost)
```

The last parameter of the function indicates the direction of the copy, from the CPU (host) to the GPU (device) and vice-versa.

Table 5.3 presents the time necessary to transfer the data corresponding to the cross-sectional and en-face OCT frames employed in this chapter. The performance of the transfer depends mainly on the access speed of the main memory, the transfer rate of the PCI Express bus, and the access speed of the global GPU memory. A linear relation can be observed between the time to transfer the data and the size of the data. This is the main and most significant overhead introduced by the parallel optimization on the GPU. A GPU application can access two types of CPU-based memory: standard and pinned. Pinned memory is defined as not pageable and therefore has faster access.

Table 5.3: CPU (host) to GPU (device) data transfer time in milliseconds (cudaMemcpy)

OCT frames (data points)	Size of OCT frame	CPU-GPU transfer time in milliseconds	
		Standard	Pinned (CUDA)
100×1024	400 KB	0.29	0.22
200×1024	800 KB	0.52	0.36
300×1024	1200 KB	0.75	0.51
400×1024	1600 KB	1.01	0.66
500×1024	~2 MB	1.33	0.82
200×192×1024	150 MB	101.52	58.31

The processing of these OCT frames results in cross-sectional and en-face images. These images are presented as one-byte gray scale values presenting intensities to be displayed by OpenGL or saved as image files. Therefore, the latencies of the GPU to CPU copy, presented in Table 5.4, are much smaller than the corresponding CPU to GPU ones. The processing of one en-face OCT frame can result in multiple en-face images, which is reflected in Tables 5.4.

Table 5.4: GPU (device) to CPU (host) data transfer time in milliseconds (cudaMemcpy)

Type of OCT images	Size (data points/bytes)	Standard	Pinned
Cross-Sectional	100×512	0.11	0.08
	200×512	0.15	0.11
	300×512	0.21	0.14
	400×512	0.27	0.16
	500×512	0.29	0.19
En-face	1×S[A]	0.09	0.07
	8×S	0.41	0.25
	16×S	0.75	0.47
	20×S	0.91	0.51
	32×S	1.27	0.86
	40×S	1.41	0.92

[A] S denotes the size of one en-face image, which is 200×192 gray scale values or 38400 bytes

The much smaller latency of the GPU to CPU transfer in en-face imaging, compared to the CPU to GPU one, is due to the much smaller size of the resulting signal and the smaller size of the data type, 8-bit gray scale, compared to the 32-bit float.

The reduction of the size of the resulting signal in en-face imaging is due to the selection of a limited number of en-face images in the CFD method, up to 40 from 513 possible. In the MSI method, this reduction is due to the integration step.

5.3 Cross-Sectional Imaging in OCT

5.3.1 Structure of the OCT Signal in Cross-Sectional Imaging

In cross-sectional imaging, the OCT systems generate data points grouped into channeled spectra, which are further grouped into cross-sectional OCT frames. Each channeled spectrum (CS) can be presented as a one-dimensional array of data points (CS_DP_i) and each cross-sectional OCT frame can be presented as a one-dimensional array of channeled spectra ($OCTF\_CS_j$) or two-dimensional array of data points ($OCTF\_DP_{i,j}$).

For practical reasons, the digital signal processed in this chapter is converted by the data acquisition software into 8-byte double or 4-byte cufftReal types. These data types are used by the two FFT libraries employed in this chapter, namely FFTW and NVIDIA CUFFT respectively. Thus, no further data conversion is necessary.

During real-time operation, the OCT system studied in this research at its current settings generates one OCT frame of up to 500 channeled spectra with 1024 data points in each channeled spectrum every 10 milliseconds [11]. These values are determined by the sweeping capabilities of the OCT system. They define the real-time criterion of the system.

Two OCT methods are developed and presented in this chapter, namely CFD and MSI. Both methods utilize discrete Fourier transforms. These transforms are performed by applying multiple one-dimensional non-redundant FFT on each channeled spectrum.

Equation 5.1 defines single channeled spectrum before FFT with M data points, as generated by the OCT system. Equation 5.2 describes a single FFT applied to the CS. Equation 5.3 defines a single channeled spectrum after FFT (CS') consisting of M' data points (CS_DP'). Equation 5.4 shown the relation between the size of the channeled spectrum, or the number of data points in each channeled spectrum, before FFT (M) and after FFT (M'). This equation considers non-redundant Fast Fourier Transform. Equation 5.5 defines an OCT frame before applying FFT (OCTF) with N channeled spectra (OCTF_CS). Equation 5.6 defines the same OCT frame consisting or M×N data points (OCTF_DP). Equation 5.7 shows the OCT frame after N FFT are applied on N channeled spectra, or M×N data points. Equation 5.8 presents the digital signal before FFT, consisting of one OCT frame (OCTF'), or N channeled spectra (OCTF_CS'), or M'×N data points (OCTF_DP'). The algorithms presented in this chapter follow these notations.

$$CS = CS\_DP_i; CS\_DP_i \epsilon \boldsymbol{R}; i = 0, \cdots, M - 1 \tag{5.1}$$

$$CS' = FFT(CS) \tag{5.2}$$

$$CS' = CS\_DP'_i; CS\_DP'_i \epsilon \boldsymbol{C}; i = 0, \cdots, M' - 1 \tag{5.3}$$

$$M' = M/2 + 1 \tag{5.4}$$

$$OCTF = OCTF\_CS_j = (OCTF\_CS_0, \cdots, OCTF\_CS_{N-1}) \tag{5.5}$$

$$OCTF = OCTF\_DP_{i,j} = \begin{pmatrix} OCTF\_DP_{0,0} & \cdots & OCTF\_DP_{0,N-1} \\ \vdots & \ddots & \vdots \\ OCTF\_DP_{M-1,0} & \cdots & OCTF\_DP_{M-1,N-1} \end{pmatrix} \tag{5.6}$$

$$OCTF' = FFT_j\big(OCTF\_CS_j\big) = FFT_j\big(OCTF\_DP_{i,j}\big); i = 0, \cdots, M - 1; j = 0, \cdots N - 1 \tag{5.7}$$

$$OCTF' = OCTF\_CS'_j = OCTF\_DP'_{i,j} = \begin{pmatrix} OCTF\_DP'_{0,0} & \cdots & OCTF\_DP'_{0,N-1} \\ \vdots & \ddots & \vdots \\ OCTF\_DP'_{M'-1,0} & \cdots & OCTF\_DP'_{M'-1,N-1} \end{pmatrix} \tag{5.8}$$

Figure 5.1 illustrates a single channeled spectrum generated by the OCT system before and after Fast Fourier transform, with 1024 data points and 513 data points respectively.

5.3.2 Resolving the Depth in Cross-Sectional OCT Imaging

The CFD and MSI methods resolve the depth in the cross-sectional OCT images differently:

1. In the CFD OCT method, the intensity of a point from a particular depth is resolved by calculating the corresponding absolute value from the channeled spectrum after performing DFT. The number of different depths depends on the size of the channeled spectrum. A channeled spectrum with M data points allows the OCT system to resolve M' = M/2+1 different depths. With M equal to 1024, the OCT system is able to resolve reflectivity from M'=513 depths. Figure 5.1 illustrates a single channeled spectrum obtained during imaging a mirror. In this case, the peak of the intensity corresponds with the position of the surface of the mirror in relation with the objective lens of the OCT system, as shown in Figure 5.1.B.

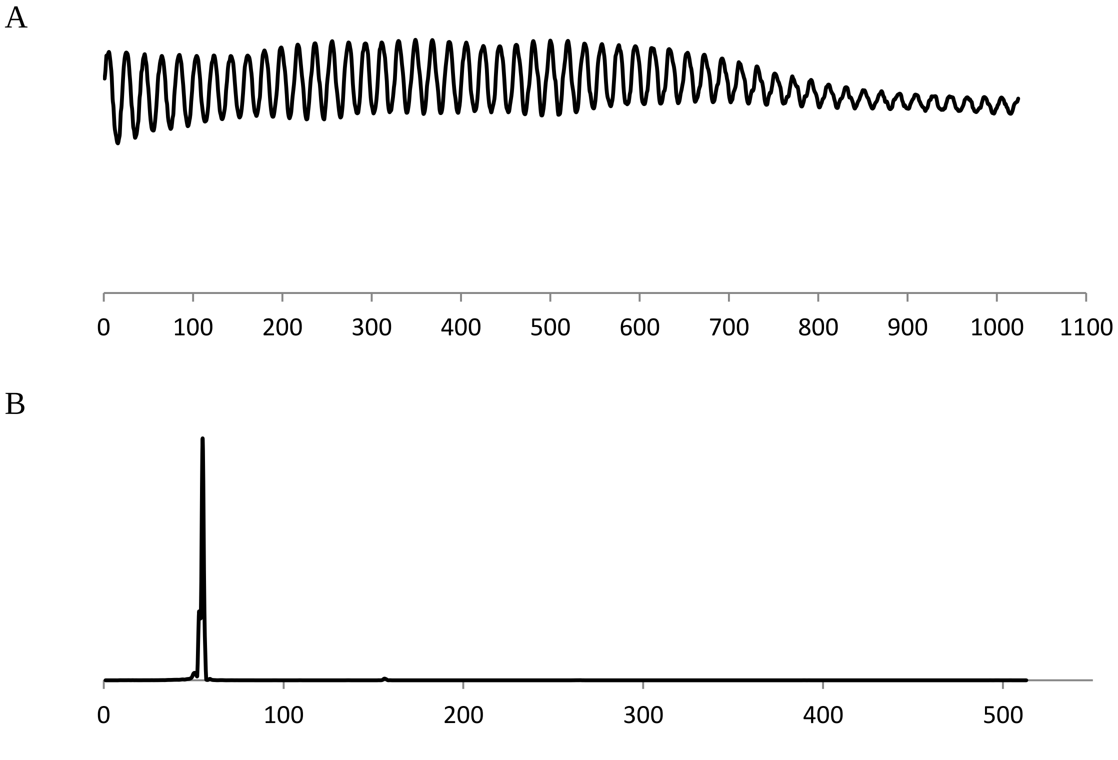


Figure 5.1: Channeled spectrum before discrete Fourier transform (A) and after discrete Fourier transform (B). Single peak in (B) indicates opaque object. Its position indicates the position of the surface of the object according to the objective lens. A semitransparent object has reflectivity from multiple depths, as seen in Figure 3.10

2. In the MSI OCT method, the intensity of each depth is obtained by cross-correlation between two digital signals, namely the channeled spectra from the OCT frame and a mask signal. The research presented in this thesis has access to a set of R=512 mask signals, acquired experimentally. Each mask signal corresponds to a particular depth. The channeled spectra and the mask signals need to be of equal size. This research operates with channeled spectra and masks with M=1024 data points.

The mask signals employed in the MSI OCT method do not change during the operation of the OCT system. Therefore, processing of the mask signals during initialization, before the OCT system is in online mode, improves the overall performance. In the coarse-grained approaches the masks are processed sequentially, since parallel optimization would not affect the performance. This processing consists of applying Hamming window (HW), FFT, and complex conjugate (CC) on the mask signal (MS_DP), as presented in Algorithm 5.1. This approach is employed in both CPU-based implementations of the MSI OCT method, the sequential and the coarse-grained.

Algorithm 5.1: Processing mask signals in sequential and coarse-grained MSI approaches

```
for r = 0 to R-1 do
    for i=0 to M-1 do
        $MS\_DP_{i,r} = HW ( MS\_DP_{i,r} )$
    end for
    $MS\_DP'_{i,r} = FFT_{r} ( MS\_DP_{i,r} )$
    for i=0 to M'-1 do
        $MS\_DP'_{i,r} = CC ( MS\_DP'_{i,r} )$
    end for
end for
```

The corresponding fine-gained approaches employ GPU-implemented FFT. Therefore, the mask signals, along with the OCT frames generated by the OCT system during online mode, are processed on the GPU. Each GPU-based parallel thread processes one data point from the mask signals. Each thread block from the GPU grid processes one mask from the set of mask signals, as illustrated in Algorithm 5.2.

Algorithm 5.2: Processing mask signals in fine-grained MSI approaches

```
1: Copy $MS\_DP_{i,r}$ from CPU memory to GPU memory /cudaMemcpy/
2: $MS\_DP_{i,r}$ = HW ( $MS\_DP_{i,r}$ ) /kernel with M×R threads/
3: $MS\_DP'_{i,r}$ = $FFT_r$ ( $MS\_DP_{i,r}$ ) /cufftExecR2C, batch of R 1D transforms/
4: $MS\_DP'_{i,r}$ = CC ( $MS\_DP'_{i,r}$ ) /kernel with M'×R threads/
```

All implementations of the MSI OCT method presented in this chapter require the completion of the processing of the mask signals. The two-dimensional array $MS\_DP'_{i,r}$ holds the processed mask signals in the CPU memory for the sequential and coarse-grained approaches and in the GPU memory for the fine-grained approaches.

5.3.3 CFD OCT Method in Cross-Sectional Imaging

In the CFD OCT method, cross-sectional images are obtained by the following steps:

1. Apply Hamming window on the data points from the OCT frame (OCTF_DP)
2. Apply one-dimensional discrete Fourier transform on all channeled spectra from the OCT frame
3. Calculate the absolute values of the data points
4. Scale to the gray scale interval [0..255] by multiplying with a gray scale coefficient (GSC)

Equation 5.9 follows these steps, where i is the index of the data point within the channeled spectrum and j is the index of each channeled spectrum within the cross-sectional OCT frame. $CrossSectional_{i,j}$ represents the gray scale values of the points from the cross-sectional image, where i is the vertical coordinate and j is the horizontal coordinate in the cross-sectional OCT image. The vertical coordinate of the point corresponds to the depth of the cross-sectional OCT image. Algorithm 5.3 implements the steps from Equation 5.9.

$$CrossSectional_{i,j} = GSC \times \left| DFT_j \left(HW\left(OCTF\_DP_{i,j} \right) \right) \right| \tag{5.9}$$

In all approaches, the result of the calculations is a two-dimensional array $\text{CrossSectional}_{i,j}$. This array contains the gray scale values of the resulting cross-sectional image.

In all cross-sectional algorithms M is the size of the channeled spectra before discrete Fourier transform, M' is the size of the channeled spectrum after Fourier transform, and N is the number of channeled spectra in the OCT frame.

Algorithm 5.3 follows a sequential approach to CFD OCT method in cross-sectional imaging. It employs three ‘for’ loops. The two inner 'for' loops process one data point per iteration. The corresponding coarse-grained approach aims at distributing these iterations among multiple parallel threads.

Algorithm 5.3: Sequential approach to the CFD OCT method in cross-sectional imaging

```
for j=0 to N-1 do
    for i=0 to M-1 do
        $\text{OCTF\_DP}_{i,,j} = \text{HW}(\text{OCTF\_DP}_{i,,j})$
    end for
    $\text{OCTF\_DP'}_{i,,j} = \text{FFT}(\text{OCTF\_DP}_{i,,j})$
    for i=Freq[1] to M' -1 do
        $\text{Intensity}_{i,,j} = \text{AV}(\text{OCTF\_DP'}_{i,,j})$
        $\text{CrossSectional}_{i,,j} = \text{GSC} \times \text{Intensity}_{i,,j}$
    end for
end for
```

[1] The multiplication skips the first Freq data points (0, 1,..., Freq-1), in order to avoid low-frequency noise introduced into the digital signal by the OCT system. By observing the cross-sectional images in Figures 5.4, 5.5 and 5.6, a value of Freq=8 yields an optimal signal-to-noise ratio

5.3.3.1 Coarse-Grained Approach

In this approach, presented in Algorithm 5.4, one parallel thread processes a section from the cross-sectional image. The size of the section, processed by a single thread, varies from one vertical line, which corresponds to one channeled spectrum, to half the cross-sectional image.

This approach reduces the number of iterations of the outer ‘for’ loop. In the case of the highest level of parallelism, the number of parallel threads is equal to the number of processes channeled spectra. In this case, the outer loop has a single iteration and each parallel thread processes only one channeled spectrum and generates only one vertical line from the cross-sectional image.

Algorithm 5.4: Coarse-grained approach to the CFD OCT method in cross-sectional imaging

```
Start Parallel Thread$_t$ ( ThreadID[1] )
    Start=(NumberOfChanneledSpectra/NumberOfThreads)×ThreadID
    End=(NumberOfChanneledSpectra/NumberOfThreads)×(ThreadID+1)
    for j=Start to End-1 do
        for i=0 to M-1 do
            $\mathrm{OCTF\_DP}_{i,j} = \mathrm{HW}(\mathrm{OCTF\_DP}_{i,j})$
        end for
        $\mathrm{OCTF\_DP}'_{i,j} = \mathrm{FFT}_j(\mathrm{OCTF\_DP}_{i,j})$
        for i=Freq to M'-1 do
            $\mathrm{Intensity}_{i,j} = \mathrm{AV}(\mathrm{OCTF\_DP}'_{i,j})$
            $\mathrm{CrossSectional}_{i,j} = \mathrm{GSC}\times\mathrm{Intensity}_{i,j}$
        end for
    end for
End Parallel Thread$_t$
```

[1] The value of ThreadID is passed to the thread as a parameter

5.3.3.2 Fine-Grained Approach

The fine-grained approach aims at parallelizing Algorithm 5.3 using the GPU. The number of possible parallel threads launched on the GPU exceeds the number of data points in the OCT frame. This allows the processing of each data point by a single GPU thread. This eliminates the need for any loops and achieves maximum level of parallelism. Algorithm 5.5 presents this approach. The performance of this approach is presented in Table 5.3.

This approach uses the following functions, part of the NVIDIA CUDA library:

cudaMemcpy: copies data from CPU memory (heap or stack) to the global GPU memory. The syntax of this function is analogues with the Standard C function memcpy.

cufftExecR2C: performs one or multiple (batch) Forward FFT

Algorithm 5.5: Fine-grained approach to the CFD OCT method in cross-sectional imaging

```
1: Copy $OCTF\_DP_{i,j}$ from CPU memory to GPU memory /cudaMemcpy/
2: $OCTF\_DP_{i,j}$ = HW ($OCTF\_DP_{i,j}$) /kernel with M×N threads/
3: $OCTF\_DP'_{i,j}$ = $FFT_j$ ($OCTF\_DP_{i,j}$) /cufftExecR2C, batch of N 1D transforms/
4: $Intensity_{i,j}$ = AV ($OCTF\_DP'_{i,j}$) /kernel calling fabsf() with M'×N threads/
5: $CrossSeciotnal_{i,j}$ = GSC×$Intensity_{i,j}$ /kernel with M'×N threads/
6: Copy $CrossSectional_{i,j}$ from GPU memory to CPU memory /cudaMemcpy/
```

5.3.4 MSI OCT Method in Cross-Sectional Imaging

The MSI OCT method, discussed in Chapter 3, uses a prerecorded set of mask signals to resolve the depth in the cross-sectional images. Each mask signal corresponds to a particular depth and must have the same number of data points M, as each channeled spectrum.

The intensity of a point (pixel) with horizontal position **j** and depth **r** is obtained by the following steps:

1. Hamming Window applied on data points from the OCT frame (OCTF_DP) and the mask signals (MS_DP)

2. Cross-correlation between OCTF_DP and MS_DP, Equation 5.10. The MS_DP is conjugated, which is indicated with ‘*’.

3. Integration of the resulting Product, Equation 5.11

4. Scale to the gray scale interval [0..255] using gray scale coefficient (GSC), Equations 5.12

$$Product_{i,j,r} = IDFT\left(DFT\left(HW\left(OCTF\_DP_{i,j}\right)\right) \times DFT\left(HW\left(MS\_DP_{i,r}\right)\right)^{*}\right) \qquad (5.10)$$

$$Intensity_{j,r} = \sum_{i=W_1}^{W_2} \left|Product_{i,j,r}\right| \qquad (5.11)$$

$$CrossSectional_{j,r} = GSC \times Intensity_{j,r} \tag{5.12}$$

Where:

i is the index of the data point within each channeled spectrum and within each mask signal, used both before and after discrete Fourier transform

i = 0, ..., M-1 before discrete Fourier transform

i = 0, ..., M'-1 after discrete Fourier transform

j is the index of channeled spectrum from the OCT frame, j also indexes the horizontal coordinate of the point from the cross-sectional image

j = 0, ..., N-1

r is the index of mask signal from the set of mask signals, r also indexes the vertical coordinate of the point from the cross-sectional image

r = 0, ..., R-1

Algorithm 5.6 presents a sequential implementation of the MSI OCT method by following Equations 5.10, 5.11 and 5.12. This algorithm assumes that the necessary mask signals are already processed and copied to the two-dimensional array MS_DP'$_{i,r}$, as shown in Algorithm 5.2.

Algorithm 5.6: Sequential approach to the MSI OCT method in cross-sectional imaging

