Building an image- and video processing application on Apple iOS platform using a parallel programming model
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 and general purpose parallel computation through application programming interfaces such as OpenGL ES and Apple Metal.

Using a narrative literature review this study viewed 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). While the research on the context is an emerging topic, there still is a limited amount of research available on the topic. As acknowledged in the literature and in the practice, the OpenGL ES programming model for computing tasks can be a challenging environment for many programmers. With OpenGL ES, the paradigm shift from serial- to parallel programming, in addition to changes and challenges in used programming language and the tools supporting the development, can be a barrier for many programmers.

In this thesis a Design Science Research (DSR) approach was applied to build an artefact – an image- and video processing application on Apple iOS software platform using OpenGL ES parallel programming model. An Open Source Software (OSS) parallel computing library GPUImage was applied in the implementation of the artefact filtering- and effects functionality. Using the library, the process of applying the parallel programming model was efficient and productive. The used library structures and functionality were effectively suppressing the complexity of OpenGL ES setup- and management programming and provided efficient filter structures for implementing image- and video filters and effects. The application filtering performance was measured in real-time- and post-processing cases and was perceived as good, alongside the feedback collected from demonstration sessions and end-users.

However, designing new custom cinematic filters using OpenGL ES Shading Language is a challenging task and requires a great deal of specific knowledge of technical aspects of the OpenGL ES domain. The used programming language (OpenGL ES Shading Language) and tools supporting the work process of design, implementation and debugging of the GPU program algorithms were not optimal in terms of applicability and productivity. Findings note, that more generic and applicable language would benefit the development of parallel computation applications on PMD platforms.

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

**Supervisor**
Ph.D Candidate, MScEE, 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 literature review 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
Oulunsalo, April 25, 2018
### Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>2D</td>
<td>Two dimensional</td>
</tr>
<tr>
<td>3D</td>
<td>Three dimensional</td>
</tr>
<tr>
<td>API</td>
<td>Application programming interface</td>
</tr>
<tr>
<td>APU</td>
<td>Accelerated Processing Unit</td>
</tr>
<tr>
<td>AR</td>
<td>Augmented reality</td>
</tr>
<tr>
<td>CPU</td>
<td>Central processing unit</td>
</tr>
<tr>
<td>CUDA</td>
<td>Compute Unified Device Architecture</td>
</tr>
<tr>
<td>DCT</td>
<td>Discrete cosine transform</td>
</tr>
<tr>
<td>DIP</td>
<td>Digital image processing</td>
</tr>
<tr>
<td>DSR</td>
<td>Design Science Research</td>
</tr>
<tr>
<td>ER</td>
<td>Entity-Relationship (model)</td>
</tr>
<tr>
<td>GPGPU</td>
<td>General purpose computing on a graphics processing unit</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphics processing unit</td>
</tr>
<tr>
<td>HPC</td>
<td>High-performance computing</td>
</tr>
<tr>
<td>ILP</td>
<td>Instruction-level parallelism</td>
</tr>
<tr>
<td>MAR</td>
<td>Mobile augmented reality</td>
</tr>
<tr>
<td>MIMD</td>
<td>Multiple instructions multiple data</td>
</tr>
<tr>
<td>MISD</td>
<td>Multiple instructions single data</td>
</tr>
<tr>
<td>MPI</td>
<td>Message Passing Interface</td>
</tr>
<tr>
<td>MPMD</td>
<td>Multiple programs multiple data</td>
</tr>
<tr>
<td>MPP</td>
<td>Massively Parallel Processing</td>
</tr>
<tr>
<td>MPSD</td>
<td>Multiple programs single data</td>
</tr>
<tr>
<td>MVC</td>
<td>Model-View-Controller</td>
</tr>
<tr>
<td>NWP</td>
<td>Numerical weather prediction</td>
</tr>
<tr>
<td>OpenGL ES</td>
<td>Open graphics language embedded system</td>
</tr>
<tr>
<td>OSS</td>
<td>Open source software</td>
</tr>
<tr>
<td>PC</td>
<td>Personal computer</td>
</tr>
<tr>
<td>PMD</td>
<td>Personal mobile device</td>
</tr>
<tr>
<td>POSIX</td>
<td>Portable Operating System Interface</td>
</tr>
<tr>
<td>PU</td>
<td>Processing Unit</td>
</tr>
<tr>
<td>RLP</td>
<td>Request-level parallelism</td>
</tr>
<tr>
<td>SDK</td>
<td>Software development kit</td>
</tr>
<tr>
<td>SE</td>
<td>Software engineering</td>
</tr>
<tr>
<td>SIMD</td>
<td>Single instruction multiple data</td>
</tr>
<tr>
<td>SPSD</td>
<td>Single program single data</td>
</tr>
<tr>
<td>SPMD</td>
<td>Single program multiple data</td>
</tr>
<tr>
<td>SMP</td>
<td>Symmetric Multiprocessor</td>
</tr>
<tr>
<td>SoC</td>
<td>System on a Chip</td>
</tr>
<tr>
<td>TBB</td>
<td>Threading building blocks</td>
</tr>
<tr>
<td>TLP</td>
<td>Thread-level parallelism</td>
</tr>
<tr>
<td>UI</td>
<td>User interface</td>
</tr>
</tbody>
</table>
Contents

Abstract ................................................................................................................... 2
Foreword .................................................................................................................. 3
Abbreviations ......................................................................................................... 4
Contents .................................................................................................................. 5
1. Introduction ........................................................................................................ 6
2. Prior research ...................................................................................................... 8
   2.1 Parallel computing architectures .................................................................. 8
      2.1.1 Instruction level parallelism (ILP) ....................................................... 9
      2.1.2 Vector architectures and Graphics Processing Units ....................... 11
      2.1.3 Thread-level parallelism (TLP) ......................................................... 12
      2.1.4 Request-level parallelism (RLP) ....................................................... 12
      2.1.5 Parallel High-Performance Computing (HPC) architectures ............ 13
      2.1.6 Example applications of parallel computing .................................... 14
   2.2 Levels of parallel programming models ....................................................... 15
      2.2.1 Classification of parallel programming models .................................. 17
   2.3 Parallel computing in embedded and mobile systems ................................. 18
      2.3.1 Parallel programming models in personal mobile devices ................ 18
      2.3.2 Research of parallel image processing on mobile systems ............... 20
3. Research method ............................................................................................... 23
   3.1 Design Science Research (DSR) .................................................................. 23
   3.2 DSR Frameworks ......................................................................................... 24
   3.3 DSR Process ............................................................................................... 25
   3.4 Application of DSR in this thesis ................................................................ 26
      3.4.1 Literature review ............................................................................... 26
      3.4.2 Design process .................................................................................... 28
      3.4.3 Evaluation methods ............................................................................ 28
4. Design and development .................................................................................... 30
   4.1 Application overview ................................................................................... 30
   4.2 Application requirements ............................................................................ 32
   4.3 Application architecture overview ............................................................. 33
   4.4 OpenGL ES 2.0 data processing .................................................................. 34
      4.4.1 GPU-program flow ............................................................................. 34
      4.4.2 Processing overview ........................................................................... 36
      4.4.3 Image processing using OpenGL ES 2.0 API .................................... 37
      4.4.4 Fragment processing requirements .................................................... 39
   4.5 GPUImage library ....................................................................................... 40
      4.5.1 Structural composition of the library ................................................ 40
      4.5.2 Using filters and effects ..................................................................... 41
      4.5.3 Framebuffers and memory management ......................................... 43
   4.6 Effect design process .................................................................................... 44
5. Evaluation ........................................................................................................... 49
   5.1 Application performance ............................................................................. 49
      5.1.1 Video preview performance ............................................................... 49
      5.1.2 Post-processing performance ............................................................ 50
   5.2 Application user evaluation .......................................................................... 52
6. Discussion and conclusions ............................................................................. 53
   6.1 Literature summary ..................................................................................... 53
   6.2 Findings of the design process .................................................................... 54
References ............................................................................................................. 56
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 tablets 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 et al., 2008).

The motivation for the study emerged from author’s personal interests towards image processing on mobile devices and the ambition to build a real-time video- and image processing application for Apple iOS-platform smartphones and tablets. Furthermore, study’s contribution and relevance for author’s employer was considered important from technical knowledge- and commercial opportunity viewpoints. The smartphone application market had been booming with image processing applications for a good while and one incentive was to peek into technical- and commercial potential to join the app movement to build a productive image processing application with engaging- and
user-friendly functionality. Strictly from scientific viewpoint, the emerging modern heterogeneous CPU-GPU -parallel computing and programming, and its application in Software Engineering (SE) provided tempting properties over the general and dominant serial computing and -programming approaches.

The purpose of this study was to explore how to build an artefact, an image- and video processing application for Apple iOS software platform using parallel programming paradigm. This thesis was continuing the work on author’s bachelor’s thesis. That thesis set the groundwork as a literature review part of this thesis. The thesis is using the Design Science Research (DSR) methodology and namely the DSR Framework by Hevner, March, Park and Ram (2004). The main contribution of this thesis is the built artefact itself, and the main research question for this study was:

*RQ1: How to build an image processing application on Apple iOS platform using an OpenGL ES parallel programming model?*

A good deal of SE disciplines was required in building of the artefact. Reflection of those disciplines, problems and challenges in building the artefact contributes knowledge for the process of applying the parallel programming model on this specific platform. In that regard, an assistive research question was:

*RQ2: What are the challenges and barriers in the applied OpenGL ES programming model?*

This thesis is organized as follows: Chapter 3 describes the used research method in detail. Chapter 2 makes an outlook to the prior research describing principles of parallel computation, describing the fundamentals of the hardware architectures and software programming models and implementations in mobile context. Chapter 4 describes the design and development of the artefact. Chapter 5 presents the evaluation of the artefact, and Chapter 6 summarizes the work of thesis.
2. Prior research

First, an in-depth look is made into terminology and literature of generic parallel computing followed by definition of parallel programming models and finally their application in mobile- and Personal Mobile Device (PMD) environments.

2.1 Parallel computing architectures

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. The classical taxonomy of computer architectures, adapted from Flynn (1972) and Ilg, Rogers and Costello (2011).](image)

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.

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. 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.

