HTW Berlin (University of Applied Sciences) WS 2013 / 2014
Media and Computing (Master)
Faculty of Business Sciences II

Master's Thesis

Parallel Computing for Digital Signal Processing on Mobile Device GPUs

Markus Konrad, B.Sc.
Mat. No. 517910

March 2, 2014

Supervisors:
Prof. Dr. Klaus Jung
Prof. Dr. David Strippgen

Abstract

This thesis studies how certain popular algorithms in the field of image and audio processing can be accelerated on mobile devices by means of parallel execution on their graphics processing unit (GPU). Several technologies with which this can be achieved are compared in terms of possible performance improvements, hardware and software support, as well as limitations of their programming model and functionality. The results of this research are applied in a practical project, consisting of performance improvements for marker detection in an Augmented Reality application for mobile devices.

Contents

1. Introduction

This thesis explores the possibilities of parallel computing on mobile devices by harnessing the power of their built-in multi-core graphics processing unit (GPU). It shows by which means and technologies algorithms for digital signal processing (DSP), especially in the field of image processing, can be made to execute in parallel on such graphics units in order to improve the overall performance of a mobile application.

1.1. Motivation

One of the basic principles of computer engineering and hardware design is *Moore's Law* [Moo98], which states that the number of transistors on integrated circuits doubles about once every two years. This observation is connected to the increasing processing power of integrated circuits and their ongoing miniaturization. These advances made small handheld devices, such as mobile telephones and later modern mobile devices like smartphones and tablets, possible.

At the same time it should be clarified that Moore's "Law" is an observation rather than a natural law and as such it is not universally true, although many software developers have been accustomed to what Sutter [Sut05] described in 2005 as the "free lunch": Ever increasing processor clock-speeds resulted in "automatically" faster running programs for each chip generation. As Sutter describes, this can no longer be taken for granted, since the processor manufacturers "have run out of room with most of their traditional approaches to boosting CPU performance." Due to this, they changed their strategy to produce multi-core processors that share the workload. Since "[a]ll computers are now parallel" as McCool et al. say, it is "necessary to use explicit parallel programming to get the most out of such computers" [MRR12, p. 1]. Applications do not receive automatically improved performance merely with the addition of cores, as will later be shown. Thus, one solution for improving the performance of algorithms even as Moore's Law produces diminishing returns or halts completely, is to better use the available power of current and future hardware when writing *portable* and *scalable* software.

GPUs are high-end multi-core hardware that can run arbitrary parallel computations via *general purpose computing on graphics processing units (GPGPU)*. With this it is possible to turn normal desktop computers into small "supercomputers" with hundreds of parallel floating-point operation units. Since most modern mobile devices also contain a GPU, it should be possible to harness the power of their parallel processing capabilities for all kinds of purposes. The motivation of this work is to show that this can be done, and furthermore that this is worth doing – especially in mobile device development where central processing unit (CPU) performance is still a major bottleneck. Fields of application such as image filtering, video analysis or augmented reality could greatly benefit from GPGPU on mobile devices, especially when considering increased camera resolutions which leads to higher computing complexity for the mentioned fields of application.

1.2. Aims of this Work

As indicated in the previous section, the aims of this thesis are finding ways for faster computation of DSP problems by using techniques of parallel computing on mobile device GPUs. The performance impact of this approach is to be examined and compared in the following fields:

- image processing (linear filters, histograms)
- computer vision (extracting geometric features)
- audio synthesis

An important part of this thesis will be to find out the proper prerequisites for GPGPU on mobile devices: Which hardware can be used, i.e. which devices with what kinds of GPU hardware? What are the advantages and disadvantages of these devices in terms of parallel computing on their GPU? This leads to the next question: What needs to be expected from the software side? On which kind of platform, i.e. on which mobile device operating system (OS) is it possible to use GPGPU? What kind of frameworks, libraries and application programming interfaces (APIs) can be used in the development process and what are their advantages and disadvantages?

Besides these practical questions, this work also needs to consider the positives and negatives of this approach on mobile devices, including benefits and limitations. How can a developer handle the issue of portability when dealing with technologies that might be very hardware-specific and dependent on special APIs? How can one design possible hardware-independent fallbacks? May the effort put into designing and writing such applications negate the outcome of possible higher performance?

Furthermore, this thesis tries to postulate about the future and predict what one might reasonably expect of current trends in GPGPU on mobile devices. Where is this technology heading and what kind of developments can be anticipated?

1.3. Organization of this Thesis

Since many different fields of computer science are mentioned in this thesis, a comprehensive overview of each field's respective fundamentals cannot be given. Common knowledge about digital image and audio processing is assumed.

In general, the work can be outlined as follows: An introduction about the fundamentals of parallel computing that are important for this thesis is given in chapter 2, since this topic is still often regarded a specialized subject in computer science. The explanations also cover different machine models that are important in order to understand the hardware architecture of graphics units. Because this thesis focuses on performance measurements, concepts such as complexity and speedup are discussed in this chapter, too. Chapter 3 then focusses on parallel computing with GPUs and outlines the state of this field on mobile devices for the area of DSP. Available parallel programming technologies for these platforms are introduced and an overview about related research is given. The main part of this thesis consists of chapter 4, where a study of the different technologies for GPU computing on mobile devices is conducted. The comparisons are made using different algorithms of the mentioned DSP fields. Their theoretical background is explained before the different implementation approaches are discussed. Therefore, some important DSP fundamentals are given in the beginning of this chapter. Based upon the results of this evaluation of GPGPU technologies, a practical project is implemented, where parts of an existing application are accelerated by sourcing them out to the graphics unit. This project shows the potential but also the issues for the practical application of this technology. To conclude, chapter 6 gives a summary of this work and outlines future perspectives for GPGPU on mobile devices.

2. Fundamentals of Parallel Computing

In this chapter the fundamentals of parallel computing are introduced. This theoretical background is important for designing, analyzing and verifying parallel algorithms. The introduced models and theories are device-agnostic and not bound to mobile systems. Special considerations for this group of devices will be made in the next chapter.

First, some common vocabulary will be defined before moving on to the examination of different machine models and hardware architectures. Some laws for performance theory in parallel systems will be described as well as fundamental concepts of algorithmic complexity. After that the focus is set on parallel programming techniques as well as the involved problems and limits.

2.1. Important Vocabulary and Definitions

For better understanding of the following concepts, it is important to provide some common vocabulary definitions.

2.1.1. Concurrency and Parallelism

Although often used as synonyms, concurrency and parallelism are not the same. In fact, parallelism is a subset of concurrency, which in turn is the opposite of the *serial* execution model. Parallel execution means that two or more actions can be executed *simultaneously*, whereas a concurrent execution allows to run two or more actions *in progress* at the same time [Bre09, p. 3]. In practice this means that a program using two threads on a single-core machine employs concurrency but not parallelism. The instructions in its threads are executed in an interleaved manner and hence are in progress at the same time, but two instructions cannot be executed simultaneously on a single-core machine and hence the application is not parallel.

2.1.2. Workers, Tasks, Data and Dependencies

Algorithms implement a procedure for solving a problem by using *tasks* that operate on *data* in most cases. These tasks are processed by one or several *workers*, which are an abstraction of the available processing units in a piece of hardware. In contrast to a serial program, concurrent algorithms may execute their instructions in a nondeterministic order. At the same time two kinds of *dependencies* may exist between tasks in such algorithms which restrict the running order: Data dependencies and control dependencies. The former describes the problem in which one task may need to work on data that is generated by another task and therefore needs to wait for its completion or that memory accesses need to be synchronized in order to maintain memory consistency. The latter describes a dependency of a task on events, states or other side effects created by another task [MRR12, p. 39; SW11, p. 4; Bre09, pp. 5,27].

A task implements serial control flow as a sequence of instructions and can detach a new task (a concurrently running serial control flow) using a *fork point*. These tasks can be synchronized again by uniting them at a *join point* [MRR12, p. 40]. *Threads* use the same terminology of "forks" and "joins" but are not to be confused with tasks. They refer to what is explained as *mandatory parallelism* in the following section.

2.1.3. Mandatory and Optional Parallelism

McCool et al. [MRR12, p. 19] compare *mandatory* and *optional* parallelism. These terms are also referred to as explicit and implicit threading models by Breshears [Bre09, p. 77]. The former describes a parallel programming model in which the system forces concurrent (not necessarily parallel) execution of instructions. An example for this is POSIX threads[1], where spawning and destroying of threads is directly specified in the program. In contrast to that, optional parallelism defines *opportunities* for parallel execution. The system itself decides if parallel execution is beneficial under the current circumstances, taking the number of idle processing units into account, for example. This approach has strengths in scalability and portability but gives less control to the software developer. The different parallel programming models and available libraries will be described in more detail in section 2.4.1.

2.2. Machine Models

A machine model provides the theoretical background for a computer hardware architecture. It explains how data is organized, transported and processed in such a machine.

2.2.1. Flynn's Taxonomy and Related Classifications

Flynn [Fly72] has introduced a classification of computer architecture that is now known as *Flynn's taxonomy*. For parallel computing there are two categories of interest: Single

[1]See [Jos13] on the POSIX standard and [Bla13] on its *pthreads* API.

Instruction, Multiple Data (SIMD) and Multiple Instruction, Multiple Data (MIMD). The former describes a system that can perform one instruction stream simultaneously on multiple data elements. This can be implemented as *instruction level parallelism* or *vector instructions* in a single processing core where multiple functional units process a whole vector of data at once [MRR12, p. 44]. The latter is a system that is capable of handling separate instruction streams that operate on separate data. This can be realized with different hardware configurations, including computers in a network that form a computing cluster or, more important for this thesis, multiple processing cores in a single computer.

There is a related classification called Single Instruction, Multiple Threads (SIMT) that forms a subset of SIMD and is used in GPUs. It specializes in executing massively parallel computations that run hundreds of hardware threads all operating on different parts of data. However, these threads need to run the same instructions together in order to achieve the best performance [Zwa10]. This is due to the different hardware architecture of GPUs compared to CPUs, which will be explained in more detail in the following section.

2.2.2. General CPU and GPU Architecture

Modern consumer CPUs have made excellent progress in terms of performance and efficiency, while their fundamental design is still based on the classic Von-Neumann-architecture of 1945 [Neu45]. This shows how successful this architecture still is for general-purpose computers. As shown in figure 2.1, a CPU is connected via a data bus to a memory unit which allows the reading or writing of data. The CPU itself consists of a control unit and an arithmetic/logic unit (ALU). The former interprets instructions of a program and controls the communication to and from the ALU which itself executes these logic and arithmetic instructions. Modern CPUs additionally have a data cache and typically consist of many ALUs. However, the basic design of Von-Neumann is laid out for strict sequential running order, which places it in the *Single Instruction, Single Data (SISD)* category in Flynn's classification. This influences the programming model (see section 2.4.1) of the programming languages that can be used on these machines. According to Backus [Bac72], this causes problems with programming language design: Besides the "literal [Von Neumann] bottleneck" (which refers to the data bus) there is the "intellectual bottleneck" that encourages sequential programming.

GPUs have an essentially different hardware architecture. They moved from a hardwired graphics pipeline with several highly specialized processors to an "unified grid of processors, or *shaders*" with a large array of freely programmable processing units [LH07]. They are designed to perform a massive amount of floating point number calculations in parallel by executing a large number of threads that operate in groups on "nearby" memory (see next section) with similar instructions thus minimizing control logic and long-latency memory accesses. Figure 2.2 shows the architecture of a modern GPU.

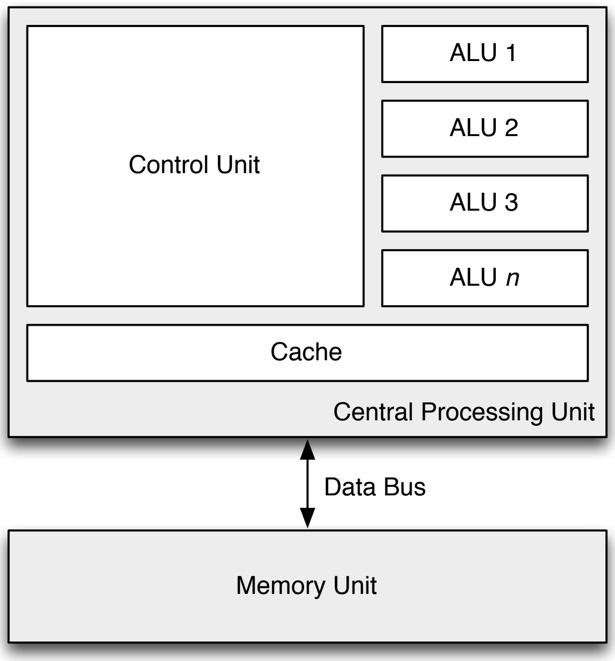


Figure 2.1.: Modified Von-Neumann-Architecture [KH12, p. 4].

It consists of several *streaming multiprocessors (SMs)* that are together organized in a building block. Each SM has multiple *streaming processors (SPs)* connected to a shared instruction cache and control logic unit. All SMs have high-bandwidth access to the video memory of the GPU, with a capacity of 128 MB to 8 GB on modern video cards [KH12, p. 5–9].

The high level of parallelism leads to an impressive raw computational power on modern GPUs. Another big difference compared to CPUs is that GPU design philosophy is aimed at high throughput rate for graphics rendering while CPU design prioritizes low latency, as Owens et al. [Owe+08] point out. A modern CPU operates in a scale of nanoseconds, but the human visual system works only in a magnitude of milliseconds. Because of this, GPUs are designed to complete the processing of a large amount of data in a comparatively long time, whereas CPUs process fewer data in very short time spans. This is one of the reasons why it is more efficient to let a GPU process one big chunk of data at one time instead of feeding it several times with smaller data chunks.

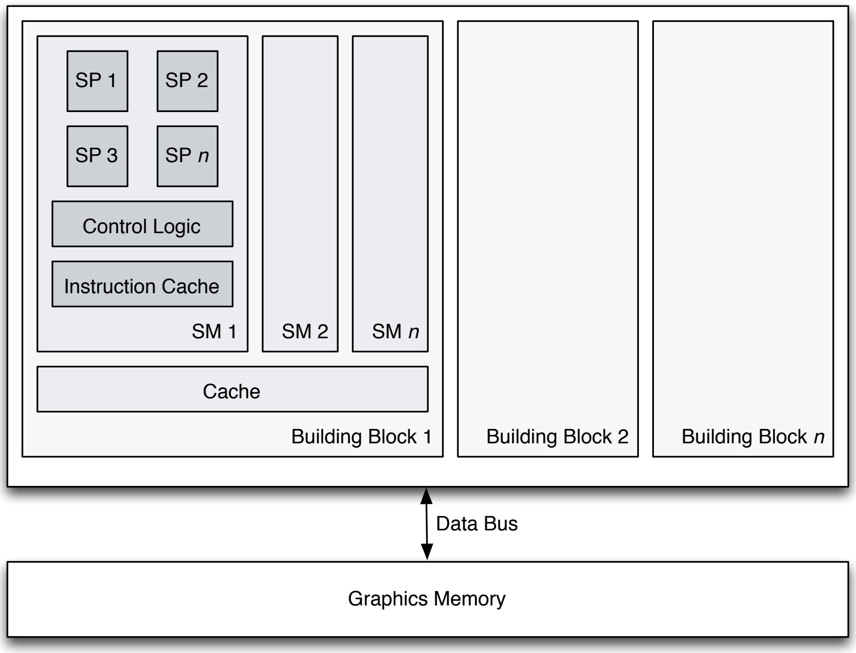


Figure 2.2.: GPU architecture schema [LH07; KH12, p. 9].

2.2.3. Memory Hierarchies and Data Locality

For efficient computation *data locality* is crucial, which is the “reuse of data from nearby locations in time or space” [MRR12, p. 50]. For this reason most computer architectures implement a multi-level memory hierarchy as described by Van Der Pas [Van02]: The closest memory units are *registers* that store a value on which a machine instruction directly operates. After that come data and instruction *caches* which are organized in several levels, where higher levels are slower to access but are larger than lower cache levels. Registers and caches are integrated into processor cores and therefore provide very fast access. The next step on the hierarchy is *main memory*, which is connected via a data bus and therefore is much slower to access but provides usually several gigabytes of memory. The next steps include disk and tertiary storage that provide long term mass storage, but are not important for this thesis.

Two important measures for memory performance are *bandwidth* and *latency*. The former describes the rate at which a certain amount of data can be transferred, whereas the latter is the amount of time until a memory transfer request is fulfilled. Both are strictly tied to the memory hierarchy level. The closer the data resides in time and space to the processing unit, the higher the bandwidth and lower the latency. Therefore, data locality is a “key feature for performance” according to McCool et al. [MRR12, pp. 46,50].

2.3. Performance and Complexity

This section covers fundamental measures and laws of performance theory and provides a short introduction to computational complexity for parallel algorithms. Although performance can (and should) also be measured empirically, the analytic models that will be introduced allow deeper insight about how performance scales under different circumstances, what improvements are to be expected from parallelism and where the limits to these improvements lie.

2.3.1. Important Measures for Performance

Culler et al. [CSG98, p. 59] give some important important performance characteristics for parallel computation: *Latency* has already been introduced in the context of memory (see 2.2.3). In performance theory it means the time a task needs to complete. *Throughput* can also be used in this context and means the rate at which a certain amount of tasks are completed. This measurement is related to the term *bandwidth*, which is used for memory transfer rates [MRR12, p. 55].

Speedup S_p is defined in a number of publications ([DAS12, p. 19; Bhu09, p. 20; MRR12, p. 56]) as the fundamental indicator for comparing the performance of a program on a multiprocessor machine to a reference (single-processor) machine. It is defined as the latency or execution time T_1 of the program using one *worker* (one processing unit) divided by the latency T_P for using P workers:

$$S_p = \frac{T_1}{T_P}. \tag{2.1}$$

Efficiency is related to this since it is defined as speedup divided by the number of workers (S_p/P). This results in a measurement for the "return on hardware investment" [MRR12, p. 56]. When a program runs P times faster with P workers, the equation would yield 1 (or 100% efficiency) which is called *linear speedup*. This is the ideal case that can usually not be achieved since every parallelization also introduces overhead for task management and synchronization. Theoretically a value over 1 is also valid and would yield *superlinear speedup*, but this is only possible if the parallel algorithm itself is more efficient than the serial algorithm due to better cache memory usage or better overall algorithm design. However, in practice *sublinear speedup* is to be expected.

2.3.2. Amdahl's Law

Amdahl [Amd67] split the overall execution time T_1 into two parts: Time s for non-parallelizable serial work and time p for parallelizable work. These are summed and inserted into the definition of speedup and result in *Amdahl's Law* for linear or sublinear speedup with P as the number of processors:

$$S_p \leq \frac{s+p}{s+p/P}. \tag{2.2}$$

When the total time is defined as $s+p = 1$, this formula gets simpler:

$$S_p \leq \frac{1}{s+p/P}. \tag{2.3}$$

Now if one considers having an infinite number of processors (such that $P \to \infty$), one can see that the maximum possible speedup is $1/s$. So the important implication of this law is that it limits the maximum possible speedup to the fraction of non-parallelizable work. Even with an infinite number of workers (processor cores), the speedup cannot exceed the limit set by the time for serial work. This is an important limitation, because most algorithms can only be partly parallelized (as explained in section 2.4.3) and it depends on the amount of remaining serial work plus parallelization (work distribution) overhead. This decides whether a satisfying speedup can be achieved.

2.3.3. Gustafson-Barsis' Law

Gustafson [Gus88] revised Amdahl's Law noting that the problem size also grows with increasing computational power: "[I]n practice, the problem size scales with the number of processors". By adding more workers, the parallelizable workload is distributed better and thereby each worker can do more parallelizable work in the same time, whereas the serial fraction of the work grows smaller or stays fixed. This results in a better speedup. Gustafson calls this *scaled speedup* S_{sc} and provides the following formula[2] giving credit to E. Barsis:

$$S_{sc} = P + (1 - P) * s. \tag{2.4}$$

Gustafson-Barsis' Law takes scalability into account, which is very important. However, the difference to Amdahl's Law is the point of view as noted in [MRR12, p. 62]: "The difference lies in whether you want to make a program run faster with the same workload or run in the same time with a larger workload."

[2]Gustafson's paper gives the formula with an *equal* sign instead of a *less-or-equal* sign, therefore providing an upper bound for ideal parallism without overhead (linear speedup) in contrast to McCool et al. [MRR12].

2.3.4. Computational Complexity for Parallel Algorithms

Computational complexities describe the requirements of an algorithm in terms of time (execution period) and space (amount of memory) subject to the problem size n. This is usually done as asymptotic complexity analysis that yields an upper and/or lower bound for time or space complexity of an algorithm. Roosta [Roo00, p. 233] emphasizes that for parallel algorithms the time complexity is harder to determine than for serial algorithms, where it can essentially be calculated by counting the number of operations and their dependency from the problem size. So for serial algorithms the complexity can be described as a function $f(n)$, whereas for parallel algorithms there is an additional dependency on the number of processors P which yields a function $f(n, P)$.

For serial algorithms the complexity of an algorithm is usually expressed in *Big O notation* resulting in an upper bound. For example, to find the index i of a number q in a vector that contains n unique unordered numbers, a possible algorithm has the time complexity $T_1(n) = O(n)$. This upper bound means that the worst case scenario for this algorithm is to perform n operations for a vector of size n. McCool et al. [MRR12, p. 66] notes that *Big Theta notation* (referring to a paper by Knuth [Knu76]) is more practical because it provides a lower bound "Ω" (the one for parallel computation) and an upper bound "O" (the one for serial computation). To determine the complexity for parallel computation it is necessary to take into account the number of processors as described in the previous paragraph. For the former example of searching in a vector of unique numbers, we can assume a parallel algorithm that divides the vector into n/P (for $n \geq P$) parts and distributes them on P processors in a tree-like manner. Roosta [Roo00, p. 235] describes this as two phases: First, the "activation steps" for distributing the work, and then the parallel computation for each processor. In the provided example $log_2 P$ activation steps are needed (this equals the tree height) and each processor works in a vector part of n/P. This results in an asymptotic complexity $T_P(n) = \Theta(n/P + log_2 P)$ for the given example.

Computational complexities for parallel algorithms depend on the parallel programming model, and the two phases that Roosta [Roo00, p. 235] describes may not be valid for all of them. But what always needs to be considered is the number of processors and the complexity of distributing the work or collecting results of the parallel computations (e.g. summing up values of all the processors' partial results). In practice, it is also important to note that some operations are considerably slower than others, which might be the result of the underlying hardware architecture.

2.4. Parallel Programming

2.4.1. Parallel Programming Models

Maggs et al. [MMT95] give a short definition for a *programming model*: According to them, it defines the fundamental rules or relationships of an abstract programming vocabulary, which are applied in a number of programming languages. As such, it is an abstraction layer above a *model of computation* which is implemented at the hardware level (as described in section 2.2.2). Bhujade [Bhu09, p. 31] notes that programming models are not tied to a certain kind of hardware, but when the programming model does not fit the hardware specifications, performance may suffer.

