Parallel Approaches to Digital Signal Processing Algorithms with Applications in Medical Imaging

Konstantin Kapinchev
School of Computing
School of Physical Sciences
University of Kent
Canterbury, England
K.Kapinchev@kent.ac.uk

Adrian Bradu
Applied Optics Group
School of Physical Sciences
University of Kent
Canterbury, England
A.Bradu@kent.ac.uk

Adrian Podoleanu
Applied Optics Group
School of Physical Sciences
University of Kent
Canterbury, England
A.G.H.Podoleanu@kent.ac.uk

***Abstract*-This paper reviews established and emerging parallel technologies, which are employed to enhance the performance of digital signal processing algorithms. Special attention is paid to algorithms with applications in medical imaging. Parallel implementations of some of the most commonly used algorithms, such as Fourier transforms, convolution and cross-correlation are discussed. Parallel optimization of a newly introduced method in optical coherence tomography is presented. Its performance, in terms of latency, is presented and discussed.**

Keywords-parallel computing, digital signal processing, medical imaging, optical coherence tomography

I. Introduction

There is a constant demand for higher performance delivered by computer systems, a trend seen in many areas, especially in real time systems. Particular need for improved computational performance can be observed in the area of digital signal processing (DSP). In this area, large volumes of data are processed by applying mathematical operations, such as Fourier transforms, matrix multiplication, convolution, cross-correlation and others.

Performance improvements, based solely on new processing units with increased number of transistors and more complex architecture, cannot meet the growing demand for computational power. Considering the current technologies and the physical limitations of the transistor-based chips, as discussed in [1], the primary way to extract higher performance is to increase the number of processing cores and to adapt the software accordingly, as described in [2]. This trend is illustrated by the introduction of the dual-core CPU, followed by further increase of the number of the CPU cores.

Later on, the highly parallel architecture of the graphics processing units (GPU), originally designed for rendering imagery, found another application, namely a parallel acceleration of computationally intensive tasks. Those tasks are not limited to a specific type, but a wide range of computational problems. As a result, the GPU assumed the role of a parallel co-processor, capable of significant acceleration of computationally intensive tasks.

Subsequently, this trend is reinforced by the introduction of the parallel co-processors, such as Intel Xeon Phi [3], combining some features from the architectures of the multi-core CPU and the many-core GPU. These parallel co-processors were followed by the introduction of bootable host processors.

The introduction of the aforementioned parallel architectures is accompanied by the development of various programming environments and languages, designed to utilize them. These programming languages allows the developer to divide a larger *task* into a number of smaller *sub-tasks* and assign each *sub-task* to a *thread of control*. The execution of these threads of control will be distributed among the processing cores, for example on a multi-core CPU, or on a many-core GPU.

These threads of control can be implemented for example as processes (UNIX), threads (Java, C), GPU threads (NVIDIA CUDA) and fibers (C++).

The following sections discuss the DSP algorithms, including those involved in medical imaging. The parallel architectures with their corresponding programming supports are presented. Special attention is paid to the GPU computing and its utilization in the area of medical imaging.

II. Digital Signal Processing Algorithms

The DSP algorithms discussed in this paper are commonly employed in the area of image generation and more specifically medical imaging. Special attention is paid to optical coherence tomography (OCT), its use of DSP algorithms and possible benefits from parallel optimization.

A. Fourier Transforms

Fourier transform, introduced by Joseph Fourier (1768-1830), is a mathematical operation which provides the conversion of signals from time domain to frequency (Fourier) domain. In case of discrete signals, discrete Fourier transform (DFT) can be applied. It is a computationally intensive algorithm, with computational complexity set at *O(n^2)*, where n is the number of data points. Numerous efforts are made to improve the performance of this algorithm, with one of the most notable implementations being the Fast Fourier Transform (FFT), proposed in [4]. This optimization of the DFT algorithm reduces the complexity to *O(n log n)*.

A number of software libraries performing FFT are developed. Examples include FFT of the West (FFTW) [5] and NVIDIA cuFFT [6].

