Parallel computing and parallel programming models: application in digital image processing on mobile systems and personal mobile devices

University of Oulu
Department of Information Processing Science
Bachelor's Thesis
Ari Ruokamo
31.1.2018

Abstract

Today powerful parallel computer architectures empower numerous application areas in personal computing and consumer electronics and parallel computation is an established mainstay in personal mobile devices (PMD). During last ten years PMDs have been equipped with increasingly powerful parallel computation architectures (CPU+GPU) enabling rich gaming, photography and multimedia experiences ultimately general purpose parallel computation through application programming interfaces.

This study views into current status of parallel computing and parallel programming, and specifically its application and practices of digital image processing applied in the domain of Mobile Systems (MS) and Personal Mobile Devices (PMD). The application of parallel computing and -programming has become more common today with the changing user-application requirements and with the increased requirements of sustained high-performance applications and functionality. Furthermore, the paradigm shift of data consumption in personal computing towards PMD and mobile devices is under increased interest. The history of parallel computation in MS and PMD is relatively new topic in academia and industry. The literature study revealed that while there is good amount of new application specific research emerging in this domain, the foundations of dominant and common parallel programming paradigms in the area of MS and PMD are still moving targets.

Keywords
parallel computing, parallel programming, image processing, signal processing, graphics processing unit, GPU, GPGPU, mobile systems, personal mobile devices

Supervisor
M.Sc. EE, PhD Candidate, Mr. Pertti Seppänen

Foreword

This thesis idea started from my personal motives towards video and image processing on mobile devices. When the first truly powerful smartphones became available during 2010's, the application idea started to materialize. From the science perspective and with only some previous experience on parallel programming, exploring this interesting area and phenomenon was, and still is an interesting task. For me it has revealed a lot of fundamental and valuable knowledge that for years I have taken for granted, without knowing the history and origins of that knowledge.

I want to thank my thesis supervisor Pertti Seppänen for accepting the challenge to guide me in the process of writing this thesis with his valuable experience in computing and science. I also want to thank Raija Halonen for supporting and motivating me writing this thesis, and Mari Karjalainen, who also provided me important knowledge and instructions during the writing process.

I want to express my dearest thanks to my love wife Teija, who has tolerated my long evenings and enthusiasm on this thesis and everything around it.

Ari Ruokamo
Oulu, 29 January 2018

Abbreviations

API	Application programming interface
APU	Accelerated Processing Unit
AR	Augmented reality
CCD	Charge-coupled device
COTS	Commercial, “Off-The-Shelf”
CPU	Central processing unit
CUDA	Compute Unified Device Architecture
DCT	Discrete cosine transform
DIP	Digital image processing
GPGPU	General purpose computing on a graphics processing unit
GPU	Graphics processing unit
HPC	High-performance computing
ILP	Instruction-level parallelism
MAR	Mobile augmented reality
MIMD	Multiple instructions multiple data
MISD	Multiple instructions single data
MPI	Message Passing Interface
MPMD	Multiple programs multiple data
MPP	Massively Parallel Processing
MPSD	Multiple programs single data
NWP	Numerical weather prediction
OpenGL ES	Open graphics language embedded system
OSS	Open source software
PC	Personal computer
PMD	Personal mobile device
POSIX	Portable Operating System Interface
RLP	Request-level parallelism
SDK	Software development kit
SE	Software engineering
SIMD	Single instruction multiple data
SPSD	Single program single data
SPMD	Single program multiple data
SMP	Symmetric Multiprocessor
SoC	System on a Chip
TBB	Threading building blocks
TLP	Thread-level parallelism

Contents

1. Introduction

The need for accelerated information processing and computation through scalable parallel execution has existed for several decades. The development and evolution of computer architectures has made the parallel computation more accessible and cost effective than ever before. The amount of created and stored information has been rapidly increasing in many areas of human life, in science and engineering, industry, medicine and entertainment. Different types of requirements drive the development and application of parallel processing. In science and engineering, High Performance Computing (HPC) and Massively Parallel Processing (MPP) systems comprise numerous processing units, often consisting of hundreds and even millions of processing cores to perform the required application execution simultaneously. Modern Numerical Weather Prediction (NWP), complex simulation models in astronomy and medicine employ massive distributed parallel processing applications utilizing enormous communication- and storage resources. Consumer-level multimedia and graphics computation, modern mobile phones, smartphones and tables often require real-time parallelism of multiple concurrent processing units to accomplish the tasks of visual content processing we are accustomed with media interfaces today (Grama, 2003).

Today, a smartphone or a tablet has become the sole personal computer for many people. Information browsing and acquisition, information consumption and processing, collaboration and sharing is increasingly conducted with smartphones and portable tablets. The ever-changing landscape of services and applications people use every day is transforming the applications and their processing and performance requirements. For example, *Mobile Augmented Reality (MAR)* has been a trending topic among mobile application developers in recent years. MAR adds a segment of mobile applications that were not possible to implement until recent developments in mobile device hardware and software technology. For example, already today furniture manufacturers provide mobile applications that allow users with a smartphone camera to augment their home spaces with furniture or lightning fixtures in real-time to help their customers to make purchase decisions. Similarly, a cosmetics company provides a smartphone app where user can apply an artificial make-up on one's face in real-time, still making it look like it was real (Wired, 2017; Modiface, 2017).

In the landscape of emerging new applications, the increasing efficiency and processing requirements push the future of computing towards parallelism. In specific computation setups, the new applications, modern games and scientific- and engineering applications have requirements that are constrained by the CPU-only performance. A new general computing model has emerged where CPU works together with *graphics processing unit (GPU)*. In these parallel configurations, typically containing several multi-core processing units (e.g. CPU and GPU), can reside potential in responding to increasing computation requirements (Owens, Houston, Luebke, Green, Stone, Phillips, 2008).

1.1 Purpose and motivation of the study

The motivation for the study emerged from author's personal interests towards image processing and the ambition to build a real-time video processing application for Apple iOS-platform smartphones and tablets. Furthermore, author has 8 years of professional application development experience on Apple iOS platform.

Application of digital image processing for video effects processing typically requires massive computation power as the information required to be processed is vast. Apple iOS Software Development Kit (SDK), and Apple devices have included a powerful programmable GPU providing since introduction of iPhone 3GS (Apple, 2017) application programming interfaces (API) for parallel computing, such as OpenGL ES 2.0 and more recently, the Apple Metal. Approximately ten years ago the potential of parallel processing on mobile devices caught the attention of academia and industry, and new research and practical applications begun to appear on hobbyists and Open Source communities' websites and leading mobile platform application market places. These new parallel computation utilities and applications utilized mobile device hardware parallel CPU-GPU processing power for practical applications such as, augmented reality (AR) applications and image- and video processing applications. The smartphones were transforming into platforms suitable for serious application of *generic purpose computation on a graphics processing unit* (GPGPU).

This thesis is organized as follows: chapter 2 makes an overview to the prior research, chapter 3 describes the used research method in detail. Chapter 4 outlines the principles of parallel computation, describing the fundamentals of the hardware architectures and software programming models and implementations in mobile context. Chapter 5 summarizes the findings. Finally, chapter 6 presents the conclusions and upcoming future work.

2. Prior research

The following chapters will create a brief outlook to the topics of this review. First, parallel computing and parallel programming models are presented followed by their applicability on current mobile systems and personal mobile devices. Furthermore, digital image processing is briefly described.

2.1 Parallel computing

Universally, parallel computing drives simultaneous execution of a computer program using multiple parallel, inter-communicating processing units. The computer program under execution is devised into multiple processing units by means of parallel programming, and often autonomously by the operating system and the underlying hardware system architecture. There is a growing array of applications in need to pursue towards computing parallelism to challenge larger, more complex and ever more demanding computing problems in need to solve them faster, more cost efficiently and more productively (Almasi & Gottlieb, 1994; Krisnamurthy, 1989).

In recent decades, parallel computing paradigm has become the choice of computing for numerous branches in engineering, scientific computing and commercial applications including fluid dynamics, structural mechanics, signal processing, weather forecasting, astrophysics, nanotechnology, datamining and gaming and multimedia. Parallel computing cost benefits, tied with hard performance requirements make it often the only choice in many application areas. Despite of the steady periodic advancements in processing performance of a modern micro-processor, the stabilized and standardized parallel programming interfaces greatly favour parallelism over serial processing approach in seek for greater performance (Grama, 2013).

In the growing variety of application areas in scientific computing, parallel computer architectures and parallel computing are the only practical choice to work on specific scientific research problems; working on areas where obtaining empirical evidence is very hard – if not possible – through traditional scientific data collection methods. For example, understanding how galaxies form and evolve over long periods of time would not be possible to observe other than through massive computer simulation of the event. Similarly, parallel computing helps scientists to gather simulated data on how ocean water currents behave and evolve and interact with the atmosphere and other parts of the Earth's ecosystem when the global climate around us is changing. The size of the ocean physical simulation models alone needs massive parallelism in computation. (Culler, Singh & Gupta, 1999).

2.2 Parallel programming

A parallel programming model strictly defines its parallel behaviour and functionality on basic operations such as new task allocation and message passing and memory access, how these structures affect the computation, how they can be composed and when they can be applied on the computation (Kessler & Keller, 2007).

Traditional software programming is serial where algorithms and problems are solved sequentially in software program. Compiling and executing a sequentially written program on a uniprocessor computer exploits only some type of parallelism, typically ILP on a superscalar processing system. For a long time, evolution of processors and gained computation speed was sufficient in many applications. However, when scaling

up the performance to involve multiple processing units over distributed or shared memory systems, appropriate parallel programming models are required to achieve the increased computation resources with gained performance and the same functionality (Diaz, Muoz-Caro & Nio, 2012).

2.3 Parallel computing and programming model in Mobile Systems and Personal Mobile Devices

Many embedded devices and Personal Mobile Devices (PMDs) today are energy-efficient computers hosting multi-core CPUs and GPUs with integrated peripheral devices, cameras, sensors and rich wireless connectivity. PMD parallel computing environments share many of the same principles as those applied in the desktop and server systems. A typical PMD is powered from a rechargeable battery power source and the main system components including processing units, memory and peripheral devices are typically coupled inside a single physical device. Furthermore, there are several constraining differences between the two environments. For example, a mobile device has a battery power source, meaning the power provided for all the system components is only a fraction what is available on a desktop computer. Mainly due to this fundamental power constraint, processor cores' operating speeds, memory bus width and -speed typically fall greatly behind of those available in a desktop computer (Singhal, Yoo, Choi, Park, 2010).