Since most computers are still based on the model described by Neumann [Neu45], most programming languages in the serial computing domain are also tied to this model. For parallel computing, the programming models differ stronger since they are optimized for different parallel computer architectures. As already mentioned, a programming model should fit to its underlying hardware architecture or the performance will suffer. Such models that either support optional or mandatory parallelism (see section 2.1.3) can be further divided into the following different programming models described by Bhujade [Bhu09, pp. 31-34]:

Shared Memory is a programming model that allows tasks that run on a multiprocessor system to have direct access to a globally shared memory for asynchronous reading and writing. The program development is simpler than with other models but CPU-memory connection latency and cache coherence are crucial [HHG99, p. 419]. Parallelism can be either mandatory (via *inter-process communication (IPC)*) or optional (for example in Intel Threading Building Blocks (TBB) or Cilk Plus).

Message Passing supports optional parallelism and describes a model where tasks can be distributed on the processing cores of one or several machines (e.g. via a computer network). They communicate by sending and receiving messages. An example is *Message Passing Interface (MPI).*

Data Parallelism describes a model where tasks work collectively on a large data set by dividing this set into smaller parts for each worker. It is important to note that each task performs the *same* operations on a different part of the data set. This programming model provides optional parallelism since data partitioning and hence parallelism is dynamically determined depending on problem size and the available number of workers [MRR12, p. 24]. Examples include the already mentioned TBB, Cilk Plus and a number of programming languages and libraries designed for GPGPU, such as OpenCL or CUDA.

Task Parallelism is usually related to the above data parallel model, but both are distinct. In a task parallel model the focus for parallelism is not set on the data, but on the particular tasks that perform operations on this data. Such tasks are decomposed so that they can run concurrently. Each task might involve a different work-load, so distributing them properly is crucial [MGM11, p. 9]. An example might be a task-graph consisting of connected and interdependent tasks. With respect to their dependencies, such tasks could be parallelized, for example with OpenCL [MGM11, p. 28].

Threads implement mandatory parallelism. Each task is a thread that runs concurrently (not necessarily in parallel) with other threads. The number of threads might by determined dynamically depending on the problem size and number of workers but they must be explicitly created and destroyed. The scheduling of these threads is handled automatically by the OS. An example is the already mentioned *POSIX pthreads API*.

As noted, task and data parallelism are sometimes hard to distinguish and because of this McCool et al. [MRR12, p. 41] call it a "troublesome term." They favor the terminology of *regular* and *irregular* parallelism. In the former type tasks behave similar and therefore have predictable dependencies. The later type is the opposite: they may behave in a different way depending on the input data, and therefore may introduce unpredictable dependencies.

Some of the above models are general enough to be used as parallel programming models for CPUs or GPUs. However, the data parallel model suites GPUs best as will be shown in section 3.1.2 in the next chapter.

2.4.2. Decomposition Techniques for Parallel Programming

Task Decomposition

Most algorithms are at first devised for sequential processing in order to solve a given problem. To parallelize such an algorithm, it must at first be decomposed. There are several techniques for this task. A very common approach is called *task decomposition* or *divide-and-conquer* strategy [Bre09, pp. 22-32; Gas+13, p. 3], which is connected to task parallelism as described in the previous section. By applying this strategy, a problem is broken down into a series of sub-problems that are suited for parallel processing, meaning that little or no dependencies exist between these sub-tasks. So the key for task decomposition is to identify independent sub-tasks in an algorithm in order to parallelize them. This strategy fits best for algorithms whose run-time is very dependent on the input data (not only the problem size) and which have various dynamic side effects. Examples might be a parallel search algorithm, which terminates dynamically depending on when the search term was found [Bre09, p. 23], or a parallel shortest-path algorithm that operates on graph data structures [MGM11, pp. 411-416].

When decomposing a serial algorithm into independent sub-tasks, it is very important to keep a good balance between task *granularity* and task management overhead [Bre09, 25f.]. It is of course important to divide big tasks into smaller independent ones in order to achieve good scalability, because these tasks can be efficiently distributed on the available processing units. However, if these tasks are too fine-grained, meaning that the amount of work in each task is very small in relation to the overhead that is produced by managing this large number of tasks, the overall performance may suffer.

Having completely independent sub-tasks in a parallel algorithm is often not the case, since many algorithms involve at least one of the already described dependencies (see section 2.1.2). An example might be a parallel algorithm that needs to increment a value in a field of a global array. If it is possible that several parallel tasks might want to increment the same field simultaneously (which is for example the case in a Hough voting map as described in section 4.3.3), a data dependency is given and the parallel program must ensure synchronized memory access in order to maintain data consistency.

Data Decomposition

Data decomposition, also known as *scatter-gather* strategy, applies data parallelism by dividing the input data into subsets (data chunks) which are distributed on the parallel processing resources (scattering phase) [Bre09, p. 32; Gas+13, p. 3]. After processing, the partial results are collected and form the final result (gathering phase). Many types of data structures can easily be used for this strategy, such as arrays, matrices or lists. Many image processing tasks, such as linear filters, can be implemented as data parallel algorithms since they work independently on single pixel values.

As with task decomposition, two factors are important to keep in mind: First, order and data dependencies should be considered. In data parallel algorithms there can also be situations when tasks have to be ordered or memory accesses need to be synchronized. An example is given by Breshears [Bre09, pp. 38-41] who designs a parallel algorithm for the well-known “Conway’s Game of Life.” Because in each game round every cell has to check the status of neighboring cells in the “universe” matrix and then has to update its own status, there is a data dependency between individual cells (which also exists in the traditional serial algorithm). Because of this, two matrices have to be used in order to read the status from one matrix containing the previous game round and write the result to another matrix for the new round.

The other important factor is again granularity, which is in this case related to the amount of data in each chunk. Because decomposing a data set is often easier than decomposing tasks at run-time, this strategy allows dynamic reaction to the amount of available processing resources. This means that the amount of work can be efficiently distributed to the processing units at run-time in order to achieve minimum overhead and maximum resource utilization.

2.4.3. Problems and Limits of Parallel Computing

There are many cases where it is either not possible to decompose a serial algorithm into parallel sub-tasks or not beneficial to do so [Bre09, pp. 43-47]. Algorithms that are inherently not suited for parallel processing are the ones that employ a *state machine*. Since there can be only one state at a time, it excludes the possibility of concurrent operations that might produce different states at the same time. In some cases it is possible to prevent this by mutual exclusion and synchronization operations, but this can lead to parallel tasks that eventually execute serially.

Another complicated case is *recurrence relations*, which are present when a calculation result within an iteration of a loop is dependent on the result of the previous iteration. An example are loops that refine a calculation result until a convergence value is reached, as for instance the update and refinement step in the *Linde-Buzo-Gray (LBG)* algorithm [LBG80]. However, it is still possible to parallelize such algorithms. Often this means to apply task decomposition one level above (i.e. calculating the centroids for each cluster in parallel in the mentioned LBG algorithm).

All in all, it must be kept in mind that parallel computing is not a universal cure to increase the performance of every program. This is especially the case when specialized hardware such as GPUs are used, as Luebke and Humphreys [LH07] noted: "The GPU's specialized architecture isn't well suited to every algorithm. Many applications are inherently serial and are characterized by incoherent and unpredictable memory access."

3. Parallel Computing on Mobile Device GPUs for DSP

In this chapter the general concepts of the previous chapter are applied to the special case of parallel computing on a mobile device GPU. Its special hardware characteristics and their implications for parallel programming on such a device are examined. The question of which GPGPU technologies are supported is settled, as well as what their advantages and disadvantages are. Prior research by others on this topic will be presented and analyzed. In the last section some problems in the broad field of DSP will be selected for further investigation in the context of parallel programming on mobile device GPUs.

3.1. GPU Architecture on Mobile Devices

GPUs on mobile devices must meet special requirements since both power consumption and chip size are crucial for a battery powered handheld device. Still, as demands for elaborate graphics in user interfaces and games have grown, they must provide excellent performance. Since mobile devices are a growing market [Kur13], big efforts have been put into the improvement of all hardware components of these devices, including the low-power GPUs. For example, the first generation of Apple's iPhone had a *PowerVR MBX Lite* supporting OpenGL ES 1.1 with one GPU core running at 103 MHz, while the iPhone 5 came out about five years later with a *PowerVR SGX543MP3* with three GPU cores at 266 MHz and support for OpenGL ES 3.0 [Tal13; Shi12]. So, in the past few years mobile device GPUs have made tremendous progress in terms of performance and API support, but still the architectural differences in relation to desktop GPUs remain.

3.1.1. Architectural Differences between Desktop and Mobile GPUs

On desktop and laptop computers, a GPU is often attached to the system as a dedicated graphics card via PCI-e bus. Another possibility is to integrate the GPU on the motherboard or on the CPU die *(integrated CPU/GPU system)*. The differences between mobile and desktop systems in terms of hardware architecture come from the need to save space and energy. The strongest distinction is probably the *system on a chip (SoC)* design, which integrates all processing, data bus and memory components of a computer on one chip, not just the CPU and graphics unit. Regarding the GPU design for mobile devices, Akenine-Möller and Ström [AS08] provide an overview describing some of the main distinctions compared to desktop GPUs:

Memory access: As already explained, desktop computer systems usually come with a dedicated graphics card. This also means that data needs to be transferred via some kind of data bus from the main (or: "host") system to the GPU. Although this design provides a dedicated data bus for the graphics unit and is therefore very fast, it is not practical for mobile devices for reasons already explained. The SoC design provides a more compact layout, which saves space. This comes with a price – the GPU needs to share the system bus with all other components to access the main memory, so the memory bandwidth for it is lower as also pointed out by Cheng and Wang [CW11]. Furthermore, memory accesses are also described as "very expensive in terms of energy" [AS08] and therefore should be minimized.

Tiling: Many GPUs implement a *tiling architecture* that splits the rendering area into small tiles that fit into the cache memory of the GPU's processing units. This prevents expensive off-chip memory accesses [AS08].

API and data type limitations: Mobile device GPUs only support the *ES* subset of OpenGL which will be explained in more detail in section 3.2.3. Until version 1.1 only a fixed rendering pipeline was supported. Since version 2.0 it has been replaced by a programmable pipeline. Some earlier mobile device GPUs only supported fixed-point data types (with OpenGL ES 1.x Common-Lite Profile [BML08, 6f.]), but all OpenGL ES 2.0 capable devices must support floating-point types [ML10, 6f.].

3.1.2. Implications for Parallel Programming

Section 2.2.2 has introduced the hardware architecture of modern GPUs, which is essentially different from the architecture of CPUs. A parallel programming model for GPUs must reflect these differences to achieve the best performance on such a system. Due to the massive number of processing units that should run similar instructions on organized data sets to make best use of the instruction and data caches, a data parallel programming model with regular parallelism is highly favored. This programming model has already been introduced in section 2.4.1 and provides best scalability since the level of parallelism grows dynamically with increasing problem size and/or increasing number of workers. This enables programs to run faster on future hardware with more processing units [MRR12, p. 24].

The previously mentioned features of a GPU's hardware architecture also describe the separate memory spaces for the GPU and CPU. Before a GPU can operate on data, it must be transferred from the host memory to the graphics unit memory. After the calculations there are done, results may be written back to the host memory again. Although separate memory spaces for GPU and CPU are not the case for SoC environments often used in mobile devices, the data must still travel to or from the GPU via a data bus, which can quickly become a bottleneck. So careful design of algorithms to use the least possible amount of memory accesses is crucial. The problem of memory transfers can directly cause parallel algorithms to run slower than their serial counterparts, no matter how well they are optimized.

3.2. Available Parallel Programming Models, APIs and Technologies

Since embedded low-power GPUs are primarily designed for high-performance 2D and 3D graphics rendering, GPGPU is often not natively supported via an API. This was also the case for the beginnings of GPU computing on desktop computers. In 2008, Owens et al. [Owe+08] wrote that "[u]ntil recently, GPU computing could best be described as an academic exercise". But this has changed dramatically as GPGPU technology became easier to use for programmers. Owens et al. [Owe+08] show that GPGPU is now included in consumer applications such as games that use the *Havoc FX*[1] physics engine.

In the following sections, APIs and technologies supported on mobile devices are introduced, although some of them are only available on a small set of models. Nevertheless, the author of this thesis is sure that GPGPU will become as popular as popular for mobile devices as it is in the desktop world in the future and one or more of these technologies will eventually prevail. In all of the following APIs the preferred parallel programming model for GPU computing (the data parallel model) is directly supported or can be emulated.

3.2.1. OpenCL

OpenCL is a framework for parallel computing on *heterogenous systems*. Such systems consist of multiple processing units, with each including potentially different specifications and properties. An example can be a computer with a multicore CPU, two dedicated GPUs and a DSP accelerator. OpenCL provides a uniform access to the computing capacities of all these different processing units in a computer that support the OpenCL industry standard defined by the Khronos Group.[2] This allows software developers to implement parallel algorithms on a wide range of devices.

[1]See product website `http://www.havok.com/products/physics`.
[2]See `http://www.khronos.org/opencl/` for OpenCL specifications and documentation.

Conceptional Foundations and Supported Programming Models

To support such a tremendous variety of hardware, the authors of the framework chose a low-level approach: “OpenCL delivers high levels of portability by exposing the hardware, not by hiding it behind elegant abstractions.” [MGM11, p. 4]. This means that the implementation itself has to be aware about the capabilities of the hardware it is running on, and with this information it has to decide about the optimal distribution of work. Hence, OpenCL is “counter to the trend toward increasing abstraction”. The conceptional foundation of OpenCL is defined by four models [MGM11, pp. 11-29]:

Platform model: Describes the heterogenous system (the main system or *OpenCL host*) and the capabilities of each computing device, which is called an *OpenCL device*. Each device consists of one or more computing units (e.g. the cores of a multi-core CPU), which in turn have one or more *processing elements (PEs)* (e.g. SIMD vector units of a CPU core).

Execution model: In OpenCL, an application includes a *host program* that runs on the main system and one or more *kernels* that are executed on the OpenCL devices. These kernels are usually written in *OpenCL C* and implement the computations that are supposed to run on the parallel PEs of one or more OpenCL devices. Just like OpenGL shaders, they are compiled on-the-fly for the respective hardware. The distribution of work for these kernels is specified with an hierarchical model consisting of *work-items* and *work-groups* in an *NDRange* (see figure 3.1). Work-items are single instances of a kernel and operate on each input data item. They are grouped together in work-groups that share group-local memory (see next description). Kernel programs have access to memory via *memory objects*. The whole set of devices, kernels and memory objects is called an *OpenCL context*.

Memory model: OpenCL defines a hierarchical model of memory regions, shown in figure 3.2. It is divided into *host memory* and levels of *device memory*, each with its own scope. Host memory can be copied or mapped to device memory (*global* and *constant* (read-only) memory) and is available for all work-groups with their respective work-items. Inside such a group, work-items can access *(group-)local memory*. *Private memory* is only accessible by its work-item. This hierarchy can be mapped very efficiently to memory and cache models of different hardware resources. It also guarantees consistency on each of these levels: Local and global memory is consistent within the items of a work-group on synchronization points *(barriers)*. However, it is important to note that consistency for global memory cannot be enforced for work-items in different work-groups.

Programming models: Two parallel programming models are supported by OpenCL 1.1: Data and task parallelism. Both have already been introduced in section 2.4.1.

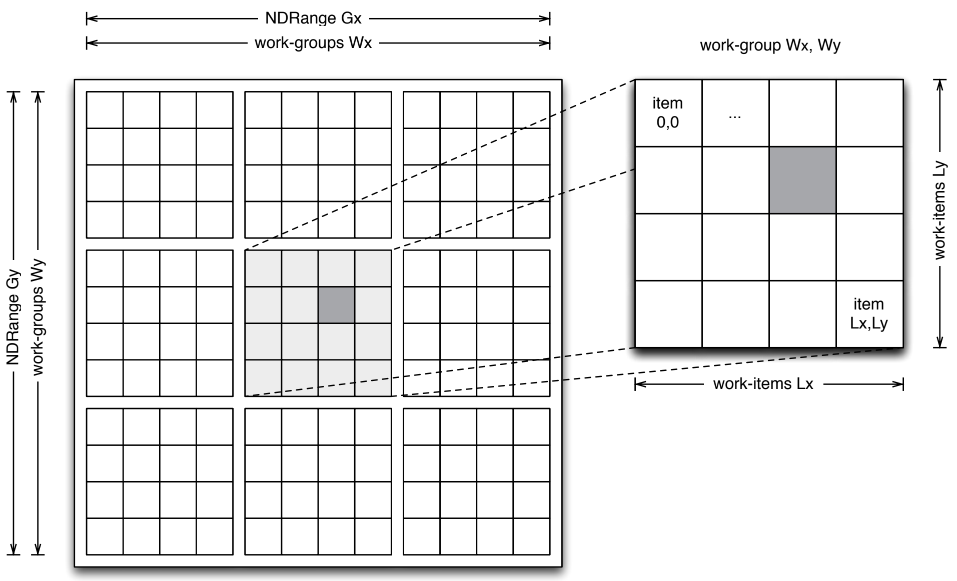


Figure 3.1.: Kernel execution model with two-dimensional NDRange index space. Here each work-item has a global index G_x, G_y and belongs to a work-group W_x, W_y. As member of a work-group, a work-item also has a local index L_x, L_y. In the example, the highlighted work-item has the indices $G = (6,5); W = (1,1), L = (2,1)$ [MGM11, p. 16]. The indices start with zero.

Supported Devices

The OpenCL standard defines an *Embedded Profile (EP)* for handheld devices that relaxes some of the requirements in the standard for such systems. OpenCL officially supports Android devices since version 2.0 [Gro13]. However, the real situation regarding OpenCL support on smartphones is more complicated. The Android devices Nexus 4, Nexus 10 and Sony Xperia Z have been found to come with OpenCL EP 1.1 drivers [Pic13; Sca13; com13]. Unfortunately, support is only available with Android version 4.2 and has been removed with version 4.3 in favor of RenderScript [Gar13]. Although Google received complaints from many developers about this step, future support of OpenCL on Android stays unclear at the time of writing since its execution model is believed to be too hardware-centric by Google officials. Hindriksen [Hin13] gives a good overview about the discussion.

Although OpenCL itself is backed by Apple, iOS devices do not have official support for the framework so far. But apparently there is a way to interact with OpenCL via private API calls as shown by Yang [Yan13] on iOS devices.

For Windows Phone, no reports about OpenCL support could be found.

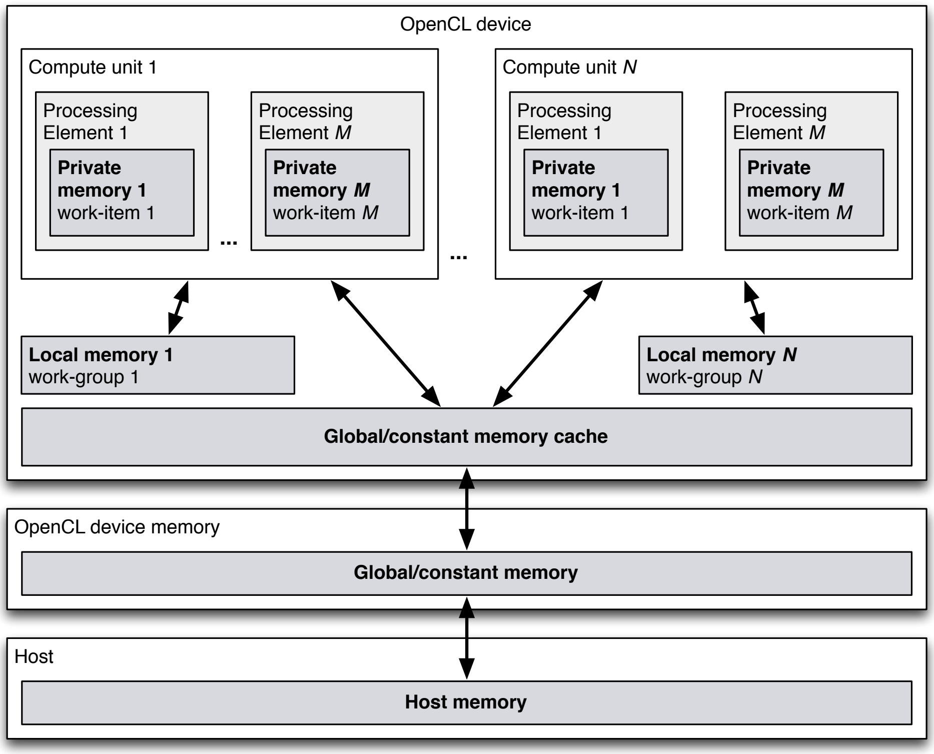


Figure 3.2.: OpenCL memory model hierarchy [MGM11, p. 23].

3.2.2. RenderScript

RenderScript (RS) was originally introduced by Google with Android 3.0 for high performance computations and 3D rendering. Since version 4.1 the 3D rendering API is declared deprecated and the focus is now solely on heterogenous high speed computation [Goo].

Only recently was GPU computing support added to RS with Android 4.2. Therefore most articles about this technology make performance comparisons only between Android Java software development kit (SDK), native development kit (NDK) and RS running on a mobile multicore CPU [QZL12; AAB12; Kem+13]. However, the article by Sams [Sam13] suggests a strong performance increase for RS computations running on a heterogenous CPU/GPU environment exemplified by the Nexus 10.

Conceptional Foundations and Supported Programming Models

The above article also points out the fundamental difference to OpenCL's approach towards hardware abstraction. While OpenCL exposes low-level hardware features and expects the developer to make intelligent decisions about which algorithms are to run on which hardware component, for RS "[n]o effort is required on an app developer's part to enable this [GPU] acceleration." Here, the developer lays the decision about hardware responsibility in the hands of an (undocumented) RenderScript compiler/analysis tool in conjunction with GPU vendor driver specifications. This has pros and cons, such as easier and faster development while assuring portability on the one side, but nontransparent work distribution on the other side.

Regarding the execution model, RS follows the kernel concept also used by OpenCL and CUDA. Such kernels include the high performance compute tasks and are written in a C99-like language. Their lifetime, execution and host memory access is controlled in the Android application side via a Java API. As already mentioned, the work distribution cannot be controlled by the developer and is instead performed automatically by RS under unknown assumptions. In the application build process, all RS kernels are compiled to device-independent *bitcode*, which in turn is compiled on-the-fly to native code on the device [Gui12, p. 233]. This model similar to the Java virtual machine (VM) assures portability (at least between Android devices), but in contrast to Java on Android it does not compile to native code during execution (*just-in-time*) but only once at startup (*ahead-of-time*), which delivers better performance.

The programming models of RS can also be compared to OpenCL or CUDA. Both data and task parallelism can be implemented, but it remains unclear if the latter also utilizes the GPU. In respect to the described execution model, one cannot surely say if the RS compiler can automatically generate efficient native code for GPUs for anything besides data parallel algorithms, since task parallelism usually includes a large amount of conditionally executed code and unpredictable memory accesses. For highly data parallel algorithms *Filterscript* was introduced with Android 4.2, which is a more restrictive subset of RS suited for GPUs and aimed at image processing algorithms like convolutions [Voi13].

OpenCL gives full freedom to the developer, potentially burdening her or him with hardware details. RS lifts this burden but introduces non-transparency regarding GPU utilization and load-balancing. One of the goals of this work is to find out which of these two approaches is best under what kind of circumstances.

Supported Devices

Since RS is an Android API, only Android devices are supported. It requires at least OS version 3.0 and for GPU computing support version 4.2 or above is needed. Furthermore, the GPU vendor must support this technology in its drivers. Unfortunately there is currently no list of supported Android devices for RS computing using the GPU, but Nexus 4, 7 and 10, as well as Motorola Xoom are reported to work. GPU utilization decisions stay obscure with RS and only very little documentation is given on the topic at the time of writing.