```
for j = 0 to N-1 do
    for i = 0 to M-1 do
        $OCTF\_DP_{i,j} = HW ( OCTF\_DP_{i,j} )$
    end for
    $OCTF\_DP'_{i,j} = FFT ( OCTF\_DP_{i,j} )$
    for r = 0 to R-1 do
        for i = Freq to M'-1 do
            $Product'_{i,j,r} = OCTF\_DP'_{i,j} \times MS\_DP'_{i,r}$
        end for
        $Product_{i,j,r} = IFFT ( Product'_{i,j,r} )$
        $Intensity_{j,r} = 0$
        for i = $W_1$ to $W_2$ do
            $Intensity_{j,r}$ += AV ( $Product_{i,j,r}$ )
        end for
        $CrossSectional_{j,r} = GSC \times Intensity_{j,r}$
    end for
end for
```

The core of the algorithm is the multiplication of the data points from the OCT frame and the set of mask signals, line 8 in Algorithm 5.6. Three nested ‘for’ loops carry this multiplication. The total number of these iterations is equal to the product of the number of channeled spectra, number of mask signals, and the number of data points per channeled spectrum. The following coarse-grained and fine grained approaches aim at reducing the number of these iterations and thus improving the performance.

5.3.4.1 Coarse-Grained Approach

A coarse-grained parallel approach of the MSI OCT method is proposed in this research. In this approach, a number of CPU-based parallel threads perform the processing. Their purpose is to reduce the iterations of the loops in Algorithm 5.6.

In this implementation, all parallel threads receive a unique identifier (ThreadID), which is passed as a parameter, apart from the ones associated automatically by the operating system. These identifiers start from 0 and are incremented with 1. Each process preforms End-Start iterations of the outer for loop. A larger number of parallel threads results in a smaller number of iterations and vice versa. The same algorithm applies for a single thread implementation by setting ThreadID=0 and NumberOfThread=1. In this case, Start and End will obtain the following values:

Start = (NumberOfChanneledSpectra/1)×0 = 0

End = (NumberOfChanneledSpectra/1)×(0+1) = NumberOfChanneledSpectra

Algorithm 5.7: Coarse-grained approach to the MSI OCT method in cross-sectional imaging

```
Start Parallel Thread$_t$ ( ThreadID[1] )
    Start=( NumberOfChanneledSpectra/NumberOfThreads )×ThreadID
    End=( NumberOfChanneledSpectra/NumberOfThreads )×( ThreadID+1 )
    for j = Start to End-1 do
        for i = 0 to M-1 do
            $OCTF\_DP_{i,j}$ = HW ( $OCTF\_DP_{i,j}$ )
        end for
        $OCTF\_DP'_{i,j}$ = $FFT_j$ ( $OCTF\_DP_{i,j}$ )
        for r = 0 to R-1 do
            for i = Freq to M'-1 do
                $Product'_{i,j,r}$ = $OCTF\_DP'_{i,j}$×$MS\_DP'_{i,r}$
            end for
            $Product_{i,j,r}$ = $IFFT_{j,r}$ ( $Product'_{i,j,r}$ )
            $Intensity_{j,r}$ = 0
            for i = W_Start to W_End do
                $Intensity_{j,r}$ += abs ( $Product_{i,j,r}$ )
            end for
            $CrossSectional_{j,r}$ = GSC×$Intensity_{j,r}$
        end for
    end for
End Parallel Thread$_t$
```

[1] The value of ThreadID is passed to the thread as a parameter

5.3.4.2 Fine-Grained Approach

The fine-grained approach is carried out by parallel GPU threads launched by the kernels presented in Algorithm 5.8. Like the previous approaches to the MSI OCT method, this algorithm assumes that the mask signals are already processed, as presented in Algorithm 5.2.

In this approach, the result of the multiplication ($Product_{i,j,r}$), line 4 in Algorithm 5.8, is a three-dimensional array, where i is the index of the data point within the channeled spectra, j is the index of the channeled spectra which corresponds to the horizontal index, and r is the index of the mask signal used to resolve the depth of the image, which is the vertical index. This array is mapped on a two-dimensional GPU grid as follows:

index i - threadIx.x (the index of the thread with the blocks)

index j - blockIdx.x (the X index of the two-dimensional block)

index r - blockIdx.y (the Y index of the two-dimensional block)

This mapping allows the elimination of all 'for' loops from the previous approaches.

Algorithm 5.8: Fine-grained approach to the MSI OCT method in cross-sectional imaging

```
Copy $OCTF\_DP_{i,j}$ from CPU memory to GPU memory /cudaMemcpy/
$OCTF\_DP_{i,j} = HW ( OCTF\_DP_{i,j} )$ /kernel with M×N/
$OCTF\_DP'_{i,j} = FFT_{j} ( OCTF\_DP_{i,j} )$ /cufftExecR2C, batch of N 1D transforms/
$Product_{i,j,r} = OCTF\_DP'_{i,j} \times MS\_DP'_{i,r}$ /kernel with M'×N×R threads/
$Product_{i,j,r} = IFFT_{j \times r} ( Product'_{i,j,r} )$ /cufftExecC2R, batch of N×R 1D transforms/
$Intensity_{j,r} = Integrate ( abs ( Product_{i,j,r} /) )$ /kernel with N×R threads/
$CrossSectional_{j,r}=GSC \times Intensity_{j,r}$ /kernel with N×R threads/
Copy $CrossSectional_{j,r}$ from GPU memory to CPU memory /cudaMemcpy/
```

5.3.5 Performance and Results of Cross-Sectional Imaging in OCT

The OCT system studied in this research scans up to 500 points during cross-sectional imaging. The parallel optimizations are tested with three digital signals (OCT Frames):

OCT Frame A: Human eye, 100×1024 data points resulting in an image of 100×513 points (pixels) in CFD OCT method and 100×512 points (pixels) in MSI OCT method

OCT Frame B: Human eye, 200×1024 data points resulting in an image of 200 ×513 points (pixels) in CFD OCT method and 200×512 points (pixels) in MSI OCT method

OCT Frame C: Laminated paper, 500×1024 data points resulting in an image of 500×513 points (pixels) in CFD OCT method and 500×512 points (pixels) in MSI OCT method

Tables 5.5 and 5.6 present the performances of the CFD and MSI methods measured in term of latency. The reported elapsed time does not include the time to copy the data to and from the global GPU memory. These latencies are presented separately in Tables 5.3 and 5.4.

Table 5.5: Performance of the CFD OCT method in milliseconds

Approach		Width of Cross-Sectional CFD OCT Image (number of channeled spectra)				
		100[1]	200[1]	300[2]	400[2]	500[1]
Single Thread (CPU)		3.0	4.6	6.4	6.6	7.2
Coarse-Grained	2 Threads	4.2	4.2	4.8	6.4	7.0
	4 Threads	4.0	4.0[3]	4.2[3]	4.6[3]	4.8[3]
	5 Threads	4.4	4.6	5.0	5.2	7.4
	10 Threads	7.8	8.2	8.6	7.6	10.4
	20 Threads	13.6	14.8	17.6	19.8	22.8
	25 Threads	19.8	21.6	24.2	26.8	34.8
	50 Threads	47.8	59.4	72.2	85.0	91.4
	100 Threads	177.8	182.4	204.2	210.2	216.2
Fine-Grained		0.096	0.098	0.12	0.16	0.198

[1] OCT Frames collected from OCT system. The corresponding images are displayed in Figures 5.4, 5.5 and 5.6

[2] OCT Frames obtained by reading the first 300 and 400 respectively channeled spectra from the OCT Frame with total amount of channeled spectra of 500

[3] Optimal coarse-grained performance used to evaluate the speed-up

Table 5.6: Performance of the MSI OCT method in milliseconds

Approach		Width of Cross-Sectional MSI OCT Image (number of channeled spectra)				
		100	200	300	400	500
Single Thread (CPU)		477.4	898.2	1362.4	1823.4	2372.0
Coarse-Grained	2 Threads	262.4	514.4	748.6	989.0	1228.2
	4 Threads	162.6	288.2	414.4	538.2	657.8
	5 Threads	123.2	235.8	335.4	439.8	552.2
	10 Threads	86.8	173.4	234.2	313.8	402.4
	20 Threads	76.2[1]	140.8[1]	218.2	294.8	368.2[1]
	25 Threads	83.2	155.4	211.8[1]	292.8[1]	380.8
	50 Threads	119.2	176.2	225.8	334.6	402.8
	100 Threads	195.8	227.2	276.0	366.8	418.4
Fine-Grained		6.4	13.8	21.2	28.4	35.6

[1] Optimal performance of the coarse-grained approach used to evaluate the speed-up

Figures 5.2 and 5.3 present the performance in terms of latency of the coarse-grained approaches to CFD and MSI methods. The MSI method, unlike the CFD one, benefits significantly from this parallel approach. It demonstrates the scalability discussed in Chapter 2 and presented in Figure 2.1.A.

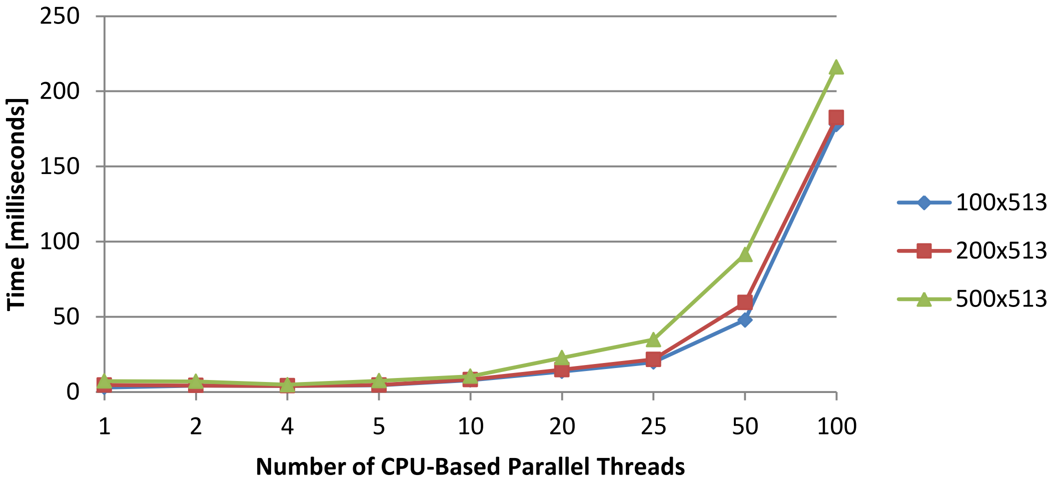


Figure 5.2: Performance in terms of latency of coarse-grained approach of the CFD OCT method

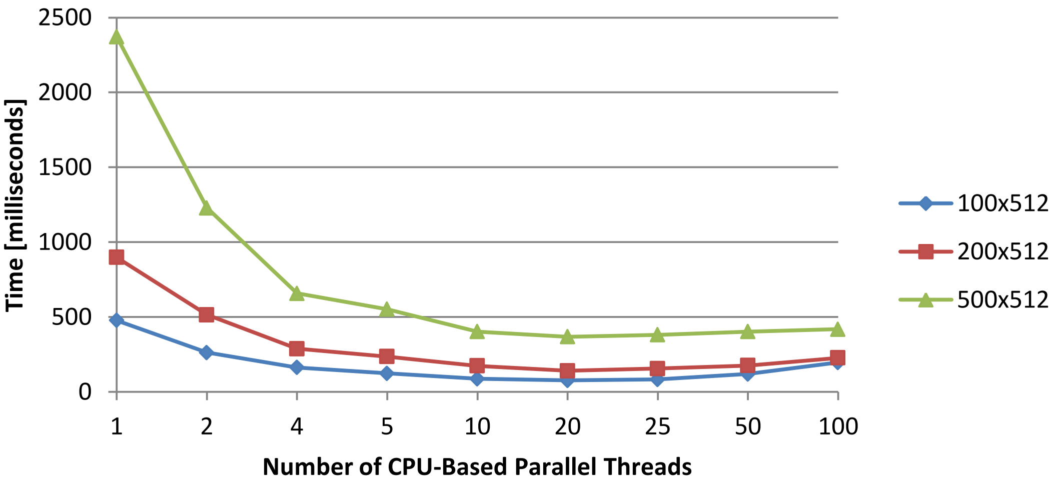


Figure 5.3: Performance in terms of latency of the coarse-grained approach of the MSI OCT method

The following conclusions can be made, based on Tables 5.5 and 5.6 and Figures 5.2 and 5.3:

- In all cases, the fine-grained parallel optimization outperforms the coarse-grained one.

- Unlike the coarse-grained, the fine-grained approach in all cases launches one parallel thread per data point in both the CFD and MSI OCT methods.

- The CFD OCT method does not significantly benefit from the coarse-grained parallel optimization. This is due to the smaller number of computations, compared with the MSI OCT method, and the time needed to launch the CPU-based parallel threads presented in Chapter 4.

- The MSI OCT method benefits significantly from the coarse-grained approach, as seen in Figure 5.3 and reach optimal performance with 20-25 parallel threads. This performance depends on the multi-core architecture and the thread management provided by the operating system.

- The number of threads which delivers optimal performance in the coarse-grained approach depends on the OCT method and the size of the processed signal:

- The CFD OCT method reaches optimal performance with 2-4 parallel threads.

- The MSI OCT method reaches optimal performance with 20-25 parallel threads.

Table 5.7 illustrates the improvements in the performances, presented as speed-up, achieved by coarse-grained and fine-grained parallelization.

Table 5.7: Speed-up of the coarse-grained and fine-grained cross-sectional imaging

Width of cross-sectional image	CFD OCT method		MSI OCT method	
	Coarse-grained speed-up	Fine-grained speed-up	Coarse-grained speed-up	Fine-grained speed-up
100	-	31.25	6.27	74.59
200	1.15	46.94	6.38	65.09
300	1.52	53.33	6.43	64.26
400	1.43	41.25	6.23	64.20
500	1.67	36.36	6.44	66.63

As seen in Table 5.7, the GPU-based fine-grained approaches deliver significant improvement of the performance. Two factors contribute to this improvement. The first one is the higher level of parallelism of the fine-grained approach, where each data point is processed by one parallel thread. The second one is the utilization of the NVIDIA CUFFT library, which is optimized for NVIDIA GPU.

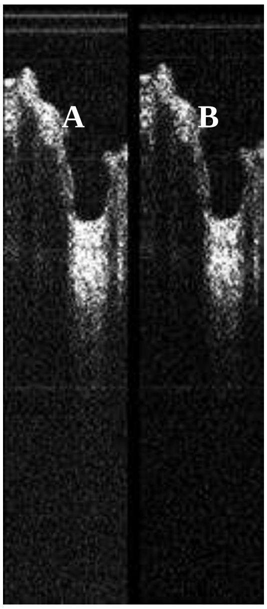


Figure 5.4: Cross-sectional OCT images of the human eye. A: CFD OCT method 100×513, B: MSI OCT method 100×512

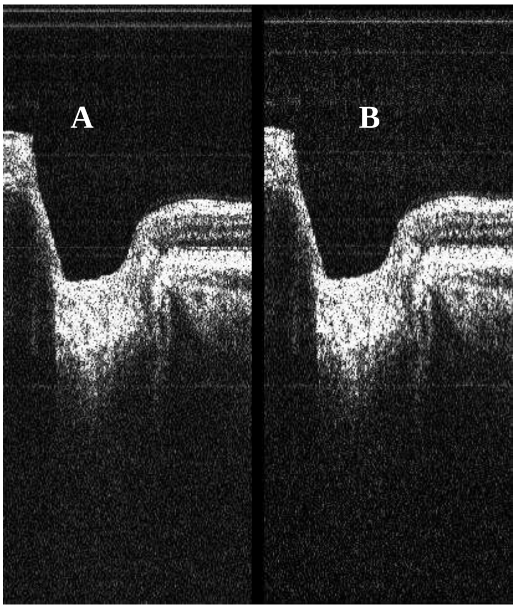


Figure 5.5: Cross-sectional images of the human eye: A: CFD OCT method 200×513, B: MSI OCT method 200×512

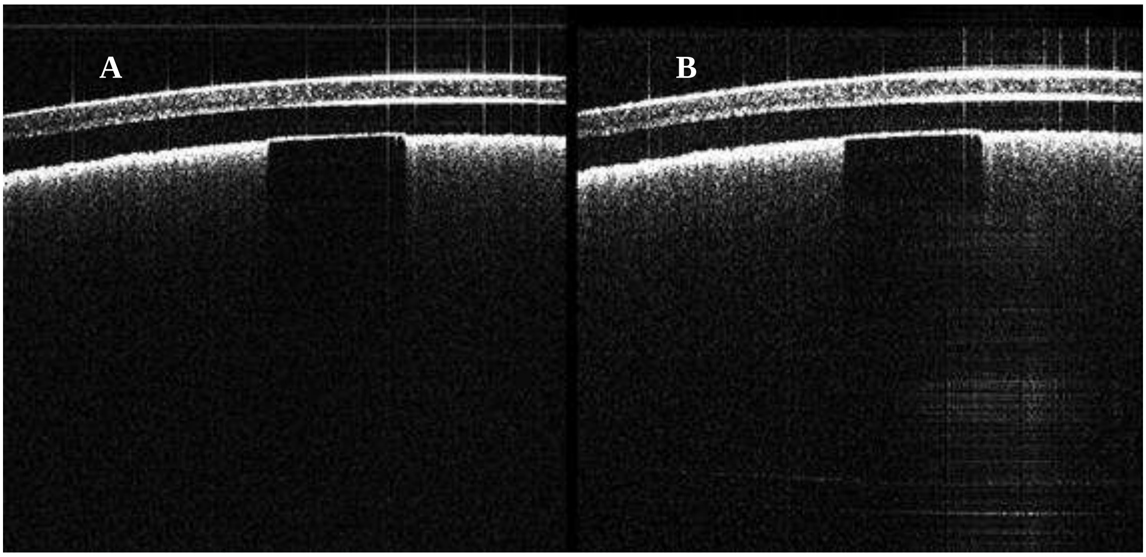


Figure 5.6: Cross-sectional OCT images of laminate paper. A: CFD OCT method with 500×513 pixels, B: MSI OCT method with 500×512 pixels

5.4 En-face Imaging in OCT

In en-face imaging, the OCT system generates multiple en-face images from different layers below the surface of semitransparent objects. Both CFD and MSI methods are capable of producing en-face images. Sequential implementations of these methods limit the number of en-face images generated in real-time. This is especially the case in the MSI OCT method. The coarse-grained and fine-grained parallel approaches aim at increasing the number of simultaneously generated images within the real-time criterion of the OCT system.

5.4.1 Structure of the OCT Signal in En-Face Imaging

In en-face imaging, an OCT system during a single scan of the beam of light over the object acquires one OCT frame of M×N×L data points, where M is the size of the channeled spectrum, N is the number of points per line, and L is the number of lines. The OCT system studied in this research acquires one OCT frame of 1024x200x192 data points in 0.8 seconds [12]. As in the cross-sectional imaging, these values are determined by the sweeping capabilities of the OCT system and define its real-time criterion.

The OCT frame, as generated by the OCT system, can be presented as two-dimensional array of N×L channeled spectra, or three-dimensional array of M×N×L data points. The corresponding OCT frame after discrete Fourier transform has M'×N×L data points, where M'=M/2+1.

$$OCTF^{TD} = CS_{j,k}^{TD} = DP_{i,j,k}^{TD}; DP\epsilon\boldsymbol{R}; i = 0, \cdots, M-1, j = 0, \cdots, N-1; k = 0, \cdots, L-1 \qquad (5.13)$$

$$OCTF^{FD} = CS_{j,k}^{FD} = DP_{i,j,k}^{FD}; DP\epsilon\boldsymbol{C}; i = 0, \cdots, M'-1; j = 0, \cdots, N-1; k = 0, \cdots, L-1 \qquad (5.14)$$

The main differences between the generation of cross-sectional and en-face images are the size of the OCT frame, the three-dimensional volumetric character in en-face imaging, and the ability to produce multiple en-face images based on a single OCT frame.

A screen with a standard resolution of 1920×1080 pixels is suitable for displaying simultaneously in one window up to 40 (8×5) en-face images of 200×192 pixels, which amounts to (8×200)×(5×192) pixels, or 1600×960 pixels. The subsequent parallel optimizations aim at reaching this number of en-face images in real-time.

5.4.2 Stages in En-Face Imaging

In en-face OCT imaging, a single en-face OCT frame can generate multiple en-face images, unlike the cross-sectional OCT imaging, where only one image can be generated by one cross-sectional OCT frame.

Figure 5.7 presents the processing steps of the two OCT methods studied in this thesis, namely the CFD and the MSI. The first two steps, namely HW and FFT, are the same for both methods. These two steps do not depend on the number of processed and generated en-face images. The amount of calculations remains the same, whether a single image or multiple images are generated. In the CFD OCT method, the number of en-face images depends on the number of data points selected at the AV steps. In the MSI OCT method, the number of images is determined by the number of complex multiplications, performed between the OCT frame after Hamming window and FFT and the selected mask signal. Each multiplication expects a different mask signal. Based on this, the en-face imaging can be divided into two stages: first one consisting of HW and FFT with fixed number of computations, and a second one with computations that depend on the number of en-face images, illustrated in Table 5.8.

Table 5.8: Stages in the CFD and the MSI methods

Stage	Steps	
	CFD OCT method	MSI OCT method
1	Hamming window	Hamming window
	FFT	FFT
2	Absolute value	Complex multiplication
		IFFT
	Scale	Integration
		Scale

The fine-grained approach launches one kernel (or FFT/IFFT call) per each step and applies the same parallelization strategy. This strategy aims at processing each data point by one parallel thread from the GPU grid. The only exception is the integration step, as discussed in Chapter 4.

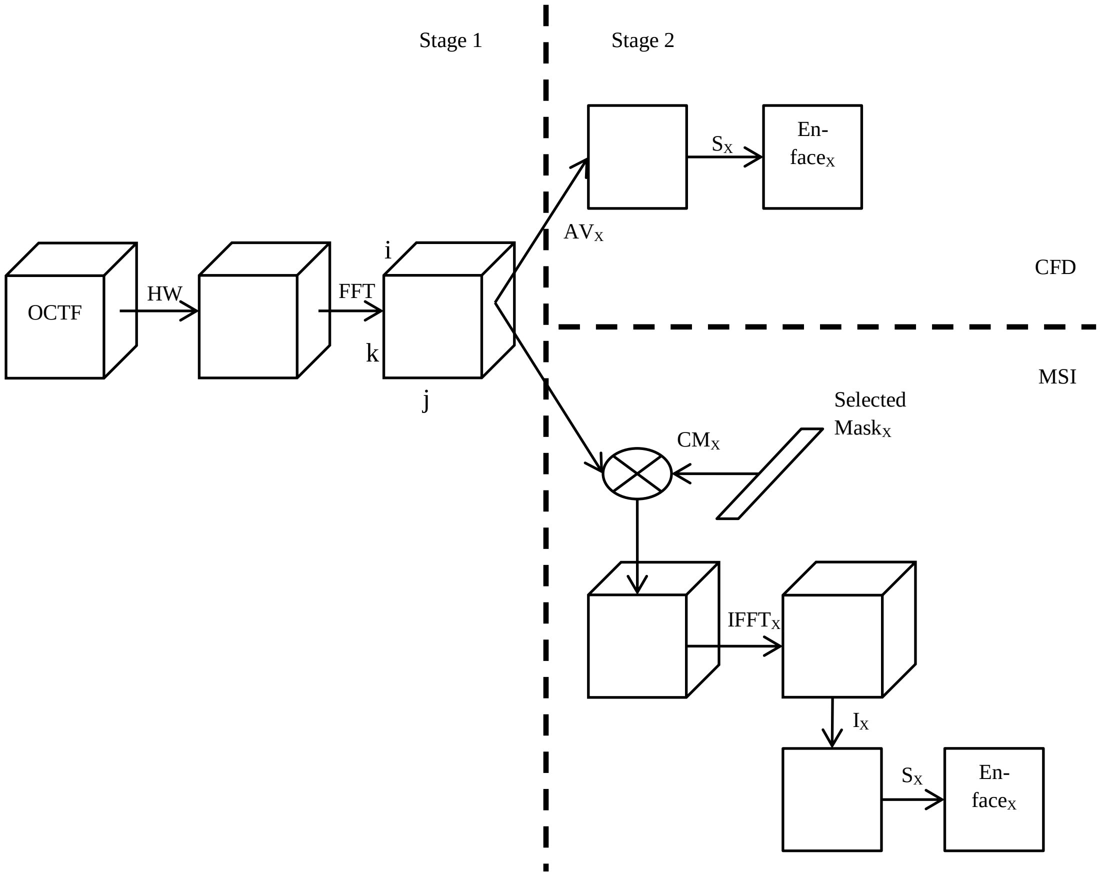


Figure 5.7: Stages in CFD and MSI methods. X is the index of the en-face images. In the CFD method, X is chosen from the **i** coordinates from the OCT frame after HW and FFT. In the MSI method, X corresponds to the index of the selected mask from the set of mask signals. In the MSI method, HW, FFT and CC are already applied on the mask signal during Offline mode

In the coarse-grained approach, on the other hand, optimal parallel performance can be achieved by much smaller number of parallel threads. Each of these parallel threads performs all steps of the methods, processing fraction of the digital signal. As a result, stages 1 and 2 are expected to achieve optimal parallel performances with a different number of CPU-based parallel threads. Therefore, an optimal performance of the overall en-face imaging can be achieved by parallelizing the two stages separately.

Algorithm 5.9: Coarse-grained approach to Stage 1 (Hamming window and FFT)