Modern PMD platforms support many types of parallelism on hardware- and programming level. To reach the additional processing power provided by the mobile heterogeneous system, programming access to the device GPU is required. The GPU hardware programming interfaces on common PMD platforms, such as Android and iOS, are mainly supporting graphics programming language oriented models (Singhal, Yoo, Choi, Park, 2010).

2.4 Digital image processing

Digital image is a bounded, two-dimensional numerical representation of a real-world image. This discrete image has been constructed into pixels (individual points in two-dimensional plane). In digital image, each pixel has a discrete value of *intensity* or *gray level,* which defines the information value of the pixel. *Digital image processing* means processing of (these) digital images by means computer programs and algorithms. Digital image processing is a vast science and practice extending beyond human senses, capable of applying methods into discrete data captured on almost entire electro-magnetic spectrum from gamma- to radio waves. Digital image processing is often discussed with other image processing sciences such as *digital image analysis* and *computer vision.* One of the first practical applications of digital image processing were employed by U.S. space exploration in the mid-1960's. The available computing power and development of image processing algorithms brought the science into practical significance; distorted images from space probes could be recovered and meaningful information could be restored by means of digital image processing. Today, the science of digital image processing is vast and is providing invaluable meaning to almost any area of industry, science and human life (Gonzalez & Woods, 2017).

Gonzalez & Woods (2017) group the application of digital image processing into ten sub-categories:

1. Image acquisition
2. Image enhancement

3. Image restoration
4. Wavelets and other image transforms
5. Color image processing
6. Compression and watermarking
7. Morphological processing
8. Image segmentation
9. Feature extraction
10. Image pattern classification

Image acquisition is the process of capturing image from specific wavelength signal (e.g. visible light, x-ray, infrared). Charge-coupled device (CCD) is a common digital image acquisition component found in today's digital cameras and smartphones. A CCD captures the incoming signal at specified wavelength and ultimately converts it into numerical representation. *Image enhancement* techniques seek to improve or alter the desired properties of a captured image. For example, noise filtering is a common process to improve low-light astronomy images thus enhancing the image quality and usability. *Image restoration* is a commonly used process in for example restoration of degraded photographs. Processing the digitized photograph can remove the artefacts of the original photograph such as stains, dirt and malformations caused by material decay. *Image transformations* such as discrete Fourier transform (DFT) and discrete cosine transform (DCT) are foundational algorithms in image processing. For example, DCT is the essential procedure in JPEG image compression scheme. *Color image processing* provides methods to change digital image color spaces, change image color properties and tonal balances. Color space conversion is an essential task where e.g. RGB (red, green, blue) color model is converted to e.g. HSI (hue, saturation, intensity) model. This way, for example, the task of color segmentation benefits working with a single colour value (hue) instead of all three channel values. The process of *image compression* means, that the redundant data in the image can be discarded for storage or transmission purpose. For displaying the image, the process need to be reversed. The technology of image compression is in widespread use – in internet, digital TV and -video and smartphones. *Morphological processing*, *image segmentation and feature extraction* are selective operations that typically extract information out of the original image. Image segmentation typically requires use of morphological processing such as image dilation and erosion combined with the process of thresholding. The task of feature extraction typically starts with the results from image segmentation. For example, a set of buildings have been extracted from a satellite image and one needs to (automatically) detect all building corners and their orientation from the segmented image. *Image pattern classification* is a trending topic in image processing. The process is used to recognize desired data from digital images automatically. The classification systems can be divided into three types: 1) Prototype image matching 2) Optimal statistical formulation 3) Neural networks. (Gonzalez & Woods, 2017).

2.5 Scope of the research

This study serves as a literature part of author's bachelor's and master's theses. In those theses, the purpose was to build a real-time video effect processing application that exhaustively utilize the GPGPU programming model on Apple iOS environment. While the review of available literature seeks to be broad, specific narrowing criteria and selection is applied to limit the scope of literature to serve those aforementioned theses.

The research questions for this study are:

RQ1: What parallel computing architectures and parallel programming models are available today?

RQ2: What parallel programming models are available on mobile systems and personal mobile devices to apply heterogeneous CPU-GPU computing, and specifically on Apple iOS platform?

3. Research method

The research method for this study consisted of search of available *multivocal* literature for the topics of the research. The type of this review can be described as *generic narrative* review (Green, Johson & Adams, 2006) where the author synthesizes the text through development and evolution of the knowledge around the study topic phenomena.

3.1 Sources of literature

The following sources for literature- and information search were used:

1. *Scopus* abstract and citation database.
   Scopus was by far the dominant source for literature references and the information was most of the time available through direct links to publishers' full-text archives such as IEEE and ACM. For most of the time Scopus worked reliably.
2. IEEE Xplore Digital Library.
   If, for some reason, Scopus failed to work or was suspected to not to give correct information, direct access to IEEE Xplore interface was used.
3. ACM Digital Library.
   Same as IEEE Xplore, this database interface was used directly only if it was suspected that the Scopus did not work as expected.
4. Google Scholar search.
   If the three-aforementioned failed to give results, Google Scholar sometimes was able to find unofficial links/scans/photographs of documents of interest.
5. Oulu University Library - Oula Finna.
   Oula Finna access was used mostly in search for text- and course books available at University Library for study, and in some cases links to e-book publishers' content.
6. Google internet search.
   Many times, Google was the fastest source to find some bits of initial information on some specific topic.

3.2 Literature types and time period

The main source of literature was from scientific publisher organizations such as IEEE and ACM. However, to broaden the sources of knowledge, information from practice (e.g. programming libraries, manufacturer's references) was included in the search process. Using *multivocal literature*, it is possible to explore knowledge that would otherwise be new to have reached a publication, or would present significant and meaningful knowledge of the topic that otherwise would have been excluded from a publication (Ogawa & Malen, 1991). Generally, during the search phase most of the included literature was known to be of high quality, and in case of doubt a check was made at Julkaisufoorumi (http://www.julkaisufoorumi.fi/) for publication reputation.

Within the subject of the study the time period studied consisted the literature from 1960s to present year 2017. Some themes spanned through-out the time period providing view to changes and evolution in their contexts, while some specific themes have been under active research during only last decade and limited amount of research was found.

As a limitation, research papers concentrating on parallel image processing on mobile platforms were limited to a period of years 2013-2017, since already one review was available on the matter consisting of earlier years by Thabet et al (2014). Overview of those results shall be presented in in chapter 5.

3.3 Search process and keywords

Literature searches were conducted using the information systems listed in chapter 3.1. Since the topic categories were vast, the search process loosely followed a systematic mapping study protocol as presented by Kitchenham, Budgen & Brereton (2014). Several sub categories were identified such as *(generic) parallel computing, parallel programming, image processing.* The initial set of base documents were identified during a rigorous initial research phase using the empirically identifier article keywords, see below. The search process then continued following the *snowballing* practice presented by Wohlin (2014). In the process of *backward snowballing,* the base articles' reference lists were browsed and interesting articles were chosen by their title and publication year. In *forward snowballing,* articles containing citations made to base articles were examined for title and publication year. A great number of articles were chosen to be examined. With those articles, their abstract was read and if it contained relevant and interesting information for the topic, the paper was chosen to be examined in further detail. If it was containing interesting information article was chosen to be used in the study. The snowballing procedure iterated a number of times during the search phase until a sufficient number of documents were selected for the thesis.

During the process, citation information was exported to *Refworks* for easier follow-up on found and used articles. Furthermore, the reference sections of numerous articles were examined for further interesting article titles to be searched and studied. Depending on the literature topic, a vast amount of titles was found and author's subjective criteria was used to select the literature for the study.

The main keywords used in the search were: *parallel computing, parallel programming, image processing, signal processing, graphics processing unit, GPU, GPGPU, mobile systems, personal mobile devices*. Appropriate available Boolean operators (AND, NOT) were often used in search engines to narrow down the results.

4. Parallel computing theory

The commonly used classification of computer architectures and their data processing structure was presented by Michael Flynn, and is often referred as *"Flynn's taxonomy"* (Flynn, 1972) and the classification is still in use today in industry and scientific literature. Figure 1 depicts the following classification of processing element's capability to process program instructions and data:

- **SISD**
  Single instruction stream, single data stream. This is a *serial processor* and it is capable of executing a single program instruction and a single data element at a time. Typically, SISD approach employs no parallelism at all and is not a choice for parallel computing platform.
- **SIMD**
  Single instruction stream, multiple data stream. SIMD processor architecture promotes data processing parallelism. During the execution of one program instruction multiple data elements can be processed at a time.
- **MIMD**
  Multiple instruction stream, multiple data stream. Multiple instructions can be processed over multiple data items at one time. In practice, this means that the processing involves multiple processing units i.e. processors are working in parallel.
- **MISD**
  Multiple instruction stream, single data stream. Over one single data point, multiple instructions can be performed at one time

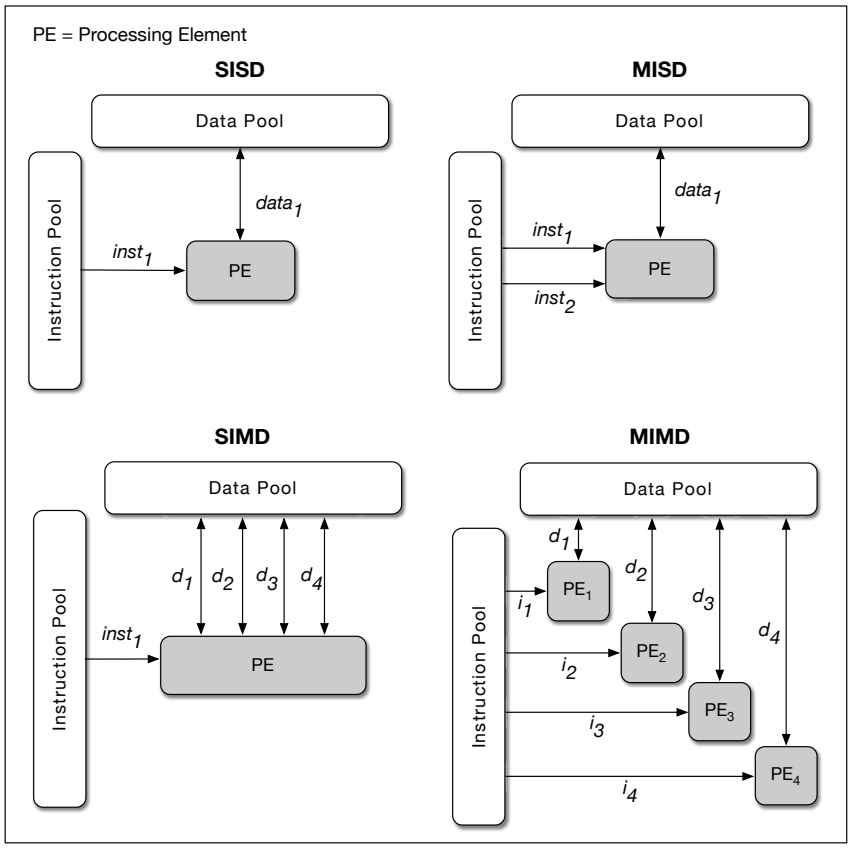

Figure 1. The classical taxonomy of computer architectures, adapted from Flynn (1972) and Ilg, Rogers & Costello (2011).