### 2.1.1 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](image-url) **Figure 2.** Processor instruction pipelining using multiple stages. Adapted from Tanenbaum (2006).
Figure 2 presents an example of an optimal and successive instruction and data processing stages pipelining inside a processing unit (PU). The PU progresses several instructions simultaneously during one single time step.

2.1.2 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 2 presents an example of an optimal and successive instruction and data processing stages pipelining inside a processing unit (PU). The PU progresses several instructions simultaneously during one single time step.

2.1.2 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 depicts a vector multiplication example of two data vectors consisting of data items of same type (e.g. an integer- or a floating-point numbers). A result of vector data items multiplication is computed with a single instruction during a single time step.

The second, modern and hybrid design of the vector processor architecture is a Graphics Processing Unit (GPU) - a vector 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 on desktops and servers (Nickolls & Dally, 2010). 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).

2.1.3 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).

2.1.4 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).

2.1.5 Parallel High-Performance Computing (HPC) architectures

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).
In addition to Flynn’s taxonomy, Figure 4 depicts four software computing architectures as suggested by Limet et al. (2015). 1) Single program single data (SPSD) 2) Multiple programs, multiple data (MPMD) 3) Single program, multiple data (SPMD) 4) Multiple programs, multiple data (MPMD).

2.1.6 Example applications of parallel computing

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 examples demonstrate 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 (Bauer, Thorpe & Brunet, 2015).
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 et al., 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).

### 2.2 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, depicted in Figure 5 (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 pursuit for additional performance by parallelism. Figure 5 (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 5 (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).

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) enlisted performance, productivity and portability as important properties of a parallel programming model. First, the programming model should be performance-predictable to scale up to larger system sizes. The model should also 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. And finally, to maximize re-use of existing software the model must be portable across variable hardware systems now and into the future.
2.2.1 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).

2.3 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).

2.3.1 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 (Munshi, Ginsburg & Shreiner, 2009).

Figure 6. OpenGL ES architecture overview.

A high-level overview of the architectural rendering pipeline of the OpenGL ES is depicted in Figure 6. Conceptually OpenGL ES API and its enclosed framework binds together the CPU and GPU domains in a client-server model.

Figure 7. CPU-GPU tasks overview in an OpenGL ES system.

In Figure 7, a high-level overview of the tasks and stages of OpenGL ES command- and data procedure are presented; CPU is responsible only for task and data allocation for
GPU computation, GPU stages perform the actual work of computation leaving CPU available to other application tasks (Apple, 2017).

**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 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).

### 2.3.2 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 et al., 2014).

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 the 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 performance 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

---

\(^1\) 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.
only two cases of application of a more higher-level OpenCL. For OpenGL ES, most of the cases were computer vision related such as 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. 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.

Baek, Lee and 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 the 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 OpenGL ES 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 (640×480, 1280×720 and 1920×1080 pixels) and the performance gain increased dramatically the more computation was required with larger image sizes.

OpenGL 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, Ddllner and 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 and 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.

Hewener and Tretbar (2015) 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. The system demonstrated 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.
3. Research method

This thesis work was built upon methods and guidelines defined by disciplines in Design Science Research (DSR) frameworks and models. The following sections and chapters describe the fundamental aspects of DSR, a number of its currently popular research frameworks, the common design process and the method of conduct of the literature review and the artifact evaluation.

3.1 Design Science Research (DSR)

DSR can be defined as an empirical constructive method used in research in disciplines of Information Systems (IS), Software Engineering (SE) and Computer Science (CS). In application of DSR, the emphasis is to empirically reveal utility of new innovations. DSR aims to report how does one build and evaluate an artefact – a man-made artificial construct that has relevance and purpose for both the practice and the body of scientific knowledge (Hevner, March, Park & Ram, 2004).

While the scientific, documented process of conducting the research and development to produce the desired artefact remains in the core of the DSR process, the outputs of a DSR processes - artifacts themselves - often embody a major part of the scientific contribution produced during the process. In the field of Information Technology (IT) and engineering of the Information Systems (IS), March & Smith (1995) proposed the types of DSR artifacts as follows:

**Constructs**
Constructs can define the used vocabulary used to describe the problem or to specify the solution for the context domain. Construct define the highly specific vocabulary or terminology – the conceptualization – of the domain problem or solution. For example, a definition of a formalized modeling syntax language and vocabulary for a textual, machine-readable representation of a User Interface construct could be a DSR construct artifact.

**Models**
Models embody information of domain problems and solution statements including detailed information of construct relationships and data attribute descriptions. A model is well suited to abstract domain situations and relationships with representations. An example model artifact is an Entity-Relationship (ER) model of a proposed database system.

**Methods**
A method can be a detailed procedure to accomplish a technical task (e.g. an algorithm) or a practice to produce the desired output. For example, in case of a practice, a method could be a procedure or steps to convert given user-level requirements of desired application views into a prototype application.

**Instantiations**
Instantiations are the implemented prototypes, systems and tools.

Hevner (2007) presented the principle of DSR activity cycles, as depicted in Figure 8. The model portrays three cycles, the relevance cycle, design cycle and rigor cycle depicting the major activity contexts in DSR clearly differentiating it with from
Qualitative- and Quantitative Research disciplines. The relevance cycle acts as an initiator of the DSR process. Problem ideation and challenge identification, elicitation of central user requirements and acceptance criteria can be collected during this cycle to form the grounding knowledge of the problem identification. Furthermore, during demonstration and testing of the artefact, its relevance and fit can be rigorously evaluated inside its target context.

**Figure 8.** The three-cycle view of Design Science Research. Figure adapted from Hevner (2007).

*Rigor cycle*, as depicted in Figure 8, serves two main purposes; first, the literature search and gaining understanding of the existing state-of-the-art research in the problem context, and secondly, rigorous study of the current designs and solutions. It is essential that a researcher can explore and select innovations from routine designs and thus prepare for the design cycle (Hevner, 2007). Design cycle is the essential part of the three cycles, where iteration between the design, implementation and evaluation is the fastest. The loop iterates as long as it is seen beneficial by the researcher e.g. user requirements have been fulfilled, or the design has relevance and serves a fit customer purpose, or it is found to contribute enough new knowledge to the discipline. It is important that the design and evaluation are balanced against each other firmly enough – strong arguments for a specific design must include equally strong reasoning in the evaluation phase.

### 3.2 DSR Frameworks

There is a number of different guiding frameworks to apply on DSR work. Many of these frameworks resemble each other in multitude of ways, the newer framework models complement their predecessors’ designs and processes introducing newer specific viewpoints in the work process steps and design and emphasizing different activities in different phases of the design process. Generally, the practice advises one to select a specific framework to fit one’s working methods, or if the context problem area promotes specific methods in a specific process approach. The common DSR frameworks include,

- The General Design Cycle (Takeda, Veerkamp, Tomiyama & Yoshikawa 1990)
- Systems Development in Information Systems Research (Nunamaker, Chen & Purdín 1991)
- Design Science in Information Systems Research (Hevner, March, Park & Ram, 2004)
- Design Science Research Methodology for Information Systems Research (Peffers, Tuunanen, Rothenberger & Chatterjee, 2007)
Each framework embodies a guiding work process how to conduct DSR. The following chapter outlines the common DSR process and its essential, common activities present in most of the DSR frameworks.

3.3 DSR Process

Upon their extensive literature research on current DSR frameworks, authors Peffers, Tuunanen, Rothenberger and Chatterjee (2007) suggested a six-step process for conducting a DSR research. The following common activities in several different frameworks were introduced:

Activity 1: Problem identification and motivation
It is essential to identify the research problem and questions to justify the value of the planned solution. Building upon the purpose and relevance of the planned solution serves multiple purposes, including motivating the researcher and the audience. It is important to realize what already has been researched, what is the state-of-the-art situation in the context - this phase typically includes a rigorous knowledge research of the problem domain in form of e.g. a systematic literature review.

Activity 2: Definition of the objectives
Definition of the objectives follow closely the problem identification phase. Objectives can be quantitative e.g. measurable to the existing solutions in terms of performance and efficiency, or qualitative where researcher forms a written description how the planned solution will address the needs from practice and fulfils the requirements.

Activity 3: Design and development
In this phase, the primary focus is in actual process of creating the artefact according to the set objectives, including all design- and implementation activities. Artefact can be anything – a design, a model, an implementation – as long as it embodies the essence of the targeted research problem.

Activity 4: Demonstration
A demonstration is a proof for audience that the design works as intended and solve one or more of the research problems defined in the first activities.

Activity 5: Evaluation
This step is the analysis of the artefact design and implementation to fit for its purpose. Generally, evaluation should account for comparing the artefact properties, performance etc. against set objectives of the process. The evaluation should involve usage of proper analysis methods, metrics, tools, it also can be in form of customer feedback or survey feedback. A firm empirical or logical proof that provides the researcher a possibility to evaluate the conducted work in order to make decision to continue the search for better solution or to complete the development.

Activity 6: Communication
Communication should be done according to proper and rigorous scientific practices. The text should convey enough technical details to be repeatable by other professionals and researchers and also provide an accurate overview for non-technical readers that could utilize the information for e.g. management purposes.
3.4 Application of DSR in this thesis

This thesis applies the DSR framework by Hevner et al. (2004) and the work followed
the activities presented in the previous chapter 3.3. The problem statement, motivation
and setting of the objectives are more closely discussed in the opening chapters of this
thesis.

The produced artefact was a video- and image processing application for Apple iOS
portable devices. The relevance requirements originated from personal learning
incentives and author’s employer’s needs to explore and find feasible ways to adopt
efficient development practices in the application context and enter the market with an
initial, yet competitive application product. The process of designing the new artefact
aimed to contribute to the knowledge with the completed artefact itself and valuable
knowledge of properties of the chosen implementation method such as method
applicability and efficiency. One of the key processes was the examination of the current
state-of-the-art knowledge in the form of literature review. The following chapters define
the used literature review method, the design process and the outline of the evaluation
methods.

3.4.1 Literature review

The literature review provided this research with background knowledge for the empirical
part. The most relevant findings of the literature review are presented in section 2. 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.
The following list presents the databases and data sources used during the literature search
process:

1. Scopus abstract and citation database was by far the dominant source for searching
literature references and abstracts. Furthermore, search results often contained a
direct link to publishers’ full-text archive.
2. IEEE Xplore Digital Library and ACM Digital Library were often directly used if
the document was known to be found in these databases.
3. Google Scholar search was many times explored for alternative academic sources
for data of interest in case the two first options failed to provide document sources.
4. Oulu University Library (Oula Finna) access was the main source for printed- and
electronic text- and course books available through Oulu University Library.
5. Google internet search was the fastest source to find initial information on a
specific topic.