3.2.3. OpenGL ES

OpenGL for Embedded Systems (ES) is an API specification for hardware-accelerated graphics rendering on handheld systems such as mobile devices. There are several versions that are widely supported on mobile devices: Versions 1.0 and 1.1 implement the fixed rendering pipeline of the original OpenGL 1.x API but have an overall stripped down, lightweight interface. Since the rendering pipeline is not programmable, these versions are of no use for serious GPGPU programming. OpenGL ES 2.0 changed this and introduced a programmable pipeline via *shaders* – a concept drawn from the desktop OpenGL 2.0 specification [MGS09, p. 3]. The programmability of the pipeline was further extended with the introduction of OpenGL 3.0.

Conceptional Foundations and Supported Programming Models

Shaders determine the functions of certain stages in the graphics pipeline and can be written in OpenGL Shading Language (GLSL) which employs a C-like syntax. Just like OpenCL kernel programs, these shaders are built at run-time. The OpenGL ES 2.0 rendering pipeline can be seen in figure 3.3 where the two programmable shaders, at vertex and fragment stage, are highlighted. The former allows per-vertex operations such as transformations, whereas the latter is used for per-fragment operations (i.e. operations on each pixel) that produce the final pixel colors for the *framebuffer*. A framebuffer is the final rendering destination in OpenGL and is usually displayed on screen, but it is also possible to specify a *framebuffer object (FBO)* as rendering target. Such an FBO can then for example be used for subsequent rendering passes or for copying back its contents to the main memory. For a detailed description of the OpenGL ES 2.0 rendering pipeline see dedicated literature such as [MGS09].

OpenGL as such does not directly support GPGPU computation or parallel programming models in the stricter sense, but it was found that some of its features for graphics rendering can be exploited for GPU-accelerated, data-parallel calculations. Göddeke [Göd06] provides four main concepts to turn the OpenGL graphics rendering pipeline into a GPU-based accelerator for general purpose calculations:

Arrays as textures: Textures can be used to pass data to the GPU. The most obvious and directly supported use is to pass image data as textures for image processing. But it is also possible to pass arbitrary data as long as it complies with the supported image formats of the GPU. The API defines a function `glTexImage2D` for copying (image) data to the GPU to be used as a texture. To read back the result

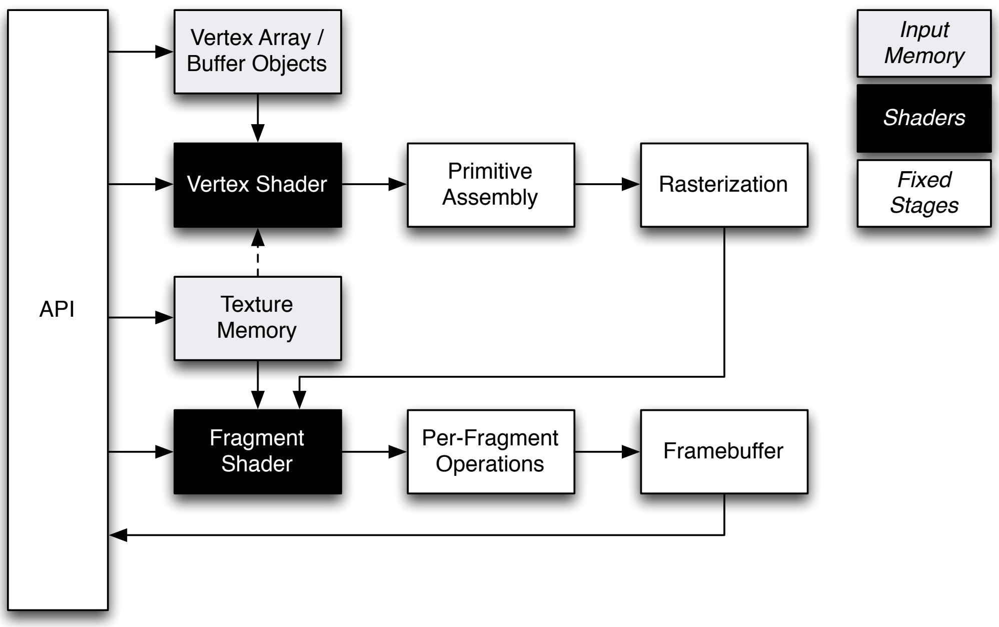


Figure 3.3.: OpenGL ES 2.0 render pipeline [MGS09, p. 4].

after rendering, `glReadPixels` can be used to transfer the framebuffer output to the main memory.

Kernels as shaders: OpenGL shaders implement the operations that are performed on the vertex and texture data. They are written in GLSL and are compiled during run-time. As with OpenGL ES 2.0 two main approaches can be used (and combined). Data can be passed in as vertex buffer being processed in the vertex shader, and as texture being processed in the fragment shader (more common) *or* vertex shader (less common). With this, data-parallel operations defined in the respective shaders can be performed on the input data.

Computing is drawing: For the calculations to be executed, an output image needs to be rendered by OpenGL. This is done just as with usual graphics rendering, only that off-screen rendering into a framebuffer object might be performed. In many cases it is sufficient to specify a simple geometry for a quad that covers the whole display area on which the data will be rendered as texture. To specify the dimensions of the generated output, `glViewport` is used. For example, in image processing, the viewport dimensions can be set to the resolution of the desired output image – OpenGL scales the rendered image on the fly. Another example might be calculating the histogram of a grayscale image. Here a viewport of 256x1 pixels can be set where each pixel in width represents a histogram bin.

Feedback: Often, more complex computations require multiple rendering passes, which means that several different rendering passes are chained together so that the output of one pass is the input of another. An example might be low-pass filtering with a Gauss kernel in the first pass and thresholding in a subsequent rendering pass. With FBOs this can be easily achieved since one rendering pass can use the generated output "texture" as its input "texture".

Implementing this concept results in a program flow depicted in figure 3.4. Using multiple rendering passes as shown in the schema is of course optional but is usually required. Often, images need to be rotated before being processed in the shader. This can easily be achieved by setting the proper texture coordinates.

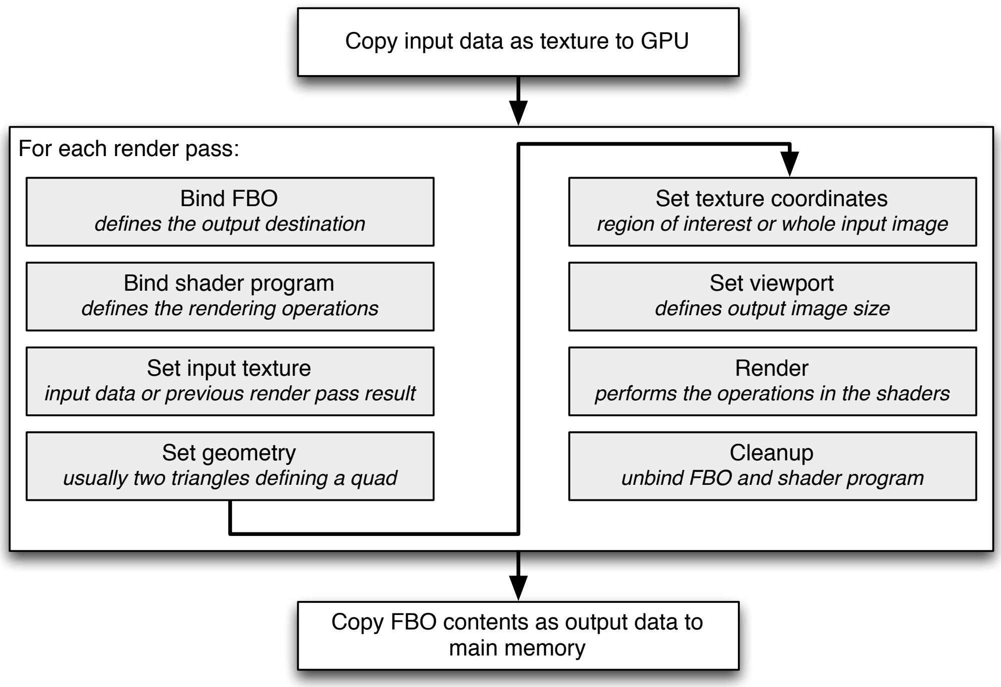

Copy input data as texture to GPU
For each render pass:
Bind FBO
defines the output destination
Bind shader program
defines the rendering operations
Set input texture
input data or previous render pass result
Set geometry
usually two triangles defining a quad
Set texture coordinates
region of interest or whole input image
Set viewport
defines output image size
Render
performs the operations in the shaders
Cleanup
unbind FBO and shader program
Copy FBO contents as output data to main memory

Figure 3.4.: Basic process sequence of a GPGPU program using OpenGL.

Limitations

There are several limitations for GPU computing with OpenGL in general and with the ES 2.0 variant in particular. While advanced versions of desktop OpenGL (version 4.4 and above) have *compute shaders* for arbitrary computations that can be divided into work-groups, no OpenGL ES version supports any GPGPU functions out of the box at the time of writing. Some of these functions can be simulated. For example, multiple rendering passes have practically the same effect as memory barriers. Others, like work distribution or shared memory for work-groups, cannot be replicated. Whilst *geometry shaders* could be exploited for some GPU computing tasks with desktop OpenGL (see [Dia07]), they are not available in OpenGL ES. With vertex and fragment shaders being the only programmable pipeline stages, only limited data-parallel algorithms can be implemented efficiently. Another issue is that only textures can be handled as input data, which means that the data to be processed must conform to a certain image format or must otherwise be converted, which introduces additional overhead. Because the supported image formats for FBOs are very limited in OpenGL ES 2.0, it is hard to process arbitrary (non-image) data. And even image data can sometimes not be as efficiently handled as it could be. For instance, the `GL_LUMINANCE` format for single-channel grayscale images is not supported for FBOs (as found out in experiments), so shaders must always operate on four-channel color data even if this is not necessary. This is not such a big problem for the rendering performance, but it is problematic for the image data transfer times when FBO contents need to be copied back to the main memory. Another point that adds some overhead is the FBO rendering itself and copying vertex buffer data to the GPU, which is necessary because the textures must be rendered to some primitive at the end. Although very little data is copied, it nevertheless adds additional instructions, that OpenCL or RS do not require.

One should keep in mind that OpenGL is designed for rendering 2D and 3D graphics, so exploiting it for general-purpose computations is a rather daring interpretation of its specification and features. The techniques described by Göddeke [Göd06] can be seen as workarounds that originate from a time prior to GPGPU technologies such as OpenCL, CUDA or compute shaders. However, since such technologies are only widespread on desktop or server systems, OpenGL-based GPU computing stays a viable solution for portable mobile applications.

Supported Devices

OpenGL ES is the most widespread graphics API for mobile devices, because at least version 1.1 is available on all Android and iOS devices. Apple states that version 2.0 is available on all iOS devices since the iPhone 3GS and version 3.0 is supported by all devices with an Apple A7 GPU (iPhone 5s, iPad Air, iPad Mini Retina) [App13]. Google's *device dashboard* [Goo14] shows a figure displaying a large majority (93.5%) of OpenGL ES 2.0 devices. Version 3.0 is supported by 6.4% of all Android devices, whereas only 0.1% support solely version 1.1. Mobile operating systems added OpenGL ES 3.0 support during 2013 (Android 4.3 and iOS 7). Summing this up, OpenGL ES 2.0 is the most widely used technology that can be exploited for general-purpose GPU computing on mobile devices, at least within some limitations. Since RenderScript can only be used on Android systems and OpenCL is hardly supported at all, OpenGL ES 2.0 is the only portable option for GPGPU at this point in time.

3.2.4. Other Parallel Programming Technologies

There are many other parallel programming technologies and APIs, but none of them has widespread support for GPU computing on mobile devices. *CUDA (Compute Unified Device Architecture)* is the original GPGPU platform developed by NVIDIA. Although it is available on mobile development platforms [Har13], it has no widespread support on mobile device operating systems, since it is bound to NVIDIA GPUs like *Tegra*. Microsoft's OpenCL competitor *DirectCompute* does not target mobile devices but GPGPU could be emulated on Windows Phones using *Direct3D* shaders.

3.3. Related Work about GPGPU on Mobile Devices

A lot of research has been done in the field of GPU computing on stationary computers. [Owe+05; Owe+08; ND10] give a good overview of recent developments. Research about the GPGPU capabilities of mobile devices has just begun, but some interesting reports and case studies have already been published on this topic. Many authors tried to utilize the GPU for image processing or computer vision tasks, but not all of them reported an increase in performance. In this section an overview of articles on the topic is given and their results are compared.

3.3.1. Selected Works

The first publications on the topic appeared in 2009. Bordallo López et al. [Bor+09] used the GPU of a Nokia N95 mobile phone to speed up the image warping and interpolation processes in a "document panorama builder" application. The user captures a video while moving the mobile phone camera over a document and the program will select several still images that form a mosaic of the complete document. The suggested algorithm uses scale-invariant feature transform (SIFT)[3] for feature extraction of the selected frames for later stitching. To correct varying orientations and letter sizes due to the changing viewing angle and distance, each frame is warped and interpolated. Bordallo López et al. [Bor+09] at first suggested to utilize the GPU for SIFT, but report that the fixed rendering pipeline of OpenGL ES 1.1 is not flexible enough for this task. Therefore only the warping and interpolation is done on the GPU, resulting in an "about four times" faster computation compared to the implementation on the CPU. The authors also consider that the parallel programming model of OpenCL and CUDA would be beneficial for their use case. However, since that model was not supported on their device, the results are based on an OpenGL ES 1.1 implementation.

Leskela et al. [LNS09] experimented with OpenCL EP 1.0 on mobile devices in 2009. However, due to the lack of OpenCL support on consumer devices their research was

[3]See [Low99].

conducted using a Texas Instruments (TI) OMAP 3430 development board. Performance measurements were carried out using an example algorithm with three image processing tasks: geometry adjustment, Gaussian blur and color adjustment. Good results (3.4x speedup) are achieved with the OpenCL implementation running on the CPU and GPU concurrently and slightly better results (3.6x) were reported using exclusively the GPU for the image processing tasks. In terms of energy efficiency the GPU did also outperform the CPU in their experiments and uses only 14% (GPU only) or 33% (heterogenous CPU/GPU utilization) of the energy per image compared to the algorithm running on the CPU alone. However, the authors note that the comparative implementation on the CPU was not optimized for ARM processors.

Another paper that reports taking advantage of programmable shaders for GPGPU with OpenGL ES 2.0 was presented in 2010 by Singhal et al. [SPC10]. They propose a shader-based image processing toolkit with several features including image scaling, color transformation, convolution operations, edge detection and much more. These functions can be applied for real-time video processing as three example applications show: video scaling, cartoon-style non-photorealistic rendering and Harris corner detection.[4] For the latter two applications the performance appeared not to be good enough for real-time processing on their hardware. The authors report between 6 and 20 frames per second on a TI OMAP 3430 development board, depending on the video resolution (320x240 up to 640x480 pixels). Unfortunately this paper does not contain a comparative implementation running on the CPU alone.

The first comparative results on GPU computing with OpenGL ES 2.0 shaders were published in 2011. Surprisingly, little or no performance or energy improvements were reported. Bordallo López et al. [Bor+11] have further deepened their research in the field and implemented image scaling and the Local Binary Pattern (LBG) algorithm [5] using OpenGL ES 2.0 on several devices.[6] The expected performance boost was not observed, though. The implementation running on the GPU was slower than on the CPU, but at least "the GPU performance increase[d] as the picture size [grew]", which is believed to be "due to improved parallelization." However, an implementation with heterogenous CPU/GPU utilization was able to moderately outperform the CPU-only solution. Furthermore they observed that the GPU gave much better performance results on per-pixel tasks such as scaling and color preprocessing than did the CPU. This supports the assumption that not all algorithms are suited well for GPU computation, especially not on mobile devices. Then again, their implementation apparently did not use optimal memory access patterns for GPUs, which might cause the partly disappointing results.

Canny edge detection[7] is believed to be an algorithm that is not ideal for GPU computation, because its CPU implementation usually involves many conditional statements.

[4]See [HS88].
[5]See [OPH96].
[6]Namely: Beagleboard rev. C, Zoom AM3517 EVM board, Nokia N900 Mobile phone.
[7]See [Can86].

Nevertheless an implementation on mobile device GPUs was conducted by Ensor and Hall [EH11] and is described in their article. The authors note that implementing this algorithm using OpenGL ES 2.0 shaders is quite challenging since it includes “a large amount of conditionally executed code and dependent texture reads.” Several strategies are depicted on how to handle these circumstances efficiently. With this they achieved to implement Canny edge detection “without any conditional statements whatsoever, ideal for a GPU shader-based implementation on OpenGL ES.” Tests on a wide set of consumer devices showed diverging results. Many devices like the Nexus One or Galaxy S did not profit from GPU computation in the experiments; the iPhone 4 speedup was almost not measurable (1.03x). By contrast, the application running on a Desire HD sped up the algorithm by 1.4x. Moreover, the authors report that the algorithm scales with improved hardware resources so that for example the Galaxy S II achieved a speedup factor of 2.4x. This clearly shows that even algorithms that seem not to be suited well for GPGPU can nevertheless be ported for it and achieve good speedup rates, although potentially difficult to implement.

Cheng and Wang [CW11] can also confirm this as they described the possibilities of GPGPU on mobile devices for face recognition based on Gabor wavelets.[8] Fast Fourier Transform (FFT) is used to perform the Gabor kernel operations in Fourier space, which is done on the GPU via OpenGL ES 2.0 shaders on a Tegra 2 development board. This task is reported to be 4.3x faster than the CPU implementation and uses only 26% of the energy per image.

Since 2012 more and more papers have reported performance improvements for GPU computing on mobile devices. Hofmann et al. [HSR12] have implemented Speeded Up Robust Features (SURF)[9] with OpenGL ES 2.0 shaders. They presented an interesting approach where the shader programs are generated at run-time for device specific adjustments. Their tests on several consumer devices[10] have shown impressive speedup factors ranging from 2x (Galaxy Nexus) to 14x (iPad 4G).

Rister et al. [Ris+13] reworked the SIFT algorithm first ported for mobile device GPU computation by Kayombya [Kay10]. Their implementation was aimed at minimizing memory transfers between CPU and GPU, better workload balancing and efficient image data reordering. Just as did Hofmann et al. [HSR12], shader programs are generated on-the-fly with user-defined parameters, which results in “branchless convolution” – no loop conditions need to be checked by the hardware. They report speedup factors from 4.7x (Galaxy Note II) to 7.0x (Tegra 250). Even better results are reported by Wang et al. [Wan+13] for an object removal and region filling algorithm[11] implemented with OpenCL on a Snapdragon S4 development board. When running on the CPU, the algorithm took 393.8s compared to only 4.3s on the GPU for the best configuration, which yields 92x

[8]See [Su+09].
[9]See [BTG06].
[10]See table A.1 for details.
[11]As described in [CPT03].

speedup. However, it should be kept in mind that the OpenCL implementation on the CPU might not have been optimized for the processor.

To conclude, Pulli et al. [Pul+12] give an extensive overview of the topic with a focus on computer vision (CV) algorithms in OpenCV.[12] They compare heavily optimized CPU implementations (using NEON[13] optimizations and TBB multithreading) of several algorithms with variants that utilize the GPU for applicable tasks like image resizing and warping. In the example applications that included panorama stitching and video stabilization on a Tegra 3 development board, the GPU powered implementations achieved a total speedup of 1.5-2.0x and 5-6x, respectively.

3.3.2. Conclusive Thoughts about Related Research

Table A.1 in the appendix provides an overview of the discussed papers and their results. This extensive review of prior art about GPU computing on mobile devices shows common approaches and certain trends that developed in the last four to five years. Most researchers focused on GLSL implementations for their experiments, since it is supported by almost every mobile device with a GPU (except for Windows Phone). Some results were rather disappointing but might be caused by non-optimized shaders with a large amount of control sequences and irregular memory accesses [Kay10; Bor+11]. This underlines that shader programming is challenging and requires a good understanding of the special GPU architecture on mobile devices. Having jumped this hurdle, GPGPU delivers very promising results for specific domains. Many papers discussed low-level image processing tasks like resizing, warping or color conversion. Some even successfully implemented high-level computer vision algorithms like SIFT or SURF. The majority of articles reports performance gains and those that measured power consumption also reported more energy efficiency in most cases. Although many experiments were conducted with development boards, successful implementations on a wide range of consumer devices could also be shown [EH11; HSR12; Ris+13].

OpenCL on mobile devices was rarely examined for GPU computing, which is probably due to limited support on such devices [LNS09; Wan+13]. However, tests have shown that OpenCL might be a suitable solution if a wide range of manufacturers decided to add support. Its advantage is that it is easier to implement and less dependent on GPU hardware, because the OpenCL standard specifies certain hardware requirements. But since OpenCL is not widespread, RS is supposed to step into the breach, at least for Android devices. Unfortunately GPU computing support is still quite new in RS, so no publications closely analyzed its potential so far. This thesis will try to fill this gap and will also provide comparative results with RS and OpenCL. Furthermore, the application domain for GPGPU should be broadened so that algorithms from other DSP areas such as audio processing can be taken into consideration.

[12]OpenCV is an open-source computer vision framework. See `http://opencv.org`.
[13]NEON is an SIMD instructions extension for ARM processors.

4. Analysis and Comparison of GPGPU Technologies on Mobile Devices

Available GPGPU technologies and their programming models on mobile platforms have been introduced in section 3.2. This chapter focuses on a practical view and compares the mentioned technologies in terms of their performance, features, developmental support and documentation. Several fundamental problems in different fields of digital signal processing are selected for comparison purposes and are implemented in prototypes using respective GPGPU technologies. The results of this chapter affect the decision which technology to use in the practical project presented in chapter 5.

4.1. Selected DSP Tasks and their Implementation for the Purposes of Analysis and Comparison

Three popular tasks in the field of DSP are chosen to compare different GPGPU technologies on mobile platforms. Of course, a vast number of problems in such a big area of computer science exists. Multiple solutions in the form of algorithms are already present. Choosing only three of these solutions cannot be considered representative. Instead, an overview is given here of the different possibilities, features and implementation difficulties of several technologies. Possible performance improvements that can be expected with comparable problems are examined.

4.1.1. Generic Kernel-based Image Convolution

Convolution is a fundamental image processing task that takes the *neighborhood* of pixels into account for image transformations [RR08, 57f.]. It is widely used for several purposes such as blurring, sharpening, edge-detection and many more. Convolution is a linear filter operation that can be performed by applying a matrix or *kernel* K to all pixels of an image I_{in} to produce a convoluted image I_{out} as in equation 4.1.

$$I_{out}(x, y) = \sum_{i,j} K(i, j) I_{in}(x + i, y + j). \tag{4.1}$$

Since the kernel operations can run independently and require few conditional statements (usually only for image borders) it is optimal for parallel processing and can be implemented by using a data decomposition approach (see section 2.4.2). The image can be divided in N equal-sized regions of pixels that are handled by N workers in parallel. Another optimization is separating a kernel convolution into two convolutions, one for vertical and one for horizontal direction, and apply only an one dimensional kernel row respectively. This reduces the order for this operation from $O(N^2)$ to $O(N)$ for a NxN sized kernel. The problem is that this optimization can only be adopted for separable kernels, i.e. for a matrix K where $rank(K) = 1$ [Edd06]. Examples are averaging kernels or Gauss kernels [RR08, p. 79].