While this classification is coarse and abstract, it is established mainstay knowledge and terminology when discussing computer architectures. From these definitions only MISD does not have any modern practical implementations. Today, computer and parallel architectures are hybrids and composites of more than one class, typically SIMD and MIMD (Hennessy & Patterson, 2011). Later, on the course and evolution of parallel processing and programming technology additional classes have been proposed to the architectural taxonomy from the software viewpoint. However, an important notion is that these classes require involvement from the executing program thus it is negotiable if these are more paradigms of programming models than classes of computing architecture. For the sake of completeness, the additional suggested architectures are defined as follows:

- **SPSD**
  Single program stream, single data stream. This is equivalent to SISD hardware architecture, where computer program is progressed by sequential program stream addressing shared data source. The program may be parallelized by the functionality available in the functional unit such as *instruction level parallelism* and pipelining.
- **MPSD**
  Multiple program streams, single data stream. This model of parallel computation is rare and is typically engaged only in software testing (Limet, Smari & Spalazzi, 2015)
- **SPMD**
  Single program stream, multiple data streams. In this model, a single instruction stream (the program) is devised and executed on several *independent* processing units simultaneously each of them accessing different elements on the same data stream. (Duclos, Boeri, Auguin, & Giraudon, 1989).
- **MPMD**
  Multiple programs streams, multiple data streams. Multiple, different co-operating programs are executed in parallel each executing non-overlapping sections in the data stream. Independent programs use novel synchronization procedures to accomplish the tasks co-operatively (Limet et al, 2015)

These numerous models of computing architectures exploit parallelism in various ways, each of them enabling parallel computation with differing scalability. In current literature, there are several viewpoints and architectural paradigms how parallel architectures has been defined. Hennessy et al (2011) defined four categories of parallel architectures: 1) Instruction-level parallelism 2) Vector architectures and Graphics Processing Units 3) Thread-level parallelism 4) Request-level parallelism. The following chapters briefly outline the essential definitions of these concepts.

4.1.1 Instruction level parallelism (ILP)

SISD architecture is based on the key principles of the serial computer or, *uniprocessor* definition by Von Neumann (1945) where during execution of one program instruction, one data element is processed. Since the mid-1980s uniprocessor architectures have included techniques that have allowed pipelined overlapping in instruction execution thus providing a technique called *instruction-level parallelism* (ILP). As defined by Hennessy et al (2011) the established approaches for implementing ILP in uniprocessor designs are:

- Hardware-based techniques including dynamic-time detection of program sections that can be parallelized. This is the dominant solution used in CPUs in desktop- and server computing
- Software compiler-based techniques, where optionality for parallelism is examined during software compilation being typical for CPUs in PMD class of computing

Essential concepts in these optimization attempts include *pipelining* and *data dependency*. Pipelining is a sequential process in CPU where sequential stages of instruction fetch and decode, execution and writing the result back to memory, see Figure 2 for illustration. For maximum efficiency of the process, each of these stages need to be continuously populated by succession of the next program instruction. However, in this process it is typical that many of the instructions can be *data dependent* of the result of the previous instruction execution. If the successive command in the pipeline requires the result from the previous instruction execution, this may cause stall in the program execution (Tanenbaum, 2006).

Both techniques apply a great degree of intelligence in analysis and optimization of the program instructions towards ILP in program execution including loop unrolling, branch prediction and dynamic scheduling with renaming (Hennessy et al, 2011). ILP is strictly a hardware SISD practice occurring inside a processing unit. The increase in the CPU processing performance followed long *Moore's law* (Moore, 1998) up until 2005 when the amount of required electrical computation power and related mandatory system maintenance forced chip manufacturers started shifting towards multiprocessors designs. The era of continuous evolution in ILP in uniprocessors was coming to an end and shift towards other parallel paradigms - such as SIMD and MIMD - in seek of ever greater performance, had started (Hennessy et al, 2011).

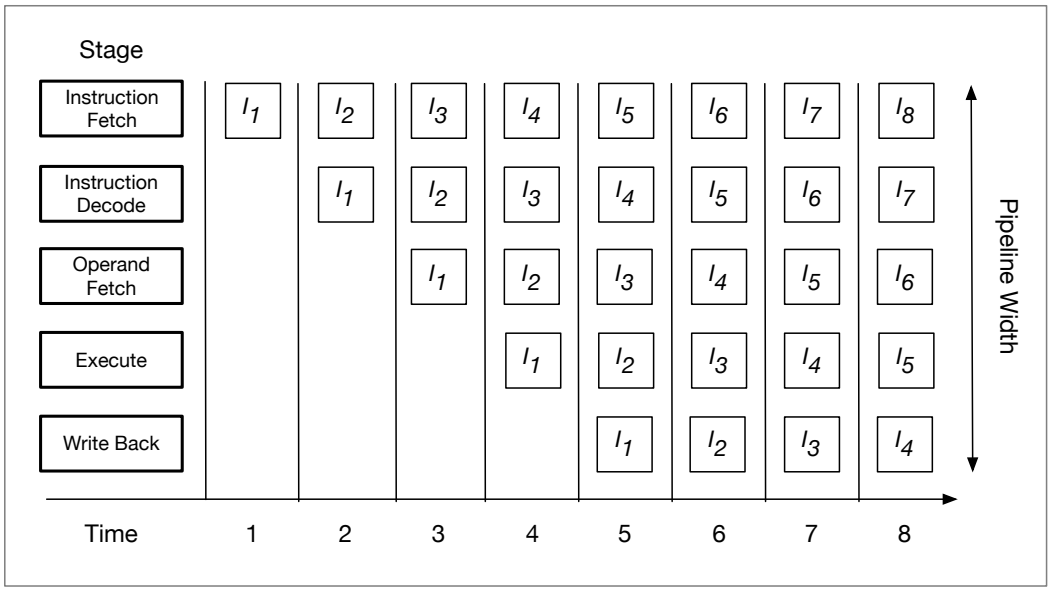

Figure 2. An example of an optimal processing unit's successive instruction and data processing stages pipeline. Processing unit progresses several instructions simultaneously during one time step. Adapted from Tanenbaum (2006).

4.1.2 Vector architectures and Graphics Processing Units

The emerging need for scientific and engineering applications requiring massive matrix and vector calculations emerged the first pipelined vector array computing architectures

in the turn of 1970s. Generally, a vector computer is a special purpose computer system designed especially for numerical calculations, namely simultaneous arithmetic and Boolean operations on vectors and matrices. The system designs were soon also incorporated to include general scalar functional units in addition to vector computation. A typical vector computer has one or more functional processors, that are capable of performing simultaneous arithmetic operations on vector data. Each processor includes multiple pipelined functional units capable of performing various arithmetic vector operations simultaneously, see Figure 3 for illustration. Vector architectures became the dominant architectures of scientific supercomputers until 1990s and were generally capable of exploiting *data-level* parallelism, and also *task-level* parallelism in its multiprocessor (MIMD) configurations (Duncan, 1990).

The principle of vector-arithmetic has been later included in many microprocessor designs as an additional functional unit. One example is the modern ARM-technology - based embedded CPU design used in mainstream smartphones such as Apple iPhones.

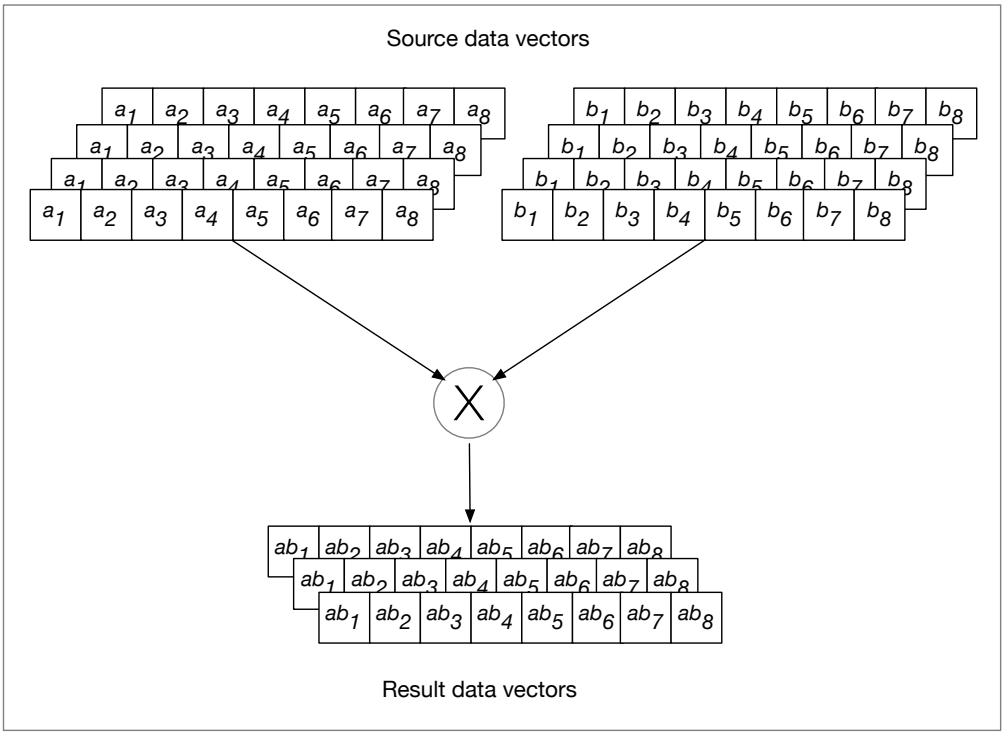

Figure 3. Vector parallel arithmetic example; two arrays of same type of data (e.g. floating-point number, integer) are computed with a single instruction.