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). However, this study was included in the review literature contents.

**Search process and keywords**

Literature searches were conducted using the scientific information search systems available at Oulu University. These included Scopus, IEEE Xplore, ACM Digital Library and Google Scholar. 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 identified article keywords. The search process then continued following the snowballing practice presented by Wohlin (2014). In the process of backw...
3.4.2 Design process

The design process of the artifact was effectively an iterative approach alternating between design and implementation and continuous evaluation of the achieved artifact progress, as suggested by Hevner et al. (2004). The design process was empirical and explorative, using a formal, informal and novel approaches to search and solve problems encountered in the implementation process. First, the design concentrated on the application user interface (UI) basic functionality, alternating between prototype interfaces, building up the initial views and methods for changing filters quickly and intuitively, and building the basic application constructs such as settings. From there each iteration added more functionality to the application structure and gradually the work was starting to include also the filter design and implementation. Filter design, including most of the application of parallel programming paradigm, was an empirical and explorative process. Only once the artifact functionality was starting to reach practical usability and stability, more formal evaluation was performed during the design process.

3.4.3 Evaluation methods

The goal of the DSR process was a commercially available application. For the artifact evaluation purposes quantitative and qualitative evaluation criteria was defined in the beginning of the design phase of the application.

Table 1. Artifact evaluation criteria.

<table>
<thead>
<tr>
<th>Evaluation method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance measurement</td>
<td>Completed application filters and effects were benchmarked and their resource usage was measured using the available development tools on three different iPhones; iPhone 5s, iPhone 6s Plus and iPhone X.</td>
</tr>
<tr>
<td>Application requirements</td>
<td>During the development start phase, a small set of critical requirements were defined for the application. As the application development was testing the method of parallel programming, no extensive set of “hard-requirements” were seen necessary for the application.</td>
</tr>
<tr>
<td>Feedback from demonstrations</td>
<td>Application progress was demonstrated frequently to colleagues working at the same office space, and valuable feedback and comments was collected. Also, application was frequently installed on co-workers iPhones to get feedback on usability and performance on different iPhone and iPad hardware.</td>
</tr>
<tr>
<td>End-user external feedback</td>
<td>Feedback collection from end-users realized only after the application was released to the Apple AppStore. Valuable feedback and ideas for improvement was gathered using this method.</td>
</tr>
</tbody>
</table>
Table 1 outlines the used artifact evaluation criteria. The written criteria were laid out to examine the completed application properties such as performance (e.g. targeted and achieved video frames-per-second for an effect). During the application development, evaluation of the achieved progress and artifact quality was a continuous procedure taking place every time a demonstrable progress was completed. For user-experience, feedback was collected within own company colleagues, family and friends about application performance and visual effects quality. Furthermore, valuable external feedback was received from application users who downloaded the application through Apple Appstore when it was finally available.
4. Design and development

This chapter provides description of the application design and implementation.

4.1 Application overview

The result artifact of the DSR process, RECS (Boogie Software, 2018), is an image- and video capture- and processing application for Apple iOS portable devices with functionality in form of number of image and video filters and effects. The application allows user to preview filters with an intuitive User Interface (UI) and capture the photo or video, applying the filter effect directly to the captured media. The filters and effects, total of 26, vary from simple color distortion and manipulation, such as black & white filter or brown sepia, to multi-pass cinematic effects applying many types of processing operations simultaneously, such as blurring of the source image followed by number color manipulation passes, yet followed by an addition of image artefacts such as noise. In addition to the real-time capture and processing of the media, application allows also processing of user’s existing stored photos and videos, like in several similar applications available out there. The application also supports sharing user’s media to Social Media (SM) such as Facebook and Twitter. All the application filters and effects are listed in Appendix A, B and C.

Video- and photo shooting- and editing applications have long been a highly popular category in PMD platform manufacturer’s application download stores, such as Apple AppStore and Android Market. The modern powerful PMD devices embedded with many high-resolution cameras, audio capture and rich online connectivity provide an ideal- and natural platform for these applications. Advancements in UI- and usability technologies during the last ten years make using these applications on a small portable device feel natural and productive – applications that were long a privilege of media professionals working on a high-end desktop- and server computers. Now PMD users can easily enhance and edit their media on-the-go and share the results easily across various Social Media.

Aside from the main purpose of the DSR process of researching and exploiting the parallel programming model, one main target was to emphasize good overall usability following the iOS-platform’s essential usability idioms and also provide good user experience in terms of application performance. One main difference in the application structure design, in comparison to the similar available applications was, that due to the GPU-enhanced performance the effect could be applied to the photo or video in real-time during the shooting. This design choice was made on purpose based on two factors; first, and highly debatable one, was the direct application of the effect to the captured media that merged the shooting & post-processing -phases into one thus making final application of the video (e.g. sharing it to SM or by messaging) faster to access. Second factor was simply motivated by the design and development process of the artifact; new filters and effects could simply be tried and demonstrated immediately after building the artefact on the development workstation. This idiom was left into the application even though it differed from the mainstream “first capture then post-process” -practice.

The main viewfinder UI actions are defined as follows:
1. Change the filter or effect: User can change the current image filter either by swiping left or right on the screen, or by tapping a button on top of the screen.

2. Show filter settings: User can bring filter settings visible by swiping up or down on the screen or by tapping the “Menu”-button.

3. Toggle device camera: User can switch the camera viewfinder feed between device front- and back cameras by switching the camera button.

4. Capture or recording mode: User can change between photo capture and video recording modes by swiping left or right on the screen above the recording action button.

5. Encode photos and videos: User can open an additional processing view and post-process user’s photos and videos using the application.

6. Capture photo or video: Pressing the button captures a photo or starts/stops a video capture using the filter visible on the viewfinder.

7. Application settings: User can open settings to engage assistive application functions such as sharing invitations to social media or messaging applications.

Figure 9. The application main UI functions and an example filter settings view.

Figure 9 portrays the application main viewfinder- and filter settings UI providing a familiar looking touch UI controls found in many modern smartphones.
4.2 Application requirements

In the beginning of the design process a small set of central requirements was defined. Initially the target was to build an artefact, a prototype candidate testing the method of parallel programming on Apple iOS. Therefore, the need large number of directing requirements were not seen as necessary. The following core requirements were defined for the functionality, performance and usability of the application:

Functionality:

FR1: The application shall provide functionality to record video and take photos functionally in similar fashion as device’s own Camera application provides it.

FR2: The application shall provide functionality for user to post-process user’s own videos and photos.

FR3: The application shall provide video and photo filters emulating a long era of cinema and video visual quality properties such as color (e.g. black & white, old color techniques) and technical imperfections (e.g. audio-visual noise).

FR4: The application shall provide filters emulating pixelated videos and photos.

FR5: The application shall provide imaginary filters emulating some entertainment phenomenon (e.g. comic-book style, a theme from a movie).

FR6: The application shall provide functionality for user to adjust individual filter’s meaningful filter parameters in real-time.

FR7: The application shall provide functionality for user to change camera video resolution and filters must support this in their processing.

Performance:

PR1: The application shall target for filter performance of 30 fps during video and photo preview i.e. the User Interface shall remain responsive at all times.

PR2: The application shall provide filter performance higher than real-time for video post processing.

PR3: The application shall run on Apple iPhones, iPad and iPod Touch devices having back- and front facing cameras.

Usability:

UR1: The application usability shall follow Apple iOS Human Interface Guidelines.

UR2: The application User Interface (UI) shall remain responsive at all times during video and photo filter preview, video recording and when device is post-processing and saving a video-file.
4.3 Application architecture overview

The application was implemented using Apple’s development software and tools including Apple iOS Software Development Kit (SDK). Essential tools include numerous software simulators (all iPhones and iPads) of devices supporting the iOS’ minimum OS-version requirement. The iOS SDK native development is supported in Objective-C and Swift-languages but also support for C- and C++ - programming languages is included with some of the system frameworks. The application covered in this thesis was mainly implemented in Objective-C but with some of the audio processing was done with C-language.

Figure 10 outlines the application main use cases with a UML component diagram depicting the main roles of the application components in the processing chain:

Use Case 1: Browse and preview effects on screen

The application main view displays a live preview of device’s selected built-in camera processed by the chosen filter. By swiping left or right on the application main UI user can change the filter applied currently on the camera image data.

Use Case 2: Take photo or record video

In addition to Use Case 1, user can tap the capture button on bottom of the device screen UI to capture a photo or start a video recording applying currently active effect on the media. The media is stored in device persistent memory.

Use Case 3: Post-process user’s photo or video

The media browser view begins by displaying user’s all videos and photos in the device and in the Apple’s cloud infrastructure. By selecting a media, a processing view is displayed where user can browse, preview and apply an effect on the chosen media.

The basic application structure was forced by the standard iOS-programming paradigm where the application structure and its building components dictate the use of model-view-controller (MVC) paradigm. While the core application functionality is driven on the CPU, all of the application photo- and video effects processing is explicitly offloaded to the GPU-domain through the OpenGL ES programming API. This functional separation of processing not only added challenges to the application’s technical design but also shifted the whole paradigm of problem implementation from serial computing to parallel computing, as intended in the thesis motivation and setting of the objectives. While adding structural and architectural complexity to the application design and development, the CPU-GPU separation introduced challenges in common development practices such as step-debugging of the source code. Architectural and tool issues will be discussed in thesis’ Findings section.

During the initial problem identification phase, an Open Source Software (OSS) library – GPUImage - was researched and chosen to help reduce the programming overhead between the CPU- and GPU-domains and ease the design and development of GPU image processing tasks. GPU-image library will be described in chapter 4.5.
4.4 OpenGL ES 2.0 data processing

The following sub-sections describe the principles of OpenGL ES 2.0 processing applied in the application context.