B. Matrix Multiplication

Matrix multiplication, strictly speaking classified as an algebraic operation, is included in this discussion due to its common occurrence alongside other typical signal processing operations. In many cases in practice, matrix multiplication *($Z=X\times Y$)* involves processing large amount of data points, in other words, relatively larger values of p, q and r in Equation 1. This makes it computationally intensive operation. As a result, significant efforts are made to optimize this operation, especially for real time applications.

$$z_{ij} = \sum_{k=1}^{r} x_{ik} \times y_{kj} \qquad \text{where: } i = 1 \,..\, p,\ j = 1 \,..\, q \tag{1}$$

Based on the proportion of zero-valued elements, matrices are divided into two groups, dense matrices, with more than half none-zero elements, and sparse matrices, with less than half none-zero elements. This distinction leads to separate optimization strategies for these two types of matrices. In the case of dense matrices, the optimization methods usually exploit the data locality and aim at involving a higher number of parallel threads of control [7]. In the case of the sparse matrices, one of the main objectives is to eliminate operations which do not affect the result, such as storing and processing zero-valued elements, as discussed in [8].

A number of software solutions offer matrix multiplication tools. Those with industrial strength qualities include NVIDIA, MATLAB, LabVIEW and others.

C. Convolution

Convolution is a mathematical operation with two input signals (functions) *X* and *Y* and one output signal (function). The convolution itself can be treated as a function of the lag *L*, which is a delay applied to one of the input signals, as seen in Equation 2. In practice, convolution calculates the output of a linear system, based on the impulse response of that system and an input signal.

$$(X \cdot Y)_L = \int_{-\infty}^{\infty} X(t)Y(L-t)dt \tag{2}$$

If *X* and *Y* are digital signals, convolution can be calculated by applying the Convolution Theorem as illustrated in [9] and stated in Equation 3. In this case, the convolution of two digital signals, *X* and *Y*, can be calculated by applying inverse Fourier transform (IFT) on the product of the Fourier transform (FT) of the two input signals. This product can be implemented as *dot product*. In many cases in practice, this is implemented by using multiply-accumulate operations to improve the speed of the dot product operation.

$$X \cdot Y = IFT\big(FT(X) \times FT(Y)\big) \tag{3}$$

The Convolution Theorem allows the utilization of optimized FFT libraries, such as the NVIDIA cuFFT. This provides additional avenues to performance improvement of the convolution algorithm.

D. Cross-Correlation

Cross-correlation, similarly to convolution, processes two input signals (functions), *X* and *Y*, and produces third signal (function), as shown in Equations 4. The resulting signal measures the level of similarity between the input signals *X* and *Y*. Equation 4 expresses the definition of the cross-correlation operation and Equation 5 denotes the Cross-Correlation Theorem, as discussed in [10]. According to this theorem, the cross-correlation of two signals can be obtained by the inverse Fourier transform of the product of the complex conjugate of the Fourier transform (FT) of signal *X*, and the Fourier transform of signal *Y*. Similarly to the convolution, the product can be implemented as dot product.

$$(X \cdot Y)_L = \int_{-\infty}^{\infty} \overline{X(t)}Y(t+L)dt \tag{4}$$

$$X \cdot Y = IFT\left(\overline{FT(X)} \times FT(Y)\right) \tag{5}$$

The Cross-Correlation Theorem allows, as in the case of the Convolution Theorem, alternative implementations based on Fourier transforms.

E. DSP in Medical Imaging

Medical imaging concerns with the generation of graphical representation of organs and tissues, by processing signals captured by some form of sensors. Those signals are collected and digitized by data acquisition systems. Those systems made these signals available for further processing, which is transforming a stream of numerical data into information ready to be visualized, as illustrated in Figure 1.