```
Start Parallel Thread$_t$ ( ThreadID[1] )
    Start=( NumberOfChanneledSpectra/NumberOfThreads )×ThreadID
    End=( NumberOfChanneledSpectra/NumberOfThreads )×( ThreadID+1 )
    for j = Start to End-1 do
        for i = 0 to M-1 do
            $OCTF\_DP_{i,j}$ = HW ( $OCTF\_DP_{i,j}$ )
        end for
        $OCTF\_DP'_{i,j}$ = $FFT_j$ ( $OCTF\_DP_{i,j}$ )
    end for
End Parallel Thread$_t$
```

[1] The value of ThreadID is passed to the thread as a parameter

This algorithm divides the total number of channeled spectra equally among the parallel threads. The OCT system studied in this research generates OCT frames with N=200 lines and L=192 columns and has a total number of 38400 channeled spectra. As seen from Table 5.9, the optimal number of parallel threads to process a single OCT frame is 8. Each parallel thread applies HW on 1024×4800 data points and performs 4800 FFT.

Table 5.9: Performance of coarse-grained parallel optimization of HW and FFT in milliseconds

Number of parallel threads	1	2	4	6	8	12	16	32	64
Number of Channeled Spectra/FFT Per Thread	38400	19200	9600	6400	4800	3200	2400	1200	600
Time [m]	248.8	220.8	218.8	218.2	216.4	228.4	301.8	325.2	376.2

During Stage 2, the amount of computations depends on the number of processed en-face images. The CFD method performs absolute value (AV) and scale (S). The MSI method performs complex multiplication (CM), inverse FFT (IFFT), integration (I) and scale (S). In this stage in the coarse-grained approaches both methods launch one parallel thread per en-face image. Two factors determined this decision:

1. The structure of the algorithm is more consistent, if each en-face image is processed by one thread.

2. The number of processed en-face images can be controlled directly by selecting the number of parallel threads processing Stage 2. Smaller number of images provides better performance but less information and vice versa. This allows a trade-off between speed and amount of information.

5.4.3 CFD OCT Method in En-Face Imaging

The CFD OCT method employed in en-face imaging, presented in Equation 4.15, is based on the same principle as in the cross-sectional imaging. The maximum number of possible en-face images, generated by the CFD OCT method, is equal to the number of data points in the channeled spectrum after discrete Fourier transform (M'), as seen in Equation 5.4 and Figure 5.1. In this implementation M' is equal to 513.

Algorithm 5.10: Sequential approach to the CFD OCT method in en-face imaging (Stages 1 and 2)

```
for k=0 to L-1 do
    for j=0 to N-1 do
        for i=0 to M-1 do
            $OCTF\_DP_{i,j,k} = HW ( OCTF\_DP_{i,j,k} )$
        end for
        $OCTF\_DP'_{i,j,k} = FT_{j,k} ( OCTF\_DP_{i,j,k} )$
    end for
end for
for k=0 to L-1 do
    for j=0 to N-1 do
        for i=0 to NumberOfEnFaceImages do
            $ImageID^{A} = Start + Step \times i$
            $Intensity_{ImageID,j,k} = AV ( OCTF\_DP'_{ImageID,j,k} )$
            $EnFace_{ImageID,j,k} = GSC \times Intensity_{ImageID,j,k}$
        end for
    end for
end for
```

[A] The first Freq data points correspond to the low frequency components of the signal. The selection of Start larger than the value of Freq in the previous methods implements a high-pass filter

Algorithm 5.10 presents the sequential approach to the CFD method. In this algorithm, the data is processed by a single thread of execution. Therefore, the distinction between the stages does not apply in this case.

$$Image_i(P_{jk}) = GSC \times \left|FT_{jk}\left(HW\left(DP_{ijk}^{TD}\right)\right)\right|; i = 0, \cdots, M' - 1 \tag{5.15}$$

5.4.3.1 Coarse-Grained Approach

The coarse-grained approach divides the processing into the aforementioned two stages. During the second stage, illustrated by Algorithm 5.11, two global variables are employed to define the selected en-face images: Start and Step. The first one defines the index of the first image, the second one the step between successive images, Figure 5.8 An ID of the image is calculated as described in Equation 5.16 and passed to each parallel thread as a parameter.

$$ImageID_i = START + STEP \times i; i = 0, \cdots, 39, ImageID_i < 513 \tag{5.16}$$

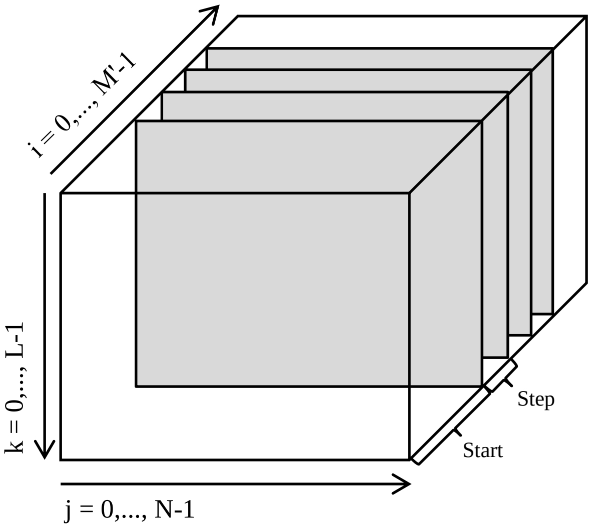


Figure 5.8: CFD OCT method generating multiple en-face images. The cube represents the data points from an OCT frame after Hamming window, FFT, absolute value and scale (multiplication with GSC). The dimension of the images is N×L, the number of possible en-face images is M'

Algorithm 5.11: Coarse-grained approach to the CFD OCT method in en-face imaging (Stage 2)

```
Start Parallel Thread$_t$ ( ThreadID[1] )
    for k=0 to L-1 do
        for j=0 to N-1 do
            ImageID[A] = Start[B] + Step[B] × ThreadID
            $Intensity_{ImageID,j,k}$ = AV ( $OCTF\_DP'_{ImageID,j,k}$ )
            $EnFace_{ImageID,j,k}$ = GSC × $Intensity_{ImageID,j,k}$
        end for
    end for
End Parallel Thread$_t$
```

[A] ThreadID is passed to the thread as a parameter. It is used to calculate the ImageID

[B] Start and Step are global variables

In an integrated solution, these global variables are part of a shared memory. In that case, their values are set in real-time by the user interface, which is a LabVIEW Virtual Instrument. In the same case, the value of GSC is also part of the shared memory, allowing the user to control the contrast of the images. The integrated implementation is discussed in Chapter 6.

5.4.3.2 Fine-Grained Approach

The fine-grained approach, presented in Algorithm 5.12, is implemented by multiple parallel threads on the GPU, launched by a number of kernels. The processing is organized in a 'for' loop. Each iteration of this loop processes one en-face image. The kernels called from the body of the loop launch one GPU-based parallel thread per each data point.

The number of processed en-face images is controlled by the number of iterations of that loop.

The three-dimensional array $EnFace_{i,j,k}$ holds the result of the processing. Its first dimension is the image ID, the second dimension is the X-coordinate, and the third dimension is the Y-coordinate of the en-face image.

Algorithm 5.12: Fine-grained approach to the CFD OCT method in en-face imaging (Stages 1 and 2)

```
Copy $OCTF\_DP_{i,j,k}$ from CPU memory to GPU memory /cudaMemcpy/
$OCTF\_DP_{i,j,k}$ = HW ( $OCTF\_DP_{i,j,k}$ ) /kernel with M×N×L/
$OCTF\_DP'_{i,j,k}$ = $FFT_{j \times k}$ ( $OCTF\_DP_{i,j,k}$ ) /cufftExecR2C, batch of N×L 1D transforms/
for i=0 to NumberOfEnFaceImages-1 do
    ImageID = Start + Step×i
    $Intensity_{j,k}$ = AV ( $OCTF\_DP'_{ImageID,j,k}$ ) /kernel with N×L threads/
    $EnFace_{i,j,k}$ = GSC×$Intensity_{j,k}$ /kernel with N×L threads/
end for
Copy $EnFace_{i,j,k}$ from GPU memory to CPU memory /cudaMemcpy/
```

In the last two kernels, which calculate the absolute values and the gray scale values, the input signal and the resulting signal have the same number of data points. They can be implemented both as one or two kernels. Both cases generate equivalent results in performance.

5.4.4 MSI OCT Method in En-Face Imaging

The implementation of the MSI OCT method, presented in Algorithm 5.13, follows Equations 5.10, 5.11 and 5.12. The maximum number images is equal to the number of prerecorded mask signals $MS\_DP_{i,r}$, where i is the index of the data point within each mask signal and r is the index of the mask itself. The size of each mask signal must be equal to the size of the channeled spectrum (M) from the OCT frame.

Algorithm 5.13: Sequential approach to the MSI OCT method in en-face imaging (Stages 1 and 2)

```
for k = 0 to L-1 do
    for j = 0 to N-1 do
        for i = 0 to M-1 do
            $OCTF\_DP_{i,j,k} = HW ( OCTF\_DP_{i,j,k} )$
        end for
        $OCTF\_DP'_{i,j,k} = FFT_{j,k} ( OCTF\_DP_{i,j,k} )$
    end for
end for
for r = 0 to Q-1 do
    $MaskID = Start + Step \times r$
    for i = 0 to M'-1 do
        $Single\_MS\_DP'_{i} = MS\_DP'_{i,MaskID}$
    end for
    for k = 0 to L-1 do
        for j = 0 to N-1 do
            for i = Freq to M'-1 do
                $Product'_{i,j,k} = OCTF\_DP'_{i,j,k} \times Single\_MS\_DP'_{i}$
            end for
            $Product_{i,j,k} = IFFT_{j,k} ( Product'_{i,j,k} )$
            $Intensity_{MaskIDr,j,k} = 0$
            for i = W_Start to W_End do
                $Intensity_{MaskIDr,j,k} += Product_{i,j,k}$
            end for
            $EnFace_{MaskIDr,j,k} = GSC \times Intensity_{MaskIDr,j,k}$
    end for
end for
```

This research uses a set of R=512 mask signals, each mask signal has the size of M=1024, which amounts to M×R=524288 data points. A smaller number of mask signals (Q<R) are selected from this set and take part in the real-time processing. These masks are selected in real-time during the operation of the OCT system, as shown in Equation 5.17. They are selected in a similar way as the ImageID is selected in the CFD-based en-face imaging. Selecting a mask signal is equivalent to selecting an image from a particular depth.

$$MaskID_r = START + STEP \times r; r = 0, \cdots, Q-1; MaskID_r \leq R-1; Q < R \qquad (5.17)$$

5.4.4.1 Coarse-Grained Approach

The coarse-grained approach of the MSI method is presented in Algorithm 5.14. Like the CFD, the MSI method is also divided into two stages:

1. The first stage applies Hamming window and FFT on the OCT frame. These two steps do not depend on the selected number of processed and visualized en-face images. As a result, the MSI method uses the same parallel realization of HM and FFT as in CFD.

2. The second stage applies complex multiplication between the OCT frame and the mask signal, inverse FFT, integration, and scale. The parallel optimization of the second stage launches one parallel thread per en-face image.

Algorithm 5.14: Coarse-grained approach to the MSI OCT method in en-face imaging (Stage 2)

```
Start Parallel Thread_t ( ThreadID[A] )
    MaskID = Start + Step×ThreadID
    for i = 0 to M'-1 do
        $Single\_MS\_DP'_i = MS\_DP'_{i,MaskID}$
    end for
    for k = 0 to L-1 do
        for j = 0 to N-1 do
            for i = Freq to M'-1 do
                $Product'_{i,j,k} = OCTF\_DP'_{i,j,k} \times Single\_MS\_DP'_i$
            end for
            $Product_{i,j,k} = IFFT_{j,k}( Product'_{i,j,k} )$
            $Intensity_{j,k,MaskID} = 0$
            for i = $W_1$ to $W_2$ do
                $Intensity_{j,k,MaskID} += Product_{i,j,k}$
            end for
            $EnFace_{,j,k,MaskID} = GSC \times Intensity_{j,,k,MaskID}$
        end for
    end for
End Parallel Thread_t
```

[A] ThreadID is passed as a parameter to the parallel threads

The proposed coarse-grained approach launches one parallel thread for each en-face image. Each parallel thread extracts one mask from the set of mask signals. This mask determines the processed depth visualized by the corresponding en-face image. The coarse-grained approach replaces the outer loop from Algorithm 5.13. The OCT frame and the set of mask signals are stored in the global memory, accessible by all parallel threads. Each thread receives a parameter, MaskID, which is used to locate the corresponding mask from the set of mask signals, and thus processes the respective en-face image.

5.4.4.2 Fine-Grained Approach

Algorithm 5.15 presents the fine-grained approach to the MSI method in en-face OCT imaging. The kernels used in this approach launch one GPU-based parallel thread for each data point. The only exception is the Integration, where one parallel thread is launched for every channeled spectrum, integrating all values within the channeled spectrum itself.

Algorithm 5.15: Fine-grained approach to the MSI OCT method in en-face imaging (Stage 1 and 2)