4.4.1 GPU-program flow

Here, the outline of the essential OpenGL ES 2.0 data processing concepts are briefly introduced. To perform a computing task on a two-dimensional array such as a digital image on GPU-hardware, an application offloads both the computing task/algorithm and data to the GPU through OpenGL ES programming API. The management of the task is performed by a GPU-program object attached with respective, compiled programming constructs performing the actual computing algorithms on the GPU. Munshi et al. (2009) define steps for executing a single image processing task as follows:

1. Create and compile vertex- and fragment shader objects with their embodied source code. A shader object embodies the (compiled) programming logic of a
computation task. Shaders are written in OpenGL ES 2.0 shading language which is a C-like programming language. For example, in image processing task, the main programming logic resides in a fragment shader, where target pixel color is typically computed.

2. Create and link a program object with attached shader objects. A program is an object of execution unit in GPU. The program is simply a parametrized programming construct and manages the GPU resources during the time of vertex and fragment shader execution.

3. Use program and queue data to vertex and fragment parts of the program. Once program with its attached resources has been successfully constructed, the main application can set up the GPU-program ready for execution. The data (e.g. an image) to be processed- and program parameters (uniforms) are sent through the OpenGL ES interface to the program’s vertex and fragments shaders, and source image is loaded into the program’s texture memory.

4. Execute program. Very shortly after data and parameters have been passed to the GPU and the execution has been queued, the program runs using all available parallelism in the GPU. Typically, hundreds of independent threads execute the same (fragment) program simultaneously over different source data items.

5. Utilize/read back the program result. Once GPU program completion has been signaled, the main application reads the data back to the application, or in the case of multi-pass processing, leave the result of the execution reside on the GPU-domain for the subsequent GPU program to utilize it as the source data.

![Figure 11](image.png)

**Figure 11.** The basic procedural steps of an OpenGL ES program execution flow.

An overview of a GPU program processing steps and respective OpenGL ES API function calls are depicted in Figure 10.
4.4.2 Processing overview

The OpenGL ES standard is an application programming interface designed primarily for processing of advanced 3-dimensional (3D) game graphics. Using this same interface for processing 2-dimensional (2D) data arrays (e.g. digital images) is possible due to the flexibility of the design of the interface introduced in OpenGL ES 2.0. The GPU-program processing pipeline consists of two programmable steps, the vertex- and fragment shader programs. In brief, the vertex shader is typically used to processing and projecting the 3D game world object vertices on a 2D plane to be displayed i.e. on the device screen. Following the vertex shader and the subsequent hardware rasterizer stage, the fragment shader stage colors or “skins” the game world objects. The fragment shader can either directly apply a desired color to the output pixel or sample the from a graphics texture image passed in as a fragment shader argument from the CPU domain application.

<table>
<thead>
<tr>
<th>Input data to GPU program</th>
<th>Vertex Shader</th>
<th>Rasterizer</th>
<th>Fragment Shader</th>
<th>Output Framebuffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program 1</td>
<td>Draw vertices; two triangles forming an image quad</td>
<td>Rasterize image pixels (fragments) to be processed</td>
<td>For one pixel, sample color from input texture and process</td>
<td></td>
</tr>
<tr>
<td>Parameters</td>
<td>Image to be processed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parameters</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Input Texture</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 12.** A simplified flow of GPU-program execution in an image processing- and effects task.

To fit this GPU processing scheme into a digital image processing context, Figure 12 depicts an overview of the task phases. First, a vertex shader task defines an area to be processed, i.e. typically an image rectangle. First, an image processing task vertex shader defines (draws) the four corner vertices (0,0), (1,0), (0,1) and (1,1) respectively, a quad made of two triangles forming a canvas for an output image. Thereafter, a GPU system rasterizer fills the space between each corner vertex by creating the fragments (i.e. screen
pixels) by interpolating from the corner vertices’ coordinates. After rasterization, the GPU hardware fragment processor invokes a fragment program for each fragment, in parallel fashion. Each fragment program outputs and contributes a single pixel to the output image.

4.4.3 Image processing using OpenGL ES 2.0 API

Figure 13 depicts the OpenGL ES API setup calls required to process a single input image frame through the application’s image processing structures. The setup flow reflects a high-level procedure of the function calls implemented in the GPUImage-library, omitting some details for the sake of notation clarity.

![Figure 13](image)

**Figure 13.** The processing of a single image frame on GPU using OpenGL ES API.
Reflecting the processing scenario in Figure 13 to the application’s main use cases depicted in Figure 10, the process starts from an entry of an input image frame from a data source. The source can be either an Operating System (OS) AVFoundation framework feeding camera image frames to the application at preset interval (e.g. 30 fps). The source can also be a still image or video from user’s personal assets such as device- or internet cloud storage. Rather than passing the image reference between the application entities, the input image is transferred directly to the GPU memory space and referred with its texture identifier in successive steps.

The identifier is then passed to a filter group structure. Filter group is a programming- and memory construct, a composite of one or more filters to be executed in sequence. Each of the filters inputs a texture as an input parameter, either from the initial source or the preceding filter, and outputs a texture as a result of the filtering process. As depicted in Figure 13, each of the filters perform the steps to bind resources to the rendering context, here denoting the step numbers from the figure:

7. **Activate program.** Upon first instantiation of the filter, its GPU-program is created, vertex- and fragment shaders are compiled and linked to the GPU-program structure. Thereafter, the GPU-program is stored to a program cache for iterative reuse.

8-11. **Prepare framebuffer.** The `glBindFramebuffer` attaches a GPU-memory structure to the GPU-program holding information about the size of the image to be processed and the actual texture memory attached to the framebuffer. For Figure 11 clarity, this step is simplified in the image; what is omitted from the figure, is a Frame Buffer Cache (FBC) – a structure for management of shared output framebuffers. FBC will be discussed in Chapter 4.5 subchapters. The `glViewport` command sets the size and orientation of the output buffer followed by `glClearColor` and `glClear` which initialize the framebuffer to its initial color.

12-13. **Prepare source image.** The `glActiveTexture` function assigns a hardware texture unit for the subsequent call to `glBindTexture` which in turn attaches the input texture identifier to that specific unit. This operation allows the texture to be used from within the fragment shader using a `sampler2D` data type.

14-15. **Pass fragment shader data.** The `glUniform***` functions are used to pass various type of data directly to fragment shader program variables, including the input image texture identifier used for sampling the image pixel colors.

16-17. **Pass vertex shader data.** The `glVertexAttribPointer` delivers vertex data or four corners of the image and also the four corner points of the fragment shader input texture used to interpolate the fragment coordinates by the hardware rasterizer.

18-20. **Program execution.** The `glDrawArrays` commands the GPU to execute the program stages. Upon completion, filter signals the next filter in the group and process continues from step 7. When last filter has executed, the final output frame will be displayed and/or written to file.
4.4.4 Fragment processing requirements

A central design challenge and requirement was that the video preview in video- and photo capture use cases should provide a smooth user-experience in terms of filter processing. As a design goal for this requirement, application filters and effects would process incoming data 30 frames-per-second to achieve this. The properties affecting this were the size of the input image frame and the complexity and length of the processing filter group. Table 2 summarizes the fragment computation requirements for the input image frame sizes (width x height), per each filter and their sizes (number of filters in group) and the required number of fragment operations for 30 fps.

Table 2. Fragment processing requirements for application video- and image filters.

<table>
<thead>
<tr>
<th>Filter</th>
<th>Size</th>
<th>Million fragments-per-second for 30 fps</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>352x288</td>
</tr>
<tr>
<td>Wild West 1899</td>
<td>5</td>
<td>-</td>
</tr>
<tr>
<td>New York 1905</td>
<td>5</td>
<td>-</td>
</tr>
<tr>
<td>Hollywood 1929</td>
<td>4</td>
<td>-</td>
</tr>
<tr>
<td>Chicago 1938</td>
<td>4</td>
<td>-</td>
</tr>
<tr>
<td>Pinhole 1893</td>
<td>7</td>
<td>-</td>
</tr>
<tr>
<td>Montreal Sun 1945</td>
<td>5</td>
<td>-</td>
</tr>
<tr>
<td>Noir 1947</td>
<td>5</td>
<td>-</td>
</tr>
<tr>
<td>London 1956</td>
<td>5</td>
<td>-</td>
</tr>
<tr>
<td>San Francisco 1964</td>
<td>4</td>
<td>-</td>
</tr>
<tr>
<td>Paris 1978</td>
<td>4</td>
<td>-</td>
</tr>
<tr>
<td>Halftone 1970</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>Pola 1981</td>
<td>5</td>
<td>-</td>
</tr>
<tr>
<td>Moon 1969</td>
<td>6</td>
<td>-</td>
</tr>
<tr>
<td>Moon 1972</td>
<td>6</td>
<td>-</td>
</tr>
<tr>
<td>Mars 1976</td>
<td>3</td>
<td>-</td>
</tr>
<tr>
<td>VHS 1984</td>
<td>9</td>
<td>-</td>
</tr>
<tr>
<td>Toy-Cam 1987</td>
<td>4</td>
<td>12.165</td>
</tr>
<tr>
<td>Commodore 64</td>
<td>3</td>
<td>-</td>
</tr>
<tr>
<td>Apple II</td>
<td>3</td>
<td>-</td>
</tr>
<tr>
<td>ZX Spectrum</td>
<td>3</td>
<td>-</td>
</tr>
<tr>
<td>VIC-20</td>
<td>3</td>
<td>-</td>
</tr>
<tr>
<td>BBC Micro</td>
<td>3</td>
<td>-</td>
</tr>
<tr>
<td>The Matrix</td>
<td>4</td>
<td>-</td>
</tr>
<tr>
<td>Teletext</td>
<td>3</td>
<td>-</td>
</tr>
<tr>
<td>LED Scoreboard</td>
<td>3</td>
<td>-</td>
</tr>
</tbody>
</table>

As depicted in Table 2, all filters support video frame sizes of 640x480 (width x height) and 1280x720, except one, supporting only a frame size of 352x288.
4.5 GPUImage library

GPUImage is an Open Source Software (OSS) video- and image processing library for Apple iOS- and macOS platforms. The library enables massive parallel computation through the use OpenGL (macOS) and OpenGL ES x.0\(^1\) (iOS) programming interfaces. The library is licensed under a permissive Berkeley Software Distribution (BSD) license and is free to use for individuals and enterprises in their software products (Larson et al., 2016). Since the introduction of the first release in 2012, the library has been authored, developed and maintained by its author and a group of contributors. Due to great performance advantages, the library has gained popularity among the Apple platform developers.