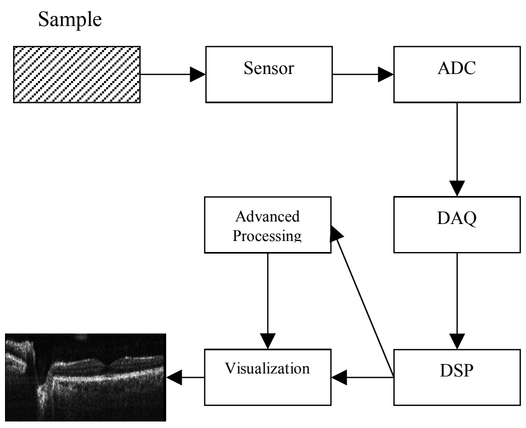

Figure 1: Building blocks in medical imaging. In case of MRI the sensor is realized as a radio frequency receiver, in the case of OCT as a photodetector

An emerging concept in medical imaging is the introduction of artificial intelligence (AI) and machine learning (ML) tools, denoted as Advanced Processing in Figure 1. These tools transform the concept of medical imaging and extend its role. The introduction of AI and ML in medical imaging has the potential to improve the diagnosis and the monitoring of treatments, as presented in [11].

III. Concurrent Computing and Parallel Computing

The division of a computational task among multiple threads of control underpins the concept of both concurrent and parallel computing. Although these two concepts share some similarities, these is one significant difference, which describes how the threads of control are handled by the processing units.

For both concurrent and parallel computing, each thread of control has the ability to advance independently, unless some form of synchronization is put into place.

In terms of concurrent computing, these threads of control are not expected to advance simultaneously. A software solution, although organized into multiple threads, may complete all statements from all threads of control sequentially. At any given moment, only one of them will be running, while the rest will be waiting. Concurrent design can be achieved on a single processing unit with the ability to switch between statements, or instructions, from different threads of control.

In parallel computing, on the other hand, a number of threads of control are expected to run simultaneously. This imposes the need for parallel multi-core or many-core architectures [12].

While concurrency brings certain advantages, such as isolating and encapsulating different types of computations, it has the ability to improve the performance in very limited cases. For example, a thread of control dedicated on computations may advance while another thread of control is waiting for an input or output operation to complete. On the other hand, introducing parallel computing has the capability to improve the performance of otherwise sequential computations. Therefore, the remaining of this paper will focus on the parallel computing, its theoretical and practical aspects and its utilization in the area of DSP and medical imaging.

IV. Classifications of Parallel Implementations

Parallel implementations can be significantly different. They utilize various architectures of the processing units, different programming languages, and most notably, they are designed for different purposes.

There are a number of ways to classify parallel implementations, most commonly by the way threads of control communicate with each other, the overheads introduced by the management of the parallel threads of control, the granularity of the parallelized task and the ability of the parallel solution to scale to different sizes of input data.

A. Communication

Parallel threads of control, while working on their sub-tasks, need to exchange information with each other. The most common reasons for this exchange are to synchronize the execution process and to share the results of the computations. The two most common ways to achieve this communication are by shared memory and message passing [13].

In the case of shared memory, the parallel threads of control have access to the same memory locations, which can be used to exchange the state of the computational process and to exchange data. For example, a thread of control generates output data, which another thread of control uses as input data.

In the case of message passing, the parallel threads of control rely on a mechanism provided by the operating system to exchange the aforementioned state and data. A notable example for message passing mechanism is the Inter-Process Communication in UNIX environment.

B. Overheads

The parallel optimization of sequential software solutions comes with a price in the form of the overheads introduced by the management of the parallel threads of control. Extensive comparative studies are conducted on the selection of types of threads based on the nature of the computations [14].

Based on this property, threads of control can be part of one of the two categories, namely heavyweight and lightweight threads. Heavyweight threads of control usually have separate virtual address space. Switching between those types of threads of control takes significant part of the overall computational process. Lightweight threads of control, on the other hand, usually share the same address space. Switching between lightweight threads of control is expected to be significantly faster, compared with the heavyweight ones.

C. Granularity

A key problem in parallel computing is to identify the optimal number of threads of control which will deliver the highest performance. This optimal number depends on a number of factors, most importantly the targeted parallel architecture, the programming language, the nature of the task and the input data. This division can be performed in different ways. It determines the relative number of the parallel threads of control and the granularity of the parallel solution. Depending on this relative number, a parallel solution can be considered as coarse-grained or fine-grained, [15].

In the case of a coarse-grained approach, the task is divided into relatively small number of sub-tasks. This approach is suitable for cases in which the threads of control have significant overheads, or the limit of possible parallel threads can easily be reached. Examples include UNIX-based processed, POSIX threads and Java threads.