```
Copy $OCTF\_DP_{i,j,k}$ from CPU memory to GPU memory /cudaMemcpy/
$OCTF\_DP_{i,j,k}$ = HammingWindow ( $OCTF\_DP_{i,j,k}$ ) /kernel with M×N×L threads/
$OCTF\_DP'_{i,j,k}$ = $FFT_{j\times k}$ ( $OCTF\_DP_{i,j,k}$ ) /cufftExecR2C, batch of N×L 1D transforms/
for r = 0 to Q-1 do
    MaskID = Start + Step×r
    $Single\_MS\_DP'_{i}$ = $MS\_DP'_{i,MaskID}$
    $Product'_{i,j,k}$ = $OCTF\_DP'_{i,j,k}\times Single\_MS\_DP'_{i}$ /kernel/
    $Product_{i,j,k}$ = $IFFT_{j\times k}$ ( $Product'_{i,j,k}$ ) /cufftExecC2R, batch of N×L 1D transforms/
    $Intensity_{j,k}$ = Integrate ( $Product_{i,j,k}$ ) /kernel/
    $EnFace_{j,k,r}$ = $GSC\times Intensity_{j,k}$ /kernel with N×L threads/
end for
Copy $EnFace_{j,k,r}$ from GPU memory to CPU memory /cudaMemcpy/
```

The fine-grained processing is organized in a 'for' loop, processing one en-face image per iteration. Although it is possible to process more than one image per iteration, this approach is chosen for two reasons:

1. This requires allocation for variables for only one en-face image. The only exception being the EnFace variable, which accumulates the gray scale values for all images. As seen in Table 5.10, the GPU memory needed for larger number of en-face images is significant.

2. It gives direct control in real-time over the number of processed and displayed images.

Table 5.10: Size of GPU memory needed in en-face OCT imaging using the MSI method

	Size of signals for 1 en-face image (data points)	Size of signals for 40 en-face images (data points)	Data type and size
Input signal in TD	1024×200×192 =39,321,600	1024×200×192 =39,321,600	Single 4 bytes
Input signal in FD	513×200×192 =19,699,200	513×200×192 =19,699,200	Complex 8 bytes
Product in FD	513×200×192 =19,699,200	40×513×200×192 =787,968,000	Complex 8 bytes
Product in TD	1024×200×192 =39,321,600	40×1024×200×192 =1,572,864,000	Single 4 bytes
Integrated product in TD	200×192 =38,400	40×200×192 =1,536,000	Single 4 bytes
Scaled product in TD	200×192 =38,400	40×200×192 =1,536,000	Integer 4 bytes
Total memory (megabytes)	~600	~12,232	-

5.4.5 Performance and Results of En-Face Imaging in OCT

The OCT system during en-face imaging generates OCT frames at the rate of 800 milliseconds, which is the real-time requirement of the system. All approaches allow direct control over the number of en-face images, and thus allowing the operator of the system to choose optimal usage.

Tables 5.11 and 5.12 and Figures 5.9 and 5.10 present the performances of the sequential, coarse-grained, and fine grained approaches of the CFD and MSI methods. The maximum number of generated en-face images is 40, as presented in Figures 5.11 and 5.12.

Table 5.11: Performance of the CFD OCT method in en-face imaging, in milliseconds

Approach	Number of en-face images								
	1	2	4	8	12	16	24	32	40
Sequential approach	218.2	222.0	224.6	228.4	230.2	231.8	233.0	237.8	252.4
Coarse-grained	-	215.6	218.4	232.2	235.8	240.8	261.4	296.2	307.8
Fine-grained	9.0	10.2	15.2	21.8	32.2	36.4	56.4	74.8	85.0

Table 5.12: Performance of the MSI OCT method in en-face imaging, in milliseconds

Approach	Number of en-face images								
	1	2	4	8	12	16	24	32	40
Sequential approach	478.2	746.4	1303.8	2290.2	3368.2	4392.4	6687.0	8887.2	10955.8
Coarse-grained	-	553.4	559.6	655.2	731.2	741.6	1156.2	1441.8	1747.8
Fine-grained	11.6	17.2	29.0	52.2	75.0	98.4	144.0	190.2	236.2

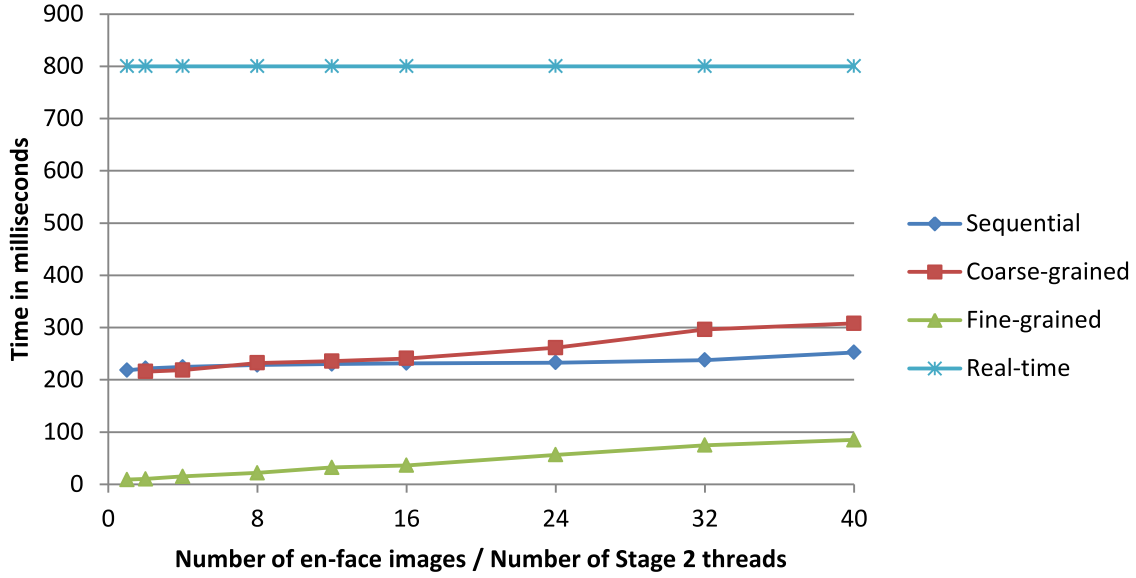


Figure 5.9: Performance of the CFD OCT method generating 1 to 40 en-face images. Coarse-grained approach launches one thread per image in Stage 2

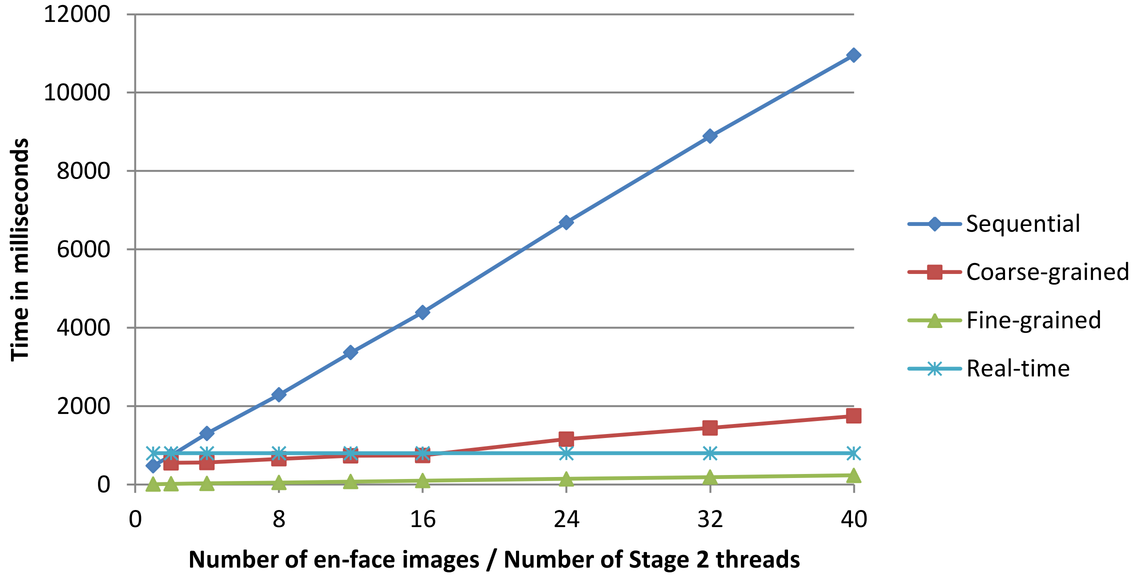


Figure 5.10: Performance of the MSI OCT method generating from 1 to 40 en-face images. Coarse-grained approach launches one thread per image in Stage 2

Unlike in the cross-sectional imaging where different coarse-grained approaches were proposed, in the en-face imaging the size of the data, which is the number of processed en-face images, determines the number of the parallel threads. Therefore, Figures 5.9 and 5.10, besides the performance in terms of latency, also illustrate the scalability of the parallel approaches as defined in Figure 2.1.B.

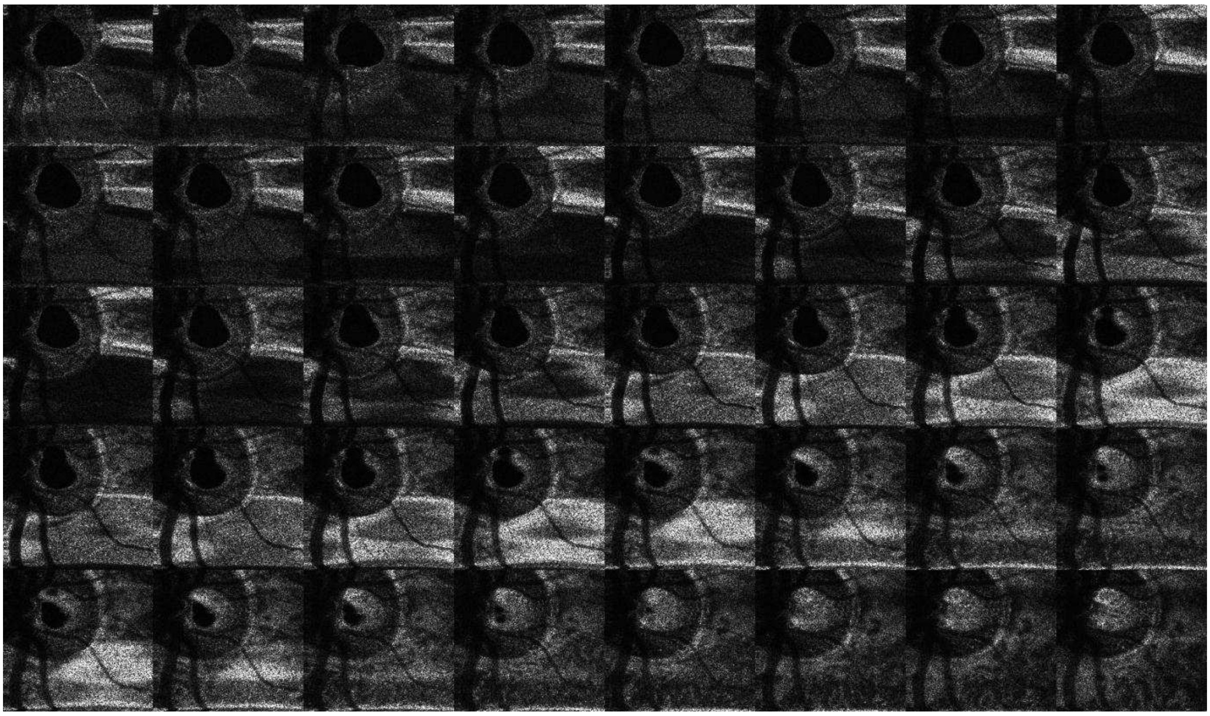


Figure 5.11: 40 en-face OCT images of the human eye

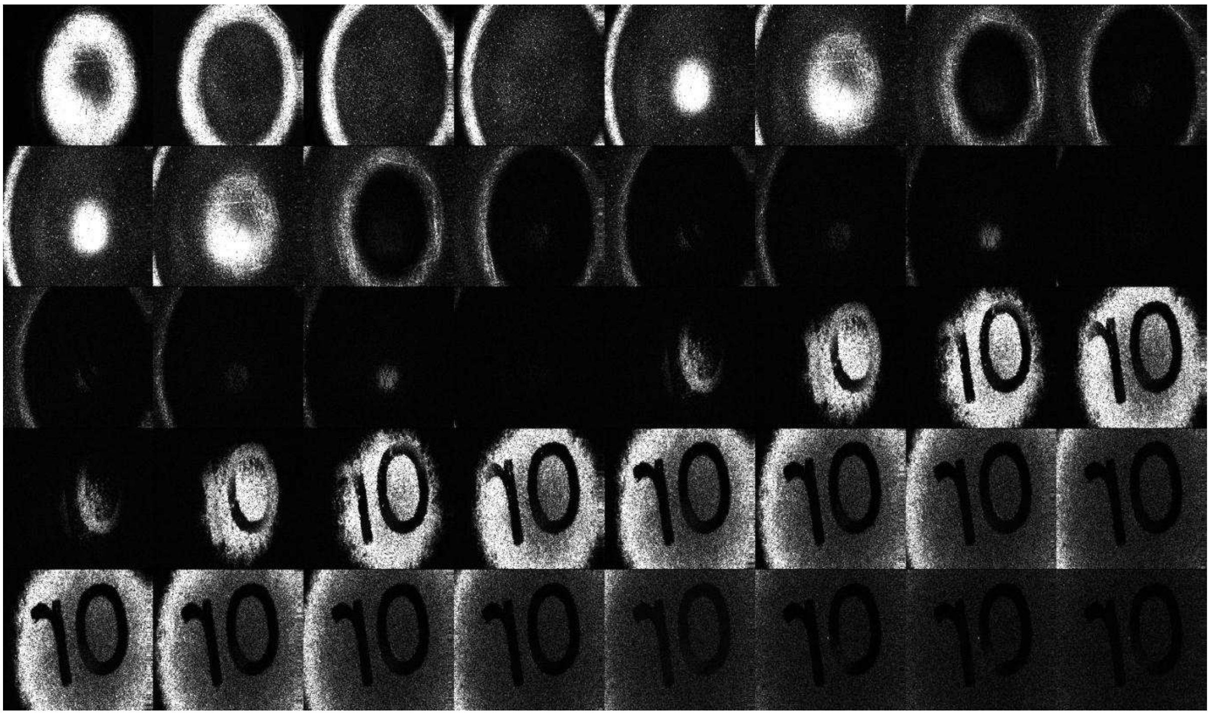


Figure 5.12: 40 en-face OCT images of laminated paper

Based on these results, the following conclusions can be made:

1. The CFD OCT method reaches real-time operation in all approaches. The actual

computations in the CFD method take less time than the CPU-GPU copy. For a small number of en-face images, up to four, the coarse-grained approach outperforms the sequential. For the remaining cases, the overheads from the multithreading prevents the coarse-grained approach to outperform the sequential.

2. In the MSI OCT, the fine-grained optimization delivers real-time performance for all sizes of processed signals. The corresponding coarse-grained optimization reaches real-time operation with up to 16 en-face images.

3. As seen from the speed-up presented in Table 5.13, the less computationally intensive CFD method does not benefit from the parallel optimization. The MSI method on the other hand improves its performance significantly.

Table 5.13: Speed-up of the coarse-grained and fine-grained en-face imaging

Number of en-face images	CFD OCT method		MSI OCT method	
	Coarse-grained	Fine-grained	Coarse-grained	Fine-grained
2	1.03	21.76	1.35	43.40
4	1.03	14.78	2.33	44.96
8	0.98	10.48	3.50	43.87
12	0.98	7.15	4.61	44.91
16	0.96	6.37	5.92	44.64
24	0.89	4.13	5.78	46.44
32	0.80	3.18	6.16	46.73
40	0.82	2.97	6.27	46.38

5.5 Confocal Imaging in OCT

Scanning laser ophthalmoscopy (SLO), along with the OCT imaging, is widely used in ophthalmology. It has a typical axial resolution, along the orthogonal axis, of approximately 200 µm and is typically used to locate layers and tissues [82]. In some cases, it is generated along with en-face OCT images, as presented in [83] and [84].

This research presents a parallel approach to the construction of images corresponding to confocal SLO images. They are generated during en-face OCT imaging by averaging (integrating) multiple successive en-face images. In this thesis, these images are denoted as confocal.

The OCT frames used in this research cover a depth of 2.7 mm (2700 µm). Similar OCT frame is presented in [12]. Displaying this depth with 513 lines in the CFD method and with 512 lines in the MSI method, results in an axial resolution of approximately 5.3 µm. Therefore, a confocal image with approximate resolution of 200 µm would need the integration of approximately 38 en-face images.

Both the CFD and MSI OCT methods can produce confocal images based on multiple en-face images. The two methods follow different routes in the en-face imaging, and consequently, different routes to confocal imaging, as illustrated in Figure 5.13.

The generation of confocal images can be divided into three phases, as seen in Figure 5.13. The order of these phases depends on the OCT method. In the CFD OCT method, the phases are: en-face generation, selection of en-face images, and integration. In the MSI OCT method, the phases are selection of mask signals, en-face generation, and integration.

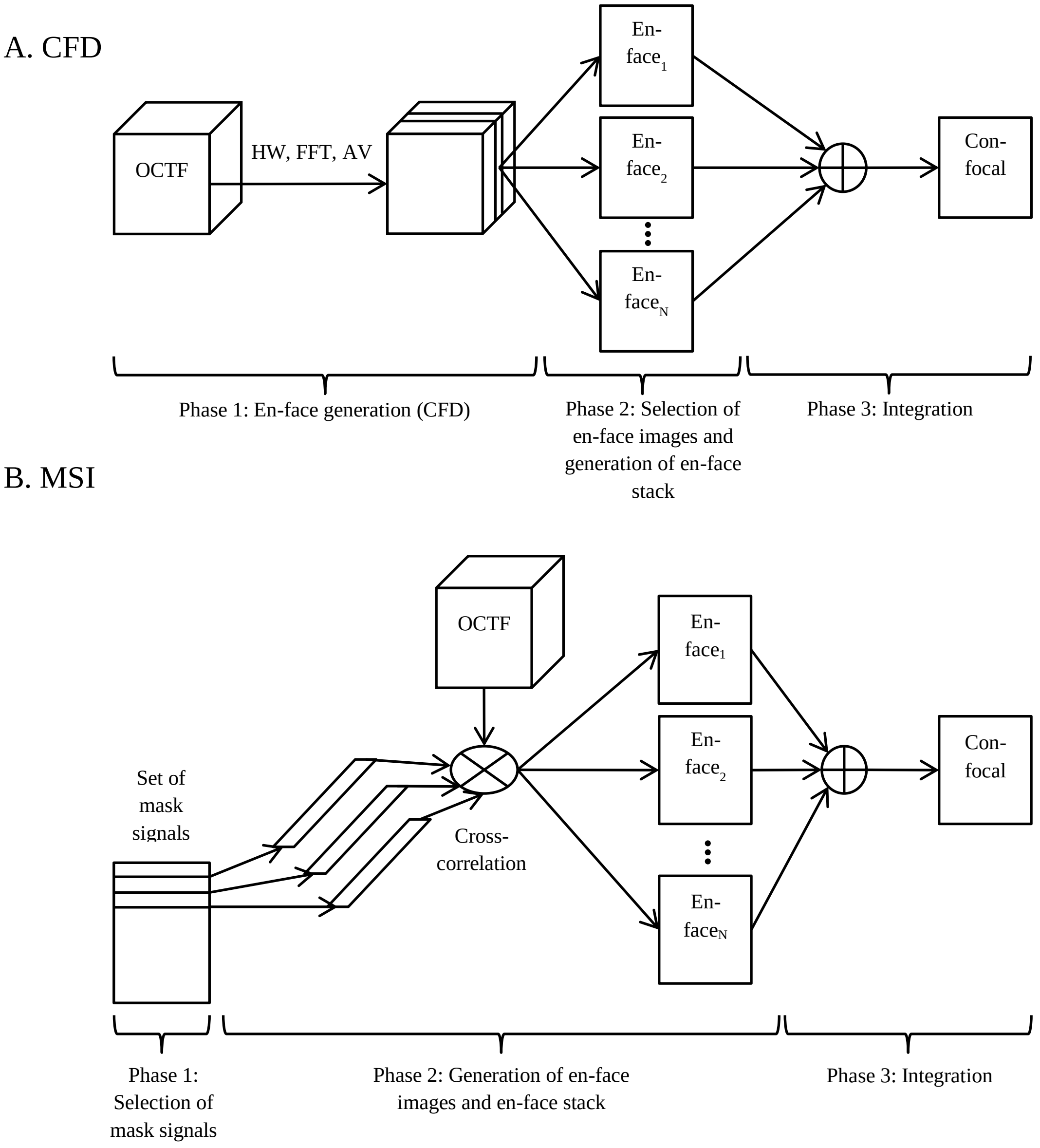


Figure 5.13: Phases in confocal imaging based on the CFD (A) and MSI (B) methods

The confocal images are built on consecutive en-face images. In the CFD OCT method, these en-face images are selected directly during Phase 2. In the MSI OCT method, the en-face images are selected indirectly by selecting corresponding masks from the set of mask signals during Phase 1. In both OCT methods, the selection of the en-face images is controlled by two variables: Start and Range. The first variable determines the first en-face image. The second one determines how many en-face images will take part in the processing.

5.5.1 Approaches to Integration

In both methods, the completion of the first two stages results in a set of en-face images, denoted as the en-face stack. The en-face stack can be considered as a three-dimensional array $EnFaceStack_{i,j,k}$, where:

- **i** is the index of the en-face image
- **j** is the X coordinate in the en-face image in the OCT frame, j = 0, ... , XSize-1
- **k** is the Y coordinate in the en-face image, k = 0, ... , YSize-1

In this case, the integration needs to be according the **i** index. The result will be a single confocal image with XSize×YSize points.

This organization of the en-face stack is followed by both OCT methods. As a result, the employed parallel approach does not depend on the OCT method. Figure 5.14 illustrates the en-face stack containing the selected en-face images. All approaches to the integration require the completion of the calculation of the en-face stack with the three-dimensional array $EnFaceStack_{i,j,k}$ holding the intensities of the selected en-face images.

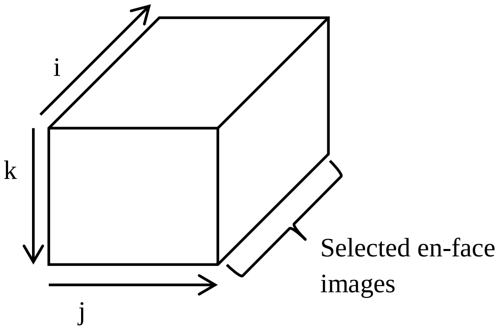


Figure 5.14: En-face stack

The confocal imaging is based on the en-face imaging discussed in the previous section of this chapter. The overall performance of the confocal imaging depends heavily on the performance of the en-face imaging. Considering the need to copy the OCT frame to the global GPU memory only once for both the en-face and the confocal imaging, and the superior performance of the fine-grained approach, the confocal imaging presented and studied here is built on the GPU-based fine-grained approach. Four GPU-based approaches are proposed, namely, a Sequential Iterative, Partially Parallel Iterative, Parallel Reduction, and Zero-Frequency Component (ZFC) The Sequential Iterative approach illustrates computations performed by a single thread. In order to avoid a possible loss of precision, the multiplication with the gray scale coefficient (GSC) is performed at the end of Phase 3 after the integration of all intensities of the selected en-face images.

5.5.1.1 Sequential Iterative

This approach is implemented with sequential iterations, as demonstrated in Algorithm 5.16. It is used as a baseline for the comparison with the other parallel approaches. The algorithm is carried by a single thread of execution on the GPU. Table 5.11 presents the performance of this approach with different number of en-face images.

Algorithm 5.16: Sequential Iterative approach to integration in confocal imaging