4.1.2. Hough Transform for Line Detection

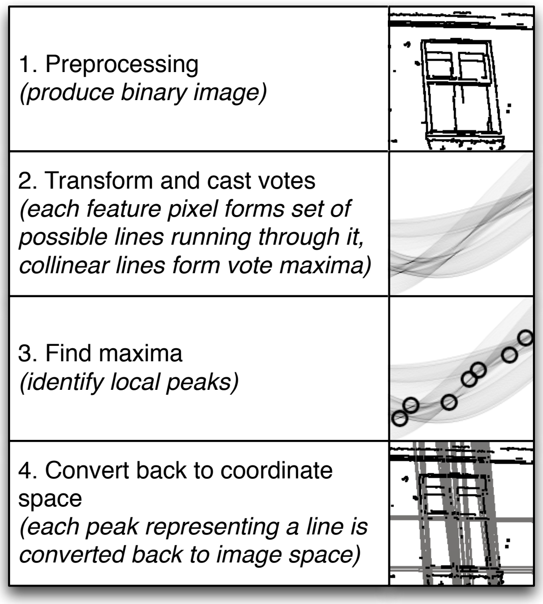

1. Preprocessing
(produce binary image)
2. Transform and cast votes
(each feature pixel forms set of possible lines running through it, collinear lines form vote maxima)
3. Find maxima
(identify local peaks)
4. Convert back to coordinate space
(each peak representing a line is converted back to image space)

Figure 4.1.: Generalized Hough transform approach.

Identifying geometric primitives such as lines or circles and determining their properties is a very common problem in computer vision [Rus07, p. 560]. This step is usually performed after potentially interesting features such as edges have been detected in a previous step. Such features are represented in a binary image. For the case of identifying straight lines in such an image, Hough [Hou59] produced the basic idea of representing them in parameter space. Here, each feature pixel (i.e. edge pixel) is mapped to a set of parameters which form lines that potentially pass through that pixel. The parameter space is discretized and each additional feature pixel adds a vote for its parameters in a *vote map* (also called *accumulator map*). By finding peaks in this map, collinear pixels that form a straight line can be identified. The generalized idea is depicted in figure 4.1 and is the basis of many variants. A significant extension was suggested by Duda and Hart [DH72]. Hough's initial approach was unable to find vertical lines, since the slope m cannot be unbounded during computation. Duda and Hart eliminated this problem by using polar coordinate representations of lines:

$$r = x \cos\theta + y \sin\theta. \tag{4.2}$$

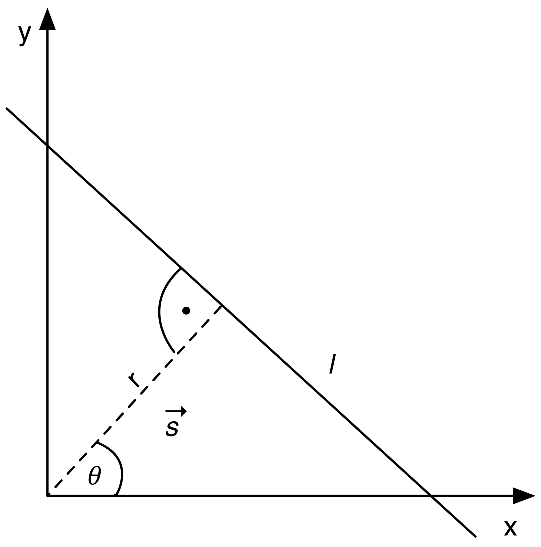


Figure 4.2.: Line l with support vector s, described by polar coordinates r and θ.

In this equation, a line is described by the help of a support vector s, as shown in figure 4.2. Parameters r and θ describe the length of s (Euclidian distance between a line and the origin), and its angle. It is therefore possible to describe each line in an image with a unique pair of θ and r parameters. Hence, with $\theta \in [0, \pi]$ and $r \in \mathbb{R}$ as the discretized parameter space, each feature pixel generates a set of sinusoidal curves $r_{x,y}(\theta) = x\cos\theta + y\sin\theta$ in this space, which represent all potential line orientations for a feature pixel at x, y. For each feature pixel collinear with another in the original cartesian image space, the related curves cross one another, producing peaks in the vote map. By applying some sort of threshold these maxima can be identified so that for each vote map maximum, a line i with parameters r_i and θ_i is found.

In practice, θ is set to have a certain resolution to represent line orientation, such as a resolution of $1°$ steps with $0° \leq \theta < 180°$. The domain of r must similarly be restricted in practical applications. Since it describes the distance from the image center at $W/2, H/2$ (with W and H being the image dimensions), its minimum and maximum value R can be easily calculated as Euclidian distance to the farthest point from the center: $R = \pm\sqrt{(W/2)^2 + (H/2)^2}$.[1]

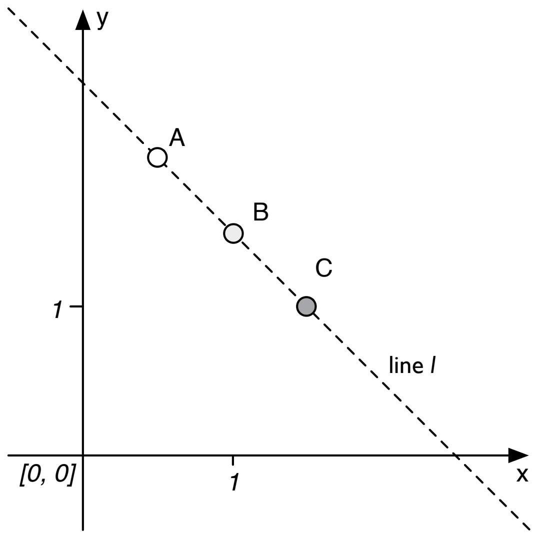
Hough transform can be efficiently parallelized by using data decomposition (see section 2.4.2) because it operates independently on each pixel. However, synchronized memory access to the vote map is crucial in this case. Furthermore, the problem with most variants of the Hough transform is that they are computationally expensive, because they use a "brute-force voting scheme" [FO08], calculating each possible parameter (in a certain discretization step) of each feature pixel. Enough memory to represent the entire voting map in parameter space is therefore required. Fernandes and Oliveira [FO08] further note that since discretization and pixel segments are not exactly collinear, "secondary peaks" are created which result in duplicate or bogus lines. Variants exist that attempt to mitigate some of the described problems. Some of these variants are are discussed in the next section.

Hough Transform Variants

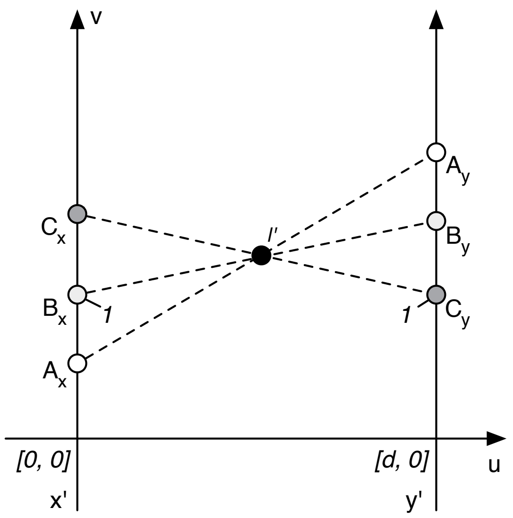
Fernandes and Oliveira [FO08] question the "brute-force" method of the classic Hough transform and suggest an approach that at first tries to identify "clusters of approximately collinear pixel segments". Out of these segments an "elliptical-Gaussian kernel" is generated, which represents the dispersion of feature pixels around the best-fitting line. The larger the dimensions of the kernel, the more dispersed are the feature pixels in the cluster. This means that it is less likely to contain a well-formed straight line. The authors state that especially by forming clusters of possible lines, less spurious lines are detected and the performance is increased. The first steps of this algorithm, which produce the mentioned clusters, include linking together neighboring feature pixels as strings and then subdividing them into segments. The first step, which is basically a border following technique [SA85], is especially hard to parallelize on GPUs since it involves branching and dynamic loops or recursion. Nevertheless it could be possible to parallelize the voting scheme with the elliptical-Gaussian kernel for each identified cluster. The worth of parallelization in this case depends on delivering enough speedup when compared to variants using the basic "brute-force" approach, which might be suited better for GPU computing since they do not involve additional steps for identifying clusters.

An approach that is well optimized for line detection on GPUs is suggested by Dubska et al. [DHH11]. The basic idea is to optimally exploit the capabilities of a graphics accelerator by utilizing its fast drawing functions for geometric primitives. While rasterizing sinusoids (as required by the parameter space introduced by Duda and Hart [DH72]) is a comparatively slow process, lines can be rendered very efficiently on a GPU. To achieve this, the authors introduce a new parameter space to be applied in the Hough accumulator map, which is based on *parallel coordinates (PC)*. As depicted in an example in

[1]In practice, negative values for r are usually avoided by choosing $r \in [0, 2R]$.



(a) Three collinear points and line l passing through them in cartesian x, y-space.



(b) Parallel coordinates u, v-space. Line l is the intersection l' between the collinear points.

Figure 4.3.: Relation between cartesian space and parallel coordinates space [DHH11].

figure 4.3, a point $P_{x,y}$ in cartesian space becomes a line p' from P_x to P_y, which are at separate axis x' and y' in parallel coordinate space u, v. An intersection of these lines happens when two or more points are collinear in cartesian space. These intersection points, however, could exist in infinity for cases such as $y = x$ or $x = a$. The authors therefore used homogenous coordinates in *projective space* where a line can be described as $l : ax + by + c = 0$ (simply written as $[a, b, c]$), which allows representation of the previously mentioned cases. The relationship of l referring to its representation l' in parallel coordinate space is defined by the authors as:

$$l : [a, b, c] \rightarrow l'(db, -c, a + b). \tag{4.3}$$

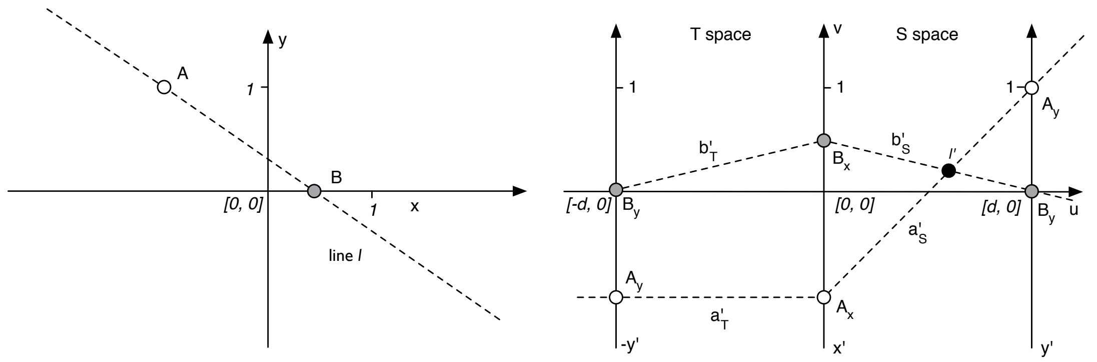
A line which is denoted in the slope-intercept form $l : y = mx+b$ can then be transformed to a point $l' = (d, b, 1 - m)$ in projective space, where d is the distance between the x' and y' axes in parallel coordinate space.

Since l' can only lie between the axes x' and y' if $-\infty < m < 0$, the authors propose to use a *twisted* space T with the axis $x', -y'$ besides the already introduced *straight* space S with x', y'. A point l' lies in this space T if $0 < m < \infty$. This results in two equations for l', depending on m:

$$l'_S = (d, b, 1 - m), -\infty \leq m \leq 0, \tag{4.4}$$

$$l'_T = (-d, -b, 1 + m), 0 \leq m \leq \infty. \tag{4.5}$$

So a line l in cartesian space can either lie as l' in S or T space. In the special cases where $m = 0$ or $m = \pm\infty$, l' is located on either both y' axes (for $m = 0$) or on both x' axes ($m = \pm\infty$). For proper visualization, S and T space can be attached together, creating a TS space depicted with examples in figure 4.4. As can be seen, each point $P_{x,y}$ in cartesian space produces two lines: A line p'_t from P_{-y} to P_x and a line p'_s from P_x to P_y. The intersection points in the TS can be accumulated in a vote map as in the standard Hough transform approach. The discretizations of its dimensions can be chosen on the basis of memory requirements and the requested precision.



(a) Two collinear points and line l passing through them in cartesian x, y-space.

(b) TS space. Since $-\infty < m < 0$, the intersection l' lies in the S space.

Figure 4.4.: Relation between cartesian space and TS space.

The advantage to the proposed approach is its high-performing implementation on graphics accelerators. The vote map in TS space can be seen as a frame buffer, in that a GPU-accelerated program draws semi-transparent lines which accumulate at intersections. The peaks in this vote map can be extracted to find the dominant lines of the input image. The authors present an implementation of their approach based on OpenGL 3.3 shaders and desktop graphics hardware. This achieves real-time performance – even on images with as many as 100,000 edge points and 4 megapixel resolution. The question remains if and how this approach can be transferred to mobile systems with limited resources and restricted APIs.

4.1.3. Audio Synthesis with Real-Time Waveform Generation

In digital audio synthesis, real-time waveform generation is a fundamental approach to generate sounds that can be dynamically controlled and modified by the user. In its simplest form, a basic waveforms such as a sine, square or triangle wave is generated (or taken from a *wavetable*) with a certain frequency which results in a tone at a certain pitch. Such waveforms can be used in conjunction with synthesis techniques – including additive synthesis, frequency or amplitude modulation – to produce a rich variety of sounds. See Roads [Roa96, chapt. 2] for more information about sound synthesis.

In practice, real-time generated sounds are usually produced by filling an *audio buffer* of a certain size with sound samples. These sound samples can be generated using a waveform generation formula $w(t)$ that will produce (usually periodic) discrete samples at a continuous phase t as shown in figure 4.5.[2]

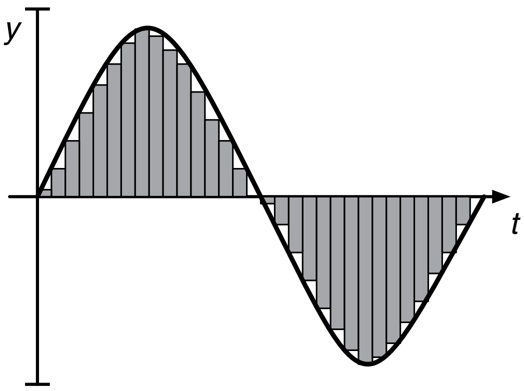


Figure 4.5.: A waveform curve $w(t)$ and its discrete samples.

To obey the Nyquist-Shannon sampling theorem, audio samples must be generated at a sampling frequency of more than 40 kHz for the entire range of human hearing [Sha49;

[2]Often, two stereo channels are represented in the audio buffer as interleaved samples.

Lab03].[3] This, together with complex waveform generation rules results in a computationally expensive task, especially for mobile devices. Parallelizing the generation of samples for the audio buffer should however be possible, as long as the samples can be calculated independently which omits techniques such as feedback loops.

4.2. Testing Environment

4.2.1. Testing Device

A *Google Nexus 10* was the device available for testing all mentioned technologies. Like the Nexus 4 and Sony Xperia Z, the Nexus 10 is one of select devices that come with OpenCL 1.1 system drivers as described in section 3.2.1. Since Google blocked OpenCL support starting with Android 4.3, the device OS has not been updated and therefore remains at version 4.2.1. RenderScript and OpenGL ES 1.1 and 2.0 are available.[4] The following list contains the main hardware features as depicted in the official hardware specifications [Goo13]:

CPU: 1.7 GHz dual-core Cortex-A15

GPU: Mali T604 with 4 shader cores [ARM13]

RAM: 2 GB 12.8 GB/s LPDDR3 RAM

4.2.2. Test Data and Measurements

Only execution time was measured, since memory or power consumption are of less priority in this work. The measurements were taken using direct calls to `System.nanoTime()` (in Java) or `clock()` (in C/C++) and the difference was converted to milliseconds. The calculated average of 10 test runs in the image processing tasks and 100 test runs in the audio synthesis tasks were used to determine the final result. Unless otherwise noted, only the execution time of the main calculations was gauged, without potential image format conversions, initializations, memory allocations, etc. In the GPGPU experiments, the time for memory transfers (host memory to GPU memory and vice versa) was also measured and is noted separately because it is an obligatory task in this context. Furthermore, OpenCL and OpenGL cache the execution of commands. Thus, in related experiments `clFinish()` or `glFinish()` was called to force immediate command execution for exact time measurements.

[3] 44.1 kHz is the industry standard for CD quality and will also be used in this work as default sampling rate.

[4] OpenGL ES 3.0 is also implemented in the Mali T604 GPU but the API is only available from Android 4.3 on which, in turn, disables OpenCL support.

For the image processing tasks, tests were conducted using four different image resolutions (256x256, 512x512, 1024x1024 and 2048x2048 pixels). Three representations of each image were used during the tests: A multi-channel color image[5], and a single-channel grayscale variant for the image convolution task, and a binarized variant with edge features for the Hough transform experiments. Three different sizes (3x3, 5x5, 7x7) of a Gauss kernel were used in the image convolution task.[6] There, speedup per image and kernel size are given for comparison with the CPU-based reference implementation. In the case of the Hough transform mentioned binary images of different sizes are used for performance comparisons. For the audio synthesis task the time for generating 16-bit samples for an audio buffer of different sizes was gauged. Two variants of this experiment were conducted: Generation of a simple sine-wave at a certain pitch, and creation of square waves via additive synthesis as described by Burk et al. [Bur+]. The square waves are constructed with three components (three harmonics) resulting in three additional sine calculations for this variant.

The term *speedup* and its relation to the number of processing cores in the context of parallel computing was introduced in the fundamentals chapter in section 2.3.1. Traditionally it refers to a performance comparison between a single- and multi-core system that employ the same kind of processor or at least the same basic processor architecture. However, in this case speedup rates are given for comparison between an implementation running on the CPU and on the GPU, respectively. Although the test device has two CPU cores and a GPU with four shader cores, a direct relation between the number of cores and the performance cannot be drawn because of the fundamentally different hardware architecture as described in section 3.1.

4.2.3. CPU-based Reference Implementations

To compare the performance results of GPGPU-powered programs with those that run on a CPU only, the performance of CPU-based reference implementations must be measured. Whenever possible, a well-tested, optimized open-source implementation of a certain algorithm was chosen for performance testing on the device.

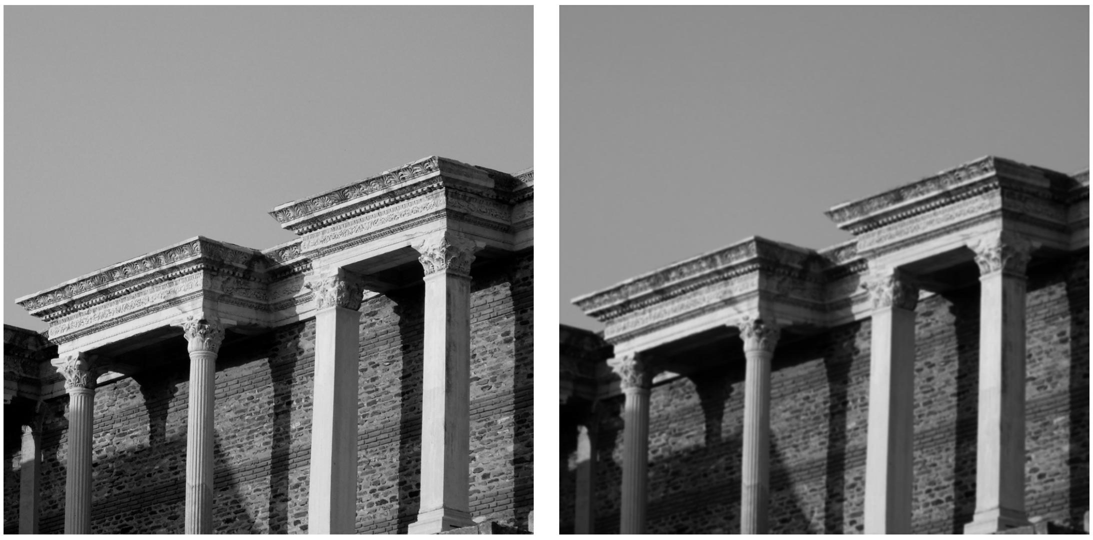
Image Convolution

Kernel-based image convolution has been tested with an implementation that exists in the *Imgproc* package of the open-source computer vision library *OpenCV*.[7] The function

[5]For CPU-based experiments RGB images were used whereas GPU-based tests use RGBA images because of preferred 4-byte format support (see `http://www.opengl.org/wiki/Common_Mistakes#Texture_upload_and_pixel_reads`).

[6]Gauss kernels were used because they contain only non-zero values, which otherwise could cause omitted operations due to optimizations done by the compiler. Although possible, the kernel convolution was not separated into one-dimensional components as shown in section 4.1.1.

[7]See project website `http://opencv.org/`.


(a) Input image.

(b) Output image. Blurring is especially visible in the ornaments.

Figure 4.6.: Example and result image with OpenCV filter2D using a 7x7 Gauss kernel.

is named `filter2D`[8] and can be called with an input image and a kernel matrix. The tests have been conducted using the Java bindings for OpenCV version 2.4.7. The results are depicted in figure 4.7 for four-channel (color) and single-channel (grayscale) images.

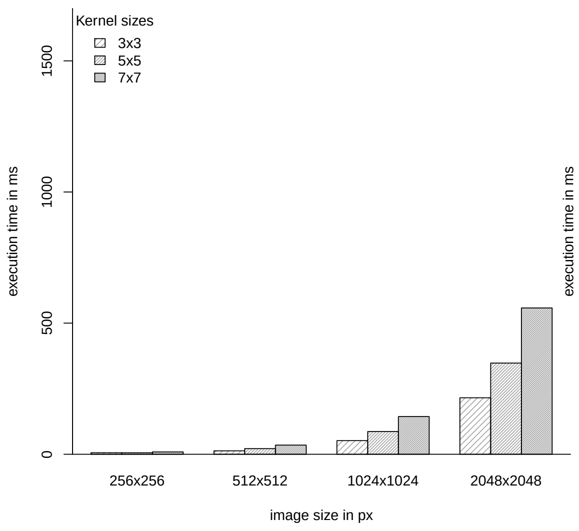
As can be clearly seen, the execution time rises linearly by both the amount of pixels to be processed (determined by the image size) and the kernel size. That three-channel images take about three times longer to process than single-channel images is apparent.[9] Overall, the calculation time is considerably high, especially when working with color images. A test case with 1024x1024 pixels and a 3x3 kernel takes 52ms (about 19 frames per second (fps)) to compute for a color image, which is close to the minimum threshold for real-time applications.