In the case of the fine-grained approach, the number of tasks, and corresponding threads of control, is relatively large. The GPU threads exemplify the fine-grained approach. Equation 6 illustrates the maximum number of GPU threads *T* launched by a single CUDA kernel. Not all threads are expected to run simultaneously, as some form of serialization will be introduced by the programming environment.

$$T = Block_M \times Dim_X \times Dim_y \times Dim_Z \qquad (6)$$

Where:
$Block_M$ is the maximum number of threads per block (1024)
Dim_x is the maximum X dimension of the GPU Grid (2^{32}-1)
Dim_y and Dim_z are the maximum Y and Z dimensions of the GPU Grid (2^{16}-1)

D. Scalability

An important feature, denoted as scalability, describes the ability of a computer system to absorb increased amount of computations in a predictable and manageable way. This feature is of critical importance in parallel systems, which are, in most cases, designed to enhance the performance. If presented, this feature would allow the parallel systems to work well with different sizes of input data. Scalability requires support from both the architecture of the processing units and the organization of the parallel threads of control.

In general, two types of scalability are recognized. Those are strong scalability, illustrated by the Amdahl Law [16] and weak scalability, illustrated by the Gustafson Law [17]. In the first case, the scalability of a computer system is evaluated by observing how various sizes of input data affect the performance of the system. In the second case, the performance of the system is measured by changing both the size of the input data and the number of processing units.

This feature is of critical importance in the parallel implementations of DSP algorithms. In most cases in practice, those implementations are expected to provide high performance regardless of the size of the input data. Otherwise, the usefulness of these implementations would be reduced.

V. Programming Languages with Parallel Support

If a larger task is implemented as a single thread of control, it would benefit very little in terms of performance from a multi-core or many-core architecture, as observed in [2]. Improved performance, based on parallel architecture, requires multi-threaded approach, in which a number of parallel threads of control are executed simultaneously by the processing cores. Each thread of control performs part of the larger task, denoted as a sub-task. This approach requires support from the operating system, the programming language and the software solution itself, as presented in [18]. The following is a non-exhaustive review of some of the most popular programming languages, which offer parallel support.

A. C/C++

The C programming language was introduced to allow operating system developers to use high-level programming language, instead of an assembly language. This resulted in solutions with greater portability. The language gained significant popularity over the years and set the foundations of the development of a large family of C-like programming languages, such as C++, C#, Objective C and others.

The language specification of the standardized ANSI C does not include multithreaded parallel functionality, as discussed in [19]. Access to multithreaded support is provided via libraries, such as pthread (POSIX thread).

The standard C++ language, on the other hand, provides multi-threaded support as part of the language specification since the standardized version C++ 11.

B. Java

The Java language specification includes multi-threaded support, as described in [20]. It is done by allowing classes to either extend a predefined class *Thread*, or to implement an interface called *Runnable*.

In both cases, the parallel threads are encapsulated as class methods. These approaches highlight the object-oriented nature of the Java programming language.

C. OpenMP

Open Multi-Processing (OpenMP) is a parallel extension to some of the most popular programming languages, such as C/C++ and Fortran. It is implemented as application programming interface (API). If utilized, it organizes the computer programs into two parts, sequential and parallel. Originally, OpenMP was designed for data parallelism, mainly by parallelizing the iterations of loops. Later on, task-based parallelism was included, which extended the application area of OpenMP.

D. GPU Computing

The constant demand for computer generated graphics with higher quality, in terms of resolution and refresh rate, resulted in the introduction of graphics processing units and the advancements in their architectures. They emerged not only as devices specialized in graphics, but also as parallel co-processors, having as many as 4608 CUDA Cores and 576 Tensor Cores in NVIDIA TITAN RTX. The following key events shaped the current state of the GPU computing.

In 1999, the NVIDIA Corporation released GeForce 256, an event widely accepted as the introduction of the first GPU.

In 2007, NVIDIA introduced the CUDA programming environment, which made the parallel architecture of the GPU available for general computations. As a result, the term General Purpose GPU (GPGPU) computing was coined.