A powerful feature of the library is that it hides the laborious setup- and configuration operations often typical for OpenGL- and OpenGL ES programming disciplines. The library provides easy-to-use APIs for constructing image filters and filter groups, allocation of the camera- and device data resources, execution processing of the filters and displaying and writing the results into device screen and persistent memory. Another important feature of the library is that the filter implementations are based on standard OpenGL GLSL and OpenGL ES x.0 shading languages. This paradigm allows developers – with a little effort - to recycle their existing shader implementations and build upon a great number of example- and reference shader implementations found in the literature and online- and community resources.

Two versions of the GPUImage library exists: the original GPUImage, introduced in 2012, is implemented completely in Objective-C programming language. It is available for Apple’s macOS and iOS platforms. The newer version, GPUImage 2 was introduced in 2016, created by the author of GPUImage, and is written in Swift programming language. The library can be utilized in computing environments that support the Swift programming language and OpenGL/ES x.0 programming interface including Linux OS and some microcomputer systems. Both versions of the library provide near identical functionality partly differing in the implementation language and -specifics and optimization maturity of the library internal structures. This thesis is referring to the original GPUImage.

The following chapters describe and illustrate basic components and structures of GPUImage, how the library processing operations take place and a description of library components’ extensibility.

4.5.1 Structural composition of the library

GPUImage is implemented in Objective-C language and can be included in application development as an Objective-C framework library. The library contains functional wrapper application programming interfaces (API) for encapsulating larger structural- and operational OpenGL ES functionality inside a single API-call. For example, a creation and initialization of an image filter object performs all the required OpenGL ES function calls to setup a glProgram ready for an immediate execution.

The library consists of the following types of structural components for managing the parallel computation:

---

\(^1\) The first Apple iOS-device supporting the OpenGL ES 2.0 was the iPhone 3GS in 2009. The Apple A7 system chip, in year 2013 was the first Apple iOS system chip supporting the OpenGL ES 3.0.
• **GPU-program and EAGLContext management**: GLProgram encapsulates structures and functionality to create, compile and link OpenGL ES programs with their attached resources. GPUImageContext class manages the necessary EAGLContext responsible for GPU-device drawing and memory resource management (i.e. textures and framebuffers).

• **Framebuffer and framebuffer cache management**: A single filtering pass of a multi-stage image filter group requires an output frame memory for each stage output. Summing up this up to 30 times a second requires and effective memory caching and recycling system.

• **Sources**: Sources refer to a group of GPUImage-library classes that bind the source image data into the OpenGL ES pipeline. These sources include the device camera (GPUImageStillCamera and GPUImageVideo camera, respectively), a videofile, (GPUImageMovie), a photo (GPUImagePicture) and an iOS User Interface (UI) class (GPUImageUIElement).

• **Filters and processors**: The library provides various filter design structures such as one-, two-, three- and four-input texture filters, single- and multipass filters and a filter group. Also, the library contains numerous image processing filters for the following image processing categories: color processing such as GPUImageBrightnessFilter and GPUImageContrastFilter, image processing such as GPUImageGaussianBlurFilter and GPUImageLowPassFilter, blends such as GPUImageAddBlendFilter and GPUImageMultiplyBlendFilter and effects such as GPUImagePixellateFilter and GPUImageToonFilter.

### 4.5.2 Using filters and effects

GPUImage library eases up the management of the OpenGL ES contexts allowing programmers and designers to concentrate more on working on the filter- and effects design. For example, the following simplified code-excerpt portrays a procedure of creating a Black & White TV filter group, a composite of multiple sequential filters comprising and emulating a monochrome color TV picture with adjustable tonal properties and some artificial image artifacts such as vertically moving “scanlines”:

**Effect header file:**

```objective_c
@interface GPUVideoBWTVColorFilter : GPUImageFilter
@property (readwrite, nonatomic) CGFloat brightnessValue;
@property (readwrite, nonatomic)CGFloat contrastValue;
@property (readwrite, nonatomic) CGFloat noiseValue;
@end

@interface GPUVideoBWTVFilter : RC_FilterGroup
@property (readwrite, nonatomic)GPUVideoBWTVColorFilter *grayFilter;
@property (readwrite, nonatomic)GPUImageGaussianBlurPositionFilter *blurFilter;
@property (readwrite, nonatomic)GPUImageLine2Generator *lineGenerator;
@property (readwrite, nonatomic)GPUImageAlphaBlendFilter *sumFilter;
@property (readwrite, nonatomic)GPUImageVignetteFilter *vignetteFilter;
@property (nonatomic) GLfloat vignette;
@end
```

**Effect implementation file, initialization:**

// 1. Monochrome filter
self.grayFilter = GPUVideoBWTVColorFilter.new;
self.grayFilter.noiseValue = 0.3;
self.grayFilter.contrastValue = 0.71264;
[self addFilter:self.grayFilter];

// 2. Blur filter
self.blurFilter = GPUImageGaussianBlurPositionFilter.new;
[self blurFilter setBlurSize:0.32];
[self addFilter:self.blurFilter];

// 3. Noise generator
self.lineGenerator = [[[GPUImageLine2Generator alloc]
initWithLineType:NoiseBWTV];

// 4. Blend filter for blur and monochrome
self.sumFilter = GPUImageAlphaBlendFilter.new;
[self.sumFilter setMix:1.0];
[self sumFilter disableSecondFrameCheck];
[self addFilter:self.sumFilter];

// 5. Smooth-corner filter
self.vignetteFilter = GPUImageVignetteFilter.new;
[self addFilter:self.vignetteFilter];
[self.vignetteFilter setVignetteStart:0.75];
[self.vignetteFilter setVignetteEnd:1.00];

// Set routing, group start- and termination filters
[self.grayFilter addTarget:self.blurFilter];
[self.blurFilter addTarget:self.sumFilter];
[self.lineGenerator addTarget:self.sumFilter];
[self.sumFilter addTarget:self.vignetteFilter];
self.initialFilters = @[self.grayFilter];
self.terminalFilter = self.vignetteFilter;

Application Main View Controller:

self.filter = GPUVideoBWTVFilter.new;
[self.activeCamera addTarget:self.filter];
[self.filter addTarget:self.filterView];

The previous code-excerpt depicts the procedural steps, yet in greatly simplified form, how programming structures of GPUImage-library help and compact the instantiation and setup of a filter.

The following code shows the fragment shader code of a one filter group filter, the GPUVideoBWTVColorFilter. The varying data type is passed, computed and interpolated through vertex shader and system rasterizer, the uniform variables are passed directly from CPU-side application using glUniform*** OpenGL ES API calls:

Fragment shader:

NSString *const kGPUVideoBWTVColorFragmentShaderString = SHADER_STRING
{
precision highp float;

varying highp vec2 textureCoordinate;
uniform sampler2D inputTexture;

uniform float brightnessValue;
uniform float contrastValue;
uniform float noiseValue;

void main()
{

float SepiaValue = 0.6;
vec3 finalColour;

// Mono RGB value
vec3 sepia = vec3(112.0 / 255.0, 66.0 / 255.0, 20.0 / 255.0);

// Step 1: Sample source pixel and convert to grayscale
vec3 colour = texture2D(inputTexture, textureCoordinate).xyz;
float gray = (colour.x + colour.y + colour.z) / 3.0;
vec3 grayscale = vec3(gray);

// Step 2: Apply sepia overlay
finalColour = grayscale;

// Step 3: Tune final sepia colour
finalColour = grayscale + SepiaValue * (finalColour - grayscale);

// Step 4: Add noise
finalColour += noiseValue;

// Step 5: Apply contrast here
finalColour = (finalColour.rgb - vec3(0.5)) * contrastValue + vec3(0.5);

// Step 6: Apply brightness
finalColour = finalColour.rgb + vec3(brightnessValue);

// Apply to fragment color
gl_FragColor.xyz = finalColour;
gl_FragColor.w = 1.0;
}

The fragment program samples the input pixel color from the input texture using the fragment coordinate received from the vertex shader and the rasterizer. Thereon, user-entered values of brightnessValue, contrastValue and noiseValue contribute to the fragment color.

Many library filters provide similar interfaces for adjusting the basic filter parameters for controlling a specific filter property e.g. the level of brightnessValue in the above example. Furthermore, following the library filter’s implementation, customizing new parameters can be accomplished by adopting the library examples. Library provides numerous filters for combining and chaining up two or more filters into more complex composite filters and using their built-in fragment shader uniform-variables to customize filter behavior and visual properties. To design new or customized filters, programmer can utilize the implementations provided in the library filters and e.g. elaborate from customizing thereon.

4.5.3 Framebuffers and memory management

Processing a video stream image frame requires the image frame to be held in memory for the processing time. For practical reasons, dynamically allocated heap memory could be considered as the only choice for efficient memory allocation strategy. A digital image is formed of continuous memory array of 32-bit pixels holding four 8-bit values of red-, green-, blue- and alpha-values. That implies, that an image size of 640x480 pixels requires 640x480x4 bytes of memory totaling 1228200 bytes (1.2 MB) and 1280x720 image consumes 1280x720x4 bytes i.e. 3686400 bytes (3.6 MB) of memory for a single frame. To put these consumption requirements into perspective in 30 fps processing case,
the amount of memory needed during one second is 36 MB for 640x480 image size, and 108 MB for image size of 1280x720 in cases where each image frame would be processed by a single pass filter. As all of the application filters were composed of at least 3 stages and 9 stages at most, the memory consumption would multiply respectively.

In an ideal memory management strategy, an application would reserve and free memory per need basis, frame by frame thus totaling the total consumed device memory to zero after the processing has been finished. However, in practice, allocation and freeing memory is not a resource free operation but requires memory management from the underlying operating system (OS) services. On iOS freeing memory after processing would crawl behind the allocation quickly running into case where application’s reserved memory at one single time instant exceeds the OS maximum allowed i.e. 50% of RAM available on the specific device hardware. On such occasion, an iOS application would be terminated by the OS.

To overcome the aforementioned and potential memory exhaustion, GPUImage implements a novel cache system of recyclable output texture memory buffers. Upon creation or activation, a filter program would request a buffer from the FrameBuffer Cache (FBC) – if no suitable buffers is available, a new buffer would be created. Once processing has been completed and the result used by the successive filter or entity, the buffer would be returned to the FBC. The cache also includes a timely purge system to avoid the cache of growing too big. The texture memory buffers are held in GPU-memory space, and the CPU-side application holds only reference identifiers to the buffers.