Hough Transform

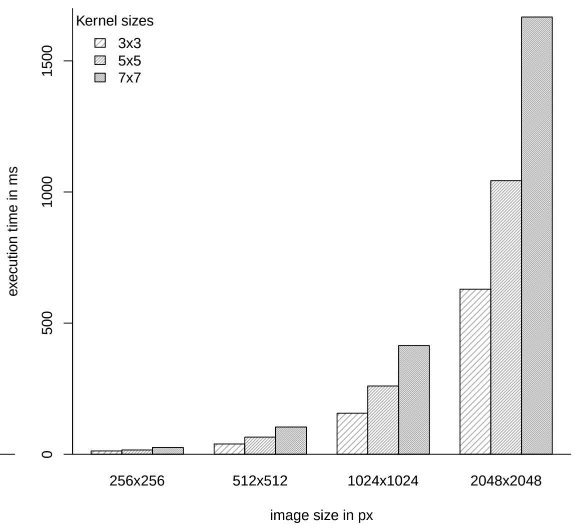
OpenCV also includes two variants of the Hough transform in its *Imgproc* package. The standard algorithm based on [Hou59] and [DH72] is implemented in the function `HoughLines`. A probabilistic approach suggested by Matas et al. [MGK00] is used in the function `HoughLinesP`. Although the latter is described as being "more efficient"

[8]See `http://docs.opencv.org/modules/imgproc/doc/filtering.html#filter2d` for documentation.

[9]Using RGBA images took about four times longer, which indicates that the function also processes the alpha channel and furthermore does not take advantage of 32-bit SIMD instructions on the CPU.



(a) Grayscale images.



(b) RGB images.

Figure 4.7.: Image convolution using OpenCV filter2D.

in the official OpenCV tutorial [Ope13], it was found to be around 30% slower than `HoughLines`. This may be because `HoughLinesP` also calculates the origin and endpoints of each line, whereas the other variant only provides the straight line parameters as polar coordinates.

For testing, the same version of the Java bindings for OpenCV has been used as in the image convolution task. Each input image is at first preprocessed with the Canny algorithm [Can86], which produces a binary image with detected edges. The results are provided by measuring the execution time of the `HoughLines` function and can be seen in figure 4.9. Even for very small images of 256x256 pixels the function takes 80ms to complete, which cannot be considered real-time. That the Hough transform also requires other computationally expensive steps such as edge detection beforehand should also be kept in mind.

Audio Synthesis

A simple sine wave and an additive square wave audio synthesis variant as described in the previous section were implemented using the Android *AudioTrack API*[10] in stream-

[10]See `http://developer.android.com/reference/android/media/AudioTrack.html` for documentation.

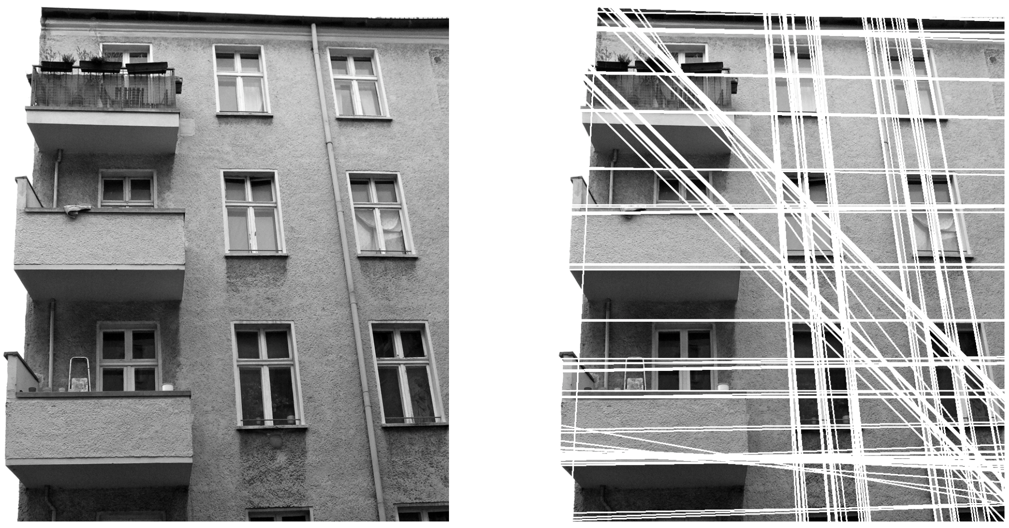

(a) Input image. (b) Output image with detected lines.

Figure 4.8.: Example and result image with OpenCV HoughLines.

ing mode. Sample buffers with sizes of 4096, 8192 and 16384 samples were generated using the two variants by calculating each sample in a `for`-loop with a shifting phase for the sine calculation(s). `Math.sin()` was used instead of a (probably faster) sine lookup-table. The results are depicted in figure 4.10.

As expected, calculation time rises linearly with the sample buffer size since the buffer is not divided into different ranges that could be calculated independently on multiple processing cores. Furthermore, the additive variant which involves three more sine calculations per sample is about two times slower. The calculations are performed quite fast nevertheless, allowing real-time synthesis. Using optimizations such as lookup-tables for sine and cosine calculations or CPU SIMD instructions (in C/C++ via Java Native Interface (JNI)) could further improve performance.

4.3. OpenCL

Please note: To avoid confusion, *kernel* here always refers to an image convolution matrix whereas *OpenCL kernel programs* will be simply called *OpenCL programs* throughout this section.

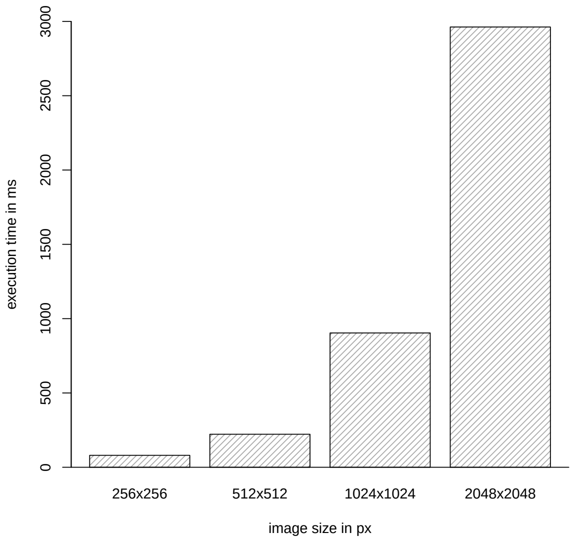


Figure 4.9.: Hough transform using OpenCV HoughLines.

4.3.1. Special Characteristics

As described in section 3.2.1, OpenCL support on mobile devices is rare and is in an experimental stage where available. Therefore, the special characteristics of this technology and its implementation on the Nexus 10 should be described.

Setting up an OpenCL development environment manually is necessary since it is not officially supported. This could be achieved by using the instructions provided by Scarpino [Sca13]. OpenCL calls could thereafter be issued using its C API. The header files for OpenCL must be obtained from the ARM Mali Developer Center[11] and the Mali GPU driver library *libGLES_mali.so* must be linked to the executable.

The Nexus 10 comes with an OpenCL EP 1.1 environment described at compubench.com [com14] and which can be confirmed by querying the properties on the device. Important environment variables are `CL_DEVICE_MAX_COMPUTE_UNITS`, which indicates the maximum number of available parallel processing units and is set to 4 on the Nexus 10, and `CL_DEVICE_MAX_WORK_GROUP_SIZE` which indicates the maximum work-group size and is set to 256. As already described in section 3.2.1, this property describes the maximum work-items a work-group may house. One can specify the number of overall work items *(global work size)* and the number of work-items per group *(local work size)* (and thereby indirectly the number of work groups) for OpenCL program execution to define shared memory between work-items. For tasks with independent work-items such as with these

[11]See `http://malideveloper.arm.com/develop-for-mali/sdk/mali-opencl-sdk/` for more information.

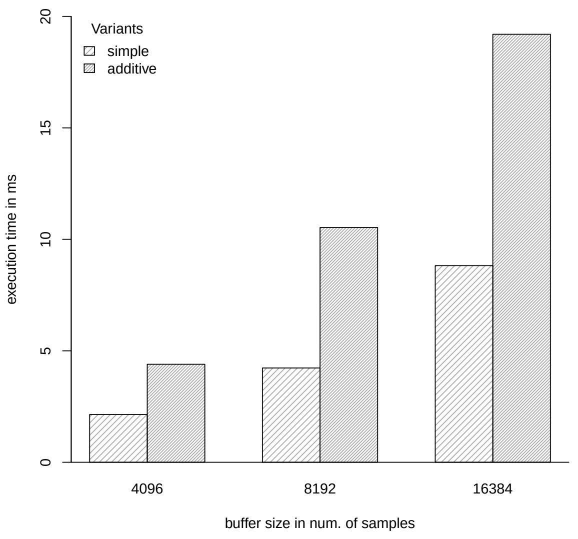


Figure 4.10.: Audio synthesis (sample buffer calculation) on the CPU.

tests, explicit declaration of specifications can be omitted, allowing those properties to be chosen by the system.

Image support queries on the Nexus 10 revealed that RGBA and BGRA image formats can be used in OpenCL kernels. It was possible to copy and fetch images to and from the GPU using the respective `clCreateImage2D` and `clEnqueueReadImage` functions. With `clCreateSampler` creation of an image sampler object for the OpenCL kernel to read and write image data was possible.

Problems were encountered during OpenCL kernel compilation which is done on-the-fly during application run-time. Building a kernel from a single source code string that is larger than 1024 bytes would fail because it is truncated to this maximum string size. Fortunately, `clCreateProgramWithSource` permits passing an array of source code strings. Therefore, this problem can be solved by splitting the source code strings into chunks of 1024 bytes or fewer.

4.3.2. Image Convolution Task

Implementation Approach

Image convolution with different kernel sizes was implemented in a data-parallel manner so that each pixel can be processed independently in parallel by the OpenCL program defined in listing 4.1. At first each OpenCL program instance (i.e. each work-item) has to find out which pixel to operate on. This is done by calling `get_global_id(0)` and `get_global_id(1)` (line 7) which returns the global work-item ID for each dimension. Since the work-items operate on each pixel of the image, the result of the function calls is equal to the pixel location. Then, the neighboring pixels and the center pixel are read depending on the current pixel location and the size of the kernel (line 12-14).[12] These pixel values are multiplied with the correspondent convolution kernel value and summed to yield the final pixel value, which is written to the output image.

```
1  // (... 3x3 image convolution matrix k is defined as array at the beginning)
2  kernel void cl_filter3x3(read_only image2d_t srcImg,  // input image
3                           write_only image2d_t dstImg, // output image
4                           sampler_t sampler,
5                           int width, int height)      // image dimensions
6  {
7    int2 pos = (int2)(get_global_id(0), get_global_id(1));  // abs. pixel position
8    // calculate resulting pixel color
9    float4 resPx = (float4)(0.0);
10   for (int kIdx = 0; kIdx < 9; kIdx++) {  // go through the convolution matrix
11     // calc. neighborhood pixel offset
12     int2 pxOff = (int2)(kIdx % 3 - 1, kIdx / 3 - 1);
13     // read pixel, multiply by convol. matrix value, add to result pixel color
14     resPx += read_imagef(srcImg, sampler, pos + pxOff) * k[kIdx];
15   }
16   // write the result pixel color
17   write_imagef(dstImg, pos, resPx);
18 }
```

Listing 4.1: OpenCL 3x3 image convolution kernel excerpt.[13]

The distribution of work is handled entirely by OpenCL, because there is no need to strictly define work-groups due to independent work-item operations in this task.

The proposed approach is obviously a simple one and has notable weaknesses. Neighboring pixels are read multiple times from nearby work-items resulting in unnecessary memory accesses. To prevent this, local data caches could be implemented as proposed by Gaster et al. [Gas+13, p. 164]. Further optimizations could include vector-read operations and aligned memory accesses [Gas+13, pp. 167-169]. However, the simplified implementation used here for testing should be sufficient to estimate the performance improvement potential of OpenCL.

Results

At first, measurements of the data transfer time between host and GPU memory were taken by copying uncompressed RGBA images of different sizes. The results are depicted in figure 4.11. Interestingly, copying data *to* the GPU took about 4 to 6 times longer

[12]Note that image border checks can be omitted due to enabled image edge clamping.
[13]OpenCL C data types such as `int`*n* or `float`*n* describe vectors of *n* length.

than vice versa. The former yields a data transfer rate of about 200 to 300 MB/s, whereas the latter ranges from about 900 to 1700 MB/s.

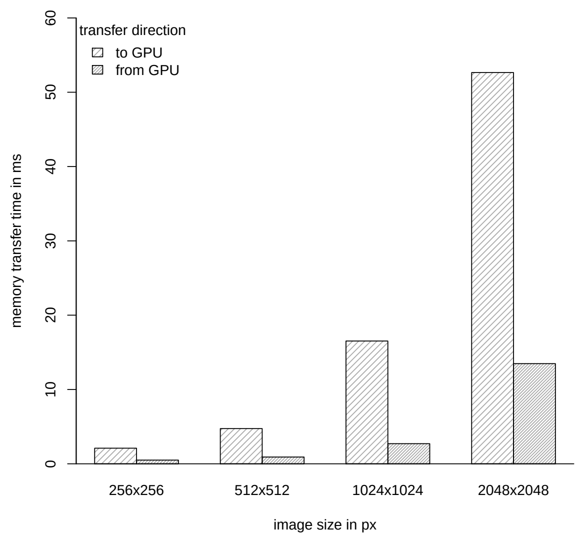
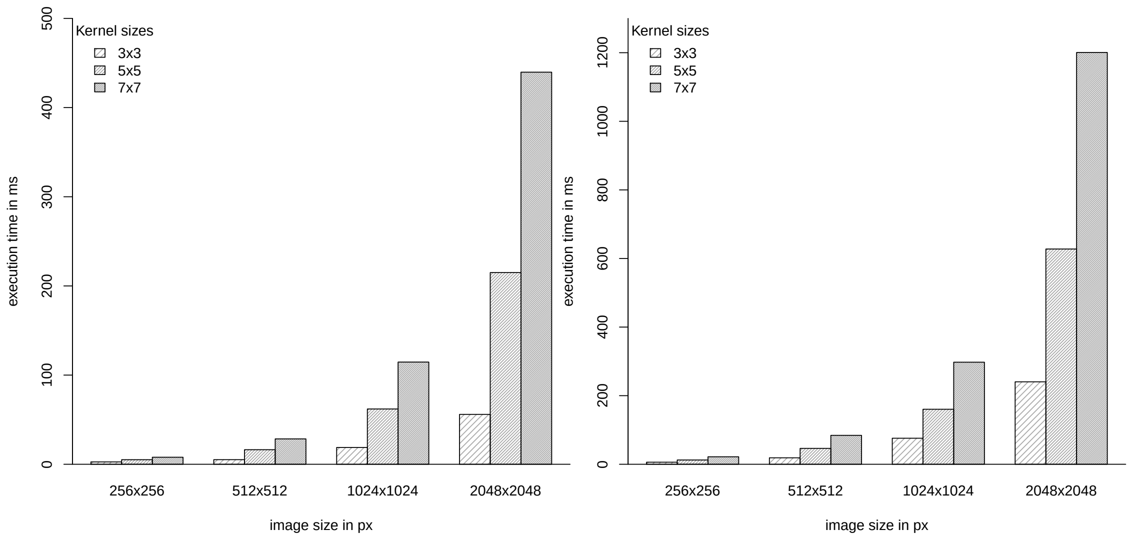


Figure 4.11.: Memory transfer times using OpenCL.

The times for the image convolution operation using OpenCL are shown in figure 4.12. During testing a significant performance drop was observed when using loop statements. At first, kernel multiplications were applied using a `for`-loop which resulted in the calculation times depicted in figure 4.12b. Then variants with an "unrolled" loop were created (using automated OpenCL C source code generation written for this purpose), which resulted in an execution time that was about four times shorter than the original program (see figure 4.12a). This optimization is also described in [Gas+13, p. 170] and is justified by insufficient ALU utilization when using "tight computation loops."

To better understand the potentials of OpenCL, the speedup rates in comparison to the CPU-based reference implementation were calculated. To calculate the correct speedup, the total time consisting of computations *and* memory transfers was used. The results can be seen in figure 4.13. Since the variants with an unrolled loop perform much better, their times were used for the comparison. All in all, the speedup rates are quite satisfying, showing a performance improvement between 2x and more than 5.5x over the CPU implementation. The speedup rises when using larger images, indicating that the GPU's performance scales better with increased problem size. On the other hand, the performance drops with bigger kernel sizes, which is caused by increasing image pixel read operations. This could be dampened by local data cache optimizations as described in the previous section.



(a) Without for-loop *(unrolled)*. (b) With for-loop.

Figure 4.12.: Execution times for image convolution using OpenCL.

4.3.3. Hough Transform Task

Implementation Approach

To implement the Hough transform as a parallel algorithm with OpenCL, the approach by Duda and Hart [DH72] (briefly introduced in section 4.1.2) seems most viable. The idea of using parallel coordinate space to draw lines instead of sinusoids into the Hough accumulator map as suggested by Dubska et al. [DHH11] seems similarly promising, but it is heavily optimized for graphics APIs such as OpenGL, which (in contrast to OpenCL) allow fast line rasterization. Therefore, the traditional polar coordinate representation (see equation 4.2) is used in the accumulator space, which generates a set of sinusoids $r_{x,y}(\theta)$ for each feature pixel.

The OpenCL implementation is a parallelized variant of the general Hough transform process and can be divided into the following steps:

Memory initialization: The binary input image of size $W \times H$ is copied to the GPU memory and an OpenCL buffer which represents the memory for the accumulator map is created. The map has the dimensions $180 \times 2R$, where the former describes the discretized values for the θ angle ($1°$ steps with $0° \leq \theta < 180°$) and the latter describes the maximum value for r with $R = \sqrt{(W/2)^2 + (H/2)^2}$. Technically, since OpenCL buffers are always one-dimensional, the memory is created as an

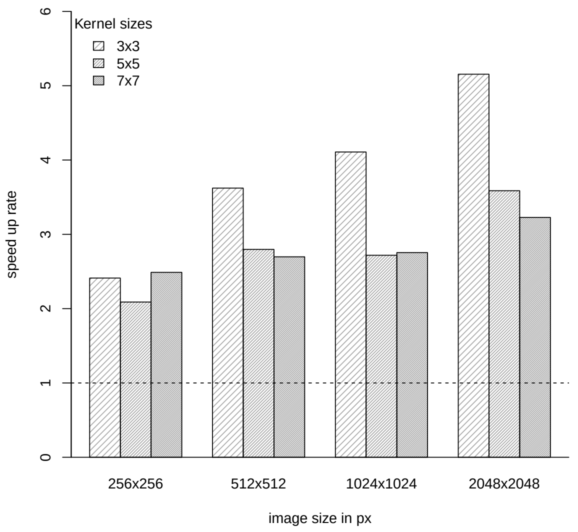


Figure 4.13.: Speedup rate for image convolution using OpenCL compared to CPU-based reference implementation.

array of $180 * 2R$ unsigned integers. Zero-filled data of this size is copied to the GPU to provide the correct buffer initialization.

Parallel Hough transform: Each pixel of the binary input image is processed in parallel on the GPU following the instructions in an OpenCL kernel program excerpted in listing 4.2. After the absolute position and the position in relation to the image origin (center point) of a pixel are determined (lines 9-12), a binary pixel value is read from the input image. If it is a feature pixel (line 14), the `r` value is calculated for each possible line direction described by `theta`. If `r` is inside the accumulator space and its absolute value is not too small (line 21)[14], a vote is cast by incrementing the value at the corresponding buffer position in the accumulator map (line 23). This operation introduces data dependency between parallel tasks (see section 2.4.2) and therefore synchronized global memory access is absolutely necessary to execute the incrementation process (which consists of fetching the buffer value, incrementing it and storing the result) *atomically*. This can be done using the `atomic_inc` function provided by an OpenCL extension.[15]

[14] It was found necessary to filter out too small $|r|$ values because otherwise stray lines were detected in the center of the image.

[15] Atomic functions are natively supported by the full profile of OpenCL [MGM11, p. 387]. For the embedded profile an extension (`cl_khr_global_int32_base_atomics`) is needed, which is present for the Nexus 10 OpenCL driver.

Fetching the result and finding maxima: The result buffer containing the vote map is read back to main memory. Here, a simple threshold is applied to find out each maximum i with parameters r_i, θ_i in the map. The parameters are translated back to coordinate space where they can for instance be drawn as lines on top of the original image. Finding the maxima could also be parallelized with OpenCL, but here the focus is set on the Hough transform and only the performance of this step should be measured.

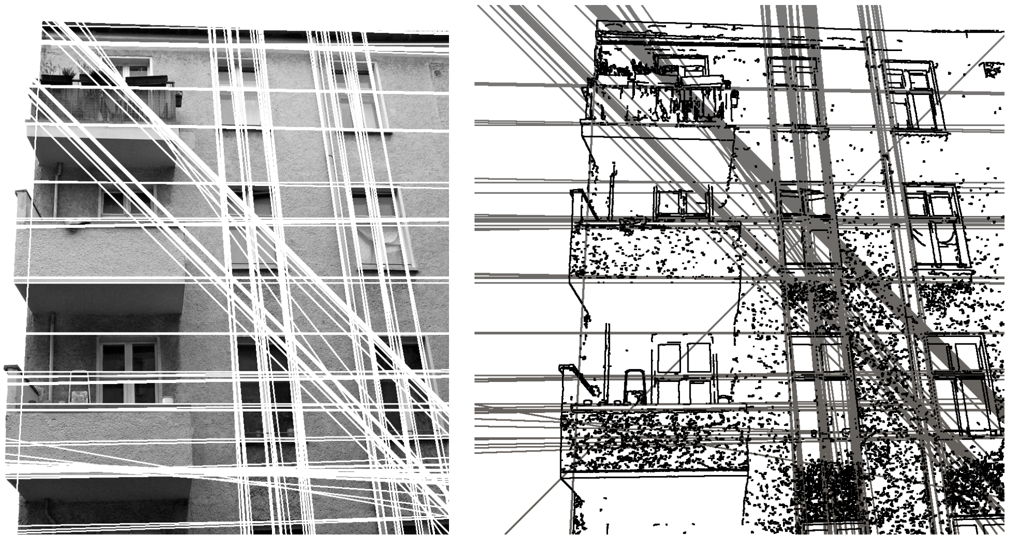
Since the work-item operations are fully independent of each other (besides the accumulator map incrementations), work distribution is handled automatically by OpenCL as is the case with the image convolution task.

```
// (THETA_STEP, THETA_MAX and DEG2RAD are defined at the beginning.)
kernel void cl_hough(read_only image2d_t srcImg,      // binary input image
                     global uint *accSpace,           // vote map (1D buffer)
                     sampler_t sampler,
                     int accSpaceW,    int accSpaceH,  // acc. space size
                     int imgW,         int imgH)       // input image size
{
    int accSpaceWHalf = accSpaceW / 2;
    // absolute pixel position
    int2 absPos = (int2)(get_global_id(0), get_global_id(1));
    // rel. pos. with origin at center
    float2 pos = (float2)(absPos.x-(float)imgW/2.0f, absPos.y-(float)imgH/2.0f);
    // get binary value at this pixel position, check if it is not "0"
    if (read_imagef(srcImg, sampler, absPos).x > 0.0f) {
        // Cast votes in Hough space for each possible line orientation
        for (int thetaStep = 0; thetaStep < THETA_MAX; thetaStep += THETA_STEP) {
            float theta = (float)thetaStep * DEG2RAD;   // convert to radians
            // calculate r (distance from origin)
            int r = (int)(pos.x * cos(theta) + pos.y * sin(theta));
            // cast vote if we are inside the vote map
            if (abs(r) > 2 && r >= -accSpaceWHalf && r < accSpaceWHalf) {
                size_t accPos = thetaStep * accSpaceW + r + accSpaceWHalf;
                atomic_inc(&accSpace[accPos]);  // use atomic incrementation
            }
        }
    }
}
```

Listing 4.2: OpenCL Hough transform kernel excerpt.