This programming environment provides extension to some of the most popular programming languages, especially C/C++. Key part of the language extension are specialized types of sub-routines denoted as *kernels*. When launched, the code within a kernel executes as a grid of multiple GPU threads with access to the global GPU memory.

In 2017, the NVIDIA Corporation introduced the Volta architecture [21]. This emerging GPU technology introduced the Tensor Cores, aimed at the growing demand for computational performance from areas such as artificial intelligence and machine learning. In these areas of computer science, the building block of the computations is the matrix multiplication. As a result, Tensor cores were specifically designed for optimized matrix multiplication by utilizing the multiply-accumulate operation [22].

VI. Parallel Optimizations of DSP Algorithms

In the process of parallel optimization, a possible way to achieve optimal performance is to implement a one-to-one correspondence of data points and parallel threads of control. In the example demonstrated in Section VII, the digital signal consists of 512,000,000 data points per image. Processing this relatively large number of data points is common in many application areas, including medical imaging and OCT. In those cases, one-to-one correspondence can be achieved by using the GPU as a parallel co-processor. This is due to the large number of threads in the GPU grid and their lightweight property. Two approaches based on the GPU are demonstrated in the next section.

VII. Applications in Medical Imaging

DSP algorithms are extensively used in medical imaging. Areas such as MRI and OCT generate two and three dimensional imagery based on data streams generated by the imaging equipment. In these cases, DSP algorithms are employed for the reconstruction of the images, for image segmentation and registration. The results are improved diagnosis and treatment monitoring.

DSP algorithms in their nature follow the Single Instruction Multiple Data (SIMD) approach to parallelism. In this type of parallelism, the same instruction is applied to large number of data points. These algorithms can be classified as embarrassingly parallel, as discussed in [23]. An optimal choice for a parallel approach would be the fine-grained GPU-based implementations.

A. Optical Coherence Tomography

By using optical coherence tomography, semi-transparent objects, denoted as samples in Figure 1, can be imaged layers under their surfaces. This technology has applications in two major areas, medical imaging [24], [25] and art conservation [26].

An OCT method, denoted as Master-Slave Interferometry (MSI), was introduced in [27]. The improvements provided by this method are presented in [28]. This method generates OCT images from different depths by applying cross-correlation of two signals, *f* and *g*. Equation 6, similarly to Equation 5, illustrates this method by taking advantage of the Cross-Correlation theorem. Since the result of the cross-correlation is a digital signal corresponding to the intensity of a point at a certain depth, the intensity of a specific point, I_p, can be obtained by the summation of the components of the signal, as shown in Equation 7.

$$I_p = \sum \left| IFT\left(FT(f) \times \overline{FT(g)}\right) \right| \qquad (7)$$

An improvement of the aforementioned MSI OCT method is the Complex MSI, introduced in [29]. Similarly to the MSI method, the Complex MSI method generates OCT images by combining two signals. Those are signal *f*, generated by the OCT system during the imaging of the sample and signal *g*, called Mask signal. Unlike the MSI OCT method in which every Mask signal is generated experimentally, the Complex MSI method requires only one experimentally generated Mask signal. Based on this signal, multiple synthetically generated Mask signals corresponding to different depths can be produced and saved. The intensity of a point is obtained by applying dot product between signals *f* and *g*. To obtain a cross-sectional OCT image, as shown in Figure 2, multiple dot products, a computation equivalent to a matrix multiplication, are performed. In this case, real time simultaneous generation of images from multiple depths would benefit from parallel optimization.

B. Parallel Implementation

The GPU implementation of the Complex MSI method can be organized into the following steps:

1. *CPU-GPU Copy*
2. *Multiplication Kernel*
3. *Integration Kernel*
4. *Scale to 256-bit gray scale color*
5. *GPU-CPU copy*

The proposed implementation organizes the matrix multiplication into two kernels, namely *Multiplication Kernel* and *Integration Kernel*. This allows more flexible and independent optimization of the two kernels. In the case of the multiplication, the kernel launches one GPU thread per data point, and thus achieves highest level of parallelism. For the integration, a number of approaches were studied with their performance compared in [30].