4.6 Effect design process

With only some experience on serial digital image processing algorithm programming on small scale hobby projects, author’s application of parallel GPU fragment shader algorithm programming was an empirical try-and-experiment process. While the initial purpose was to build “a sufficient” set of filters and effects in the application, the filter requirements kept on stretching during the process and additional features were built on the effects gradually during the design iterations.

As an example, for the “VHS 1984” filter the information gathering was a novel, experimental process. The purpose was to create an audio-visually authentic filter emulating infamous and recognizable properties of a worn VHS-tape VCR playback. These artifacts included a static “sparking” noise when playing back a worn video tape, an unstable- and unbalanced color presentation (color bleeding) in image frame objects, horizontally and vertically unstable (“shaking”) image frames, and partially blurred and sharpened regions in the image. While author was a person lived through the VHS-period of 1980s to early 2000s, no actual working VHS VCR, nor videotapes were available for studying the tape playback artifacts. Instead, a number of topic related keyword searches in YouTube produced a sufficient number of videos as learning material.

There was no intention to dive deep into the technical specifics of VHS VCR systems causing specific artefacts to the visual image and emulating these in the software. Rather, the artefacts were examined empirically and visually with sufficient analysis of possible technical issues contributing to those issues. First, in the studied material, VHS playback visual quality was low. The video images were soft and unrest thus there was not much object details visible. In some videos image scanlines were slightly “jumping” making image look unstable. Generally, playback image color saturation was low and the image frame colors looked “washed out”. Other types of artifacts were faults that were not in
the original video such as random color bursts - a horizontal band of random color would flash on screen. The other common fault was white colored video static noise sparks likely caused by a worn-out VHS-tape. Sparks were randomly appearing all around the image at random intervals. Finally, a common fault in video image was an a small band of noise in the lower part of the image. The noise seemed like real visual data but it was out-of-place, as if the scanlines had changed places in the image frame. Audio quality of the studied material was generally low, a lot of the higher sound frequencies were missing. Generally, the audio was polluted by a loud mid- and high frequency noise in all studied videos.

The design process was gradual, first starting from adding components to the filter chain affecting the blurriness and sharpness of the image quality. Those filters were already available in GPUImage – GPUImageGaussianBlurFilter and GPUImageSharpenFilter. From these filters, the blurSize and sharpness shader uniform variables were exposed to the filter settings in application UI. Then a filter for image color imperfections was gradually implemented in an interactive trial-and-error manner. In this filter – GPUVideoVHSColorFilter – were combined the manipulation of various properties of a pixel. The red-channel value was softened using a 3x3 matrix Gaussian blur and the blue-channel was accumulated with random noise. Properties contrast and saturation were also applied in the same routine – also these properties were exposed to the filter settings UI. To emphasize the unrest image quality at screen bottom, a special GPUVideoSyncDistortionFilter was created. The filter was designed to randomly sample input texture pixel data values to 15 bottom rows in the image. The algorithm used a timer and sine- and cosine functions to drive the sampling.

To create the random static video noise, an image sampling texture atlas of size 72x32 pixels was created using a Pixelmator image editing application.

![Figure 14. An enlarged static noise texture atlas holding 5 different video static spark templates.](image)

The noise texture (Fig. 14) consisted of five slightly different video static spark templates which were drawn to the output texture by a new implementation class, GPUVideoStaticGenerator. The algorithm first drew a desired number of the rectangles (vertex array) matching the sizes of the respective noise sprites in the color input texture, followed by the fragment shader, which sampled the actual noise sprite colors into the final output texture.
Figure 15. The “VHS 1984” filter block flow diagram.

Figure 15 depicts the final implementation of the “VHS1984” filter. In the figure, the filter blocks set off slightly to the right of the vertical filter flow do not have an input texture, but function as generators outputting a texture to the next filter stage. Blocks denoted with gray color depict custom filter classes implemented during the filter design process.
Figure 16a. Original input frame texture.

Figure 16b. Output texture from blurring.

Figure 16c. Output texture from sharpening.

Figure 16d. Merged textures from sharpening and overlay date text.

Figure 16e. Color processing output texture.

Figure 16f. Line generator output texture.

Figure 16g. Output texture from sync distortion.

Figure 16h. Output texture of static generator.

Figure 16i. Merged and blended output of sync and static filters.

Figure 16j. Final output texture from filter.
In Figure 16 (a-j), the output textures from respective stages of “VHS 1984” filter are presented. The images have been captured using Xcode OpenGL ES image capture tool and are presented as is. In each texture image, green lines represent the image triangles each output image has been rendered to. Note, in static generator there are multiple triangles where output texture has been rendered.
5. Evaluation

The evaluation process consisted of quantitative- and qualitative methods. First, the application filter performance was measured in video preview- and video file post-processing cases. Secondly, feedback, comments and suggestions were collected from author’s colleagues during the development phases and from application end-users after the application was made available in Apple Appstore.

5.1 Application performance

The application processing performance was monitored through-out the development process empirically during the development and during artifact progress demonstrations. During the main development phase, iPhone 5s and iPhone 6s Plus were used as the main development devices. The third benchmarking device, iPhone X, was not available during the main development period but was only used in the performance testing.

At the time the application development was started, Full High Definition (HD), 1920x1080 pixel frame size, video was a standard camera application quality format on new Apple iPhones. This source image quality was the initial design goal, but soon into the development of the application, real-time processing requirement of such large image frames were not reachable. The achieved framerate was all too low and was affecting the intended use case – previewing live video through filter in real-time. Furthermore, many of the planned filters would apply many types of quality lowering procedures (e.g. blurring, over-sharpening) and extra artifacts into the image frames, so the smaller image input size was not considered an issue. The targeted image frame sizes supported for video- and photo capture by the application were 640x480 (width x height) and 1280x720.

The tested devices are two device generations (2013, 2015 and 2017) apart from each other, iPhone 5s being the oldest and iPhone X the newest. They each contain a different generation version of the Apple A-series System-on-a-Chip (SoC) (A7 2013, A9 2015 and A11 Bionic 2017) and according to manufacturer’s information each SoC provides more CPU- and GPU performance that its predecessors.

5.1.1 Video preview performance

The performance was tested using the 720p high definition (HD) video preview for all filters, except for “Toy-Cam 1987” filter emulating the Mattel Pixelvision C-cassette tape-based toy-video camera. Table 3 lists the tested devices and the respective CPU- and GPU processing times for each filter and device capture from the Apple Xcode OpenGL ES frame capture tool, which measures the system’s used CPU and GPU execution time from the first invocation of OpenGL ES drawing calls to the final call to the GPU renderbuffer.

The magnitude of CPU processing time for each device and filter looks similar for all the filters, maintaining CPU processing time near 33 ms per frame. This could be due to the fact that the input to the application was driven by the timely arrival of a camera frame from AVFoundation subsystem, where the camera frame update frequency was set to 30 fps. Only “The Matrix” filter executed on iPhone 5s, both the CPU and GPU processing time exceed the theoretical 33 ms deadline to achieve 30 fps thus the goal was not reached in that case – this was also visually detectable portraying slowness in the output texture update to the screen.
Table 3. Video preview performance results on three different iPhones.

<table>
<thead>
<tr>
<th>Filter</th>
<th>iPhone 5s</th>
<th>iPhone 6s Plus</th>
<th>iPhone X</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CPU</td>
<td>GPU</td>
<td>CPU</td>
</tr>
<tr>
<td>Wild West 1899</td>
<td>33.2</td>
<td>22.6</td>
<td>33.1</td>
</tr>
<tr>
<td>New York 1905</td>
<td>31.0</td>
<td>25.7</td>
<td>33.1</td>
</tr>
<tr>
<td>Hollywood 1929</td>
<td>33.1</td>
<td>22.5</td>
<td>32.9</td>
</tr>
<tr>
<td>Chicago 1938</td>
<td>32.6</td>
<td>21.2</td>
<td>33.1</td>
</tr>
<tr>
<td>Pinhole 1893</td>
<td>30.5</td>
<td>25.2</td>
<td>32.9</td>
</tr>
<tr>
<td>Montreal Sun 1945</td>
<td>32.9</td>
<td>22.1</td>
<td>32.9</td>
</tr>
<tr>
<td>Noir 1947</td>
<td>32.8</td>
<td>22.7</td>
<td>32.9</td>
</tr>
<tr>
<td>London 1956</td>
<td>32.6</td>
<td>23.0</td>
<td>33.1</td>
</tr>
<tr>
<td>San Francisco 1964</td>
<td>33.0</td>
<td>21.8</td>
<td>33.1</td>
</tr>
<tr>
<td>Paris 1978</td>
<td>33.3</td>
<td>20.8</td>
<td>33.0</td>
</tr>
<tr>
<td>Halftone 1970</td>
<td>32.9</td>
<td>15.2</td>
<td>33.0</td>
</tr>
<tr>
<td>Pola 1981</td>
<td>32.5</td>
<td>23.6</td>
<td>33.3</td>
</tr>
<tr>
<td>Moon 1969</td>
<td>33.4</td>
<td>23.4</td>
<td>33.1</td>
</tr>
<tr>
<td>Moon 1972</td>
<td>31.7</td>
<td>23.5</td>
<td>32.9</td>
</tr>
<tr>
<td>Mars 1976</td>
<td>32.2</td>
<td>22.1</td>
<td>32.9</td>
</tr>
<tr>
<td>VHS 1984</td>
<td>33.2</td>
<td>25.6</td>
<td>33.2</td>
</tr>
<tr>
<td>Toy-Cam 1987</td>
<td>33.1</td>
<td>20.2</td>
<td>33.0</td>
</tr>
<tr>
<td>Commodore 64</td>
<td>32.7</td>
<td>17.7</td>
<td>33.1</td>
</tr>
<tr>
<td>Apple II</td>
<td>33.2</td>
<td>17.3</td>
<td>32.9</td>
</tr>
<tr>
<td>ZX Spectrum</td>
<td>32.8</td>
<td>16.1</td>
<td>33.2</td>
</tr>
<tr>
<td>VIC-20</td>
<td>33.0</td>
<td>17.2</td>
<td>33.0</td>
</tr>
<tr>
<td>BBC Micro</td>
<td>32.7</td>
<td>17.1</td>
<td>33.0</td>
</tr>
<tr>
<td>The Matrix</td>
<td>38.5</td>
<td>34.7</td>
<td>33.2</td>
</tr>
<tr>
<td>Tele-Text</td>
<td>33.0</td>
<td>16.5</td>
<td>32.9</td>
</tr>
<tr>
<td>LED Scoreboard</td>
<td>33.4</td>
<td>18.4</td>
<td>32.8</td>
</tr>
</tbody>
</table>

Table 3 performance results present the application processing time required both from the CPU and GPU for a single image frame from an entry to the filter group to the output of the output texture.