The second, modern and *hybrid* design of the vector processor architecture is a Graphics Processing Unit (GPU) - a vector super-computer design. GPUs were long a device category designed to compute three-dimensional video game-related data on video- and computer games' game-world objects. GPUs were designed to off-load massive, often real-time computation required by the computer CPU to be performed simultaneously lifting the game experiences to a life-like levels we experience today. During the last two decades, the GPU has evolved from PC's attached graphics processing device into general purpose parallel super-computer, empowering various computing platforms from desktops and servers to mobile platforms. The main underlying principle of GPU employs efficient, scalable SIMD vector processing through massive *data-parallelism* with SPMD approach. In modern GPU, there can be hundreds of independent processing units devised to execute a single graphics shading program over multiple

data points in the same data source independently and simultaneously, making it also a *task parallel* architecture. GPU's performance shadow modern CPU counterpart in sheer computation power providing up to tens of billions of calculations per second. This is a vast performance advantage when comparing this to serial SISD serial uniprocessor computer (Owens et al, 2008).

Many modern CPU designs drive towards heterogeneous computing often implementing a CPUs and GPUs with multiple processing units each in the same device enclosure. These enclosed GPUs –often with hundreds of processing cores empower modern consumer applications and scientific computing and computation intensive desktop applications, such as video encoding, with massive hardware level TLP, while large server farms are driven by clustered multi-computer and –processor systems, and (Blake et al, 2010).

4.1.3 Thread-level parallelism (TLP)

Thread-level parallelism is a program scheduling technique where the program under execution is devised into two or more separate running units of execution (*threads*) each progressing *simultaneously* on their own processor functional unit, or *concurrently* on a uniprocessor CPU. One essential multi-threading concept is Simultaneous Multi-Threading (SMT). SMT is a processor design that contains advanced features from both the superscalar (SISD) processor functionality as well as from the multithreaded multiprocessors. It contains hardware functionality that allows single programs to execute as efficiently as on SISD processor and capabilities to manage multiple thread information of parallel executing threads thus allowing the program run faster. On a single computer, SMT speeds up program execution 2.1-times on a multiprogramming workload, and 1.9-times on parallel execution workload (Eggers et al, 1997).

4.1.4 Request-level parallelism (RLP)

Cloud computing enclosing rich digital services and vast data resources residing in the internet cloud have become today's commodity and norm. Cloud services are being used by hundreds of millions of users every day through PMDs and other digital media devices. Data centers, or warehouse-scale computers (WSC), empower vast array of services using high-performance computing (HPC) through large-scale parallelism. A typical data center has tens of thousands inter-connected host computers employing a multiprocessor architecture capable of exploiting ILP and TLP functionality. The resources of data centers can be shared between different service provider or dedicated to depending on the resource needs. Depending on a service (i.e. a web page request, a search request), simultaneously incoming service *requests* are distributed over to different hosts for simultaneous parallel processing (Abts & Felderman, 2012).

A good example of RLP is Google search, which comprises of several geographically distributed data clusters globally, each containing thousands of host computers. Each incoming request triggers a search into the keyword index followed by calculation of the hit score within the result documents, and final calculation and formation of the result page that is displayed to the user. The keyword index and result documents both comprise tens of terabytes of data that are processed through massive parallel computation (Barroso, Dean & Hlzle, 2003).

4.1.5 Parallel High-Performance Computing (HPC) architectures

During last sixty years parallel computing has evolved from novel scientific computing phenomena into a performance commodity enabling many types of user needs ranging from simple mobile video processing application speed-up on a smartphone, to an ocean current simulation running on a warehouse scale computer cluster. Design, classification and implementation of a HPC system is identified by system's *performance, deployed parallelism, control and challenges over system latency and memory distribution and commodity* of system building blocks. For example, a single SIMD processor inside a mobile device addresses different measures for latency management and different control of memory management than highly distributed massively parallel processing (MPP) system. In turn, a scientific super computer built from custom components may employ different strategy of system implementation than, for example, a massively clustered super computer built from commodity components commonly available in industry. (Dongarra, Sterling, Simon & Strohmaier, 2005).

Limet et al (2015) defined a taxonomy for parallel computing architectures grouped by hardware and software, for illustration see Figure 4. The simplest form of parallel computing system comprises of SISD scalar processor architecture driving serially programmed software. In this type of setup parallelism arises from ILP, efficient pipelining coupled with a SMT design. Modern, notable parallel architecture models – SIMD and MIMD – are grouped in three main architecture models: *shared memory, distributed shared memory and distributed memory*. Shared memory parallel architectures are typically *tightly coupled*, meaning processing units share central memory space allowing faster access to program- and processing data. This typically means that shared memory system processing units are on a same System-on-Chip (SoC), in a same computer enclosure or closely situated in a multi-machine setup. *Distributed memory* and *distributed shared memory* are the ultimate solutions of modern High-Performance Computing (HPC) architectures such as Massively Parallel Processing (MPP) systems and large computer clusters. A modern MPP or computer cluster can contain up tens of thousands of computing nodes and up to millions of processing units within the setup. Both of these systems are typically *loosely coupled* by the memory system, meaning their program and data synchronization and communication occurs through a communication network by e.g. message passing and added coupling mechanisms e.g. by the compiler or structures enabled by parallel programming language libraries. (Limet et al, 2015).

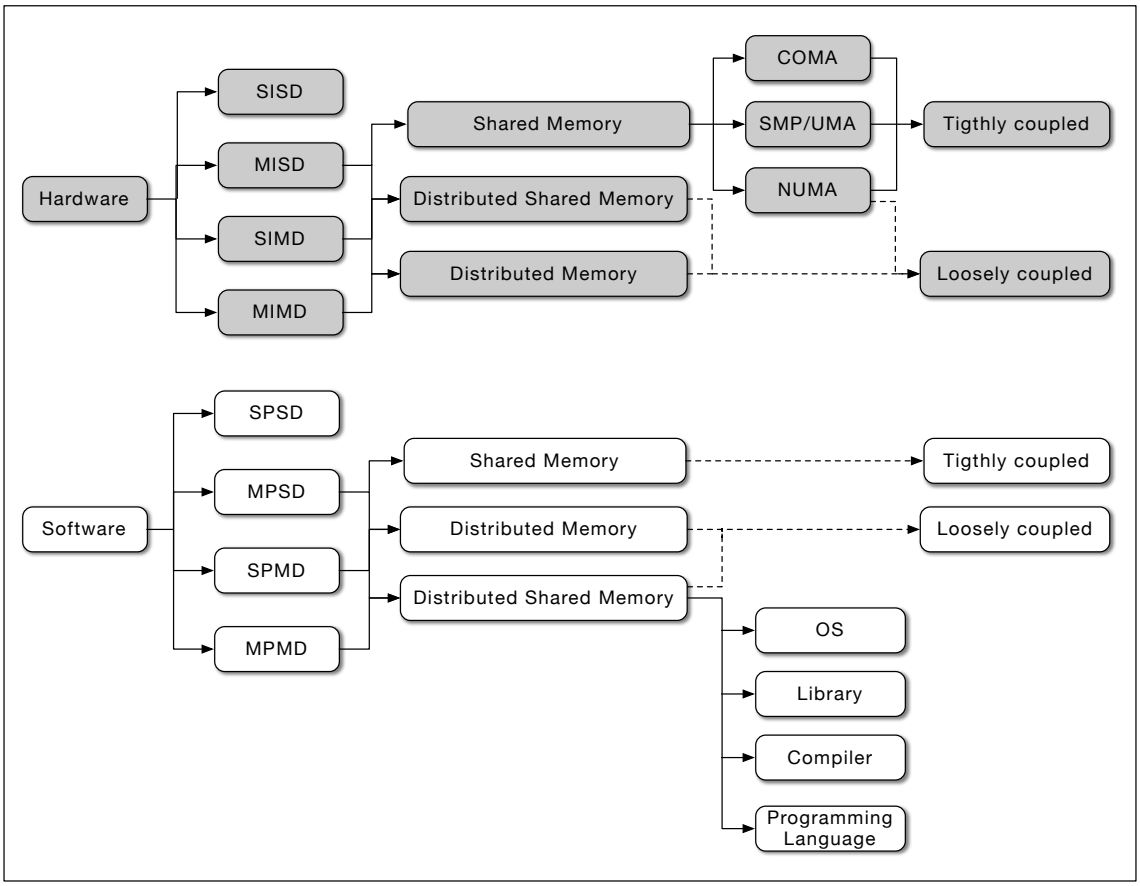

Figure 4. A taxonomy of parallel architectures grouped by hardware and software. In addition to Flynn's taxonomy, four software architectures are proposed: *1) Single program single data (SPSD) 2) Multiple programs, multiple data (MPMD) 3) Single program, multiple data (SPMD) 4) Multiple programs, multiple data (MPMD).* Figure adapted from Limet et al (2015).

4.1.6 Example applications of parallel computing

As previous chapters have demonstrated, parallel computing is a mainstay science and knowledge, and it is used across fields of science, engineering and industries. While simulations in astronomy- and computing power in medical applications provide invaluable knowledge to scientific and research audiences, the following chapters describe briefly two familiar applications of modern parallel computing.