```
Kernel SequentialIterative
Input:   EnFaceStack
Output   Confocal
    for k = 0 to YSize-1 do
        for j = 0 to XSize-1 do
            Sum = 0
            for i = 0 to NumberOfEnFace-1 do
                $Sum = Sum + EnFaceStack_{i,j,k}$
            end for
            $Confocal_{j,k} = Sum \times GSC$
        end for
    end for
End Kernel
```

5.5.1.2 Partially Parallel Iterative

The sequential iterative approach can be partially parallelized on the GPU. The stack of selected en-face images has dimension of XSize×YSize×NumberOfEnFace. In this approach XSize×YSzie parallel threads are launched. Each parallel thread integrates sequentially the values from all images with the same en-face coordinates. Each thread accumulates the integrated value of the intensity of the corresponding points from the confocal image. There are three possible cases in this implementation:

1. The integration kernel launches XSize parallel threads grouped into YSize blocks:

IntegrationKernel<<< YSize,XSize>>> (EnFaceStack, Confocal)

This approach is demonstrated in Algorithm 5.17. Each point from the confocal image obtains its intensity sequentially, line 6 in Algorithm 5.17.

2. The integration kernel launches XSize×YSize thread blocks. Each thread block launches one thread:

IntegrationKernel <<<(Xsize× YSize), 1>>> (EnFaceStack, Confocal)

3. The integration kernel launches a two-dimensional grid IntegrationGrid of XSize by YSize thread blocks. Each block launches a single thread:

dim3 IntegrationGrid(XSize,YSzie)

Integration Kernel <<<IntegrationGrid, 1>>> (EnFaceStack, Confocal)

All three approaches yield virtually identical performances.

Algorithm 5.17: Partially Parallel Iterative approach to integration in confocal imaging

```
1: Kernel PartiallyParallelIterative
2: Input     EnFaceStack
3: Output    Confocal
4:     j = threadIdx.x // parallel threads map the X coordiante
5:     k = blockIdx.x // thread blocks map the Y coordinate
6:     Sum = 0
7:     for i = 0 to NumberOfEnFace-1 do
8:         Sum = Sum + $EnFaceStack_{i,j,k}$
9:     end for
10:    $Confocal_{j,k}$ = Sum×GSC
11: End Kernel
```

5.5.1.3 Parallel Reduction

This approach is based on the parallel reduction algorithm presented in Chapter 4, in [80] and [81] . The algorithm in Chapter 4 integrates digital signal with 8 data point. To integrate a higher number of data points would require the addition of if-then operators in the kernel involving the additional values. The rest of the algorithm remains unchanged. Its improved performance is noted in Table 5.14.

5.5.1.4 Zero-Frequency Component

This approach is based on the property of the discrete Fourier transform, discussed in Chapters 3 and 4. The summation of all data points from a digital signal in time domain is hold in the zero-frequency component of the same signal in Fourier domain.

Algorithm 5.18: ZFC approach to integration in confocal imaging

```
1: $EnFaceStack'_{i,j,k} = FFT_{j×k}$ ( $EnFaceStack_{i,j,k}$ ) /cufftExecR2C, batch of XSize×YSize 1D transforms/
2: $Confocal_{j,k}$ = ( AV ( $EnFaceStack'_{0,j,k}$ ) )×GSC /kernel with XSize×YSize threads/
```

The performance of this approach depends on the performance of the employed FFT on a signal with the size of XSize×YSize×NumberOfEnFace. The number of transforms is equal to XSize×YSize. A confocal image based on 64 to 512 en-face images requires processing of a digital signal with the size of 2457600 (200×192×64) to 19660800 (200×192×512) data points. The size of the signal and the performance of the FFT presented in Table 4.2 in Chapter 4 make the GPU implementation the approach of choice for this case. This performance is reported in Table 5.14.

5.5.2 Performance and Results of Confocal Imaging in OCT

The performance of the confocal imaging is divided into two components:

1. The performance of the generation of the selected en-face images. This performance depends on the selected OCT method. In the MSI method, this performance

depends on the number of selected en-face images

2. The performance of the integration stage

Table 5.14 present the performance of confocal imaging based on different numbers of en-face images.

Table 5.14: Performance of GPU-based fine-grained confocal OCT imaging, in milliseconds

Number of en-face images	En-face generation (CFD)	En-face generation (MSI)	Sequential Iterative	Partially Parallel	Parallel Reduction	ZFC
64[A]	7.6	350	754.3	1.4	1.2	0.26
128	7.6	694	1476.1	2.6	1.2	0.48
256	7.6	1380	3012.8	5.2	6.6	0.92
512[B]	7.6	2765	6082.5	8.3	6.8	1.74

[A] Figures 5.15 and 5.17 present the confocal images generated by averaging 64 en-face images

[B] Figures 5.16 and 5.18 present the confocal images generated by integrating 512 en-face images. The level of the noise in the first en-face images, or the first lines in the corresponding cross-sectional image, is high. Therefore, the intensities from these images are set to zero

As expected, a confocal image generated by a single GPU thread is outperformed considerably by all other approaches, Table 5.14. This highlights the affinity of the GPU towards parallel computations.

The overall performance of the confocal imaging is equal to the summation of the en-face generation, the integration, and the time to copy the data to and from the GPU memory. All combinations between the two OCT methods and the four approaches to integrations are possible. Table 5.15 illustrates two cases of confocal imaging. The first case is based on 512 en-face images generated using the CFD method and integrated by the ZFC approach. The second case is based on 64 en-face images generated using the MSI method and integrated by employing the parallel reduction. The real-time criterion for confocal imaging is the same as for en-face imaging, which is 800 milliseconds. Both examples in Table 5.15 meet the real-time criterion.

Table 5.15: Performance of two approaches to OCT confocal imaging, in milliseconds

Number of en-face images	OCT method	Approach to integration	Performance in milliseconds				
			CPU-GPU copy	En-face generation	Integration	GPU-CPU copy	Overall
64[A]	MSI	Parallel Reduction	101.52[C]	350	1.2	0.09[D]	452.81
512[B]	CFD	ZFC	101.52[C]	7.6	1.74	0.09[D]	111.95

[A] Confocal image generated by integrating 64 en-face images are presented in Figures 5.15 and 5.17

[B] Confocal images generated by integrating 512 en-face images are presented in Figures 5.16 and 5.18

[C] Time to copy one OCT frame of 200×192×1024 data points, as presented in Table 5.3

[D] Time to copy one en-face (confocal) image as presented in Table 5.4

Along with the confocal image, a cross-sectional image generated from the same digital signal is visualized, as seen in Figures 5.15, 5.16, 5.17 and 5.18. The cross-sectional image provides guiding information about the axial position of the selected en-face images. The OCT frame acquired during en-face imaging basically consists of multiple horizontal cross-sectional OCT frames, in this case 192 cross-sectional OCT frames of 200×1024 data points each. Providing the index of the horizontal cross-sectional image allows the calculation of the start index of the horizontal cross-sectional OCT frame and its extraction from the en-face OCT frame. Also, the en-face and the cross-sectional images share the same Hamming window and FFT applied on the en-face OCT frame. Therefore, the extraction of the cross-sectional image is done after these steps on an OCT frame in Fourier domain in the global GPU memory. The following call to cudaMemcpy extracts the cross-sectional OCT frame from the en-face one:

```
cudaMemcpy(CSOCTF,&(EFOCTF[Mp*N*Index] ),Mp*N*sizeof(cufftComplex),cudaMemcpyDeviceToDevice)
```

Where:

Mp (M') is the size of the channeled spectrum in Fourier domain (Mp = 513)

N is the number of channeled spectra per line (N = 200)

Index is index of the selected cross-sectional image (Index = [0..191])

EFOCTF is the en-face OCT frame in the GPU memory

CSOCTF is the cross-sectional OCT frame in the GPU memory

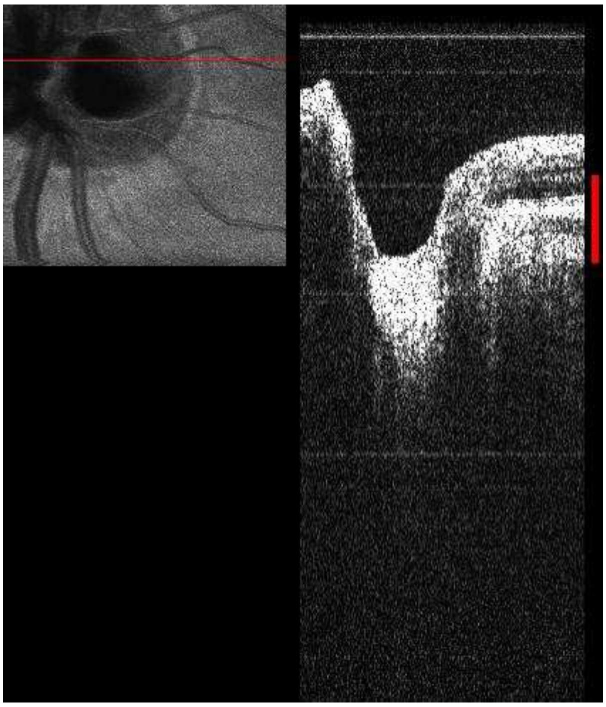


Figure 5.15: Confocal image of human eye generated by averaging 64 en-face images

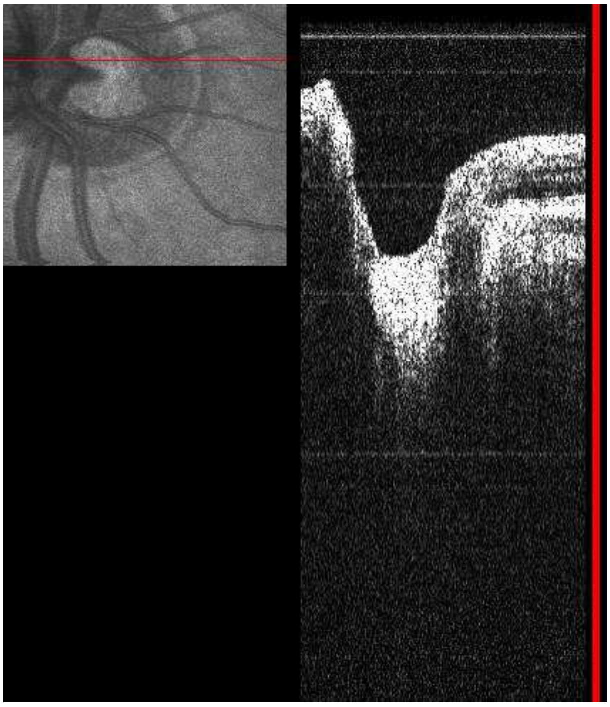


Figure 5.16: Confocal image of human eye generated by averaging 512 en-face images

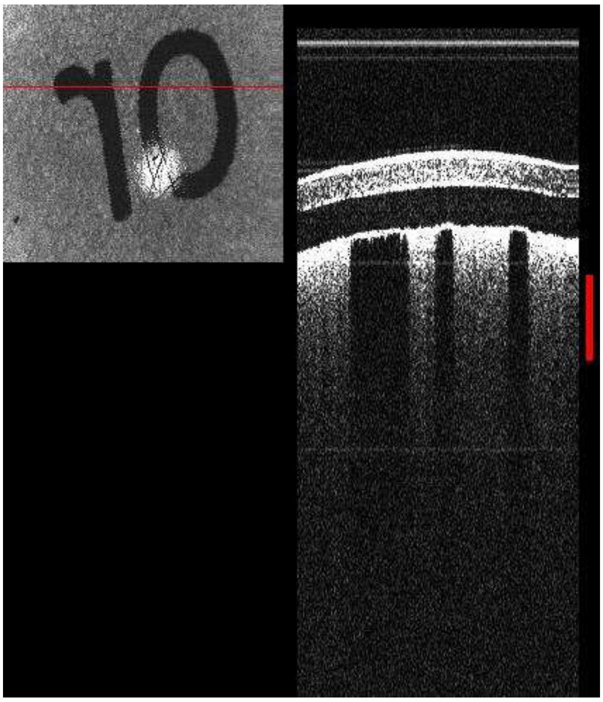


Figure 5.17: Confocal image of laminated paper generated by averaging 64 en-face images

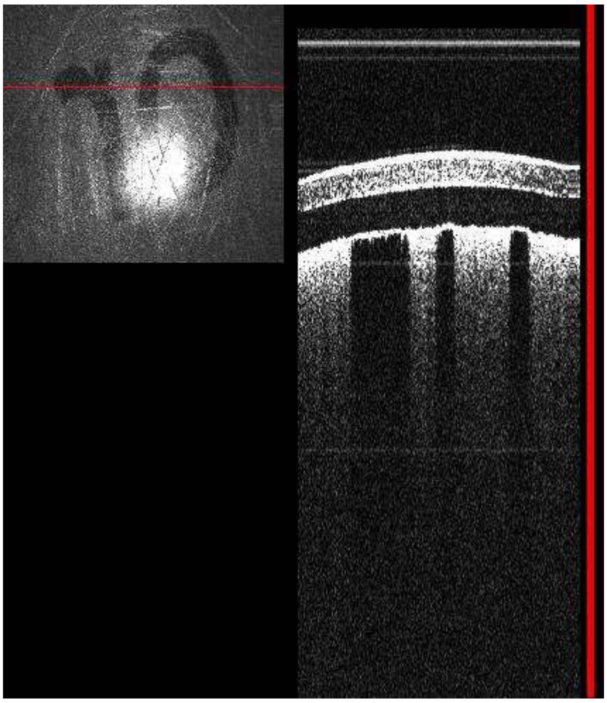


Figure 5.18: Confocal image of laminated paper generated by averaging 512 en-face images

Digital signals from two OCT frames are used. Figures 5.15 and 5.16 visualize the back of a human eye. Figures 5.17 and 5.18 present a laminated paper. A horizontal line across the confocal image denotes the position of the cross-sectional image. A vertical bar on the right denotes the region from which the en-face images are used in the confocal. Approaches to the GPU-based parallel optimizations of generating confocal images are reported in [85].

The performances presented in Tables 5.14 and 5.15 include only the time to generate the confocal image. The latency of the cross-sectional image is not included. Chapter 6 presents an integrated solution where the performances of all images, en-face, confocal and cross-sectional, are included.

5.6 Summary

This chapter presented coarse-grained and fine-grained parallel optimizations of the three principle OCT imaging, namely cross-sectional, en-face and confocal. Two OCT methods are employed, the CFD and the MSI. The MSI method, being computationally more intensive than the CFD, requires more processing time, which is especially visible in the sequential and coarse-grained implementations. The fine-grained implementations manage to absorb the increased computations and to deliver improved performance in all cases presented in this chapter. The fine-grained approach has the ability to deliver real-time operation where the coarse-grained one cannot meet this criterion.

A simultaneous processing and generation of cross-sectional images generated by the two OCT methods allows further studies on the differences and similarities between these two methods, in terms of image quality, signal-to-noise ratio and other properties.

The sequential implementation of the cross-sectional CFD OCT method has a latency of 7.2 milliseconds for image with width of 500 points. The corresponding sequential implementation of the MSI OCT method has a latency of 2372 milliseconds. It takes more than 2 seconds longer to generate MSI cross-sectional image, Table 5.6. Considering the data rate of the OCT system, which is a new OCT frame every 10 milliseconds, this difference in the latency would render this comparison ineffective.

In the GPU-based case, the latency of the CFD OCT method generating the same cross-sectional image is 0.198 milliseconds and in the MSI OCT method is 35.6 milliseconds. This amounts to a much smaller difference of approximately 35 milliseconds. This approach allows the two OCT methods to have comparable latencies and as a result to be generated from input signals generated with much smaller time difference.

Chapter 6

System Integration of Parallel Solutions

6.1 Introduction

In most cases in practice, DSP algorithms in general and OCT methods in particular are not developed for independent utilization, to be used on their own. Usually, they are part of a real-time system. This chapter presents the integration of a comprehensive GPU-based implementation of the MSI OCT method into an OCT system performing en-face imaging. This implementation provides simultaneous en-face, cross-sectional and confocal imaging. This inclusion of different types of images provides comprehensive information about the imaged objects and better understanding of the position and the shape of the features within these objects.

The fine-grained approach is selected for this implementation, based on the performance of the GPU based signal processing, as presented in Chapters 4 and 5.

The choice of OCT method employed in this implementation is based on novelty and complexity. Therefore, the newer and more complex MSI OCT method, introduced in 2013 in [4] is preferred over the CFD OCT method. Its improvements in OCT are presented in [72] and [73]. Based on cross-correlation, this method significantly increases the amount of computations necessary to generate en-face, cross-sectional and confocal OCT images.

The requirements facing the integration of a GPU-based implementation does not differ from those in the case of a CPU-based implementation. The two principle requirements for integrating external solutions are:

1. Ability to exchange data with the rest of the system. In this case the data is in the

form of a digital signal generated by the OCT system at a certain rate. Besides the digital signal, the external solution needs to read a set of control values which acts as parameters in the signal processing, such as cut-off frequency, window boundaries and others.

2. Ability to operate in a speed comparable with the speed of the system. In this particular case, this requirement is more strict. It requires an improved performance, which is the main motivation for the inclusion of external GPU-based solution into an OCT system. In other words, the speed of the GPU-based processing needs to be higher than the speed of otherwise implemented processing, for example as part of the data acquisition software, which in this case is a LabVIEW project.

6.2 Data Acquisition in OCT Systems

Data acquisition in the OCT, as in other real-time systems, involves hardware and software solutions. The purpose of these solutions is to generate numerical (digital) equivalent to specific physical properties measured over a period of time. This numerical equivalent, in the form of a digital signal, needs to be made available for further processing and visualizing.

Key components of the data acquisition process in the OCT system studied in this research are a digitizer board (AlazarTech expansion board), and a LabVIEW VI project, acting as a data acquisition software. The digitizer provides an extensive and well published software support for LabVIEW implementations [86]. In this case, the data acquisition software has three principle tasks:

- To control the various component of the OCT system, such as the optical source, the scanning mirrors and the photo detector

- To capture the digital signal generated by the digitizer board and to make it available for further processing, analysis and visualization, either within the LabVIEW project, or by external software, such as a dynamic link library or an application.

- To provide a user interface allowing the operator of the OCT system to control the processing

6.3 Integrating GPU Solution in OCT Systems

An external CUDA C or C++ code can access the digital signal generated by the LabVIEW VI via a Dynamic Link Library (DLL), which can be loaded by the LabVIEW VI in the form of Call Library Function block. Based on this, two approaches are possible:

- The CUDA C code is encapsulated within the DLL called by the VI
- The CUDA C code is implemented as a stand-alone application. This approach still needs a C++ function compiled in a DLL in order to exchange the digital signal with the LabVIEW VI.

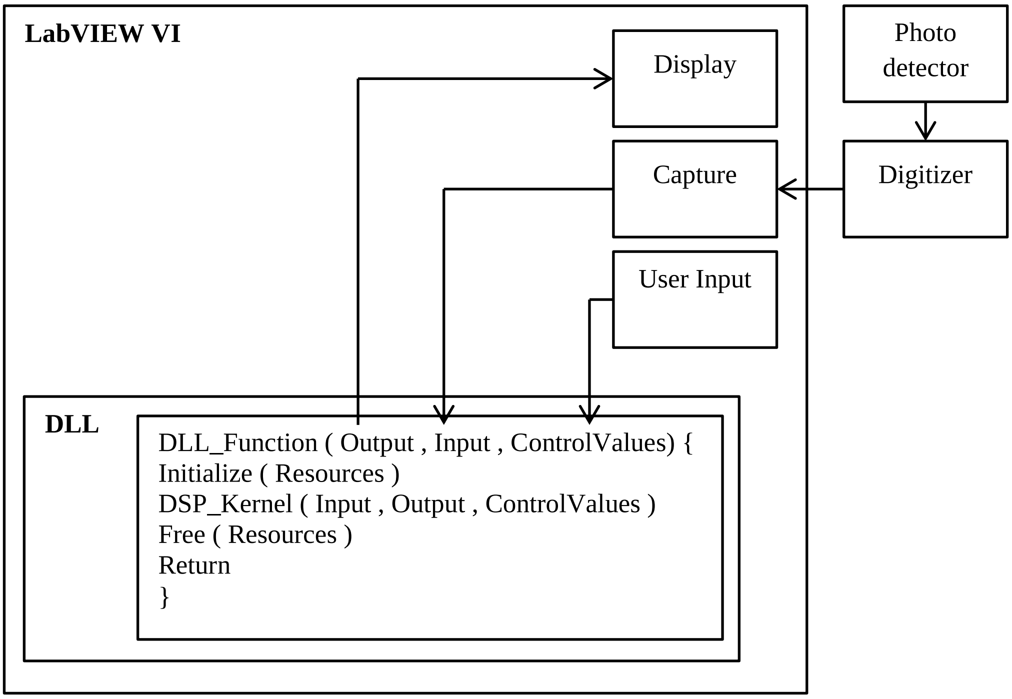
6.3.1 DLL Approach

In this approach, the CUDA C kernels are compiled to a dynamic link library [87]. This approach is used in [76], where real-time resampling is implemented on the GPU, encapsulated as a DLL library and integrated into a LabVIEW Virtual Instrument. This approach has a significant drawback in the performance. Every time a new OCT frame is generated, the LabVIEW VI calls the kernels from the DLL. Every time a kernel is called, all initializing steps must be performed. These include:

- Allocating memory for the variables, in the CUDA C case, memory is allocated both on the CPU and the GPU
- Creating FFT and IFFT plans
- Creating GPU context
- Processing the mask signals employed in the MSI OCT method

At completion, the DLL function needs to free the allocated resources. Figure 6.1 illustrates this case.

As seen in Figure 6.1, every call of the CUDA C kernel, involve the aforementioned initializing procedures. These procedures bring additional overheads to the overall latency of the processing, as discussed in [88]. A stand-alone application, presented in the next section, provides a solution to this problem.

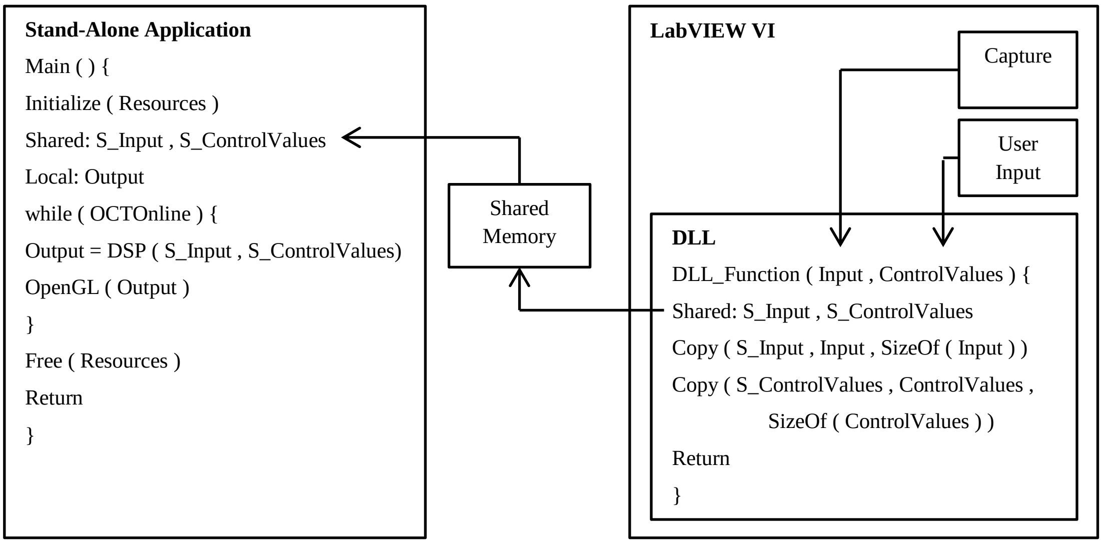