Results

The results regarding the detection of lines are similar to the CPU-based reference implementation, as can be seen for comparison in figures 4.14. Some stray lines are detected that run across the image because of the edge features produced by the Canny algorithm in the areas with coarse facade structure in the photograph.



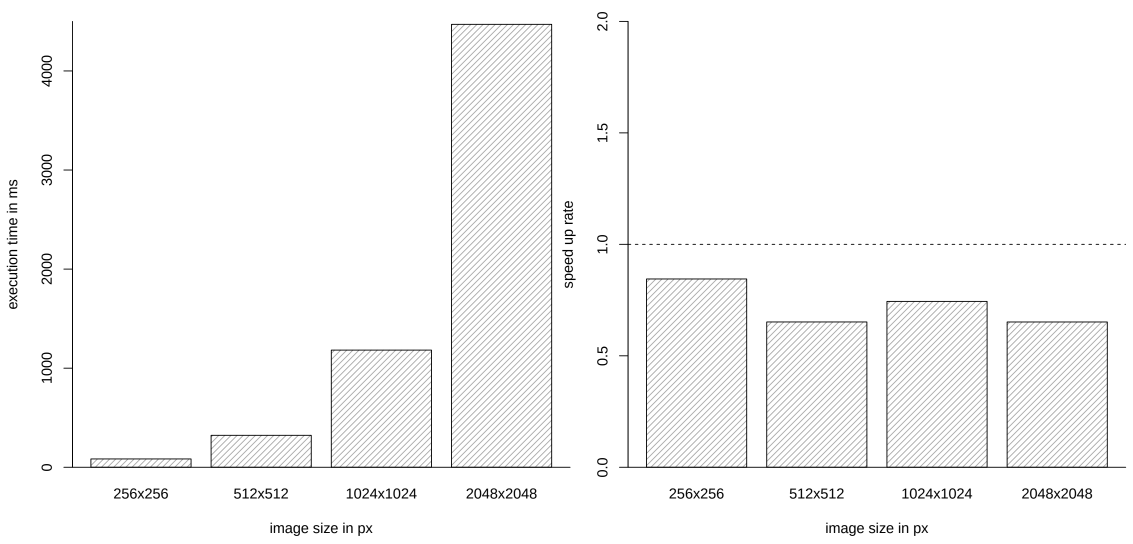
(a) CPU-based reference implementation.

(b) OpenCL-based implementation (projected upon inverted binary image).

Figure 4.14.: Detected lines in the 1024x1024 pixels test image.

The execution time for different image sizes is depicted in figure 4.15a. OpenCL performs slightly slower than the CPU-based serial variant. Adding the memory copy operations overhead, the resulting overall processing time is even worse. This is also reflected in the speedup rates in figure 4.15b. Although common-sense expectation is that a GPU-based parallel implementation performs better with bigger images, no speedup could be achieved at all. This could be caused by several factors. First, the OpenCL kernel uses a `for`-loop statement and two `if`-statements. Both branching and loops may have negative performance impact when running the OpenCL program on the GPU, as was observed with the image convolution kernel in the previous section. Unrolling the loop for all 180 possible `theta` values would produce extremely bloated source code and was not tested. Furthermore, atomic memory operations are demonstrably slower than the conventional equivalents, because the global memory accesses need to be synchronized between all work-items. Testing showed that without atomic operations the kernel executes about 10% faster – however, this produces as expected unusable results.

All in all, the Hough transform performance is rather disappointing, suggesting that more complex algorithms need further optimization to run well on the GPU with the help of OpenCL. Other variants of the Hough transform – like the one suggested by Fernandes and Oliveira [FO08] – should also be tested in the future. Their ideas were assessed in section 4.1.2 in terms of parallelization possibilities. In the scope of this thesis, however, this approach was set aside in favor of an OpenGL ES implementation



(a) Execution times.

(b) Speedup rate as compared to CPU implementation.

Figure 4.15.: Performance results for Hough transform with OpenCL.

of the ideas suggested by Dubska et al. [DHH11] (described in section 4.5.3). Since this implementation also has drawbacks, parts of it should be accelerated using OpenCL-OpenGL interoperability features [MGM11, p. 335].

4.3.4. Audio Synthesis Task

Implementation Approach

To produce sounds, OpenCL must be used in conjunction with Android's OpenSL ES API in the NDK. This low-level C API is very complex – but fortunately there is a small wrapper library written and documented by Lazzarini [Laz12]. Just as with Android's Java `AudioTrack` API, a buffer with a specified size must be filled with audio samples. An OpenCL kernel program is used to generate these samples. Therefore a one-dimensional OpenCL buffer memory object is created, whose values are calculated in parallel using the kernel program and then copied back directly to the audio buffer.

The OpenCL kernel (presented in listing 4.3) first determines at which position within the buffer to calculate an audio sample value. Since each work-item is assigned to exactly one value in the buffer (just like each work-item in the image convolution task is assigned to exactly one pixel location of an image), `get_global_id(0)` can be used to get the buffer position. The output value is dependent on this position since it determines the phase of the sine calculations in the waveform generation. This means that the kernel produces a buffer value like $buf_p = w(p * step)$, where p is the buffer position, *step* describes a factor that produces certain phase steps per position to generate a calculated pitch for the sound where w is a function used to generate the waveform, e.g. a sine function. This alone would introduce truncated phases, resulting in audible glitches due to non-continuous waveforms at the border between two subsequent audio buffers as depicted in figure 4.16. To fix this, an additional offset parameter o (or `bufOffset` in the source code) needs to be supplied to the kernel each time a new audio buffer is generated: $buf_p = w((p + o) * step)$. This offset shifts the phase to the correct position for subsequent audio buffers.

```
// phase steps for base freq. and 3 additional waves to produce a square wave
constant float phStepBase = (440.0f * M_PI_F) / 44100.0f;
constant float phStepAdd1 = 3.0f * phStepBase;
constant float phStepAdd2 = 5.0f * phStepBase;
constant float phStepAdd3 = 7.0f * phStepBase;

kernel void cl_synth(int bufOffset, global float *buf) {
    const int gid = get_global_id(0);
    const float p = (float)(bufOffset + gid - 1);     // phase position
    // calculate the sample
    buf[gid] = sin(p * phStepBase)
             + sin(p * phStepAdd1) / 3.0f
             + sin(p * phStepAdd2) / 5.0f
             + sin(p * phStepAdd3) / 7.0f);
}
```

Listing 4.3: OpenCL additive audio synthesis kernel.

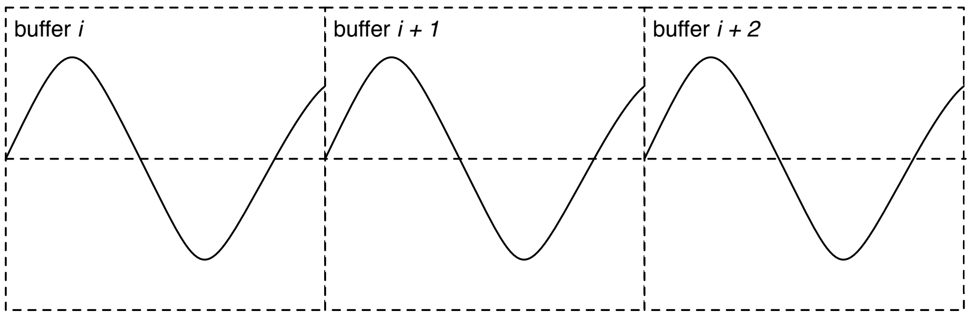


Figure 4.16.: Truncated phases problem that occurs with subsequent buffers without initial phase offset.

Results

The same test cases performed with the CPU-based reference implementation were used. Memory transfer times were not gauged separately. Figure 4.17 depicts the total time for executing the OpenCL kernel *and* copying back the OpenCL buffer memory to the audio buffer. When compared to the CPU implementation, OpenCL scales better with increased calculation complexity and buffer sizes, as shown in the speedup chart in figure 4.18. A noticeable speedup factor of about 2x is achieved with the additive synthesis variant using a buffer size of 8096 samples. Doubling the buffer size demonstrates additional increased speedup. However, doing so would introduce an audible lag for real-time audio synthesis.

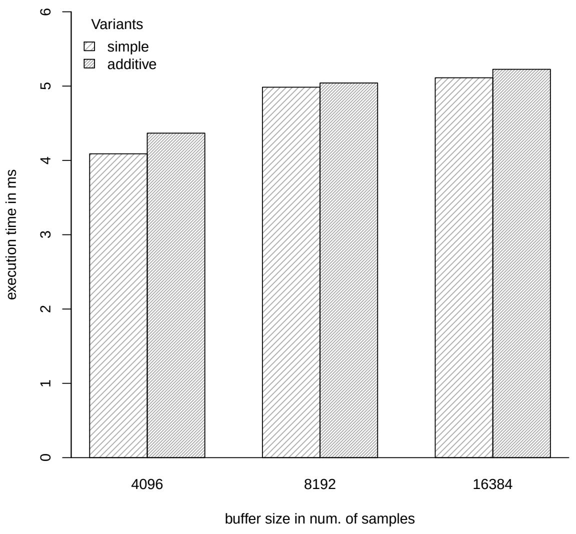


Figure 4.17.: Audio synthesis with OpenCL (waveform calculations *and* memory transfer).

It is apparent that the implementation using OpenCL has few problems handling additional sine calculations. Therefore the gap between the simple and additive synthesis variants is almost not measurable. Heavy optimization of the GPU's hardware for trigonometric functions might be why. These functions are essential for 3D graphic scene calculations. The bottleneck seems to be the data bus when copying the generated buffer values back to the audio buffer on the host side.

These results look promising for moving certain audio synthesis tasks over to the GPU. However, it should be kept in mind that the tested algorithms were quite simple and that more complex algorithms might require loops or conditional statements. These can introduce introduce substantive performance losses, as was experienced with `for`-loops in the image convolution and Hough transform task (see sections 4.3.2 and 4.3.3).

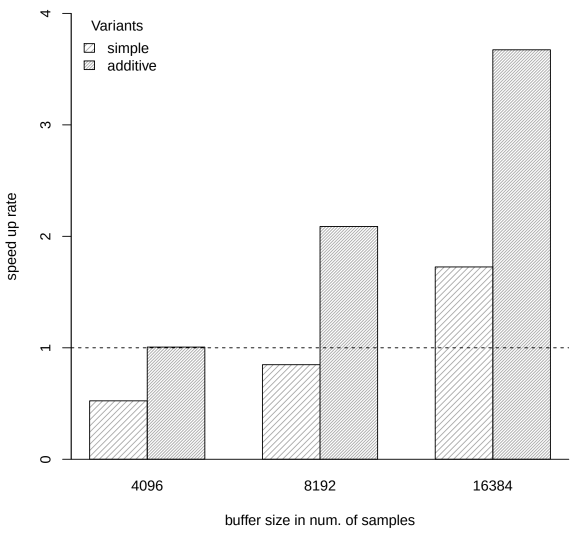


Figure 4.18.: Speed up of audio synthesis using OpenCL compared to CPU-based reference implementation.

4.3.5. Overall Experiences

The experiments have shown that OpenCL generally offers better scalability for increased problem sizes and also – in some cases – for increased algorithmic complexity. To achieve optimal performance, conditional statements and loops should be avoided since GPUs are optimized for running similar operations on big data sets. Control structures introduce branches that the GPU cannot handle optimally. Since dynamic algorithms require loops and conditions (such as image convolution with customizable kernels), it might be beneficial to generate OpenCL code with unrolled loops and conditions at run-time.

Developing Android applications that use OpenCL is challenging. The initial setup as described in section 4.3.1 is not as easy as importing a Java package. It requires using the NDK and Android's C APIs, which again introduce an additional layer that has to be addressed via JNI calls. Debugging C/C++ code is possible but requires several additional steps to set up (see [Hej11]). Debugging OpenCL code is not even possible at the moment, leaving only the less desirable path of *trial-and-error*. Since three different programming layers and languages (Java in Android SDK, C/C++ in Android NDK and OpenCL C) are involved, additional overhead and complexity is introduced. To reduce the number of layers, using an *Android Native Activity*[16] and writing the application solely in C/C++ might be considered.

[16]See documentation at `http://developer.android.com/reference/android/app/NativeActivity.html`.

4.4. RenderScript

4.4.1. Special Characteristics

As described in section 3.2.2, RenderScript (RS) has a programming model that is similar to OpenCL, but its API abstracts any low-level hardware features. This approach takes both the freedom to optimize for a specific problem or hardware and the responsibility for ensuring portability away from the developer. Since work distribution cannot be controlled or traced in any way, it is impossible to determine if the task was executed on the CPU or GPU.

During development of the prototypes, several issues occurred that made testing almost impossible in certain circumstances. Though setting up an Android application to use RS is much easier than OpenCL since the RS API is part of the Android Java SDK, RS does not offer OpenCL's stability, reliability and good documentation. Applications would reproducibly crash with a *segmentation fault* after a few test runs which made batch processing impossible in the testing environment. At time of writing, the documentation for RS that comes with Android 4.4 is very meager and examples that are shipped with the SDK are outdated. Because of this, only limited tests could be conducted. Other cases could not be tested at all due to immediate application crashes.

4.4.2. Image Convolution Task

Implementation Approach

The way how image processing can be implemented with RS is very similar to OpenCL. The syntax of the RenderScript C99 dialect is a bit different and the Java API on the host side is more restricted in terms of work distribution. In general, an Android `Bitmap` image is put into an RS `Allocation` object which is passed to the script using a `set` method. The script is then executed using a `forEach` method. Finally, the data is copied back to a `Bitmap` via `Allocation.copyTo()`. The RenderScript program (see listing 4.4) defines that it uses an *allocation* as input data (representing the input image) and expects an x and y coordinate as parameters. This is basically the same as OpenCL's global ID concept and describes the pixel location operated on by the script instance. The script reads the neighboring pixels using `rsGetElementAt()` calls and multiplies each pixel value with its respective kernel matrix value. At the conclusion, all the results are summed up and returned. The implementation is reminiscent of the method described in the experiment using OpenCL in section 4.3.2.

```
// (... 3x3  image convolution matrix k is defined as array at the beginning)
rs_allocation in;
uchar4 __attribute__((kernel)) make_gauss(uint32_t x, uint32_t y) {
  float4 f4out = { 0, 0, 0, 0};
  for (int kIdx = 0; kIdx < 9; kIdx++) {  // go through the convolution matrix
    // calc. neighborhood pixel offset
    int kX = kIdx % 3 - 1;
    int kY = kIdx / 3 - 1;
    // read pixel, multiply by convol. matrix value, add to result pixel color
    f4out += convert_float4(rsGetElementAt_uchar4(in, x+kX, y+kY)) * k[kIdx];
  }
  return convert_uchar4(f4out);
}
```

Listing 4.4: Excerpt of RenderScript program for image convolution.

Results

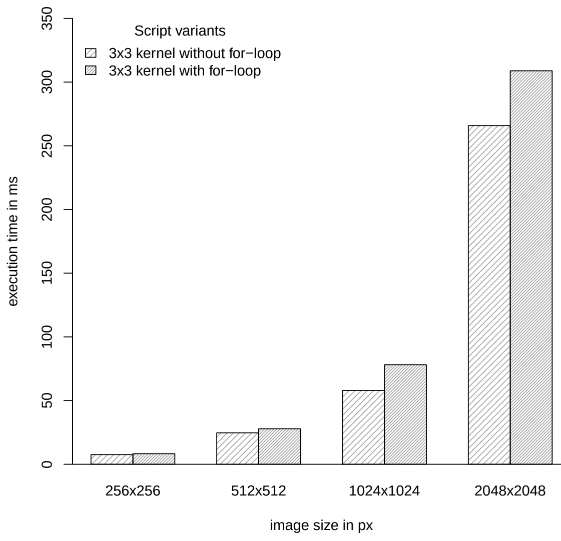
Unfortunately it was not possible to get accurate memory transfer time measurements by gauging the execution time of the respective memory transfer function calls. Such attempts yielded very different results with often unrealistically short times (below 1 ms). It is very likely that this behavior is caused by command caching, which apparently cannot be disabled.[17] Therefore the total execution time including copying the image to the GPU and vice versa was gauged instead. Due to the described issues (especially recurring segmentation faults that provoked application crashes) only an implementation using a 3x3 convolution kernel was tested. Two variants were produced, one using a `for`-loop and one with an unrolled loop.

Figure 4.19a shows the total time measurements for the two variants. Unlike OpenCL, using loop statements does not result in significant negative impact on the overall performance. On the other hand, OpenCL performs this task up to four times faster than RS. This is reflected in the speedup rates as depicted in figure 4.19b. Although RS achieves stable speedup rates over all image sizes, the top result is only half as good as OpenCL's. Interestingly, the speedup factors compared to the CPU-based implementation do not rise with image size as is usually the case in such experiments with GPU acceleration. One possible explanation might be that the calculations were not performed on the GPU at all. As noted, the work distribution is not transparent when using RS. Therefore, the possibility exists that the RS system distributed the work using the dual-core processor and (possibly) SIMD instructions for this task.

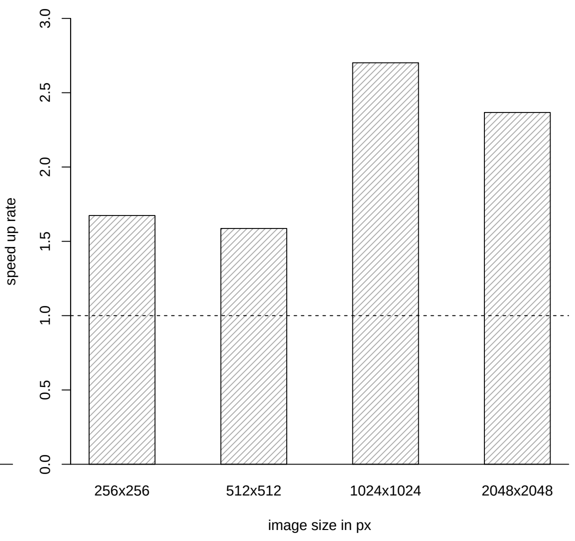
4.4.3. Overall Experiences

At time of writing, RenderScript delivers only a premature impression, suffering as it does of bad documentation and unstable run-time behavior. Since only limited tests could be conducted, no final conclusion can be drawn about its potential.

[17]OpenGL and OpenCL also employs command caching as described in section 4.2.2 but there it is possible to force immediate command execution for exact time measurements.



(a) Time measures with and without `for`-loop.



(b) Speedup rates using the "unrolled" variant.

Figure 4.19.: Execution times for image convolution using RenderScript.

4.5. OpenGL ES

4.5.1. Special Characteristics

OpenGL ES does not support GPGPU out of the box, as described in section 3.2.3. Therefore, the concepts suggested by Göddeke [Göd06] (also presented in the mentioned section), which were applied in related research such as that of Ensor and Hall [EH11], provide the basis for the GPU computing experiments with OpenGL. For the tests API, version 2.0 has been chosen because it is available by default on the test device. Also version 2.0, represents the most widely supported OpenGL ES API for all mobile devices.

There are two ways to access the API on Android. The Java API contained in the packages `android.opengl` allows all OpenGL ES 2.0 functions to be used within the Android SDK. The other method of access requires using the C API via Android NDK, defined in the header file `GLES2/gl2.h` and implemented in `libGLESv2`. At first, OpenGL's Java bindings were used in the experiments. Due to enormous performance problems with `glReadPixels` (see below), other tests were conducted using the C API – in conjunction with several extensions – to achieve better memory transfer speed.

4.5.2. Image Convolution Task

Initial Implementation Approach

The Java API was used in this initial approach to implement image convolution with OpenGL shaders. Following the concept described in section 3.2.3, input images are copied to the GPU to act as a texture by using `glTexImage2D`. A simple geometry for a quad (consisting of two triangles) is created as a vertex buffer to actually draw the result later as a texture on its surface. Two shader programs exist, each consisting of a vertex and fragment shader. One program defines the rendering pipeline for the image processing task. The other is optional as it simply displays a quad with the resulting image. This allows flexible usage, since the former shader program can be used repeatedly in conjunction with framebuffer objects to enable multi-level filtering (e.g. separated Gauss kernel convolutions). The desired resolution of the output image is set using `glViewport`, which scales the image efficiently on the GPU. At the end, the final framebuffer output is copied back to main memory using `glReadPixels`. If no further processing on the CPU side is necessary this step can be omitted, but usually some result needs to be copied back from the GPU.

The main work is done in the fragment shader of the image processing shader program. The routines in this shader are applied to each single pixel after the rasterization step, as described in section 3.2.3. Here, the image convolution is applied by fetching the neighboring pixel values and the center pixel value via texture reads (using the `texture2D` function) to multiply them with their respective kernel values. The results are summed and written to `gl_FragColor` as shader output. To achieve top performance, no loop statements were used in the shaders. For big kernel matrices this would mean a high degree of manual effort is necessary to write the shader code. Therefore, the fragment shader sources are generated dynamically at run-time depending on the provided kernel. Such a generated fragment shader is excerpted in listing 4.5.

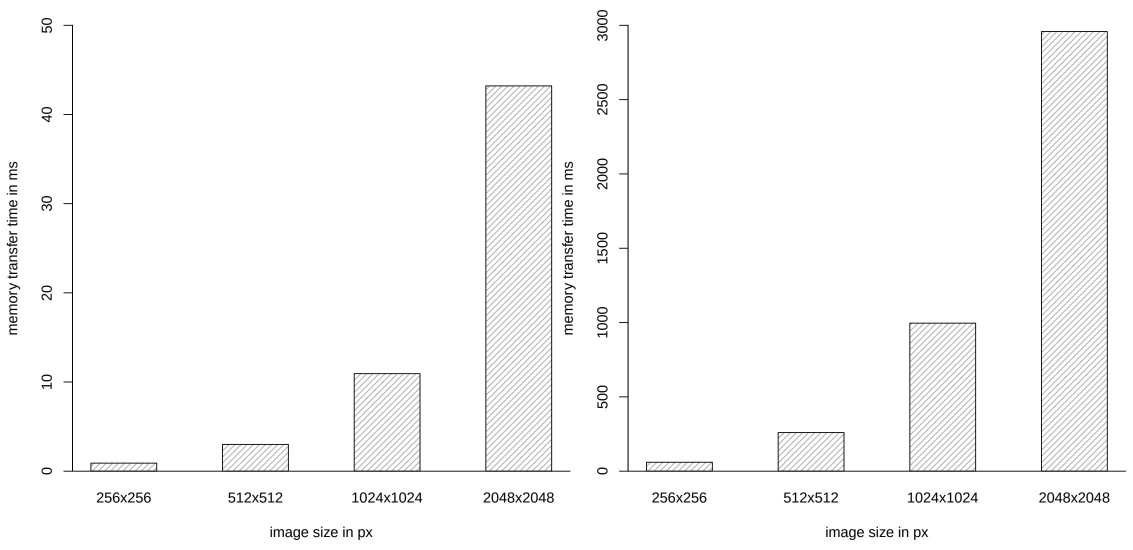
```
precision mediump float;
uniform sampler2D sTex;
uniform vec2 uPxD;  // distance between single pixels in tex. coordinates
varying vec2 vTex;  // current texture coordinate
void main() {
  // read the neighborhood pixels
  vec4 px00 = texture2D(sTex, vTex + vec2(-1.0*uPxD.x,-1.0*uPxD.y)) * 0.0625;
  vec4 px10 = texture2D(sTex, vTex + vec2(0.0*uPxD.x,-1.0*uPxD.y)) * 0.125;
  vec4 px20 = texture2D(sTex, vTex + vec2(1.0*uPxD.x,-1.0*uPxD.y)) * 0.0625;
  // ... continues with more neighborhood pixels ...
  vec4 px22 = texture2D(sTex, vTex + vec2(1.0*uPxD.x,1.0*uPxD.y)) * 0.0625;
  // sum up the result and save it as final fragment color
  gl_FragColor = px00+px10+px20+px01+px11+px21+px02+px12+px22;
}
```

Listing 4.5: Excerpt of automatically generated fragment shader code for 3x3 image convolution (comments and indentations were added manually).