Numerical Weather Prediction

Precise *weather forecasting* has a founding impact in peoples' lives, more so in areas where prediction and of severe weather conditions is important in avoidance and mitigation of possible upcoming damages to physical assets and materials, and even avoiding direct hazards to human life. The benefits of modern *numerical weather prediction* (NWP) outweigh the costs directed to the computing resources enabling it. The advancements in NWP during the recent decades have been acknowledged as among the greatest achievements in physical sciences. The premise and theory of the NWP already given at the turn of the twentieth century have turned into reality during last forty years with the availability of accessible and scalable HPC.

NWP is numerical computation based on mathematical models on earth's atmosphere, oceans and its affecting parameters, and data such as observational data from satellites in order to predict the upcoming atmospheric weather conditions. The major steps in

modelling involve combining the data from complex atmospheric *physical process* simulations with *ensemble models* portraying alternative, highly probable outcomes of the same data. Model *initialization* process provides the input data of setting up the simulation models with specific, corresponding regional data. *Parametrization* in turn, provides customization input such as geographical scale of the forecast and temporal scale defining whether the computed forecast covers upcoming hours or even weeks into the future. Today, NWP centers across the world provide short-term predictions several times a day and the level of service would not be possible without the help of parallel computation. For example, the leading NWP in Europe, European Centre for Medium-Range Weather Forecasts (ECMWF) employs petaflop-scale computers in providing up-to-date weather forecasts several times a day, for variable length time periods. The parallel computation performance needed involves computation ranked in Top10 of Top500 of world's super-computing service (Bauer, Thorpe & Brunet, 2015).

Google Internet Search

According to recent statistics (Internet Live Stats, 2017; Netcraft, 2017) there are 1.3-1.8 billion active webpages in the World Wide Web (WWW). These sites comprise of millions of petabytes of data. While only part of that data is available for searching, accessing this information would be difficult without the use of internet search service. Google Web Services (GWS) is one of the most known internet search services today in which approximately 3.5 billion searches are executed every day (Internet Live Stats, 2017). GWS has built the search service using tens of thousands low-end commodity-class computers distributed globally to numerous data centers running customized search software. While this deployment model to areas locally around the world distributes the computing power equally, GWS search programs also utilize effectively parallelism available on single computer in form of ILP and SMT. When a search request comes in at GWS, the software selects and directs the request to a suitable computing cluster based on the search request's geographical origin. In the first stage, a search is devised into multiple different computing nodes (*index shards*) each performing search on indexed webpage keywords comprised of hundreds of terabytes of data. Each node gets allocated a random number of index keywords dynamically, in helping to avoid downtime and interruption due to machine or network failures. The first stage outputs a list of document identifiers for the formation of the result page used by parallelized document servers. The search software requests page titles and other meta-data from document servers which load the data from the internet. Along the search request, multiple ancillary service requests are triggered inside Google's infrastructure such as requests for spell checking and ad services based on search keywords (Barroso et al, 2003).

4.2 Levels of parallel programming models

The number of various parallel programming models developed through the last forty decades is extensive. The existing models can be collapsed into a coarse grouping as displayed in Figure 5. In the model (a), the level of parallel programming abstraction is the highest, and the model hides the complexity of the parallelization at the hardware level. The model requires the least parallel programming and the gained parallelism is limited to specific hardware introduced parallelism such as superscalar execution and SMT. The model (b) exploits parallelism at programming library- and compiler level thus no specific parallel programming is required by the programmer. This model is typically used when programs are restructured and recompiled in pursue for additional performance by parallelism. Figure x (c) depicts a model where direct HW parallelism is hidden but its vast parallel computational functionality is provided through extensive

programming APIs. This model is used e.g. in GPGPU programming especially in mobile devices. The last one, figure x (d), depicts the most powerful parallel programming model. The model emphasizes the programmer's control over massive parallel computation by exploiting efficient programming libraries and tools. Examples of such libraries include MPI, CUDA and POSIX threading (Hwu et al, 2007).

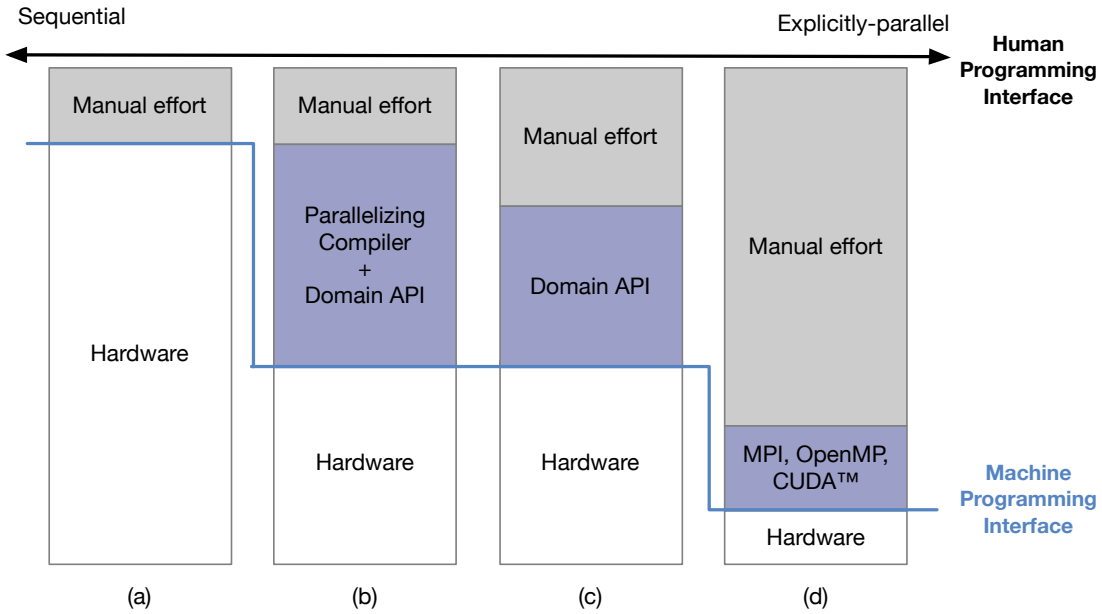

Figure 5. Parallel programming models grouped by parallel programming required by a developer: (a) Hardware introduced parallelism (b) Model with parallelizing compilers and domain API. (c) Model with extensive functional domain API (d) Explicit model, close to hardware with great control over parallelism. Figure adapted from Hwu et al (2007).

When choosing the right programming model for the task, the business domain and application of the parallel programming, available options and human and computing resources can drive the choice of an appropriate programming model. For example, implementing a distributed program over global network favors properties from another viewpoint in comparison to designing parallel algorithms for a desktop computer image processing application. There are specific *desired properties* that help to assess specific model's ability to match to a specific task or a project. McCool et al (2012) enlist the following properties important:

- *Performance*: The programming model should be performance-predictable to scale up to larger system sizes.
- *Productivity*: The model should contain sufficient tools to build performant software and monitor and debug its problems efficiently. It should provide enough productive functionality in terms or algorithms and libraries.
- *Portability*: To maximize re-use of existing software the model must be portable across variable hardware systems now and into the future.

4.2.1 Classification of parallel programming models

Parallel programming models can be classified by their essential model of programming, such as method of communication available between the computing elements, mechanisms to access the program data memory, how multiple task

management is handled, and more explicit and recent heterogeneous- and close-to-hardware models.

Message passing programming model is a distributed parallel programming model. In this model program computation processes are connected by an intermediating network or other structure, and communication and synchronization between processes occurs by passing messages. The processes typically reside physically in separate locations; they do not have global time or shared memory to access from all processes – all communication occurs by passing messages. *Message Passing Interface (MPI)* specification has become the dominant solution in distributed and HPC computing. MPI is a programming library providing extensive functionality and structures for programmers to enable scalable parallelism and is available in many favored programming languages such as Fortran and C++ (Diaz et al, 2012).

Threading programming model emphasizes task-based execution of multiple tasks simultaneously, and it is an effective shared memory parallel programming model. Thread is an independent unit of execution containing program instructions and memory structures for independent program execution by the computer CPU. Typically, threads executing on a same computer share common heap-memory allowing data sharing and synchronization between the threads. POSIX threads (IEEE, 2008) is an IEEE-standardized low-level C-programming library with a powerful programming API providing functionality and data structures to creating, managing and destroying parallel threads and their co-operation. Pthreads has long been a chosen model for task-parallel programming in shared memory systems (Diaz et al, 2012; Kessler & Keller, 2007).

Another model, *Threading Building Blocks (TBB),* is a C++ template-based library for numerous platforms in server-, desktop- and mobile computing categories. TBB extends the threading concept into *worker-tasks,* and introduces a method called *work-stealing.* This method allows parallel TBB runtime library to optimize and allocate execution load automatically between available computation resources. When the system finds processing units that are exhausted under a workload, its tasks can be divided into less busy processing units in the parallel configuration (Kim & Voss, 2011).

Shared memory programming model implies parallel execution of threads over shared memory data. *OpenMP* is shared memory application programming interface (API) providing task-level abstraction over parallel executing tasks; it is often referred to be the more sophisticated successor of Pthreads. OpenMP has been specifically designed to support creation of parallel programs, and aid in converting sequential program code into parallel code with efficient compiler pragma directives. It takes the burden of complex micro-management of individual tasks providing an efficient set of APIs, compiler directives tools and runtime support to control the program management and decomposition for parallel execution. High-level abstraction of task management, portability and support for extensive scalability makes OpenMP an especially good choice for MPP and HPC computation in shared memory systems (Diaz et al, 2012).

During the last two decades, servers, personal computers (PC), and PMDs utilizing multicore CPUs and multicore GPUs inside the same casing, and even on the same silicon die, have become more common in the computer industry. This emerging heterogeneous computer architecture has introduced a new parallel programming model – *heterogeneous programming model.* In this environment, programming model tries to harness all available system computation resources accessible through high-level functional API and tools, without the need to address paradigm specific (e.g. CPU-

specific, GPU-specific) hardware paradigms or limitations to allocate and perform computations. Computation in this model is typically managed by *host* processor which provides control over all computation *devices* in the system without distinguishing the type of the device. The parallel computation is programmed in *kernel* programs which implement the functionality to be devised to the computation devices (Diaz et al, 2012).