5.1.2 Post-processing performance

In the video post-processing use case same video file was copied to each device Photos asset storage. Video length was 4.93 seconds recorded at 30 fps containing 148 image frames. Video file frame size was identical to live preview test setup - 1280x720 pixels (width x height). The execution setup differed from the camera preview setup in important ways. First, the input to the filter group came from a movie file contributed by a GPUImageMovie class feeding video file frames to the filter group. Secondly, output from the filter group was directed to the file writer, GPUImageMovieWriter, contributing writing the filtered movie frames from GPU memory to an output file. And finally, in this use case, depending on the filter, file’s audio tracks were also filtered with various types of audio filters and effects; low-, high- and bandpass filters, noise accumulation-, down sampling- and non-linear processing algorithms.
The effect of these additional contributing factors to the performance was not measured separately. When looking at the performance results in Table 4, in comparison to the result in Table 3, the results are slightly surprising. In this use case, the read-process-write routine is not constrained by a synchronized resource (e.g. 30 fps frame input) but the routine should burst through the file as fast as possible.

On iPhone 5s, only 36% of the filters executed faster than in the video preview use case, on iPhone 6s Plus 88% of the filters were faster and on iPhone X all filters executed faster, the fastest being almost two times faster than in the video preview test (58.6 fps vs 30 fps). These results suggest, especially with iPhone 5s, that the various parts of the processing would need an examination to isolate the performance-throttling components of the process. However, optimization of the artifact was not in the scope of this work but is a subject of potential future work.

### Table 4. Video-file post processing performance results on three different iPhones.

<table>
<thead>
<tr>
<th>Filter</th>
<th>iPhone 5s</th>
<th></th>
<th>iPhone 6s Plus</th>
<th></th>
<th>iPhone X</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Frame ms</td>
<td>FPS</td>
<td>Frame ms</td>
<td>FPS</td>
<td>Frame ms</td>
<td>FPS</td>
</tr>
<tr>
<td>Wild West 1899</td>
<td>46.7</td>
<td>21.4</td>
<td>28.0</td>
<td>33.6</td>
<td>23.8</td>
<td>42.0</td>
</tr>
<tr>
<td>New York 1905</td>
<td>59.4</td>
<td>16.8</td>
<td>28.1</td>
<td>35.6</td>
<td>23.7</td>
<td>42.1</td>
</tr>
<tr>
<td>Hollywood 1929</td>
<td>35.5</td>
<td>28.2</td>
<td>24.7</td>
<td>40.5</td>
<td>17.5</td>
<td>57.2</td>
</tr>
<tr>
<td>Chicago 1938</td>
<td>35.1</td>
<td>28.5</td>
<td>35.6</td>
<td>39.1</td>
<td>17.4</td>
<td>57.6</td>
</tr>
<tr>
<td>Pinhole 1893</td>
<td>47.4</td>
<td>21.1</td>
<td>28.0</td>
<td>35.7</td>
<td>22.8</td>
<td>43.8</td>
</tr>
<tr>
<td>Montreal Sun 1945</td>
<td>43.2</td>
<td>23.2</td>
<td>36.5</td>
<td>27.6</td>
<td>21.0</td>
<td>47.7</td>
</tr>
<tr>
<td>Noir 1947</td>
<td>41.2</td>
<td>24.3</td>
<td>26.3</td>
<td>38.0</td>
<td>19.0</td>
<td>52.6</td>
</tr>
<tr>
<td>London 1956</td>
<td>43.9</td>
<td>22.8</td>
<td>26.9</td>
<td>37.2</td>
<td>22.1</td>
<td>45.2</td>
</tr>
<tr>
<td>San Francisco 1964</td>
<td>38.0</td>
<td>26.3</td>
<td>33.2</td>
<td>30.2</td>
<td>17.9</td>
<td>56.0</td>
</tr>
<tr>
<td>Paris 1978</td>
<td>39.4</td>
<td>25.4</td>
<td>24.4</td>
<td>41.1</td>
<td>17.6</td>
<td>56.8</td>
</tr>
<tr>
<td>Halftone 1970</td>
<td>25.1</td>
<td>39.8</td>
<td>20.1</td>
<td>49.8</td>
<td>17.1</td>
<td>58.4</td>
</tr>
<tr>
<td>Pola 1981</td>
<td>49.3</td>
<td>20.3</td>
<td>29.5</td>
<td>33.9</td>
<td>22.5</td>
<td>44.5</td>
</tr>
<tr>
<td>Moon 1969</td>
<td>42.6</td>
<td>23.5</td>
<td>27.3</td>
<td>36.7</td>
<td>21.3</td>
<td>46.9</td>
</tr>
<tr>
<td>Moon 1972</td>
<td>43.9</td>
<td>22.8</td>
<td>25.4</td>
<td>39.4</td>
<td>22.7</td>
<td>44.1</td>
</tr>
<tr>
<td>Mars 1976</td>
<td>41.2</td>
<td>24.3</td>
<td>26.6</td>
<td>37.7</td>
<td>21.8</td>
<td>45.8</td>
</tr>
<tr>
<td>VHS 1984</td>
<td>54.3</td>
<td>18.4</td>
<td>37.6</td>
<td>26.6</td>
<td>26.2</td>
<td>38.2</td>
</tr>
<tr>
<td>Toy-Cam 1987</td>
<td>35.4</td>
<td>28.5</td>
<td>23.8</td>
<td>42.0</td>
<td>17.6</td>
<td>56.9</td>
</tr>
<tr>
<td>Commodore 64</td>
<td>27.9</td>
<td>35.9</td>
<td>19.2</td>
<td>52.1</td>
<td>17.1</td>
<td>58.6</td>
</tr>
<tr>
<td>Apple II</td>
<td>28.1</td>
<td>35.6</td>
<td>19.8</td>
<td>50.6</td>
<td>17.2</td>
<td>58.1</td>
</tr>
<tr>
<td>ZX Spectrum</td>
<td>28.0</td>
<td>35.7</td>
<td>19.4</td>
<td>51.5</td>
<td>17.1</td>
<td>58.4</td>
</tr>
<tr>
<td>VIC-20</td>
<td>28.4</td>
<td>35.2</td>
<td>19.9</td>
<td>50.4</td>
<td>17.1</td>
<td>58.5</td>
</tr>
<tr>
<td>BBC Micro</td>
<td>27.4</td>
<td>36.4</td>
<td>19.2</td>
<td>52.1</td>
<td>17.1</td>
<td>58.3</td>
</tr>
<tr>
<td>The Matrix</td>
<td>28.0</td>
<td>35.7</td>
<td>18.7</td>
<td>53.6</td>
<td>17.1</td>
<td>58.6</td>
</tr>
<tr>
<td>Tele-Text</td>
<td>28.1</td>
<td>35.6</td>
<td>18.8</td>
<td>53.1</td>
<td>17.1</td>
<td>58.4</td>
</tr>
<tr>
<td>LED Scoreboard</td>
<td>28.2</td>
<td>35.5</td>
<td>21.8</td>
<td>45.8</td>
<td>17.1</td>
<td>58.4</td>
</tr>
</tbody>
</table>

Table 4 performance results portray the processing time required from the CPU and GPU for a single image frame from an entry to the filter to the output of the output texture.
5.2 Application user evaluation

The main period of application development was an iterative process. The work week iterated through phases of design & development–demonstration–ideation, typical to the prototype-building approach often used in Design Science approaches. The usability evaluation was empirical and performed with a group of author’s co-workers present at the office space after the demo session. Member of the group were not the same people every time, but the formation of the group was more or less voluntary and free-forming. The evaluation sessions criteria concentrated on the usability of the application features (e.g. user interface fluency) and the quality and authenticity of the application filters and effects at specific time of the development cycle. The important, direct feedback collected during the short sessions typically influenced the content to the next ideation and the following period development plan. The suggested ideas, features and requests were implemented in to the prototype application during the next period(s) and was subject to evaluation in the next session. This way in process, the evaluation was natural collaboration and it was producing valuable knowledge to the design process. When application started to mature towards releasable application, the general consensus was, that the application features and performance were mostly satisfactory and most of the application requirements were met. There were technical issues such as crashes and false behavior, but these were subjected to the normal debugging and development work.

External feedback was collected from end-users with a passive approach through Apple Appstore automatically provided mechanism and through the application’s website https://www.recsapp.com/ support email link. No forced request was made through the application to the application users to rate and evaluate the application at the Apple Appstore. This was a minor mistake. Even though many users probably would have rejected and dismissed the feedback request without answering, there still would have been chance to receive some valuable critique and feedback from some users. With direct emails through the application website, also valuable negative feedback was received exposing errors otherwise not caught during the development. With that information at hand, corrective measures could be applied to the application for the next update release.

Here are examples of application end-user comments and reviews:

User review 1:

“...Alone because of the emulation of the PXL2000 and the Polaroid camera is the app really worth gold. And then the Pixel Cams - the slogan "Art in your Hand" is totally out here."

User review 2:

“I am really impressed by the authenticity of some of the effects provided in RECS. The filters are really nice, I still miss a simple feature of no filtering at all during recording and filter the video afterwards.”

User review 3:

“I really like computer themes, make the pixel size adjustable, I want really big pixels!!"
6. Discussion and conclusions

As the result of design process an artifact was produced and ultimately released through the Apple AppStore. The application functionality and usability has proven relevance to its users, and content-wise, it has its place among the video- and image processing applications on the market. The section 4 process description and the completed artifact itself answer the first research question - how to build an image processing application on Apple iOS platform using an OpenGL ES parallel programming model?