C. Performance of Complex MSI

Two GPU approaches, namely Parallel Reduction and Zero Frequency Component (ZFC) were explored in [30]. These two approaches delivered optimal performance. Therefore, they are utilized in the proposed parallel optimization of the Complex MSI.

Table 1 presents the results of the parallel GPU-based implementations of the Complex MSI method. Tests with images with various sizes were conducted and the corresponding latencies recorded.

Table 1: Latency of the Complex MSI in milliseconds

Image size (pixels)[1]	Parallel Reduction	ZFC
100×500	12	4
200×500	16	6
500×500	19	12
1000×500[2]	24	22

[1] The number of data points is equal to the number of pixels multiplied by 1024

[2] Maximum number of data points equals to 1024×1000×500

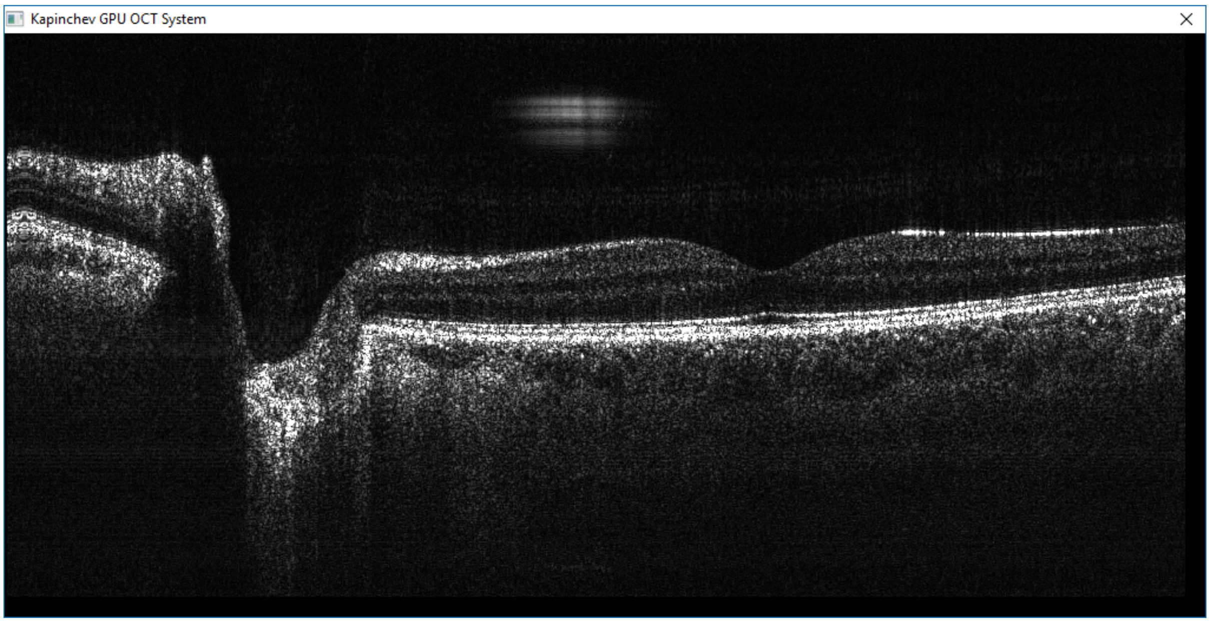

Figure 2: Cross-sectional (b-scan) OCT image of human retina with dimension of 1000×500 pixels, obtained by performing the Complex MSI method, implemented as NVIDIA CUDA C application, generated on NVIDIA GeForce GTX TITAN X and visualized by OpenGL

VIII. Conclusion

This paper reviewed established and emerging parallel technics, which are applied to accelerate the performance of DSP algorithms employed in medical imaging. Most common parallel programming environments and languages were discussed and compared.

GPU-based parallel optimization of the Complex MSI OCT method is proposed. An improved performance is expected from incoming GPU architectures. This is due to the scalability of the proposed implementation, more specifically the *Multiplication* and the *Integration* kernels.