Open Computing Language (OpenCL) is an interface and programming standard for hardware manufacturers. The language was created by Apple Inc. and was later provided for standardization for Khronos Group. The specification defines mandatory functional requirements for which all implementing devices must support. Furthermore, optional features allow manufacturers of high-functioning devices to expose optional functionality to programmers and even unique, non-standard device-specific functionality. The specification also guarantees that once written code will compile and execute on all platforms and on later time making it very tempting in terms of portability of the software. The specification defines that an OpenCL program is executed on *a computational device* defining any device enclosed in the system (e.g. CPU, GPU, APU). The devices are further decomposed into *device cores* and those further into *processing elements (PE)* where actual parallel program *kernels* are devised to execute. OpenCL is *device agnostic* letting specific system hardware OpenCL runtime environment to compile and execute program optimally on a target system and on its PEs (Stone, Gohara & Shi, 2010).

4.3 Parallel computing in embedded and mobile systems

The GPU inside a mobile device can be used for multitude of operations such as computation intensive tasks such as game graphics handling and image processing. Regardless of the active research done in the area of parallel programming in the last decades the mixture of available programming models on mobile platforms is vast and no common, fitting high-level parallel programming solutions are available to program the new heterogeneous systems. While some research exists in integrating OpenCL programming language and its runtime environment with Android OS, yet many of the platforms, and specifically their chipset suppliers mainly support mobile parallel programming through the Open Graphics Language Embedded System (OpenGL ES) programming interface (Ross, Richie, Park, Shires & Pollock, 2014).

4.3.1 Parallel programming models in personal mobile devices

Today the most common application programming interface (API) for mobile parallel computing is Khronos Group (2016) OpenGL ES 2.0 (or higher) and many device manufacturers implement support for OpenGL ES into their devices. Using this API programmers are able to write their own user-programs, or *kernels,* using specified *OpenGL ES shading* language. Within these user-programs, programmers typically implement the functionality that is executed in parallel tasks by the graphics hardware. (Singhal et al, 2010). Current popular PMD programming platforms such as Google Android and Apple iOS support explicit parallel programming models via *threading* by platform supported libraries and frameworks, and a selection of general purpose compute- and graphics programming languages. The following sections briefly describe those additional languages and models available in the respective platforms.

OpenGL ES

OpenGL ES is graphics programming standard, and it is by far the dominant graphics programming language for mobile- and embedded devices. The standard implies a

shared memory programming model and defines the programming API used to manage the OpenGL ES program contexts and execution, and the OpenGL ES Shading Language used to write the shader programs. By design, it is a data-parallel shared memory programming model. OpenGL ES (= Embedded System) is designed for low-power systems such as mobile phones, handheld consoles, media devices and vehicles. OpenGL ES allows programmer to harness the power of underlying GPU for complex 2D and 3D data computation using effective data-parallel SIMD-computation. The GPU is accessed through rich interface API and programming of *vertex- and fragment shaders.* The shader programming language resembles C-language and is highly portable across device platforms due to its strict standard. The rendering pass of the OpenGL ES is two-phase as illustrated in Figure 6. First, a vertex shader is typically used to transform properties, lightning etc. of the 3D-objects before primitive assembly and rasterization to a 2D-plane. Finally, fragment shader program colors the rasterized object using e.g. configured data or a texture from memory (Apple, 2017; Munshi, Ginsburg & Shreiner, 2009).

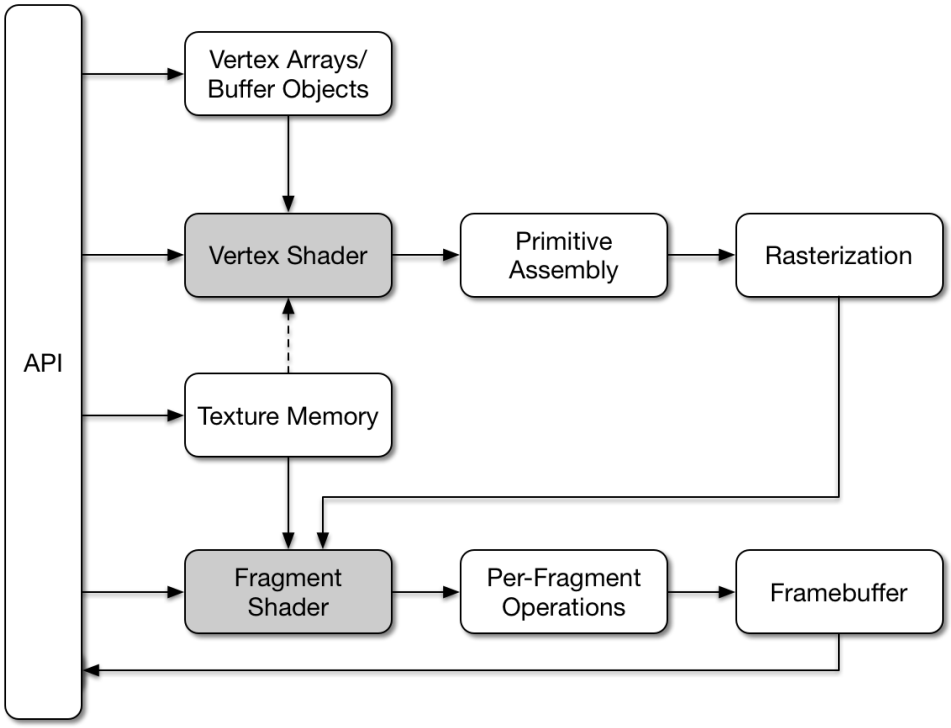

Figure 6. OpenGL ES 2.0 processing overview (adopted from Munshi, Ginsburg & Shreiner, 2008).

Apple Metal

Apple *Metal*[1] is a low-level graphics and compute API for device GPU on Apple's macOS-, iOS- and tvOS-platforms. It claims to provide high-level shared memory programming model through general applicability such as available in OpenCL programming language. Similar to OpenGL ES, it provides programmable user-programs (kernels) and efficient management API. The used programming language with Metal is C++. Being integrated into Apple's operating systems Metal gains

[1] Apple Metal framework is available for iOS devices equipped with A7 SoC or newer. Operating system release iOS 11 and onwards supports Apple Metal 2 framework.

detailed control into underlying hardware and it aims to provide more general-purpose computing than just graphics primitive manipulation. (Apple, 2017).

Google Renderscript

Google *Renderscript* is an Android operating system high performance computation framework targeted for data-parallel, intensive computation tasks such as image processing, audio processing and computer vision. Renderscript runtime environment shares and offloads the computation intensive tasks to available device processing units (CPU cores, GPU cores). Much like in OpenCL, computation is accessed through programmable *kernels* that can be written in C99-compliant programming language, and through execution management API which is available in Java, C++ and Renderscript native language (Google, 2017).

4.3.2 Research of parallel image processing on mobile systems

A number of experimental studies have been conducted on parallel CPU-GPU image processing on mobile platforms. Trending research have included topics such as exploration and *performance* comparison of various computing setups on mobile hardware such as programming models, parallel CPU-GPU programming setup against sequential CPU-only and -CPU-GPU configurations, and generic applicability of the new approach.

Thabet, Mahmoudi & Bedoui (2014) conducted a review on image processing on mobile devices viewing into serial- and parallel image processing on mobile devices. The parallel image processing methods identified were threading and CPU-GPU parallelism using OpenGL ES and OpenCL. A brief discussive summary of trend differences of that study and this will be introduced in chapter 2, while in the following sections assess other relevant developments in the mobile parallel processing.

Baek, Lee & Choi (2015) studied CPU-only and parallel CPU-GPU task allocation and configurations (*CPU-GPU sequential, CPU-GPU parallel*) on various image processing tasks using OpenGL ES 2.0. In their experimental setup CPU and GPU were allocated to process different tasks in parallel. First, the CPU decoded a H.264 image frame followed by a format conversion for GPU processor. GPU then continued to perform image processing tasks such as grayscale conversion and canny edge detection. In sequential setup, GPU did not start the work before CPU had finished the current frame. This caused unnecessary idling on both sides of the processing ends – CPU and GPU. In parallel setup, authors implemented a double-buffering between CPU and GPU. When CPU had finished conversion task, it could start the next conversion immediately after passing the buffer to GPU through API calls. Vice versa, GPU as a faster processor, did not have to wait for CPU but a new frame was available with shorter idle time than on unbuffered setups.

In another study Baek et al (2013) compared the performance of three computation configurations, *CPU-only, CPU-GPU sequential and CPU-GPU parallel*, in augmented reality (AR) feature extraction-description task using OpenGL ES 2.0. The steps required to accomplish the process were: 1) Image down-sampling 2) Grayscale conversion 3) Canny edge detection 4) Labeling 5) Contour detection 6) Rectangle check. Furthermore, in CPU-GPU tasks there was an extra image format conversion between the steps 3 and 4 performed by the CPU. For CPU-GPU parallel configuration a memory buffer solution was implemented in order to reduce idle times for both processors. The experimental results indicated that CPU-GPU parallel processing

configuration was more efficient and faster than the other two configurations in comparison on all tested image sizes (640x480, 1280x720 and 1920x1080 pixels) and the performance gain increased dramatically the more computation was required with larger image sizes.

OpelGL ES 2.0 shader programming and the split paradigm between CPU and GPU has been identified as difficult for designers and programmers. Semmo, Drschmid, Trapp, Dllner & Pasewaldt (2016) presented a research where an XML-based OpenGL ES based shader effect setup and configuration framework was designed and implemented. Using the framework programmers could create complex state-of-the-art stylization effects such as oil-painting- and water-coloring effects consisting of multiple, simpler and sequentially chained effects. The implemented framework was deployable on multiple platforms such as Android, iOS and WebCL capable of inter-operating the XML-descriptions of the OpenGL ES shaders. In the same study, a case study revealed that the user-experience on the quality of the achieved effects and filters using the framework effects was perceived as good. Furthermore, developer-experience of the framework among the students was perceived good, easing up the burden of writing OpenGL ES shaders. Developers could better focus on designing the effects and rapid prototyping. However, for real-time purposes the implemented framework was not yet providing good enough results.