In addition to the artifact contribution, the following chapters briefly discuss the central literature findings followed by the findings from this thesis, followed by the conclusions of this thesis. The findings from the artifact design and implementation process reflect the faced challenges and provide knowledge for the assisting research question - what are the challenges and barriers in the applied OpenGL ES programming model?

6.1 Literature summary

The dominant model of parallel programming model on Personal Mobile Devices (PMDs) is Khronos Group’s OpenGL ES programming interface standard. Even though the programming model is strictly oriented towards graphics programming, the introduction of the programmable graphics pipeline in specification version 2.0 makes it suitable for 2D-matrix computation such as employed in image processing. While some research-oriented efforts have been made to port OpenCL on Android platform, OpenGL ES has remained by far the dominant programming model on PMD platforms (Thabet et al., 2014).

Based on recent research on image processing on PMDs using parallel programming models, there is a potential to be explored in parallel computing in these device settings, to reach towards more higher-level portable solutions. 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 (Stone et al., 2010).

Important considerations from software engineering viewpoint was the portability and correctness of the existing software across different platforms (e.g. from desktop to mobile). The development of generic, platform independent programming languages, such as OpenCL is a promising step towards more common parallel programming language on mobile systems (Stone et al., 2010).

The programming expertise from the programmers (traditional programmer vs. graphics specialist) requires special attention. For example, 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 a 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 (Owens et al., 2007).
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 maneuvers, 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 relevant to the user context.

From performance viewpoint, available parallel computation on PMDs provided tempting advantages. In many reviewed cases harnessing the device GPU, or coupled configuration of CPU and GPU provided a significant boost in performance when compared to CPU-only configurations. (Baek et al., 2010; Kim and Kim, 2016; Cavus et al., 2014)

6.2 Findings of the design process

Similarly to the literature findings, using an assistive framework (GPUImage) to ease up the OpenGL ES programming overhead was essential for the success of the design and implementation process. The quality and functionality programmed into the library was so comprehensive that implementing that functionality alone would have not been possible or practical at all. Even though the artifact did not utilize many, but only a handful of ready-made filter components in the library vast filtering functionality, the vast OpenGL ES context and -API management was in the essence of the working application.

Apart from two most recent filters implemented into the application, no performance optimization attempts were necessary, but the filter performance achieved its set requirements. From exploratory viewpoint in some filters, such as “VHS 1984”, the functionality of selected GPUImage filters, specifically in this case the GPUImageLuminanceFilter and GPUImageColorSaturationFilter were implemented into the GPUVideoVHSColorFilter to reduce the GPU filtering passes on already big filter group. Using this type of custom, composited building approach some filtering performance improvement was noticed in the post-processing tests. From performance viewpoint, the GPU-computing proved to be efficient in each of the tested devices. For the future improvement work, the GPUImage library could benefit from moving away from supporting the OpenGL ES 2.0, and adapt the OpenGL ES 3.0. The repetitive vertex drawing stage in filter execution, could be improved by adapting the Instanced Drawing of the vertex data as defined by Apple (2017). In order to account true performance benefits from the approach used in this thesis, a comparative algorithm study between OpenGL ES, Apple Core Image and Apple Metal could be arranged.

From Software Engineering (SE) perspective, there were a number of issues affecting the design and implementation. In Apple Xcode, OpenGL ES programs cannot be examined using a functional step-debugger, like traditional programming software engineers have accustomed to during the last 30 years. This is due to the way fragment shader code is exposed to the shader compiler and linker, which are separate entities apart from the compiler and linker used for iOS application development. Furthermore, the massive parallel fragment execution may be hard to integrate into the concept of debugging. So, the lack of debugger can be very frustrating since stepping through the source code during execution is an essential method in problem solving in exposing logical faults in the source code. Xcode, however, provided a GPU Frame Capture Tool which essentially halted CPU- and GPU program execution and provided information on data items currently attached to the vertex- and fragment shaders. This was helpful, but naturally did not reveal the potential logical faults in the shader source code. This can be an issue that
diminishes over time, but can portray a substantial barrier for a programmer new to the GPU shader programming.

Another issue with programming was the lack of proper source code editor syntax checking support. In modern Integrated Development Environments (IDE), the editing of the source code is supported by a compiler exposing syntactic errors in typed source code – typically in real-time – helping to make the programming effective and productive. This was not the case with fragment shader writing on Xcode. Writing source code this way was like programming blindfolded – the syntactic errors were only exposed when the shader was explicitly compiled with the compiler. However, the compiler worked interactively and the errors could be fixed based on the findings. When considering the lack of debugger and a live-compiler - it can have impact from resourcing viewpoint, more so with programmers that are inexperienced with OpenGL ES. From personal experiences during this thesis work, solving issues with these specific tools can be very time consuming.

Another thing to consider is the OpenGL ES shading language. It is essentially a C-based programming language with custom datatypes only applying to the OpenGL ES context. The language and its specifics has a natural learning curve when programmer is coming from a different technology (e.g. Java, Javascript) and this should be considered when resourcing programmers to similar tasks.

Based on the experiences, using OpenGL ES on Apple iOS platform can be used to create an efficient and user-friendly parallel computing applications. It requires a good degree of discipline and competence from the designers and programmers. Furthermore, OpenGL ES is foremost a graphics programming language, requiring maneuvers to fit it into general purpose computation. With the help of assisting tool and libraries, such as GPUImage, the burden of the repetitive OpenGL ES management operations can be greatly reduced.

However, parallel programming on Personal Mobile Devices (PMDs) could greatly benefit from a common, standardized general purpose computing language such as OpenCL that stretches across different platforms’ boundaries. Neither of the current dominant PMD ecosystems – Android and iOS – do not officially support use of OpenCL but prioritize their own custom programming languages, RenderScript on Android and Metal 2 on iOS. Nevertheless, these custom environments are effective and provide rich functionality to users, but portability and re-usability of the source code across platform- and ecosystem boundaries can be a challenge.
References


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


## Appendix A: Table of film- and TV-filters

<table>
<thead>
<tr>
<th>Film &amp; TV filters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wild West 1899</td>
<td>Brown sepia film. Low visual quality with a lot of visual noise and image artifacts. User adjustable parameters: framerate, low-fi audio on/off, image noise, vertical stripes, blurriness, contrast.</td>
</tr>
<tr>
<td>New York 1905</td>
<td>Black &amp; white film. Low visual quality with a lot of visual noise and image artifacts User adjustable parameters: framerate, low-fi audio on/off, image noise, vertical stripes, blurriness, contrast.</td>
</tr>
<tr>
<td>Hollywood 1929</td>
<td>Emulation of TechniColor 2 color model from 1920’s. User adjustable parameters: framerate, low-fi audio on/off, blurriness, contrast, saturation and visual noise artifacts.</td>
</tr>
<tr>
<td>Chicago 1938</td>
<td>Emulation of TechniColor 3 color model from 1930’s. User adjustable parameters: framerate, low-fi audio on/off, blurriness, contrast, saturation and visual noise artifacts.</td>
</tr>
<tr>
<td>Montreal Sun 1945</td>
<td>Color film, low saturation. User adjustable parameters: framerate, low-fi audio on/off, visual noise artifacts, blurriness, saturation, contrast.</td>
</tr>
<tr>
<td>San Francisco 1964</td>
<td>Color film. User adjustable parameters: framerate, low-fi audio on/off, blurriness, visual noise artifacts, saturation, contrast.</td>
</tr>
</tbody>
</table>
Appendix B: Table of technology filters

<table>
<thead>
<tr>
<th>Technology filters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Halftone 1970</td>
<td>Emulation of old halftone printing technique, sketch image with black dots only. User adjustable parameters: adjustable dot size on/off, dot size, dot spacing, threshold.</td>
</tr>
<tr>
<td>Moon 1972</td>
<td>Emulation of the last moonwalk color TV-transmission visual quality. User adjustable parameters: framerate, artificial text overlays – Space Comms - on/off, level grid of/off, artificial text overlays on/off, image ghosting, blurriness, brightness, contrast.</td>
</tr>
<tr>
<td>Mars 1976</td>
<td>Imaginary emulation if there was a video camera on board on Mariner lander on planet Mars in 1976. User adjustable parameters: framerate, artificial text overlays – Space Comms - on/off, level grid of/off, artificial text overlays on/off, image ghosting, blurriness, brightness, contrast, saturation.</td>
</tr>
<tr>
<td>VHS 1984</td>
<td>Emulation of a VHS tape playback. User adjustable parameters: framerate, low-fi audio on/off, blurriness, sharpness, contrast, saturation, static noise, horizontal bands.</td>
</tr>
</tbody>
</table>
**Appendix C: Table of computing filters**

<table>
<thead>
<tr>
<th>Computing filters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C-64 Computer</td>
<td>Emulation of Commodore 64 computer color palette and pixelated image. User adjustable parameters: framerate, low-fi audio on/off, color, pixel size.</td>
</tr>
<tr>
<td>Apple II Computer</td>
<td>Emulation of Apple II computer color palette and pixelated image. User adjustable parameters: framerate, low-fi audio on/off, color, pixel size.</td>
</tr>
<tr>
<td>Sinclair ZX Spectrum Computer</td>
<td>Emulation of Sinclair ZX Spectrum computer color palette and pixelated image. User adjustable parameters: framerate, low-fi audio on/off, color, pixel size.</td>
</tr>
<tr>
<td>VIC-20 Computer</td>
<td>Emulation of Commodore VIC-20 II computer color palette and pixelated image. User adjustable parameters: framerate, low-fi audio on/off, color, pixel size.</td>
</tr>
<tr>
<td>BBC Micro Computer</td>
<td>Emulation of BBC Micro computer color palette and pixelated image. User adjustable parameters: framerate, low-fi audio on/off, color, pixel size.</td>
</tr>
<tr>
<td>The Matrix</td>
<td>Emulation of a Matrix movie computer world scenes. User adjustable parameters: framerate, video interlacing on/off, digital rain, digital rain intensity, ghost voice on/off.</td>
</tr>
<tr>
<td>Tele-Text</td>
<td>Emulation of a Tele-Text service color palette and pixelated image. User adjustable parameters: framerate, low-fi audio on/off, color, pixel size.</td>
</tr>
<tr>
<td>LED Scoreboard</td>
<td>Emulation of a stadium LED scoreboard. User adjustable parameters: adjustable dot size on/off, dot size, dot spacing, threshold.</td>
</tr>
</tbody>
</table>