References

[1] Mack, “Fifty Years of Moore’s Law”, IEEE Transactions on Semiconductor Manufacturing, 2011, DOI: 10.1109/TSM.2010.2096437

[2] Sutter, “The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software”, 2005

[3] Jeffers, Reinders, “Intel Xeon Phi Coprocessor High Performance Programming”, Morgan Kaufmann Publishers, ISBN: 978-0124104143, 2013

[4] Cooley, Tukey, “An Algorithm for the Machine Calculation of Complex Fourier Series”, 1964

[5] Frigo, Johnson, “The Design and Implementation of FTW3”, Proceedings of the IEEE, Volume 93, Issue 2, 2005, DOI: 10.1109/JPROC.2004.840301

[6] cuFFT Library User’s Guide, NVIDIA Corporation, 2019

[7] Garcia, Venetis, Khan, Gao, “Optimizing Dense Matrix Multiplication on a Many-Core Architecture”, Euro-Par, 2010, DOI: 10.1007/978-3-642-15291-7_29

[8] Kirk and Hwu, “Programming Massively Parallel Processors”, 2013, ISBN: 978-0-12-415992-1

[9] Arfken, Weber, “Mathematical Methods for Physicists”, Elsevier Academic Press, 2005, ISBN: 0-12-088584-0

[10] Yarlagadda, “Analog and Digital Signals and Systems”, Springer, 2010, DOI: 10.1007/978-1-4419-0034-0

[11] Comaniciu, Engel, Georgescu, Mansi, “Shaping the Future through Innovation: From Medical Imaging to Precision in Medicine”, 2016, DOI: 10.1016/j.media.2016.06.016

[12] Multithreaded programming Guide, Sun Microsystems Inc., 2008

[13] Petersen, Arbenz, “Introduction to Parallel Computing”, Oxford University Press, 2004, ISBN: 0-19-851576-6

[14] Castello, Pena, Seo, Mayo, Balaji, Quintana-Orti, “A review of Lightweight Thread Approaches for High Performance Computing”, IEEE International Conference on Cluster Computing, 2016, DOI: 10.1109/CLUSTER.2016.12

[15] Tosic, “A Perspective on the Future of Massively Parallel Computing: FineGrain vs. CoarseGrain Parallel Models”, Conference on Computing Frontiers, 2004, DOI: 10.1145/977091.977160

[16] Amdahl, “Validity of the single processor approach to achieving large scale computing capabilities”, Spring Joint Computer Conference, 1967, DOI: 10.1145/1465482.1465560

[17] Gustafson, “Reevaluating Amdahl’s Law”, 1988, DOI: 10.1145/42411.42415

[18] Mahapatra, Mishra, “Oracle Parallel Processing”, O’Reilly Media, 2000, ISBN: 978-1565927018

[19] Boehm, “Threads Cannot Be Implemented as Library”, ACM SIGPLAN Conference on Programming Language Design and Implementation, 2005. DOI: 10.1145/1064978.1065042

[20] Gosling, Joy, Steele, Bracha, Buckley, Smith, “The Java Language Specification”, Oracle Inc., 2019

[21] NVIDIA Tesla V100 GPU Architecture, NVIDIA Corporation, 2017

[22] Markidis, Chien, Laure, “NVIDIA Tensor Cores Programmability, Performance & Precision”, IEEE International Parallel and Distributed Processing Symposium Workshops, 2018, DOI: 10.1109/IPDPSW.2018.00091

[23] Herlihy, Shavit, “The Art of Multiprocessor Programming”, Morgan Kaufmann Publications, 2008, ISBN: 978-0-12-370591-4

[24] Fujimoto, “Optical coherence tomography for ultrahigh resolution in vivo imaging”, Nature Biotechnology, 2003, DOI: 10.1038/nbt892

[25] Podoleanu, “Optical Coherence Tomography”, The British Journal of Radiology, 2005, DOI: 10.1111/j.1365-2818.2012.03619.x

[26] 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

[27] 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

[28] 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

[29] Rivet, Maria, Bradu, Feuchter, Leick, Podoleanu, “Complex Master Slave Interferometry”, Optic Express, 2016, DOI: 10.1364/OE.24.002885

[30] 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