```
DLL_Function ( Output , Input , ControlValues) {
Initialize ( Resources )
DSP_Kernel ( Input , Output , ControlValues )
Free ( Resources )
Return
}
```

Figure 6.1: DLL approach to GPU OCT integration

6.3.2 Stand-Alone Application Approach

In this approach, the CUDA C code is compiled as a stand-alone application. Presently, LabVIEW does not have the functionality to access a shared memory managed by Microsoft Windows. Therefore, this approach still needs a C++ function compiled as a DLL, which allows the VabVIEW VI to access the shared memory. Figure 6.2 illustrates this approach.



```
Main ( ) {
Initialize ( Resources )
Shared: S_Input , S_ControlValues
Local: Output
while ( OCTOnline ) {
Output = DSP ( S_Input , S_ControlValues)
OpenGL ( Output )
}
Free ( Resources )
Return
}
```

```
DLL_Function ( Input , ControlValues ) {
Shared: S_Input , S_ControlValues
Copy ( S_Input , Input , SizeOf ( Input ) )
Copy ( S_ControlValues , ControlValues ,
            SizeOf ( ControlValues ) )
Return
}
```

Figure 6.2: Stand-alone approach to GPU OCT integration

This approach eliminates the need to repeatedly initialize and free the necessary resources, and hence improves the overall performance. Its implementation and integration into OCT system is reported in [88].

6.3.3 Shared Memory

The stand-alone application approach requires a shared memory, used by the DLL and the stand-alone application. Microsoft Windows OS provides a mechanism implementing a shared memory. This is done by the API CreateFileMapping OpenFileMapping and MapViewOfFile.

The shared memory is created by calling CreateFileMapping. This can be done by either the stand alone application, or by the DLL. The difference is that the code in the DLL will be called with the same frequency as the rate of the OCT frames. This leads to creating the shared memory multiple times. The stand-alone application, on the other hand, will create the shared memory once at the beginning, along the other initializing procedures. This approach is chosen for the implementation described in this chapter.

After the creation of the shared memory, the DLL function can access it by calling OpenFileMapping. This call must be supplied with the same name of the shared memory used when it was created.

Once the shared memory is accessible from both the application and the DLL, both parties can associate the shared memory with a local variable, by calling the MapViewOfFile API.

The shared memory is used for two entities: the digital signal generated by the OCT system and a set of control values selected by the user of the system and entered via the user interface in real-time.

6.3.3.1 Digital Signal

The proposed comprehensive GPU-based solution is designed for en-face imaging of the OCT system. In this case, the OCT system generates en-face OCT frames at the rate of 1.25 Hz, i.e. new OCT frame is generated every 800 milliseconds. The en-face OCT frame ($EFOCTF_{i,j,k}$) consists of 1024×200×192 data points generating by scanning an area of 200×192 points.

6.3.3.2 Control Values

A number of signal processing steps require some input from the operator of the system. The integrated solution presented in this chapter employs the following control values:

Signal Processing:

1. Cut-off frequency
2. Window type
3. Window boundaries (W_1 and W_2 used by the integration)

Display:

Index of the horizontal cross-sectional image
Index of the vertical cross-sectional image
Index of the first en-face image (Start)
Step between successive en-face images (Step)
Gray scale coefficient (GSC) for the en-face and cross-sectional images
Contrast of the cross-sectional, en-face and confocal images
Brightness of the cross-sectional, en-face and confocal images

Mode:

Online, Save to memory, Post Processing, Save to File

These control values are organized as an array. The values of this array are set by the numerical controls used in the LabVIEW VI.

6.4 Comprehensive Imaging in OCT

As reported in [72], a sequential LabVIEW-based implementation of the MSI OCT method generates four en-face images in real-time. In some area in practice, for example ophthalmology, a larger number of en-face images, combined with confocal and cross-sectional images, increases the chance of a biological feature to be detected and observed, and therefore improves the process of diagnosis. In comparison, the proposed comprehensive GPU-based implementation of the MSI method generates 40 en-face, 1 confocal, and 2 cross-sectional images, horizontal and vertical, in real-time. The proposed solution is successfully integrated into an OCT system. It demonstrates the capabilities of the GPU processing to absorb increased amount of computations within the real-time criterion.

6.4.1 Horizontal Cross-Sectional Image

The horizontal cross-sectional OCT frame (HCSOSTF$_{i,j}$) is extracted from the en-face OCT frame (EFOCTF$_{i,j,k}$) in the same way as presented in Chapter 5 and as seen in Figure 6.3. The following call to cudaMemcpy extracts the horizontal cross-sectional OCT frame from the en-face OCT frame:

```
cudaMemcpy(HCSOCTF,&(EFOCTF[Mp*N*Index_h] ),Mp*N*sizeof(cufftComplex),cudaMemcpyDeviceToDevice)
```

Where: Mp (M') is the size of the channeled spectrum in Fourier domain; N is the number of channeled spectra per line; Index_h is index of the horizontal cross-sectional image; EFOCTF is the en-face OCT frame; HCSOCTF is the horizontal cross-sectional OCT frame

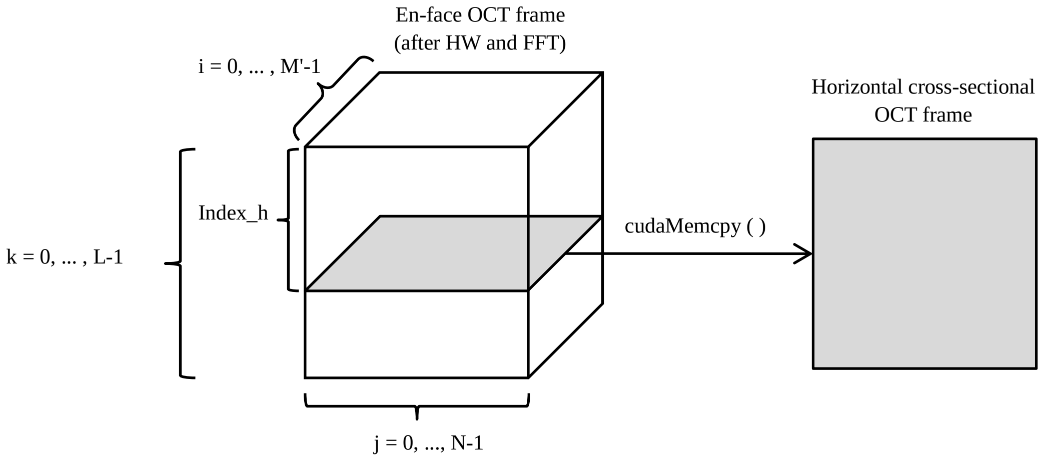


Figure 6.3: Extracting a selected horizontal cross-sectional OCT frame from the en-face OCT frame. The horizontal cross-sectional OCT image is determined by the value of Index_h

6.4.2 Vertical Cross-Sectional Image

The vertical cross-sectional OCT frame, (VCSOCTF$_{i,k}$) is chosen independently from the horizontal one by selecting the value of Index_v. It is extracted after transposing the en-face OCT frame, as seen in Figure 6.4.

Both en-face OCT frames are in the global GPU memory, which allows the transpose to be implemented as a kernel. After transposing the en-face OCT frame, the following cudaMemcpy extracts the selected vertical cross-sectional OCT frame:

cudaMemcpy(VCSOCTF,&(EFOCTF[Mp*L*Index_v]),Mp*L*sizeof(cufftComplex),cudaMemcpyDeviceToDevice)

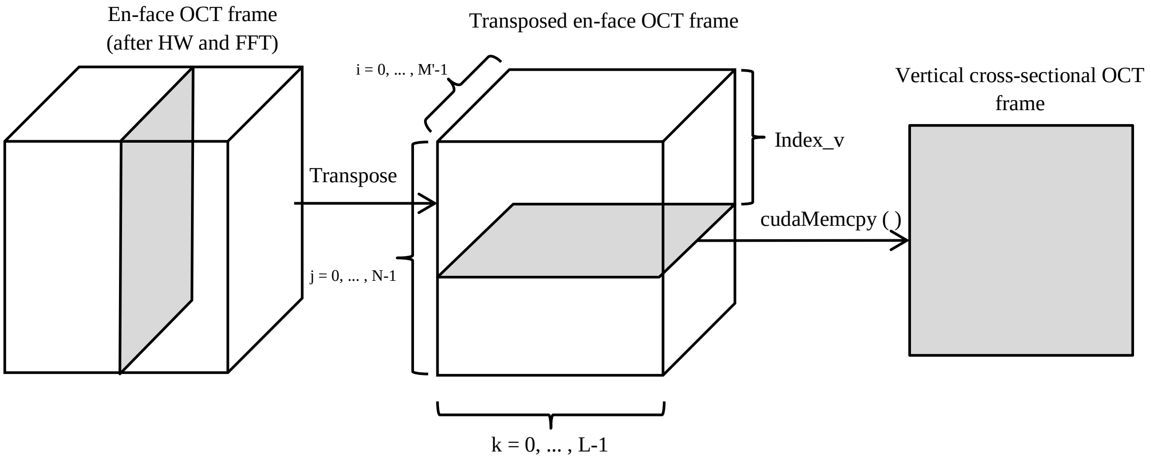


Figure 6.4: Extracting vertical cross-sectional OCT frame from the transposed en-face OCT frame

6.4.3 Confocal Image

The confocal image included in the proposed comprehensive imaging, top right in Figures 6.7 and 6.8, is based on the already calculated intensities of the selected en-face images. Thus, the confocal image presented in this solution depends on the selection of the index of the first en-face image (Start) and the step between successive images (Step). An independent confocal image, generated on an independently selected set of en-face images as presented in Chapter 5, would consume significant time, as seen in Table 5.15, and would prevent the real-time operation. The confocal image presented in this solution would meet the resolution requirements as discussed in Chapter 5, if the step between successive en-face images is 1. Otherwise, it would provide indicative information of the sample volume between the first and last en-face images.

The confocal image is generated by using the parallel reduction algorithm, discussed in Chapter 5. The number of images in this algorithm is expected to be a power of two. The proposed imaging generated 40 images. In this case, the 40-image en-face stack is complemented to 64 by zero padding, as illustrated in Figure 6.5.

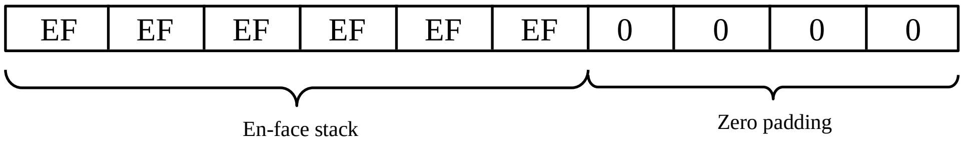


Figure 6.5: Zero padding in confocal imaging

6.4.4 Overall Performance of the GPU-Enabled OCT System

To generate images, the proposed GPU-enabled OCT system performs the following steps, as shown in Figure 6.6:

1. Acquisition of the OCT frame, which is 800 milliseconds. This latency is defined by the sweeping capabilities of the OCT system [12].

2. The DLL call by the LabVIEW VI using the Call Library Function module. The DLL copies the OCT frame from the LabVIEW VI's local memory to the shared memory in approximately 400 milliseconds [12].

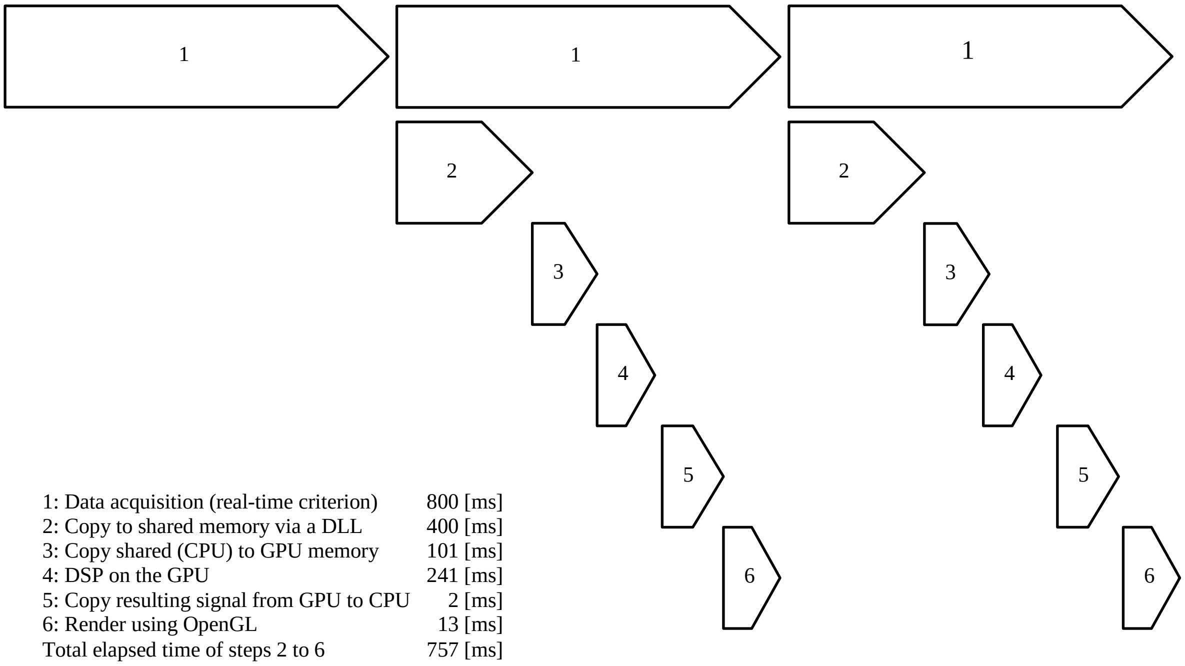
3. The copy of the OCT frame from the shared memory (CPU) to the global GPU memory. This time is averaged to 101.52 milliseconds, as presented in Chapter 4, Table 4.3.

4. The DSP of the OCT frame generating 40 en-face, 1 confocal, and 2 cross-sectional images. This time is approximately 241 milliseconds. As reported in Chapter 5, 40 MSI OCT en-face images are processed in 236.2 milliseconds, one cross-sectional image is processed in 13.8 milliseconds and the confocal image is processed in 1.2 milliseconds, which amounts to 256 (236.2+2×13.8+1.2) milliseconds. Two factors contribute in this case. Firstly, the en-face and the cross-sectional images share the same Hamming window and FFT. Secondly, the overall measurement of the elapsed time of a number of kernels is smaller than the summation of the individual elapsed times of each kernel, as seen in Equation 2.1. This is due to the asynchronous launch of the CUDA kernels.

5. The transfer of the resulting signal consisting of gray scale values, from the global GPU memory to the CPU memory, which is approximately 2 milliseconds. This is done by using the cudaMemcpy function. The function is called four times for the four arrays holding the 40 en-face images, the horizontal cross-sectional image, the vertical cross-sectional image, and the confocal image. This function, unlike the aforementioned kernels, operates asynchronously [28].

6. The display the images using OpenGL [89]. This is complete in approximately 13 milliseconds.

The total time from the signal formation to the image generation is 757 milliseconds, which is within the real-time criterion.



1: Data acquisition (real-time criterion)	800 [ms]
2: Copy to shared memory via a DLL	400 [ms]
3: Copy shared (CPU) to GPU memory	101 [ms]
4: DSP on the GPU	241 [ms]
5: Copy resulting signal from GPU to CPU	2 [ms]
6: Render using OpenGL	13 [ms]
Total elapsed time of steps 2 to 6	757 [ms]

Figure 6.6: Real-time criterion and performance of the GPU-enabled OCT system

6.4.5 OpenGL Display

Figures 6.7 and 6.8 present the images generated by the OpenGL, consisting of 40 en-face, 1 confocal, and two cross-sectional images. The employed OpenGL is part of the Microsoft Visual Studio API.

To improve the layout of the screen, the OpenGL API functions display the cross-sectional images with height equal to twice the height of the en-face images, which is 384, instead of the 512 as demonstrated in Chapter 5. This change does not affect the performance.

Figure 6.7 presents the back of the human eye of a volunteer. Figure 6.8 presents a laminated paper. Both images are acquired at the Applied Optics Group at the School of Physical Sciences at the University of Kent.

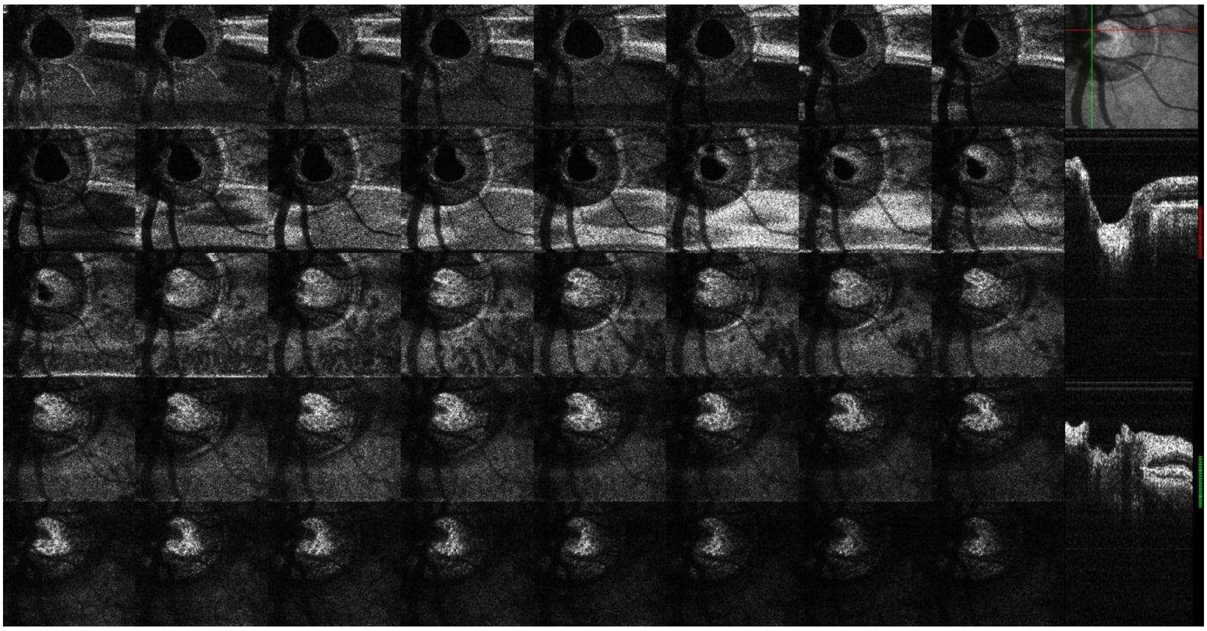


Figure 6.7: OpenGL window of the comprehensive OCT imagery generated by signal processing on the GPU displaying 40 en-face, 1 confocal and 2 cross-sectional images of a human eye

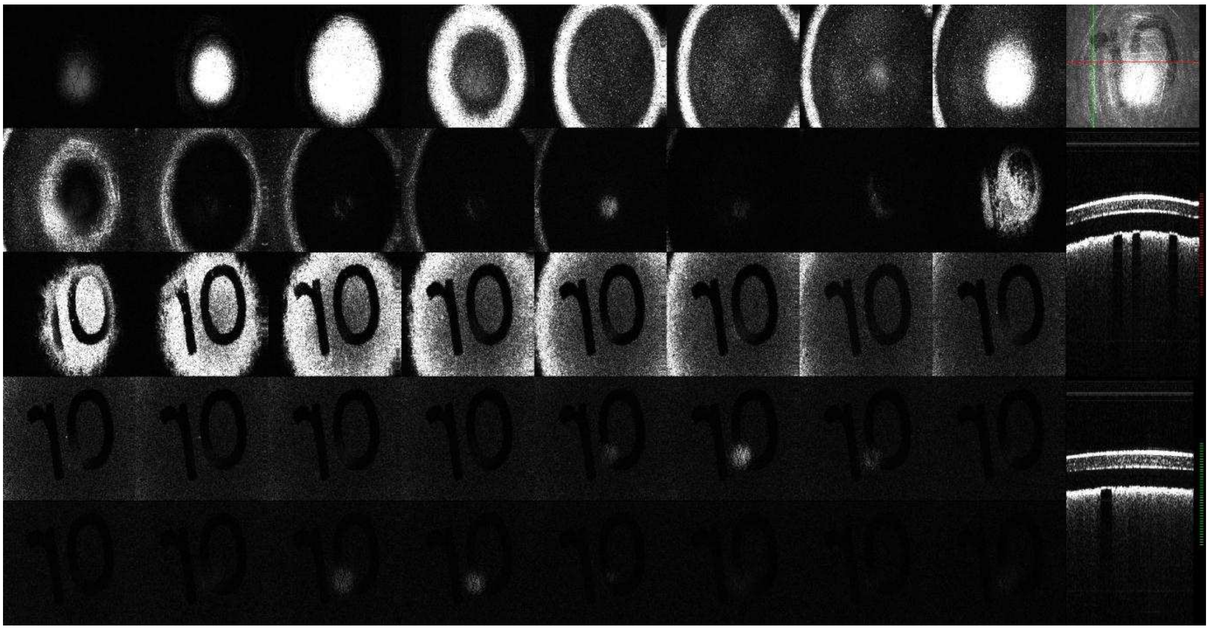
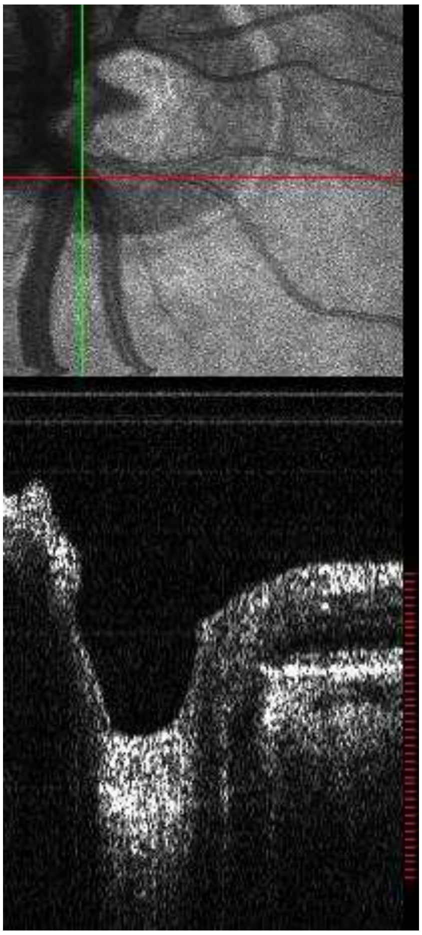


Figure 6.8: OpenGL window of the comprehensive OCT imagery displaying 40 en-face, 1 confocal and 2 cross-sectional images of laminated paper

6.4.6 Additional Guiding Information

The integrated GPU solution provides additional information about the displayed cross-sectional and en-face images. This additional information, demonstrated in Figure 6.9, allows a better understanding of the positions of the visualized images within the en-face OCT frame.



1. Horizontal red line representing the index of the horizontal cross sectional image
2. Vertical green line corresponding to the index of the vertical cross-sectional image
3. Multiple lines on the right side of the cross-sectional image, corresponding to the position of the selected en-face images as seen in Figures 6.7 and 6.8. The distance between the top of the cross-sectional image and the first line is controlled by the Start value. The distance between successive lines is controlled by the Step value. Both Start and Step values are used in en-face imaging as described in Chapter 5.

Figure 6.9: Additional Guiding Information

6.4.7 Modes of Operation

The presented solution operates in the following four modes: Online, Save to Memory, Post Processing and Save to File, illustrated by Figure 6.10. These modes allow saving and visualizing OCT frames for further examination and analysis [12].

1. Online. In this mode the GPU solution processes OCT frames as they are generated by the OCT system. This is the initial and default mode of operation.

2. Save to Memory. The GPU solution can switch to this mode only from Online Mode. In this mode, the current OCT frame is save to a separate variable in the global GPU memory. This variable is used only in post processing and Save to File mode. The save is done by calling the cudaMemcpy function. The call is done only once. After the copy is complete, the integrated GPU solution continues in Online mode.

3. Post Processing. The GPU solution can switch to this mode only from the Online mode, but will stay in this mode until the operator of the system select either Online mode or Save to File. In this mode, the Integrated GPU solution reads the Post Processing OCT frame. This OCT frame does not change, unless the solution is set to Online mode and then to Save to Memory mode.

4. Save to File. This mode can be selected only if the system is in Post Processing mode. There are two possibilities: to save the OCT frame as it is generated by the OCT system, or the save the gray scale values of the cross-sectional and en-face images, generate as a result of the GPU-based signal processing. At completion, this mode switches back to Post Processing mode. In both cases, the format of the file is tab-delimited. There are a number of solutions, which are capable of visualizing images from files containing gray scale values, including OpenGL implementations.

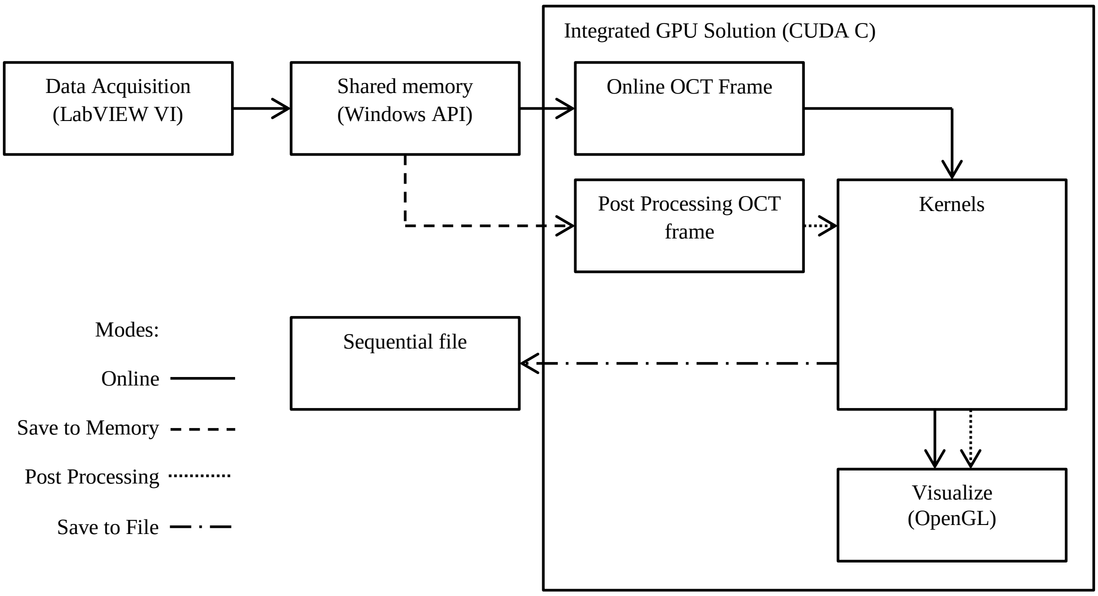

Integrated GPU Solution (CUDA C)
Data Acquisition (LabVIEW VI)
Shared memory (Windows API)
Online OCT Frame
Post Processing OCT frame
Kernels
Sequential file
Visualize (OpenGL)
Modes:
Online
Save to Memory
Post Processing
Save to File

Figure 6.10: Modes of operation of the GPU-enabled OCT system

The Save to Memory mode combined with Post Processing allows capturing an OCT frame at any time and immediately studying and analyzing the cross-sectional and en-face images. The Save to File mode allows permanent storage of the OCT frame or the images, allowing the creation of a data base. A possible utilization of this feature is in the area of ophthalmology. A number of ophthalmic OCT images of a patient's eye can be saved over a period of time and used to monitor a treatment.

6.5 Summary

The proposed integrated GPU solution provides comprehensive information about the volumetric data collected by an OCT system. The high number of en-face images is currently limited by the real-time criterion of the system and the resolution of most standard monitors.

The proposed integrated parallel solution allows a change in the real-time criterion. This may occur, if the speed of the data acquisition changes, as newer devices tend to provide higher speed. A solution to this problem is the reduction of the number of en-face images, which is currently 40. On the other hand, due to the scalability of the proposed CUDA C solution, its performance is expected to improve on next generation GPU architecture, such as the recently introduced NVIDIA Pascal GPU.

Chapter 7

Conclusions and Future Work

The research presented in this thesis is focused on parallel optimization of DSP algorithms and OCT methods. The parallelization is based on coarse-grained and fine-grained multithreaded application, designed for multi-core and many-core architectures. The parallelized computations involve signal processing algorithms applied on synthetically generated or experimentally acquired digital signals.

Chapter 4 proposed a number of parallel optimizations, based on well-known algorithms, such as cross-correlation and convolution. These optimizations were not implemented as closed solutions, but with the intention to be expanded towards the development of OCT methods. Chapter 5 presented the parallel optimization of these methods. The overall aim was to improve the output imagery of the OCT system and to meet the real-time criterion in the system when used in ophthalmology. This research resulted in the collaborative publication [90].

The performance of the proposed parallel solutions has been presented in terms of latency and speed-up and the optimal approaches have been identified. As a result of the proposed parallel optimizations, a comprehensive OCT imagery has been successfully developed and integrated into working OCT system, as presented in Chapter 6. As reported in [72], the corresponding sequential implementation provided more limited information in real-time.

The proposed parallel solutions process a number of experimentally acquired OCT frames, collected while imaging a laminated paper and a human eye of a volunteer. The resulting images were presented in Chapters 5 and 6. As reported in [12], [13], and [52], these images were result of increased computations, due to the size of the processed signals and the utilization of a more complex OCT method. Nevertheless, the parallel approaches were able to absorb the increased computations and perform the processing within the real-time criterion of the OCT system.

A number of reports on performance, in terms of latency and speed-up, provide information for a balanced choice between performance and OCT method and between performance and size of digital signal, which in many cases is linked to image quality.

The proposed parallel approaches, along with providing solutions to a range of signal processing problems, illustrate the computational characteristics, both sequential and parallel, of the utilized architectures, language specifications, and libraries. To reach an improved performance, computer programs need to reflect these characteristics.

7.1 Future Work

7.1.1 Parallel Architectures

This research involved the multi-core and many-core architectures of the CPU and the GPU. The performance of the many-core GPU architecture was reported on NVIDIA Tesla GPU. A direct competitor of the NVIDIA Tesla GPU is the Intel Xeon Phi, a parallel co-processor based on the Intel's Many Integrated Cores (MIC) architecture. Like the Tesla GPU, this co-processor extended the parallel capabilities to the CPU.

The introduction of the Intel Xeon Phi indicates the persistence of the parallel programming model. It also indicates a trend, which may lead to a possible merge of the multi-core and many-core parallel technologies. This is highlighted by the introduction of its "bootable host" version, designed to reduce the bottleneck of data transfer [91]. A viable future work is the optimization of the proposed FFT-based DSP algorithms and OCT methods for the Intel Xeon Phi and their comparison in terms of performance with the already developed coarse-grained and fine-grained approaches. Table 7.1 illustrates the similarities and differences between these two co-processors.

Table 7.1: NVIDIA-Intel parallel co-processor comparison

Vendor	NVIDIA	Intel
Product name	NVIDIA Tesla	Xeon Phi[A]
Interface	PCI Express	PCI Express
Number of cores	2880 (Tesla K40)	61[B]
Compiler/Native language	NVCC/CUDA C	Intel C++ Compiler/C++
Cross platform	Yes	Yes
Programming support	CUDA	Intel Parallel Studio[C]
FFT Library	cuFFT	Included in Intel MKL

[A] Implemented both as co-processor and bootable host processor

[B] Intel Xeon Phi 7110P [92]

[C] Includes support for OpenMP

A computational feature presented in the GPU, streams, was not utilized in the proposed fine-grained approaches. The implementation of multiple streams introduces additional level of parallelism, which is between kernels belonging to different streams. It does not have an equivalent in the API-based multithreading, which targets the multi-core CPU. Besides the asynchronous, and possibly simultaneous, execution of data transfer and GPU computations, the introduction of multiple streams is expected to be beneficial when two OCT methods are implemented to perform simultaneously. This could find application in the comparison in real-time between new OCT methods and already established ones.

7.1.2 Complex-Valued Signal Processing in OCT

An extension to the MSI OCT method, denoted as Complex Master-Slave Interferometry (CMSI) is introduced in [93]. This method processes complex-valued signals. This type of processing is already applied in other areas, such as wireless communications [94]. This enables the utilization of the phase in the acquired signal, which allows the reduction of the random component of the phase. Due to the complexity of the MSI method, the sequential implementation introduces a significant latency, which affects the performance of real-time OCT imaging. This is further highlighted by the processing of the phase in the CMSI method. The proposed coarse-grained and fine-grained approaches to the MSI method in Chapter 5 are expected to generate similar latency and speed-up in the CMSI. This would allow real-time utilization of the CMSI in OCT imaging.

7.1.3 Real-Time 3D Rendering in OCT