The performance capabilities of new emerging programming models *OpenCL* and *RenderScript* were studied by Kim & Kim (2016) on Windows and Android platforms. In the study, the performance of the two programming models was compared using common image processing routines: matrix multiplication and transpose function on both platforms. The authors conclude, that Renderscript is able to utilize better the available computing cores and was more efficient in terms of computations speed. On Windows platform, OpenCL outpaced Renderscript being almost 10-times faster. On Android devices, Renderscript was 3-to-27 times faster than OpenCL depending on the Android device. Furthermore, authors noted, that Renderscript programming was easier and relied more on its engine's and operating system's capability to automatically share the computation load between computing cores.

In a study by Hewener & Tretbar (2015) authors demonstrated a software-based Mobile Ultrasound Plane Wave Beamforming system implemented on Apple iPad Air 2 tablet device using Apple Metal framework. Authors implemented a demonstrative software-based beamforming reconstruction solution using Apple Metal computing API. The beamforming system consisted of a compute command encoder built with Apple Metal. The system was processing high-framerate ultrafast ultrasound input data from 128 channels at 40 MHz sampling rate. While the reconstruction and visualization could be computed with less frames the computation requirement was still vast. With the system, authors were able to demonstrate that mobile software-based beamforming system can effectively reduce development costs of ultrasound devices, when the specialized hardware-based solution can be transferred to parallel computation using consumer electronics.

OpenCL is a promising portable computing standard also for mobile systems. With its more general-purpose approach to programming, harnessing it to a programmable library could even more reduce the learning curve of parallel application in programming. Cavus et al (2014) presented a study where authors implemented an OpenCL-based *image processing library* (TRABZ-10) on a test mobile system equipped with heterogeneous CPU+GPU architecture. The library implementation was

benchmarked against reputable OpenCV computer vision library. In the setup, OpenCV was harnessed to run its serial computing routines on the test system CPU, while the processing library was devised to run on GPU only. The implemented and tested routines included the commonly used operations in image processing; matrix calculations, filtering, morphological operations, transformations, geometric transformations, color conversions and feature detection. The results indicated, that most operations executed by the TRABZ-10 library were significantly faster than those executed by OpenCV. On some operations, speed-up gain was up to 8-times that of compared to OpenCV.

5. Discussion

Performance computing has permanently shifted from serial computation to parallelism (Owens et al, 2008). Parallel computing is motivated by potential performance gains thus saving time and other resources. In addition, many scientific- and application areas would not be practically applicable without the use of massive, detailed computer simulations.

McCool et al (2012) state that majority of new computers are built on parallel computing. This applies to many categories of computers including desktop- and server computers, smartphones and tablets and small single-board computers. While the high-performance cloud- and cluster computing (HPC) in recent decades have been labelled by MPP scientific- and engineering computing, the push for parallelism has transformed portable devices such as smartphones and other PMDs into powerful computers. To maximally account this gained performance, *explicit* parallel programming models are required by the software practitioners. Typically, the more explicit the model is in accessing the computation properties, the more potential performance can be achieved. Software programming in software engineering has long been serial in nature; human thinking, software designs, algorithms and routines have followed serial computation principles. During the last forty decades, a vast number practical parallel programming models have been developed for parallel computation.

Embedded- and mobile systems and PMDs are often closed and constrained systems and typically battery-powered. For long, processing performance development of mobile systems was driven by gradual progress of mobile CPU evolution and -operating speeds. The introduction of mobile heterogeneous architecture started a new era in mobile parallel computation. The modern mobile CPU+GPU setup enables great potential for speeding up computation of many common and new user-tasks people consume on their smart devices. Many tasks involving massive amounts of information such as processing of high-definition video image frames can benefit greatly from a heterogeneous computing configuration.

The outlook into state of mobile image processing was made by Thabet et al (2014). The review looked into research applying both serial- and parallel computing and the different programming models. In many investigated cases, a mobile device camera was involved showing a typical use-case scenario of a mobile user-task. Many studies were examining the user-experience and performance as well as the applicability and effort of the specific approach. The serial computation models were concentrating on *Mobile Augmented Reality (MAR)*, *Mobile Visual Search (MVS)* and *Mobile Object Recognition (MOC)*. Generally, the cases identified the applicability of the mobile platform for mobile image processing and some results indicated serial computing was applicable approach. In parallel configuration, first the *threading* programming model cases were examined with cases for *Mobile Optical Character Recognition (MOCR)* and *Mobile Food Recognition (MFR)* against respective serial programmed versions. It was notable that while some cases went to extremes in performing optimization, threading programming model was perceived to provide substantial performance gains over serial computing. For CPU+GPU models, application of OpenGL ES shared memory model was a dominant model, with only two cases of application of more higher-level OpenCL. For OpenGL ES, most of the cases were computer vision related; edge detection and feature detection algorithms. The trend in the results was, that good to great performance improvements were achieved. However, review paid no mentions

about developer-experiences were reported. Furthermore, only few experimental cases explained on OpenCL. While the studies showed the performance potential of the paradigm, low support on current PMD platforms was notable, leaving the practical choice of parallel computing to threading and OpenGL ES.

In this review, similar trends were observable. Many of the described studies in chapter 4 are still observing the applicability of a specific image processing task or algorithm computation using the chosen parallel programming method. In addition, as marked by Baek et al (2015) a notable trend was an *optimization of the communication overhead between the CPU and GPU* processors. The research on this specific topic revealed that the performance potential in the programming model itself reside specifically in avoidance and optimization of idle-time of both CPU and GPU processors.

Important considerations from software engineering viewpoint was the *portability* of the existing software across different platforms (e.g. from desktop to mobile). Image processing routines are typically written in low-level languages such as C, and even assembly in embedded systems. Converting these existing algorithms to e.g. OpenGL ES shaders may not be practical; the syntax of the parallel language may differ (OpenGL ES shading language vs. ARM assembly) and the requirements in design changes due to parallelism may suggest a complete rewrite of the operation.

The *programming expertize* from the programmers (traditional programmer vs. graphics specialist) requires special attention. An OpenGL ES 2.0 is a graphics programming interface meant foremost for transforming 3D game world objects into 2D-plane for displaying objects on a device screen. Harnessing this – yet powerful scheme – into general purpose programming is problematic. First, the learning curve of OpenGL ES is steep even for seasoned programmer. Programmer can easily become detached from the problem itself forced by the difficulty of the programming interface. Furthermore, this introduces a good programming overhead irrelevant for problem solving, but yet mandatory and complex for a software engineer to account for.

Important examples to ease the burden of parallel programming models and -languages were demonstrated by Semmo et al (2016) and Cavus et al (2014). To further diminish the effort from required explicit programming manoeuvres, configuration tools and frameworks, and summing and wrapping programming interfaces can help the developer-experience and release the developers to focus more on the real software engineering problems. This type of approach will be used in author's upcoming work where an open source image processing library will be used to speed up the development of the application.

While these studies revealed important information to the audience about properties such as computation performance and developer-experience, the research of mobile parallel programming models is still in its phase of infancy. The popular commercial PMD platforms, Apple iOS and Google Android, can benefit from higher-level parallel programming model for explicit parallelism to be coupled perhaps with the platforms' dominant programming languages i.e. Java and Kotlin on Android and Swift and Objective-C on Apple iOS. While the current, only practical choice - OpenGL ES - is powerful, it requires too much specialist knowledge even from experienced software engineers.

Specifically, on Apple iOS platform, the available parallel programming models are restricted to compiler- and HW-induced ILP, threading, OpenGL ES x.0[1] and Metal, of which the two latter can be described as a shared memory model. Out of the selected studies, only one study was using Apple Metal programming model.

This literature review revealed important knowledge on the state of research of parallel computation on mobile platforms and PMDs. There is a good amount of studies on various, generic topics that provide important knowledge of parallel computation properties on those contexts. However, the PMD- and mobile system industry and its technology development and practices are rapidly developing areas and it seems that the science community interest has not yet reached to cover in-depth studies how parallel computation power could be harnessed to empower user-contexts in best possible ways in specific use-case contexts.

An interesting area of research in PMDs is the benefits of parallel computation power in user-tasks included in popular use-cases and applications, in terms of effects on user productivity and -experience. Image- and video processing applications are a popular category in PMDs. These applications often include impressive photography- and cinematic effects that require complex, novel solutions and intensive multi-pass computation on a single image frame. Investigating these complex computation setups can give more precise and more concise answer of parallel computation in PMDs. This research viewpoint, specifically on Apple iOS platform, will be part of author's upcoming Master's Thesis, continuing on the results of this thesis.

Another viewpoint to parallel processing in PMDs is Software Engineering (SE) practices required for parallel computing, specifically the programming models and potential assistive frameworks that could ease up the development. While there are numerous programming languages and -libraries and tools in the server- and desktop environments addressing parallel computing challenges, there are only few common and portable available on PMD platforms. Are the existing models of parallel programming on current PMDs sufficient, or optimal for designer- or programmer productivity? For example, what are the experiences on OpenGL ES platform programming model on Apple iOS platform – is it perceived as too technical thus hindering creativity and productivity? To overcome these problems, there are some promising open source programming libraries and -frameworks available on Apple iOS-platform. These libraries try to hide the laborious and repetitive setup and state maintenance programming typically required with OpenGL ES thus releasing developers and designers to concentrate more on the application design problems. In author's upcoming Master's Thesis, one specific programming library framework – GPUImage[2] – shall be used in construction of the application.

[1] Apple's first device to support OpenGL ES 2.0 was iPhone 3GS in year 2009. The first Apple device that contained support for OpenGL ES 3.0 was iPhone 5s in year 2013 (and all iPhones and iPads thereafter).

[2] GPUImage, https://github.com/BradLarson/GPUImage

6. Conclusions

Parallel computation is a mainstay in Computer Science and Software Engineering. In in many areas of scientific- and personal computing, the increased computation performance will come through progressions in parallelism. CPU-GPU parallelism is a well-established practice on personal- and server computing, but a relatively new phenomenon on mobile systems. For many consumers, Personal Mobile Devices (PMD) are the new Personal Computers (PC), and computer usage-paradigm shift towards PMD is emerging, yet an evident process.