The fragment shader defines a two-dimensional vector `uPxD` as a *uniform* parameter, which is passed to the shader during rendering. Since texture reads in shaders happen in texture coordinate space (s, t with values in the normalized range $[0.0, 1.0]$), a delta value needs to be specified as $\Delta_x = \frac{1}{W}$ (with W as image width) and $\Delta_y = \frac{1}{H}$ (with H as image height), so that it describes the distance between two orthogonally neighboring texture pixels in the texture coordinate space. By this, the proper neighboring pixel values can be read from texture memory. Texture borders do not have to be treated specially, since edge clamping (see [MGM11, p. 194]) is enabled.

Results under Usage of `glReadPixels`

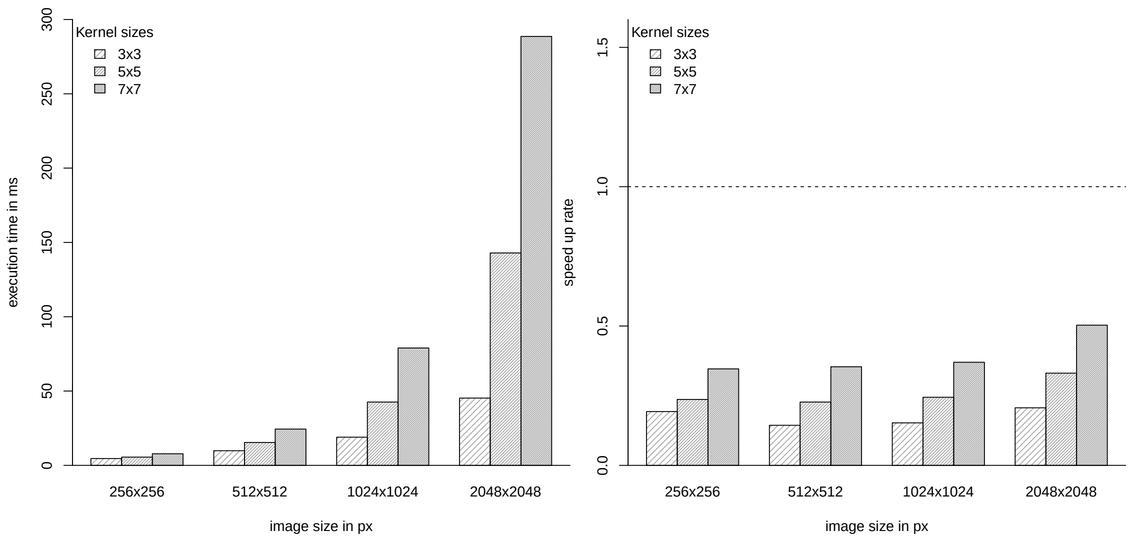
Once again, memory transfer times were measured by copying RGBA images of different sizes to the GPU memory and vice versa. The results are depicted in figure 4.20 and reveal a very ambivalent case: Copying data to the GPU happens very fast, even faster than with OpenCL, and results in speeds of about 270 to 370 MB/sec. Copying back the data from the GPU to main memory, however, delivers a very poor transfer speed between 4 and 5 MB/sec. At this speed, the data transfer for a 2048x2048 RGBA image (16 MB) takes almost 3 seconds to complete. This seems to be a known issue related to the `glReadPixels` function and is documented by Bim [Bim13] and Fung [Fun13]. Their respective articles also list several solutions, which are discussed in the next section, where an optimized implementation is presented.



(a) Copying images *to* the GPU. (b) Copying images back *from* the GPU.

Figure 4.20.: Memory transfer times using OpenGL.

The pure rendering performance, i.e. the time it takes for the image convolution task to complete on the GPU, should be examined. The overall results show that the performance is comparable to OpenCL and are displayed in figure 4.21a. Calculating the speedup between OpenGL and OpenCL for the experiment using a 3x3 kernel revealed that OpenGL delivers only slightly better results for bigger images (1.3x better than OpenCL with the 2048x2048 image), whereas it is up to 2x slower for smaller images. This might be caused by additional overhead as described in section 3.2.3.



(a) Execution time. (b) Speedup as compared to reference.

Figure 4.21.: Results for image convolution with different kernel sizes and images using OpenGL.

Clearly, the poor data transfer performance results in disappointing speedup rates. The values displayed in figure 4.21b are not surprising. The CPU-based reference implementation is between 2x and 3x faster than the OpenGL-based variant. However, OpenGL's full GPGPU potential could be unleashed using alternative solutions for reading the framebuffer.

Optimized Implementation Approach

The low data transfer rate for copying data from the GPU using `glReadPixels` makes OpenGL unusable for GPGPU in real-time scenarios. Fortunately, solutions exist as mentioned in the previous section. Bim [Bim13] recommends using `glGetTexImage` or *pixelbuffer objects (PBOs)*. Unfortunately the former is not available in OpenGL ES at all and the latter is only supported from version 3.0 onward. Fung [Fun13] additionally suggests using an EGL[18] extension called `EGL_KHR_image_base` which "achieves the same performance as PBOs, but only require[s] OpenGL ES 1.1 or 2.0." An article by Montgomery [Mon12] describes the idea in more detail and provides suggestions for implementation. Image data transfers between the host system APIs and OpenGL ES can be accelerated in both directions using an EGL image, as depicted in figure 4.22. This method enables to use an image – such as a video frame taken with the device camera – as an OpenGL texture by means of an accelerated *bit blitting* operation or `memcpy` which is "still typically faster than using `glTexImage2D` to load textures" [Mon12]. In the opposite direction, OpenGL ES can be used to render a frame into an FBO, which can then be used as EGL image to perform the same operations, transferring the data back to the host system. Image compression, which is directly supported at the hardware-level by most GPUs, could be considered another solution for decreasing the data transfer delay. Montgomery [Mon12] states that "[t]he texture compression algorithms implemented in 3D accelerators are asymmetrical, meaning that it is much more compute intensive to compress an image than to decompress the same image" and therefore advises against using this idea.

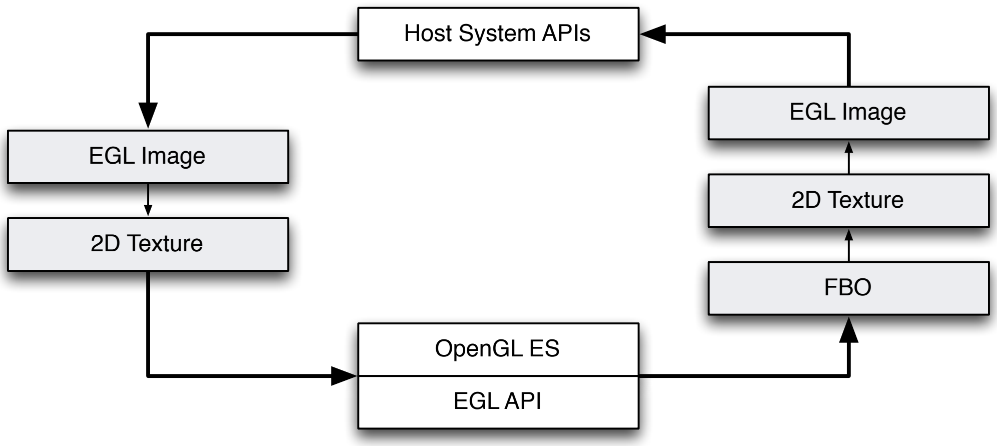


Figure 4.22.: Image data exchange between host system and OpenGL using EGL images [Mon12].

This approach is implemented on Android systems using a native Android OS API called *gralloc* or *GraphicsBuffer*. It allows creating a frame buffer in CPU/GPU shared memory. The system is documented in depth by the Android Open Source Project [AOS] but is *only* part of the operating system and therefore available neither in the SDK nor NDK. Using the API is technically possible in the NDK but requires some manual effort

[18] *EGL* is an interface that sits between rendering APIs of embedded systems (such as OpenGL ES) and the native window system [Khr14].

as described by Willcox [Wil11]. To do so involves linking two Android system libraries (`libEGL.so` and `libui.so`) dynamically at run-time. Pointers to some of the functions contained in the libraries must be acquired by using `dlsym`. Because the process of copying data *from* the GPU was found to be slow, the workaround was implemented only for that portion of the whole data transfer procedure, as shown in listing 4.6. For that the buffer can be locked to ensure consistent data when using `memcpy` to copy the data into a buffer that is then processed by the host system. Looking at source code provided by the Mozilla Community [Moz13] helped to implement this workaround. It is unclear why these functions are not provided natively in the SDK or NDK and using them requires such tricks.[19] Furthermore, this breaks portability since it only works on Android devices. For iOS devices, other solutions exist to work around the `glReadPixels` bottleneck [Lar12a].

```
// bind fbo
glBindFramebuffer(GL_FRAMEBUFFER, fboID);
// bind attached texture and set EGL image target
glBindTexture(GL_TEXTURE_2D, attachedTexId);
glEGLImageTargetTexture2DOES(GL_TEXTURE_2D, eglImg);
// lock the graphics buffer at "graphicsPtr" to safely read its contents
unsigned char *graphicsPtr;
GraphicBufLock(graphicBufferHndl, GRALLOC_USAGE_SW_READ_OFTEN, &graphicsPtr);
// copy the RGBA pixels from the graphics buffer to "buf"
memcpy(buf, graphicsPtr, texW * texH * 4);
// unlock graphics buffer again
GraphicBufUnlock(graphicBufferHndl);
// unbind FBO
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

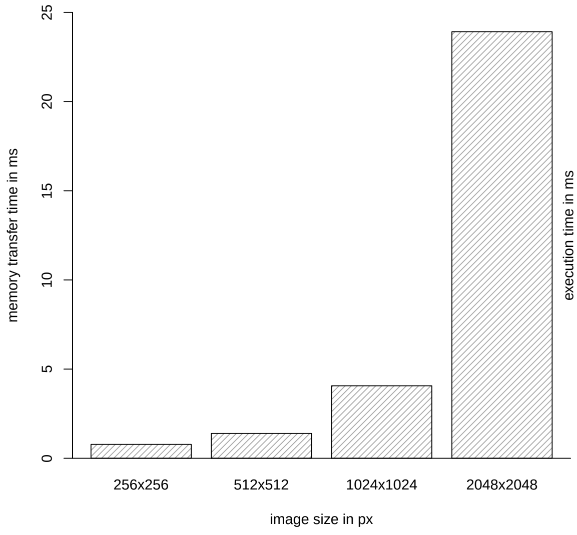
Listing 4.6: Excerpt for copying back the result image from an FBO using the EGL image extension and Android GraphicsBuffer.

An optimization of the kernel convolution was carried out to further improve the overall performance. Since Gauss kernels are used for testing, they are separated into two orthogonal convolution operations using one-dimensional kernels as described in section 4.1.1. The operations are performed in two rendering passes. The first pass operates on a 90° rotated image to benefit from linear memory access patterns as described by Rister et al. [Ris+13].

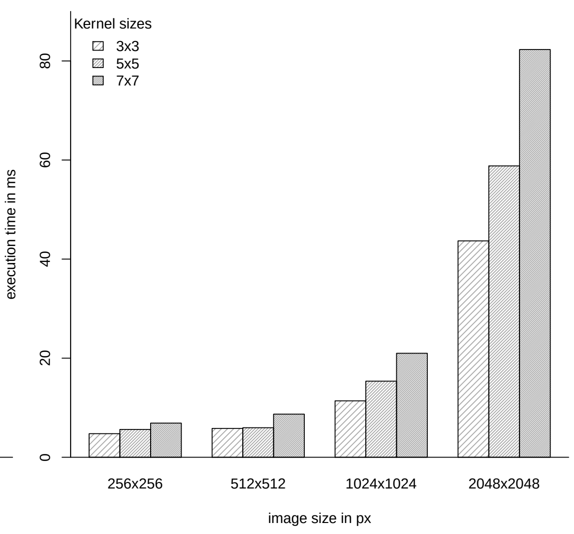
Results for Improved Implementation

As shown in figure 4.23a, the memory transfer times for copying image data from the GPU to the host system decreased dramatically. Where `glReadPixels` took about 3 seconds to copy the largest image, the revised operation using an EGL image now happens within 25 ms, which is even approximately two times faster than the image

[19]This also involves for example acquiring pointers to the API functions by their compiled symbol names such as `_ZN7android13GraphicBufferC1Ejjij`.



(a) Memory transfer times for copying data from the GPU.



(b) Execution times using separated one-dimensional Gauss kernels in two-pass operation.

Figure 4.23.: Results for image convolution using OpenGL with the improved implementation.

upload using `glTexImage2D`. The time was measured for acquiring the lock on the shared graphics memory buffer and copying the data using `memcpy`. Another improvement in performance is achieved by calculating the Gaussian image convolution in two passes with a one-dimensional kernel, as shown in figure 4.23b. That a positive effect can only be achieved with bigger images and kernel sizes is apparent. The cause of this is the second rendering pass that introduces additional overhead.

The improved overall performance is also reflected in the achieved speedup rates as shown in figure 4.24. With speedups between 2x and 8x, OpenGL partly outperforms the (not optimized) OpenCL implementation. However, it does not deliver a clear linear speedup gain on increasing image and kernel sizes, which might be connected to the two-pass rendering overhead. The results are nevertheless satisfying and could be further improved, by replacing the usage of `glTexImage2D` for copying the image to the GPU with the faster EGL image approach.

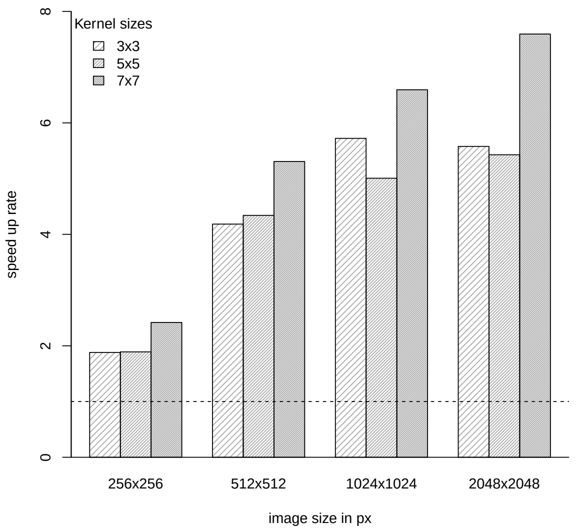


Figure 4.24.: Speedup rates of improved OpenGL image convolution implementation as compared to reference implementation.

4.5.3. Hough Transform

Implementation Approach

The implementation for line detection using the Hough transform is based on an approach suggested by Dubska et al. [DHH11] and was described in section 4.1.2. The authors of the article use OpenGL 3.3 *geometry shaders*, with which it is possible not just to modify single vertices of geometric primitives (as with vertex shaders) but to generate new vertices. These shaders are used in conjunction with *instanced rendering* [Kom13] to calculate the position of each pixel to be read from the source texture. They harness the possibility of generating vertices to create a three-point line strip in TS space (the accumulation space) for each source pixel that is identified as an edge pixel. In the rasterization process, additive blending is used so that individual line strips are accumulated and peaks are formed where many intersections occur. The results are rendered into a floating-point texture (`GL_GLOAT`) so that overflows that could quickly happen by using byte formats (e.g. `GL_UNSIGNED_BYTE`) are avoided. In the last step the local maxima in TS space are detected by another geometry shader which checks the neighborhood of each pixel in the accumulator space.

Implementing this approach on mobile devices presents several problems: First and foremost, there are no geometry shaders (or comparable features) available in OpenGL ES 2.0 (and also 3.0) since only vertex and fragment shaders can be used to program the rendering pipeline. Additionally, instanced rendering is only supported since OpenGL ES 3.0 (which is not widely available yet, as shown in section 3.2.3). Another problem is that framebuffer textures must be color-renderable, forbidding the use of `GL_FLOAT` textures as FBO targets [ML10, p. 117; Lip13, p. 203]. The iOS-based project *GPUImage* [Lar12b] addresses these issues by calculating the TS space coordinates on the CPU-side and then submitting the calculated points to the OpenGL renderer as vertex buffer, from which the line strips are drawn using a simple vertex shader. The rasterization takes place with alpha blending enabled but the rendering target is of format `GL_UNSIGNED_-BYTE`, which means that only 256 discretization steps for the voting are available. This implies that after 256 intersections happened at the same discretized spot, an overflow occurs. In this case, that means that the pixel value stays at the 8-bit maximum value of 255. The peaks are therefore trimmed, which can result in spurious lines. Another problem is that only the accumulation process is performed on the GPU and that the CPU implementation of the TS space coordinate calculation requires access to the binary image. If an application performs the edge detection process on the GPU (which can be efficiently implemented on mobile devices as shown by [EH11]), the result image needs to be copied back to main memory before it can be processed to generate the TS space coordinates. This introduces memory transfer overhead. Hence a continuous GPGPU processing pipeline is preferable.

To circumvent this drawback, a way to calculate the TS space coordinates on the GPU using an OpenGL ES 2.0 vertex shader was devised and is shown in listing A.1 in the appendix. By the help of *vertex shader texture lookups*[20] (line 22), it is possible to identify the edge pixels of the binary input image directly in the vertex shader. Then, the respective line point coordinates for the TS space can be calculated in the same shader, so that for each edge pixel four points are generated. The four points represent the beginning and ending points for the line in T and S space respectively. The rest of the algorithm works as before. The lines will be drawn with alpha blending and hence are being accumulated in the framebuffer, which acts as the vote map. Finally, a simple thresholding mechanism implemented in a fragment shader determines the peak votes. After copying back the result image to the host side, the peaks are identified and their positions (in TS space) are converted to slope-intercept line parameters m and b. The overall process is depicted in figure 4.25.

The problem with this approach is that the vertex shader needs to read the value of each pixel in the input image. What is a common task for a fragment shader is on the other hand harder to achieve in a vertex shader. It requires submission of an attribute array of texture coordinates describing the position of each pixel in the input texture (`aTexCoord` in the provided source code). The next problem is that vertex shader output is restricted to one vertex (defined as `gl_Position`) at a time, which means that the shader needs to be executed four times per input image pixel (once for each line point[21]). Texture reads

[20] Available with the GLSL function `texture2DLod` as documented in [SK09, p. 71].

[21] Note that the line strip in TS space always consists of three points (on $-y$, x and y axis), but four points are needed to construct the lines in OpenGL using `GL_LINES`. Using `GL_LINE_STRIP` is no option, because then *all* generated points would form *one* poly-line.

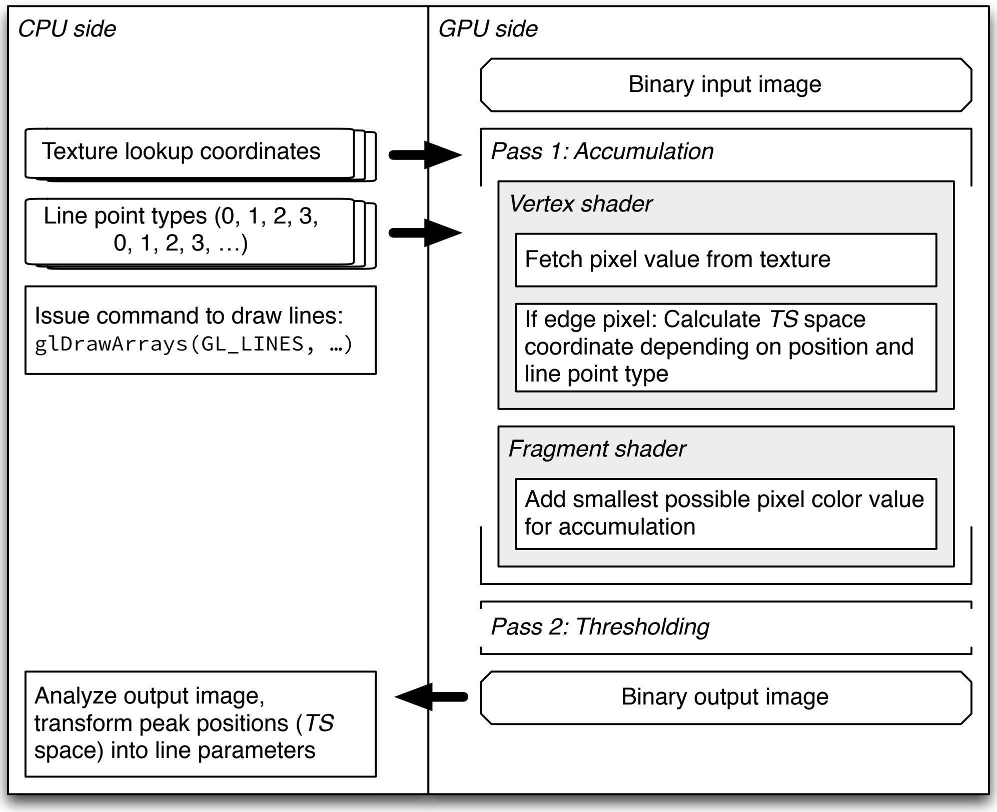
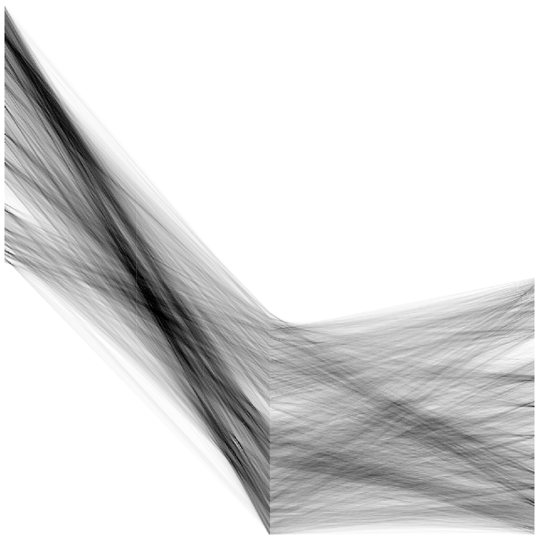

CPU side
GPU side
Binary input image
Texture lookup coordinates
Pass 1: Accumulation
Line point types (0, 1, 2, 3, 0, 1, 2, 3, ...)
Vertex shader
Fetch pixel value from texture
Issue command to draw lines: glDrawArrays(GL_LINES, …)
If edge pixel: Calculate TS space coordinate depending on position and line point type
Fragment shader
Add smallest possible pixel color value for accumulation
Pass 2: Thresholding
Analyze output image, transform peak positions (TS space) into line parameters
Binary output image

Figure 4.25.: Diagram of processing steps for the OpenGL-based Hough/PC lines transform. The single line point type values determine which point shall be calculated (begin or end, T or S space).

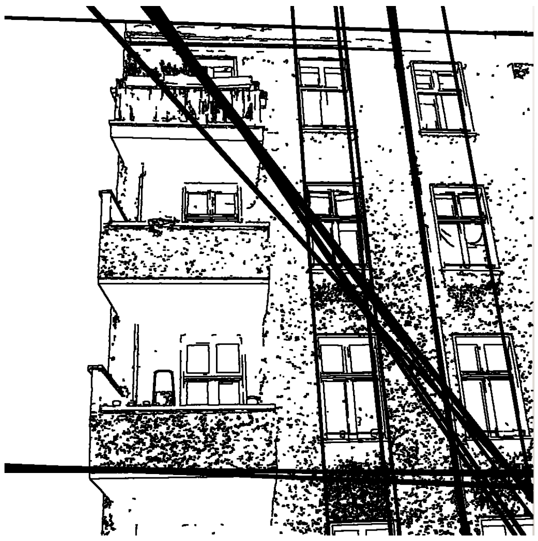
are thus quadrupled, since there is no way to directly implement caching in a vertex shader. In addition, to emulate instanced rendering in OpenGL ES 2.0 (which is needed to know what kind of line point must be calculated in the vertex shader), an additional attribute array with a continuous sequence of line point type identifiers (0, 1, 2, 3, 0, 1, 2, 3, ...) must be generated and submitted to the shader (`aTSCoordType` in the depicted shader source code).

Results

Evaluation shows that the algorithm works partially as expected and is capable of detecting straight lines in images as shown in figures 4.26. However, some incorrectly detected lines are visible crossing the image diagonally. This is most probably caused by the low number of discretization steps in the vote map, combined with the simple



(a) TS space result.

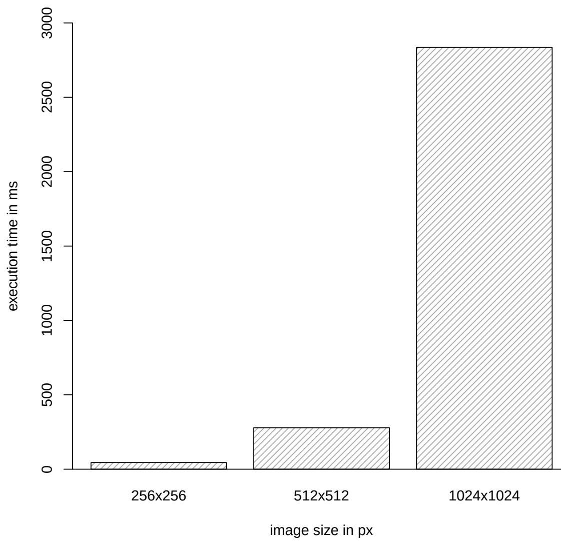


(b) Output image with detected lines, drawn upon the edge detection image.

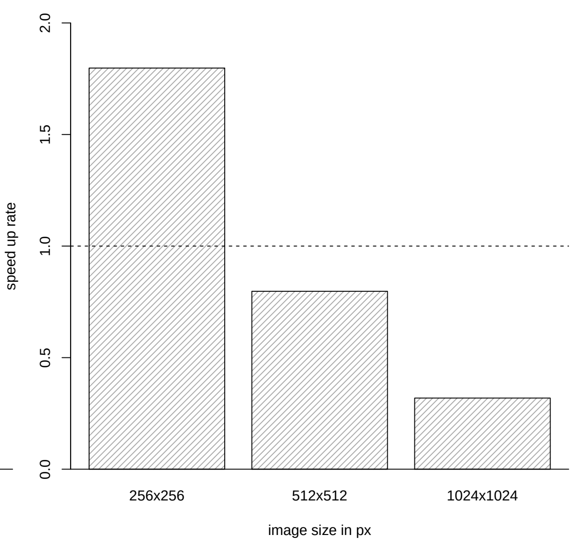
Figure 4.26.: Result images for PC lines based OpenGL Hough implementation (both images are inverted).

thresholding method. Both could be improved in the future by using floating-point textures as FBO rendering targets via OpenGL ES 2.0 extensions [Lip11] and by employing a more sophisticated thresholding algorithm. For the purposes of speed measurement, these optimizations are not necessary.

The achieved performance of the suggested implementation approach is rather disappointing. Interestingly, the initial program was not capable of handling the two bigger test images. Although no errors were thrown, the result images remained black, which is most probably because of the extensive amount of texture coordinates that are passed to the vertex shader. A “texture sampling factor” was therefore implemented, allowing the number of texture reads in the vertex shader to be reduced. When this factor is set at 2, every second pixel is omitted. Still, the tests with the biggest image size (2048x2048 pixels) fail with this setting, but at least the other three image sizes are accepted. Figure 4.27a shows the execution times of the PC lines calculation and accumulation rendering pass. As can be seen, the execution time rises exponentially with increased image size. Although a small speedup can be achieved using the smallest image, as is shown in figure 4.27b, the overall performance is poor. This is probably because the vertex shader requires an array of texture coordinates for every single pixel it needs to process. The array grows enormously with the image size. Additionally, this process has to be executed four times per pixel, further increasing the amount of texture read operations.



(a) Execution time.



(b) Speedup rates as compared to the reference implementation.

Figure 4.27.: Performance results for the PC lines based OpenGL Hough implementation. Note that the experiments were taken using a "texture sampling factor" of 2.

The results reveal that it is hard perform tasks like this Hough transform variant purely with OpenGL ES 2.0, mainly because of the lack of geometry shaders. Calculating the TS coordinates in a vertex shader is inefficient, but remains the only option when this algorithm is implemented using OpenGL ES 2.0. The TS space coordinate calculation step could be executed on the CPU side as done in the aforementioned GPUImage project. But, since copying framebuffer data back and forth from the GPU memory should be avoided, the only other possible solution would be to implement the calculation of TS space coordinates with OpenCL. By using OpenCL-OpenGL interoperability functions as described in [MGM11, p. 335] this approach could be employed but would sacrifice portability.

4.5.4. Audio Synthesis Task

Implementation Approach

The audio synthesis experiment was implemented by generating a "sound framebuffer" in the fragment shader and copying the contents of it to the audio buffer to achieve playback – an idea suggested by Borgeat [Bor12].[22]

[22]A live example using WebGL can be seen at `http://www.cappel-nord.de/glsl-audio-sandbox/`.