Chapter 5 proposed parallel implementations of synthetically generated confocal images. Each confocal image is based on multiple en-face images. The proposed parallel approaches have the ability to generate in real-time all possible en-face images, organized into en-face stacks. Alternatively, the same en-face stack can be visualized as a 3D volume. This would not require any further processing, because the en-face stack consists of the intensities of the en-face images. It would require the utilization of the 3D primitives from the used graphics library, such as the OpenGL. The generation of three-dimensional images, based on CFD-like method, is reported in [95]. A corresponding 3D imagery, based on the more computationally intensive MSI method, would require more resources, in terms of parallel threads and memory utilization. A further improvement would be the rendering of the volumetric OCT data from an arbitrary angle, selected in real-time.

7.1.4 Image Segmentation in OCT

Each en-face OCT image, presented in Chapter 5 and 6, is generated by selecting a single depth. This depth follows a plane, which is orthogonal to the point of view. When used in ophthalmology, this orthogonal plane crosses through different tissues. A generation of en-face image, which does not cross through different tissues but rather follow their curves and spans a number of depths, would vastly improve the application of the OCT system. Without crossing through tissues, a single en-face image would be able to visualize in a more complete manner features such as blood vessels and optic nerves.

This approach to visualization requires image segmentation. The segmentation needs to locate areas (segments) in the cross-sectional images acquired from the same OCT frame, calculate a curve, or set of depths, and use them in the en-face image. Manually corrected and user-guided segmentation in cross-sectional OCT images are reported, including [96] and [97].

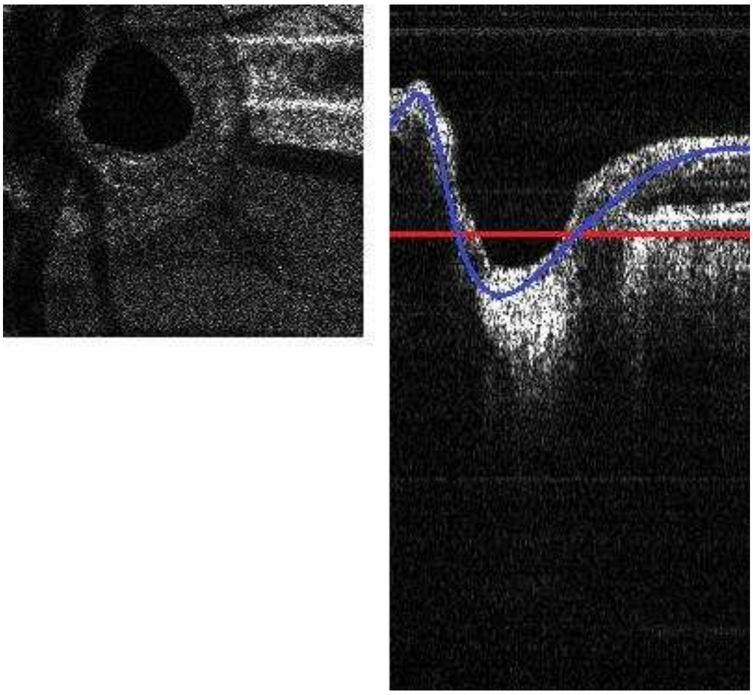


Figure 7.1: Current en-face image generated from a single depth. Image segmentation applied on the cross-sectional image would result in an en-face image generated by following a curve which approximates the geometry of the tissue

Ideally, the image segmentation would operate in real-time and be fully automated. Although, there could be some need for corrections done by the operator of the OCT system due to possible difficulties to detect edges in the image. The inclusion of image segmentation would bring additional computations. A parallel optimization is expected to absorb these additional computations without increasing the latency of the OCT imaging.

Bibliography

[1] Flynn, Hung. Microprocessor Design Issues: Thoughts on the Road Ahead, IEEE Computer Society, 2005. DOI: 10.1109/MM.2005.56.

[2] Vardi. Is Moore's Party Over?, Communications of the ACM, Volume 54, 2011. DOI: 10.1145/2018396.2018397.

[3] Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software, Dr. Dobb's Journal, 2005.

[4] Podoleanu, Bradu. Master–Slave Interferometry for Parallel Spectral Domain Interferometry Sensing and Versatile 3D Optical Coherence Tomography, Optics Express, Volume 21, 2013. DOI: 10.1364/OE.21.019324.

[5] Boppart, Herrmann, Pitris, Stamper, Brezinski, Fujimoto. High-Resolution Optical Coherence Tomography-Guided Laser Ablation of Surgical Tissue, Journal of Surgical Research, Volume 82, Issue 2, 1999.

[6] Cooley, Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. 1964.

[7] Heideman, Johnson, Burrus. Gauss and the History of the Fast Fourier Transform, IEEE ASSP Magazine, 1984. DOI: 10.1109/MASSP.1984.1162257.

[8] Frigo, Johnson. FFTW: An Adaptive Software Architecture for the FFT, IEEE International Conference on Acoustics, Speech and Signal Processing, 1998. DOI: 10.1109/ICASSP.1998.681704.

[9] cuFFT Library User's Guide, NVIDIA Corporation, 2015.

[10] Fast Fourier Transform in the West. [Online] www.fftw.org.

[11] Bradu, Kapinchev, Barnes, Podoleanu. On the Possibility of Producing True Real-Time Retinal Cross-Sectional Images Using a Graphics Processing Unit Enhanced Master-Slave Optical Coherence Tomography System, Journal of Biomedical Optics, Volume 20, Issue 7, 2015. DOI: 10.1117/1.JBO.20.7.076008.

[12] Bradu, Kapinchev, Barnes, Podoleanu. Master Slave En-Face OCT/SLO, Biomedical

Optics Express, 2015. DOI: 10.1364/BOE.6.003655.

[13] Kapinchev, Bradu, Barnes, Podoleanu. GPU Implementation of Cross-Correlation for Image Generation in Real Time, 9th International Conference on Signal Processing and Communication Systems, 2015. DOI: 10.1109/ICSPCS.2015.7391783.

[14] Multithreaded Programming Guide, Sun Microsystems, Inc., 2008.

[15] Multithreading in the Solaris(TM) Operating Environment, Sun Microsystems, Inc., 2002.

[16] Tosic. A Perspective on the Future of Massively Parallel Computing: FineGrain vs. CoarseGrain Parallel Models, Conference on Computing Frontiers, 2004. ISBN: 1-58113-741-9.

[17] Sutter, Larus. Software and the Concurrency Revolution, Microsoft Corporation, 2005.

[18] Chandra, Dagum, Kohr, Maydan, McDonald, Menon. Parallel Programming in OpenMP, Morgan Kaufmann, 2001. ISBN: 1-55860-671-8.

[19] Grochowski, Annavaram. Energy per Instruction Trends in Intel Microprocessors, Microarchitecture Research Lab, Intel Corporation.

[20] Tanenbaum, Bos. Modern Operating Systems, Pearson Education, Inc. ISBN: 978-0-13-359162-0.

[21] Microsoft API and Reference Catalog. [Online] msdn.microsoft.com/en-gb/library/.

[22] McCool, Robison, Reinders. Structured Parallel Programming, Morgan Kaufmann, 2012. ISBN: 978-0-12-415993-8.

[23] Williams. C++ Concurrency in Action, Manning Publications Co., 2012. ISBN: 860-1200915495.

[24] Flynn. Very High Speed Computing Systems, Proceedings of the IEEE, Volume 54, 1966.

[25] Flynn. Some Computer Organizations and Their Effectiveness, IEEE Transactions on Computers, 1972.

[26] El-Rewini, Abd-El-Barr. Advanced Computer Architecture and Parallel Processing, John Wiley & Sons, 2005. ISBN: 0-471-46740-5.

[27] Lindholm, Nickolls, Oberman, Montrym. Nvidia Tesla: A Unified Graphics and Computing Architecture, IEEE Computer Society, 2008. DOI: 10.1109/MM.2008.31.

[28] CUDA C Programming Guide, NVIDIA Corporation, 2015.

[29] Nickolls, Buck, Garland. Scalable Parallel Programming, ACM Queue, 2008.

[30] Gustafson. Reevaluating Amdahl's Law. 1988. DOI: 10.1145/42411.42415.

[31] GCC, the GNU Compiler Collection. [Online] gcc.gnu.org.

[32] Boehm. Threads Cannot Be Implemented as Library, ACM SIGPLAN Conference on Programming Language Design and Implementation, 2005. DOI: 10.1145/1064978.1065042.

[33] Herlihy, Shavit. The Art of Multiprocessor Programming, Morgan Kaufmann, 2008. ISBN: 978-0-12-370591-4.

[34] Akl, Selim. The Design and Analysis of Parallel Algorithms, Prentice Hall, 1989. ISBN: 0-13-200056-3.

[35] The OpenMP API. [Online] openmp.org.

[36] OpenCL The Open Standard for Parallel Programming of Heterogeneous Systems. [Online] www.khronos.org/opencl/.

[37] The Khronos Group. [Online] www.khronos.org.

[38] Daga, Scogland, Feng. Architecture-Aware Mapping and Optimization on a 1600-Core GPU, IEEE International Conference on Parallel and Distributed Systems, 2011. DOI: 10.1109/ICPADS.2011.29.

[39] Su, Chen, Lan, Huang, Wu. Overview and Comparison of OpenCL and CUDA Technology for GPGPU, IEEE Asia Pacific Conference on Circuits and Systems, 2012. DOI: 10.1109/APCCAS.2012.6419068.

[40] Weber, Gothandaraman, Hinde, Peterson. Comparing Hardware Accelerators in Scientific Applications: A Case Study, IEEE Transactions on Parallel and Distributed Systems, Volume 22, 2011. DOI: 10.1109/TPDS.2010.125.

[41] SYCL C++ Single-Source Heterogeneous Programming for OpenCL. [Online] www.khronos.org/sycl.

[42] Fang, Varbanescu, Sips. A Comprehensive Performance Comparison of CUDA and OpenCL, International Conference on Parallel Processing, 2011. DOI: 10.1109/ICPP.2011.45.

[43] Stone, Gohara, Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems, Computing in Science & Engineering, Volume 12, Issue 3, 2010. DOI: 10.1109/MCSE.2010.69.

[44] Nickolls, Dally. The GPU Computing Era, IEEE Computer Society, 2010. DOI: 10.1109/MM.2010.41.

[45] Nukada, Sato, Matsuoka. Scalable Multi-GPU 3-D FFT for TSUBAME 2.0 Supercomputer, International Conference for High Performance Computing, Networking, Storage and Analysis, 2012. DOI: 10.1109/SC.2012.100.

[46] NVIDIA’s Next Generation CUDA(TM) Compute Architecture: Kepler(TM) GK110, NVIDIA Corporation, 2015.

[47] NVIDIA Tesla P100, NVIDIA Corporation, 2016.

[48] Gregg, Hazelwood. Where is the Data? Why You Cannot Debate CPU vs. GPU Performance Without the Answer, IEEE International Symposium on Performance Analysis of Systems and Software, 2011. DOI: 10.1109/ISPASS.2011.5762730.

[49] Kirk, Hwu. Programming Massively Parallel Processors: A Hands-on Approach, Morgan Kaufmann, 2010. ISBN: 978-0-12-381472-2.

[50] NVIDIA’s Next Generation CUDA(TM) Compute Architecture: Fermi(TM), NVIDIA Corporation, 2015.

[51] Nandapalan, Jaros, Rendell, Treeby. Implementation of 3D FFTs across Multiple GPUs in Shared Memory Environments, International Conference on Parallel and Distributed Computing, Applications and Technologies, 2012. DOI: 10.1109/PDCAT.2012.79.

[52] Kapinchev, Bradu, Barnes, Podoleanu. Coarse-Grained and Fine-Grained Parallel Optimization for Real-Time En-Face OCT Imaging, Photonics West, 2016. DOI: 10.1117/12.2209560.

[53] Govindaraju, Lloyd, Dotsenko, Smith, Manferdelli. High Performance Discrete Fourier Transforms on Graphics Processors, Microsoft Corporation, 2008. DOI: 10.1109/SC.2008.5213922.

[54] Oppenheim, Willsky. Signals and Systems, Prentice-Hall.

[55] Yarlagadda. Analog and Digital Signals and Systems, Springer, 2010. DOI: 10.1007/978-1-4419-0034-0.

[56] Ingle, Proakis. Digital Signal Processing Using MATLAB, Cengage Learning, 2012. ISBN: 978-1-111-42737-5.

[57] Smith. The Scientist & Engineer's Guide to Digital Signal Processing, California Technical Publishing, 1997. ISBN: 0-9660176-3-3.

[58] Manolakis, Ingle, Kogon. Statistical and Adaptive Signals Processing, Artech House, Inc., 2005. ISBN: 1-58053-610-7.

[59] Lyon, Douglas. The Discrete Fourier Transform, Journal of Object Technology, 2010.

[60] Venugopalan. Evaluating Latency and Throughput Bound Acceleration of FPGAs and GPUs for Adaptive Optics Algorithms, High Performance Extreme Computing Conference, 2014. DOI: 10.1109/HPEC.2014.7040964.

[61] Arfken, Weber. Mathematical Methods for Physicists, Elsevier Academic Press, 2005.

ISBN: 0-12-088584-0.

[62] Subakan, Vemuri. Image Segmentation via Convolution of a Level-Set Function with a Rigaut Kernel, IEEE Conference on Computer Vision and Pattern Recognition, 2008. DOI: 10.1109/CVPR.2008.4587460.

[63] Ambardar. Analog and Digital Signal Processing, Brooks/Cole Publishing, 1999. ISBN: 053495409X.

[64] Blackman, Tukey. The Measurement of Power Spectra, Dover Publishing Inc., 1958. ISBN: 486-60507-8.

[65] Orfanidis. Introduction to Signal Processing, Prentice-Hall, 2010. ISBN: 0-13-209172-0.

[66] Podoleanu. Optical Coherence Tomography, Review Article, The British Journal of Radiology, 2005. DOI: 10.1259/bjr/55735832.

[67] Kharousi, Wali, Azeem. Current Applications of Optical Coherence Tomography in Ophthalmology, INTECH, 2013. DOI: 10.5772/53961.

[68] Liang, Peric, Hughes, Podoleanu, Spring, Saunders. Optical Coherence Tomography for Art Conservation & Archaeology, Optics for Arts, Architecture, and Archaeology, 2007. DOI: 10.1117/12.726032.

[69] Hariharan. Basics of Interferometry, Academic Press, 2007. ISBN: 978-0-12-373589-8.

[70] Fercher, Drexler, Hitzenberger, Lasser. OpticalCoherence Tomograpgy Principles and Applications, Institute of Physics Publishing, 2003. DOI: 10.1088/0034-4885/66/2/204.

[71] Watanabe, Itagaki. Real-Time Display on Fourier Domain Optical Coherence Tomography System Using a Graphics Processing Unit, Journal of Biomedical Optics, Volume 14, 2009. DOI: 10.1117/1.3275463.

[72] Bradu, Podoleanu. Imaging the eye fundus with real-time en-face spectral domain optical coherence tomography, Biomedical Optics Express, Volume 5, 2014. DOI: 10.1364/BOE.5.001233.

[73] Bradu, Maria, Podoleanu. Demonstration of Tolerance to Dispersion of Master/Slave Interferometry, Optics Express, 2015. DOI: 10.1364/OE.23.014148.

[74] Bradu, Kapinchev, Barnes, Podoleanu. Master-Slave Optical Coherence Tomography for Parallel Processing, Calibration Free and Dispersion Tolerance Operation, Photonics West, 2015. DOI: 10.1117/12.2078094.

[75] Bradu, Kapinchev, Barnes, Podoleanu. In-vivo, Real-Time Cross-Sectional Images of

Retina Using a GPU Enhanced Master Slave Optical Coherence Tomography System, Photonics West, 2016. DOI: 10.1117/12.2211072.

[76] Jeught, Bradu, Podoleanu. Real-Time Resampling in Fourier Domain Optical Coherence Tomography Using a Graphics Processing Unit, Journal of Biomedical Optics, 2010. DOI: 10.1117/1.3437078.

[77] Nam Cho, Unsang Jung, Suhwan Kim, Woonggyu Jung, Junghwan Oh, Hyun Kang, Jeehyun Kim. High Speed SD-OCT System Using GPU Accelerated Mode for in vivo Human Eye Imaging, Journal of the Optical Society of Korea, Volume 17, 2013. DOI: 10.3807/JOSK.2013.17.1.068.

[78] Radojkovic, Carpenter, Moreto, Cakarevic, Verdu, Pajuelo, Cazorla, Nemirovsky, Valero. Thread Assignment in Multicore/Multithreaded Processors: A Statistical Approach, IEEE Transactions on Computers, Volume 65, 2015. DOI: 10.1109/TC.2015.2417533.

[79] Frigo, Johnson. The Design and Implementation of FFTW3, Proceedings of the IEEE, Volume 93, 2005. DOI: 10.1109/JPROC.2004.840301.

[80] Harris, Mark. Optimizing Parallel Reduction in CUDA, NVIDIA Corporation.

[81] Che, Li, Sheaffer, Skadron, Lach. Accelerating Compute-Intensive Applications with GPUs and FPGAs, Symposium on Application Specific Processors, 2008. DOI: 10.1109/SASP.2008.4570793.

[82] Roorda, Romero-Borja, Donnelly, Queener. Adaptive Optics Scanning Laser Ophthalmoscopy, Optics Express, Volume 10, 2002. DOI: 10.1364/OE.10.000405.

[83] Podoleanu, Jackson. Combined Optical Coherence Tomograph and Scanning Laser Ophthalmoscope, Electronics Letters, 1998. DOI: 10.1049/el:19980793.

[84] Merino, Dainty, Bradu, Podoleanu. Adaptive Optics Enhanced Simultaneous En-Face Optical Coherence Tomography and Scanning Laser Ophthalmoscopy, Optics Express, Volume 114, 2006. DOI: 10.1364/OE.14.003345.

[85] Kapinchev, Barnes, Rivet, Bradu, Podoleanu. Parallel approaches to integration with applications in optical coherence tomography, 10th International Conference on Signal Processing and Communication Systems, 2016. DOI: 10.1109/ICSPCS.2016.7843350.

[86] ATS-VI Software Manual, Alazar Technologies Inc., 2009.

[87] Writing win32 dynamic link libraries (DLLs) and calling them from LabVIEW. [Online] National Instruments Corporation, 2010. http://www.ni.com/white-paper/4877/en.

[88] Kapinchev, Barnes, Bradu, Podoleanu. Approaches to General Purpose GPU Acceleration of Digital Signal Processing in Optical Coherence Tomography Systems, IEEE International Conference on Systems, Man, and Cybernetics, 2013. DOI: 10.1109/SMC.2013.440.

[89] OpenGL The Industry's Foundation for High Performance Graphics. [Online] www.opengl.org.

[90] Bradu, Kapinchev, Barnes, Garway-Heath, Rajendram, Keane, Podoleanu. Real-Time Calibration-Free C-scan Images of the Eye Fundus Using Master Slave Swept Source Optical Coherence Tomography, Photonics West, 2015. DOI: 10.1117/12.2078956.

[91] Intel Corporation. Intel Xeon Phi Processors: Your Path to Deeper Insight, Intel Corporation, 2016.

[92] The Intel Xeon Phi(TM) Product Family (Product Brief), Intel Corporation, 2013.

[93] Rivet, Maria, Bradu, Feuchter, Leick, Podoleanu. Complex Master Slave Interferometry, Optics Express, 2016. DOI: 10.1364/OE.24.002885.

[94] Martin, Kenneth. Complex Signal Processing Is not Complex, IEEE Transactions on Circuits and Systems, Volume 51, 2004. DOI: 10.1109/TCSI.2004.834522.

[95] Sylwestrzak, Szlag, Szkulmowski, Targowski. Real-time Massively Parallel Processing of Spectral Optical Coherence Tomography Data on Graphics Processing Units, Optical Coherence Tomography and Coherence Techniques, 2011. DOI: 10.1117/12.889805.

[96] Duncker, Stein, Lee, Tsang, Zernant, Bearelly, Hood, Greenstein, Delori, Allikmets, Sparrow. Quantitative Fundus Autofluorescence and Optical Coherence Tomography in ABCA4 Carriers, Investigative Ophthalmology & Visual Science, 2015. DOI: 10.1167/iovs.15-17371.

[97] Yin, Chao, Wang. User-Guided Segmentation for Volumetric Retinal Optical Coherence Tomography Images, Journal of Biomedical Optics, 2014. DOI: 10.1117/1.JBO.19.8.086020.