Based on recent research, there is a potential to be explored in parallel computing in these device settings, to reach towards more higher-level portable solutions. That said, portable devices are targeted for media consumption including camera and video. During recent years, while mobile systems hardware has strongly shifted towards parallelism with the introduction of heterogeneous computing architectures, the practices and technology in mobile parallel programming models are still in search for optimal practices and improvements in current solutions. The current explicit programming models on popular PMD platforms look primitive in comparison to the state-of-the-art available in desktop and server computing. Mobile platforms surely continue to evolve towards parallelism in programming with industry and communities pushing additional and improved solutions for these platforms.

In future research, author's M.Sc. thesis will look into a constructive research of image processing using parallel computation on Apple iOS platform. The research tries to answer how meaningful are the efforts put into parallel programming solutions available today and what could be improved in implementation in order to provide more tempting and generic solution for developers and designers to adapt to.

References

Abts, D., & Felderman, B. (2012). A guided tour of data-center networking. *Communications of the ACM, 55*(6), 44-51. doi:10.1145/2184319.2184335

Almasi, G.M., Gottlieb, A. (1994). *Highly Parallel Computing*. Redwood City, CA. The Benjamin/Cummings Publishing Company, Inc.

Apple Inc. (2017). *About OpenGL ES.* https://developer.apple.com/library/content/documentation/3DDrawing/Conceptual/OpenGLES_ProgrammingGuide/Introduction/Introduction.html Referenced: 21.11.2017

Apple Inc. (2017). *iOS Device Compatibility Reference.* https://developer.apple.com/library/content/documentation/DeviceInformation/Reference/iOSDeviceCompatibility/Introduction/Introduction.html Referenced: 20.11.2017

Apple Inc. (2017). *iPhone 3GS Technical Specifications.* https://support.apple.com/kb/sp565?locale=en_US Referenced: 27.11.2017

Apple Inc. (2017). *Metal*. https://developer.apple.com/metal/ Referenced: 21.11.2017.

Barroso, L. A., Dean, J., & Hlzle, U. (2003). Web search for a planet: The google cluster architecture. *IEEE Micro, 23*(2), 22-28. doi:10.1109/MM.2003.1196112

Bauer, P., Thorpe, A., & Brunet, G. (2015). The quiet revolution of numerical weather prediction. *Nature, 525*(7567), 47-55. doi:10.1038/nature14956

Blake, G., Dreslinski, R. G., Mudge, T., & Flautner, K. (2010). Evolution of thread-level parallelism in desktop applications. Paper presented at the *Proceedings - International Symposium on Computer Architecture,* 302-313. doi:10.1145/1815961.1816000

Cavus, M., Sumerkan, H. D., Simsek, O. S., Hassan, H., Yaglikci, A. G., & Ergin, O. (2014). GPU based parallel image processing library for embedded systems. Paper presented at the *VISAPP 2014 - Proceedings of the 9th International Conference on Computer Vision Theory and Applications, 1* 234-241

Culler, D. E., Singh, J. P., & Gupta, A. (1999). *Parallel computer architecture: a hardware/software approach*. San Francisco, CA: Morgan Kaufman Publishers, Inc.

Flynn, M. J. (1972). Some computer organizations and their effectiveness. *IEEE Transactions on Computers, C-21*(9), 948-960. doi:10.1109/TC.1972.5009071

Diaz, J., Muoz-Caro, C., & Nio, A. (2012). A survey of parallel programming models and tools in the multi and many-core era. *IEEE Transactions on Parallel and Distributed Systems, 23*(8), 1369-1386. doi:10.1109/TPDS.2011.308

Dongarra, J., Sterling, T., Simon, H., & Strohmaier, E. (2005). High-performance computing: Clusters, constellations, MPPs, and future directions. *Computing in Science and Engineering, 7*(2), 51-59. doi:10.1109/MCSE.2005.34

Duncan, R. (1990). A survey of parallel computer architectures. *Computer, 23*(2), 5-16. doi:10.1109/2.44900

Duclos, P., Boeri, F., Auguin, M., & Giraudon, G. (1988, November). Image processing on a SIMD/SPMD architecture: OPSILA. In *Pattern Recognition, 1988., 9th International Conference on* (pp. 430-433). IEEE.

Eggers S. J., Emer J. S., Levy H. M., Lo J. L., Stamm R. L., & Tullsen D. M. (1997). Simultaneous multithreading: A platform for next-generation processors. *IEEE Micro, 17*(5), 12-19. doi:10.1109/40.621209

Gonzalez, R. C., & Woods, R. E. (2017). *Digital image processing* (Fourth edition ed.). New York, NY: Pearson.

Google, 2017. *Renderscript Computation Framework.* https://developer.android.com/guide/topics/renderscript/compute.html. Referenced: 21.11.2017

Grama, A. (2003). *Introduction to parallel computing*. London: Pearson Education.

Green, B.N., Johnson, C.D., Adams, A. (2006). Writing narrative literature reviews for peer-reviewed journals: secrets of the trade. *Journal of Chiropractic Medicine*, 5 (3)

Hennessy, J. L., Patterson, D. A. (2011). *Computer Architecture: A quantitative approach*. Waltham, MA: Elsevier, Inc.

Hewener, H. J., & Tretbar, S. H. (2015). Mobile ultrasound plane wave beamforming on iPhone or iPad using metal- based GPU processing. Paper presented at the *Physics Procedia, , 70* 880-883

Hwu, W. -., Ryoo, S., Ueng, S. -., Keim, J. H., Gelado, I., Stone, S. S., . . . Patel, S. J. (2007). Implicitly parallel programming models for thousand-core microprocessors. Paper presented at the *Proceedings - Design Automation Conference,* 754-759. doi:10.1109/DAC.2007.375265

IEEE (2008). *Portable Operating System Interface (POSIX) Threads*. http://standards.ieee.org/findstds/standard/1003.1-2008.html Referenced: 14.11.2017

Internet Live Stats (2017). *Internet Live Stats.* http://www.internetlivestats.com/ Referenced: 4.11.2017

Ilg, M., Rogers, J., & Costello, M. (2011, August). Projectile Monte-Carlo trajectory analysis using a graphics processing unit. In *2011 AIAA Atmospheric Flight Mechanics Conference, Portland, OR, Aug* (pp. 7-10).

Kessler, C., & Keller, J. (2007). Models for parallel computing: Review and perspectives. *Mitteilungen-Gesellschaft Fr Informatik eV, Parallel-Algorithmen Und Rechnerstrukturen, 24*

Kim, S. K., & Kim, S. -. (2016). Comparison of OpenCL and RenderScript for mobile devices. *Multimedia Tools and Applications, 75*(22)

Kim, W., & Voss, M. (2011). Multicore desktop programming with intel threading building blocks. *IEEE Software, 28*(1), 23-31. doi:10.1109/MS.2011.12

Kitchenham, B. A., Budgen, D., & Pearl Brereton, O. (2011). Using mapping studies as the basis for further research - A participant-observer case study. *Information and Software Technology, 53*(6), 638-651. doi:10.1016/j.infsof.2010.12.011

Khronos Group. OpenGL ES 2_X – The Standard for Embedded Accelerated 3D Graphics. Retrieved from https://www.khronos.org/opengles/2_X

Krisnamurthy, E.V. (1989). *Parallel Processing*. Singapore: Addison-Wesley Publishing Company, Inc.

Limet, S., Smari, W. W., & Spalazzi, L. (2015). High-performance computing: To boldly go where no human has gone before. *Concurrency Computation, 27*(13), 3145-3165. doi:10.1002/cpe.3463

McCool, M. D., Robison, A. D., & Reinders, J. (2012). *Structured parallel programming: Patterns for efficient computation.* Waltham, MA: Elsevier, Inc.

Modiface (2017). Modiface. http://modiface.com/ Referenced: 21.11.2017

Moore, G. E. (1998). Cramming more components onto integrated circuits. *Proceedings of the IEEE, 86*(1), 82-85. doi:10.1109/JPROC.1998.658762

Munshi, A., Ginsburg, D., & Shreiner, D. (2009). *OpenGL ES 2.0 programming guide*. Upper Saddle River, NJ: Addison-Wesley.

Netcraft (2017). *October 2017 Web Server Survey.* https://news.netcraft.com/archives/category/web-server-survey/ Referenced: 4.11.2017

Ogawa, R. T., & Malen, B. (1991). Towards rigor in reviews of multivocal literatures: Applying the exploratory case study method. *Review of Educational Research, 61*(3), 265-286. doi:10.3102/00346543061003265

Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E., & Phillips, J. C. (2008). GPU computing. *Proceedings of the IEEE*, *96*(5), 879-899.

Ross, J. A., Richie, D. A., Park, S. J., Shires, D. R., & Pollock, L. L. (2014). A case study of OpenCL on an android mobile GPU. Paper presented at the *2014 IEEE High Performance Extreme Computing Conference, HPEC 2014,* doi:10.1109/HPEC.2014.7040987

Semmo, A., Drschmid, T., Trapp, M., Dllner, M. K. J., & Pasewaldt, S. (2016). Interactive image filtering with multiple levels-of-control on mobile devices. Paper presented at the *SA 2016 - SIGGRAPH ASIA 2016 Mobile Graphics and Interactive Applications*

Stone, J. E., Gohara, D., & Shi, G. (2010). OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in Science and Engineering, 12*(3), 66-72. doi:10.1109/MCSE.2010.69

Tanenbaum, A.S. (2006). *Structured computer organization*. Upper Saddle River, NJ: Pearson Prentice Hall.

Thabet, R., Mahmoudi, R., & Bedoui, M. H. (2014). Image processing on mobile devices: An overview. Paper presented at the *International Image Processing, Applications and Systems Conference, IPAS 2014,*

Top 500 (2017). *Statistics on high-performance computers*. http://top500.org/. Referred on 4.11.2017.

Wired (2017). *So Smart: New Ikea App Places Virtual Furniture in Your Home* https://www.wired.com/2013/08/a-new-ikea-app-lets-you-place-3d-furniture-in-your-home/ Referenced: 22.11.2017

Wohlin, C. (2014). Guidelines for snowballing in systematic literature studies and a replication in software engineering. Paper presented at the *ACM International Conference Proceeding Series,* doi:10.1145/2601248.2601268