```
1  uniform vec2 uRes;  // buffer resolution
2  const float phStep = 0.06268937721; // phase step (440Hz sine, 44.1kHz sampling)
3  void main() {
4    // calculate sample position
5    float smplPos = gl_FragCoord.y * uRes.x + gl_FragCoord.x;
6    // calculate sample value
7    float smpl = (sin(smplPos * phStep) + 1.0) / 2.0;
8
9    // use two channels and divide into two bytes, each
10   vec2 stereo = vec2(smpl, smpl);
11   stereo = stereo * 255.0;
12   vec2 upper = floor(stereo);
13   vec2 lower = stereo - upper;
14   upper /= 255.0;
15
16   // save the result
17   gl_FragColor = vec4(lower[0], upper[0], lower[1], upper[1]);
18 }
```

Listing 4.7: Fragment shader for simple sine wave audio synthesis.

To begin, the size of the framebuffer is specified via `glViewport`. It works similarly to the memory buffer object described in section 4.3.4 – audio samples are written to it depending on the position inside the buffer. Since the framebuffer plays the role of the audio buffer, its width and height need to be set to values that result in the proper audio buffer size. For example, setting the viewport to 128x64 pixels results in an audio buffer of $128 * 64 = 8192$ samples. The fragment shader shown in listing 4.7 works like the OpenCL kernel for audio synthesis, excepting that it does not operate on a one-dimensional, but a two-dimensional buffer. It calculates the current position inside the buffer using the `gl_FragCoord` variable in accordance with the provided sample buffer resolution `uRes` (line 5). The phase for the sine wave oscillator is derived from this position, and then the output sample for this buffer element (i.e. this framebuffer pixel) can be calculated. Since the shader produces the pixel color as a four-component RGBA vector, R/G and B/A together can form two 16-bit stereo channels with two 8-bit (upper and lower byte) fields each [Bor12].

Although this approach technically works, the results were not satisfying since audible artifacts strongly distorted the sound output. The main source of this problem is that the point in time when the audio buffer should be generated – corresponding to the rendering of the frame buffer – is not directly controllable. Although it is possible to render the framebuffer upon request[23], it is not clear when this request will be fulfilled, leading to glitches in the audio signal due to missing samples. Because of these problems, GPGPU-powered audio synthesis using OpenGL was not found to be practicable, at least on Android OS. Further evaluation is therefore set aside.

[23]This can be achieved using the rendermode “dirty” and calling `requestRender` on a `GLSurfaceView`.

4.5.5. Overall Experiences

Despite initially disappointing results, using OpenGL shaders on mobile devices for GPU computing was found to be feasible – at least for data-parallel image processing tasks. Using special shared memory API calls effectively eliminates the bottleneck of reading back pixel data from the GPU. Mapping a parallel algorithm to shader programs is at times a challenging, ineffective or impossible task due to limitations in the programmable shader pipeline of OpenGL ES 2.0. More complex algorithms must be divided into multiple rendering passes, introducing additional overhead. API extensions, undocumented or unofficial functions are necessary in order to gain decent memory transfer speeds but introduce portability issues. Because of this, it is necessary to implement platform-dependent code by querying for available OpenGL extensions. As a fallback, GPU computing support could be disabled for devices that only support OpenGL 1.x or have no alternative to the slow `glReadPixels` function. Nevertheless, OpenGL ES is still the GPGPU option offering the widest device support in the mobile world by far.

4.6. Summary of Technology Comparison Results

The comparison of different technologies that can be used for GPU computing on mobile devices revealed many problems but also a high degree of potential. First, a significant speedup can be achieved with all technologies in the case of highly data-parallel tasks like image convolution. Memory transfer costs always need to be taken into consideration. When shared memory in the SoC architecture of mobile devices can be exploited, these costs can be seriously reduced. The support on the API level is the key, since the potential in the hardware already exists. The main issue is portability – OpenGL ES is the only GPGPU option available on most mobile devices. Just this technology has no direct GPU computing support and is therefore hard to employ in many parallel algorithms. RenderScript could be a solution for this dilemma – at least for many Android devices – but fails because of bad documentation and obscured task scheduling. Alternatively, OpenCL provides an excellent programming model for GPU computing and even supports OpenGL interoperability. Although it is widely supported on the hardware – the Mali-GPU series is a notable example – no noteworthy mobile OS comes with software level support.

Until this situation changes, OpenGL ES stays the only technology usable for GPU computing in practical projects. Therefore, it was chosen to undertake optimizations for a marker detection project as described in the next chapter.

5. Utilization of GPU Computing for Marker Detection

In this chapter, GPGPU is employed in a practical software project for mobile devices in order to assess its potential in the context of real-time applications. A common computer vision task was chosen for this, which often occurs (in different variants) in the context of *Augmented Reality (AR)*[1] – real-time marker detection. The following sections briefly describe the basic algorithms that are usually involved in this task and will then introduce the CPU-based reference implementation. Then, some starting points that seem promising for optimizations in terms of GPU computing will be identified, and possible ways of implementing these improvements will be outlined. Following the presentation of selected interesting components of the implementation, the results are shown and interpreted in the final section.

5.1. Marker Detection for Augmented Reality

In AR systems, the view of the real world is enhanced with virtual objects. These objects need to be placed correctly in the real world view to achieve the effect of seamless integration of virtual elements in a natural environment. To achieve this, some kind of target or tag in the real world needs to be defined on which virtual elements can be correctly placed by calculating the visually appropriate pose. Usually, this target consists of a marker which can be detected and tracked in the camera video stream. Although there are also variants providing "marker-less" AR (meaning that arbitrary objects can be defined as markers), marker-based methods are considered faster and more reliable [Bag+12, pp. 94-95]. Such markers usually provide some unique code (a *marker ID*) on a square that can be extracted and thus enables mapping of different content to different IDs.

5.1.1. The AR Pipeline and its Theoretic Backgrounds

A complete marker-based AR system consists of several processing steps divided into three main parts: Marker detection, pose estimation and visualization. Since this work

[1]The term *Augmented Reality* describes an environment that provides a view of the real world which is supplemented with virtual elements. In contrast to *Virtual Reality*, "AR supplements reality, rather than completely replacing it." [Azu97]

is not primarily dedicated to AR but instead to image processing, the focus will be set on the first step. It includes many computer vision tasks that might be parallelized on a GPU. Furthermore, it is the most computationally expensive part of the AR system. In this section the relevant algorithms that are especially needed by this part will be explained. A fully functional AR system will be presented at the end.

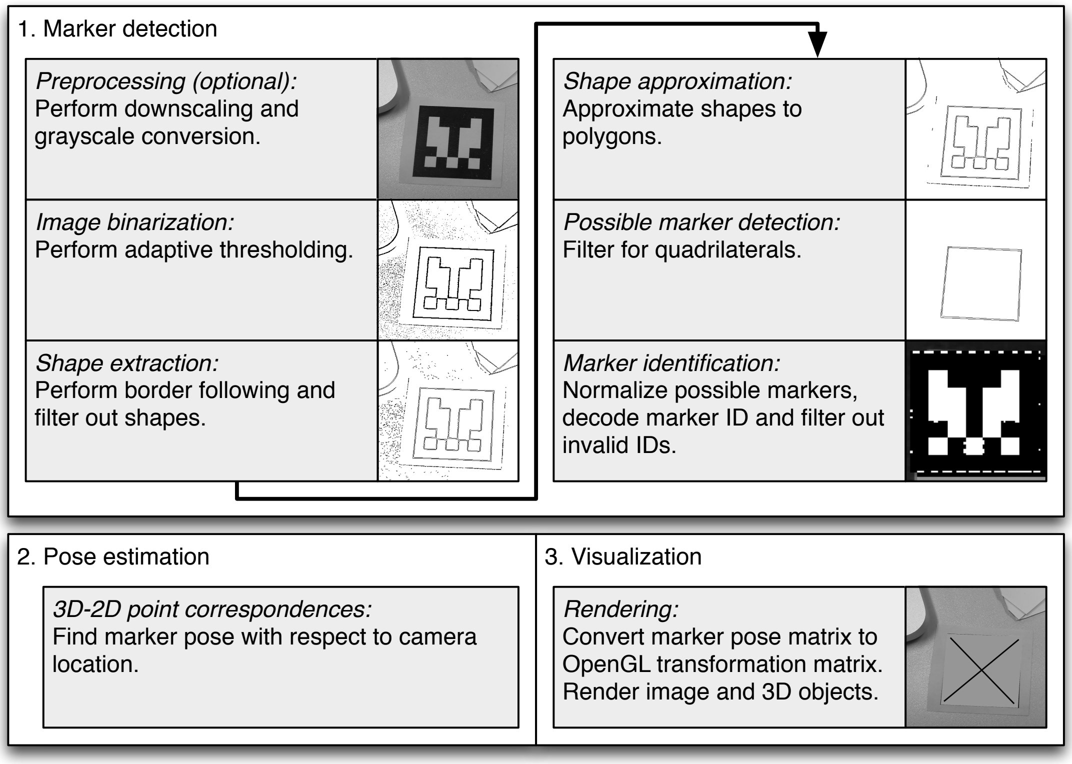

1. Marker detection
Preprocessing (optional):
Perform downscaling and grayscale conversion.
Image binarization:
Perform adaptive thresholding.
Shape extraction:
Perform border following and filter out shapes.
Shape approximation:
Approximate shapes to polygons.
Possible marker detection:
Filter for quadrilaterals.
Marker identification:
Normalize possible markers, decode marker ID and filter out invalid IDs.
2. Pose estimation
3D-2D point correspondences:
Find marker pose with respect to camera location.
3. Visualization
Rendering:
Convert marker pose matrix to OpenGL transformation matrix. Render image and 3D objects.

Figure 5.1.: Augmented Reality pipeline after [Bag+12]. Images in steps 2-5 are inverted.

There are already complete AR frameworks available for mobile devices, such as ARToolKit[2], Qualcomm Vuforia[3], AndAR[4] or ArUco[5]. For better evaluation and understanding, however, a new implementation was written with the help of OpenCV following the ideas described in [Bag+12, pp. 47-92]. The approach is based upon the popular AR system design suggested by Kato and Billinghurst [KB99], which is also implemented in ARToolKit. The marker IDs are encoded as a binary code in 7x7 cells, of which the

[2]Published under GNU General Public License. See `http://www.hitl.washington.edu/artoolkit/`.
[3]Closed source commercial product. See `https://www.vuforia.com/`.
[4]Provides an Android Java API for the underlying ARToolKit. See `https://code.google.com/p/andar/`.
[5]Published under BSD License. See `http://www.uco.es/investiga/grupos/ava/node/26`.

outer cells must be black (binary "0") for validation reasons, yielding a 5x5 bitfield. This *AR pipeline* includes several image processing steps that are outlined in figure 5.1 and are explained in more detail:

Preprocessing: In this optional step, the camera frames will be downscaled to a specific size and converted to grayscale images. This is necessary because high-resolution camera frames take too long to process, include too many details and fine-grained noise – which can have negative effect in upcoming processing steps. An output resolution of 640x360 pixels can be considered sufficient for the following steps.

Image binarization: It is necessary to binarize the image in order to find contours. Therefore, object contours should remain identifiable after this process. Absolute thresholding with a fixed value cannot achieve this because it is highly dependent on lighting conditions and produces problems with gradients. A better approach is using adaptive thresholding, because it calculates a threshold $t_{x,y}$ for each pixel in the image I by calculating the mean of the surrounding pixels in the neighborhood size B (hence employing neighborhood block-based averaging) and subtracting it with C:

$$t_{x,y} = \frac{\sum_{i=-b}^{b} \sum_{j=-b}^{b} I(x+i, y+j)}{B^2} - C, \text{where } b = \lfloor B/2 \rfloor. \tag{5.1}$$

Since the threshold $t_{x,y}$ is dependent on the block of surrounding pixels, it produces contours in the resulting binary image along strong gradient changes (i.e. edges). When a pixel at x, y is part of a homogenous neighborhood (soft gradient changes), it is not considered a contour since this pixel value will be smaller than $t_{x,y} - C$. So C can be used to control when a change in gradient is considered to be strong.

Shape extraction: This step is crucial because it extracts geometric information about the contours in the binary image. For the approach presented in [Bag+12, pp. 47-92], OpenCV's `findContours` function is used. This implements an algorithm suggested by Suzuki and Abe [SA85]. Their paper describes a way to find topological information about contours by hierarchical border following. Finally, contours with less than four contour points are dropped because they cannot form a quadrilateral.

Shape approximation: Since most contour points form polygonal curves with multiple vertices, it should be approximated to a less complex polygon. Here, another OpenCV function `approxPolyDP` can help. It implements the *Ramer–Douglas–Peucker* algorithm [DP73] for finding a similar curve with fewer points whose distance (i.e. "dissimilarity") to the original curve is less than or equal to a specified approximation accuracy ϵ.

Possible marker detection: After the polygons have been approximated, all of them that cannot form a quadrilateral are dismissed. This includes all with a polygon count not equal to four and those that do not form a convex shape. Furthermore, the quadrilaterals that are left must meet the minimum length requirement for each edge – otherwise, their shape would be too small or too tapered.

Marker identification: Now all shapes that are left are quadrilaterals. Their corner points can be normalized so that they form a perfect square of equal dimensions (just like the original marker image). Hence, any skewing or scaling caused by the perspective view is undone. This can be achieved in two steps. First, a perspective transform matrix M is obtained so that for each corner point x_i, y_i from the source image the following equation in relation to the normalized points x'_i, y'_i is satisfied:[6]

$$\begin{bmatrix} t_i x'_i \\ t_i y'_i \\ t_i \end{bmatrix} = M * \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix} . \tag{5.2}$$

This can be done using the OpenCV function `getPerspectiveTransform`. In the next step the matrix M can be applied to the source image. After this transformation the normalized square image S of the found quadrilateral can be obtained.

$$S(x, y) = I(p_x, p_y), \text{where } p = M * \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} . \tag{5.3}$$

OpenCV's `warpPerspective` function can be used for this calculation with each discovered potential marker. To obtain the marker ID encoded in its binary image, at first another thresholding operation needs to be applied to the extracted square image S. This time, a fixed-level threshold is necessary since image segments are binarized, but not contours. To avoid miscalculation, this step must not be prone to lighting changes – so a method must be applied which takes the overall tonal distribution into consideration. Otsu thresholding [Ots79], among others[7], provides these qualities. For this, a grayscale histogram of each extracted image S is generated and an optimal threshold value for separating the binary classes (black and white) in the image is calculated with Otsu's method by minimizing their intra-class variance. Since the threshold is not applied to the whole camera frame but only to the extracted squares of possible markers, even such areas that have a suboptimal tonal distribution due to different lighting conditions within the picture, can be binarized. After this step, the binary code in the image can be efficiently extracted by dividing the image into 7x7 cells and counting the white pixels (and hence also the black pixels) in each cell. Depending on the majority of pixels belonging to one class or the other, the cell yields a binary "0" or "1", resulting in a marker code.[8] If no valid marker ID could be found in a square, it is dismissed so that only correctly identified markers are left.

[6] This is the general equation. In the context it is used here, t_i is always 1.

[7] Another example is the k-means approach [Llo82] applied to a grayscale image histogram.

[8] The algorithm further checks the code against a small set of possible codes for each possible marker orientation. For details about the binary code extraction and validation see [Bag+12, pp. 72-74].

After the markers have been detected, their position and orientation in relation to the camera pose can be estimated. This is done by finding 3D-2D point correspondences between the object in 3D space and its 2D projections (*extrinsic* matrix), and calculating the inverse of this matrix.[9] It requires a matrix of camera parameters (*intrinsic* parameters), which among other things describe the focal length of the camera. OpenCV provides a function `solvePnP` to calculate the extrinsic matrix. This matrix can be converted to an OpenGL transformation matrix that can be used in the visualization process to render virtual objects on top of the found markers.

5.1.2. CPU-based Reference Implementation

Now that the involved algorithms in the marker detection process have been briefly described, the CPU-based reference implementation and its results are presented. As already mentioned, the AR system design described by [Bag+12, pp. 47-92] provides the outline on which a reference system has been implemented. The main steps in the marker detection part of the AR pipeline have been employed by using the OpenCV functions named in the previous section. An Android application was written which uses the native OpenCV 2.4.7 C++ API via the Android NDK. The camera frames are obtained using OpenCV's `VideoCapture` class. The requested camera frame size is 1280x720 pixels. It is halved during preprocessing to yield frames with pixel dimensions 640x360 to use in the remaining steps of the marker detection pipeline. Adaptive thresholding is performed in a 5x5 pixel neighborhood. All possible markers are deskewed and normalized to 64x64 pixels in order to read their marker code.

After marker detection and pose estimation, each marker found is drawn as a square with a with a different color determined by its ID. OpenGL ES 2.0 is used for the visualization step.

5.1.3. Evaluation of the Reference Implementation

Testing Environment and Undertaken Measurements

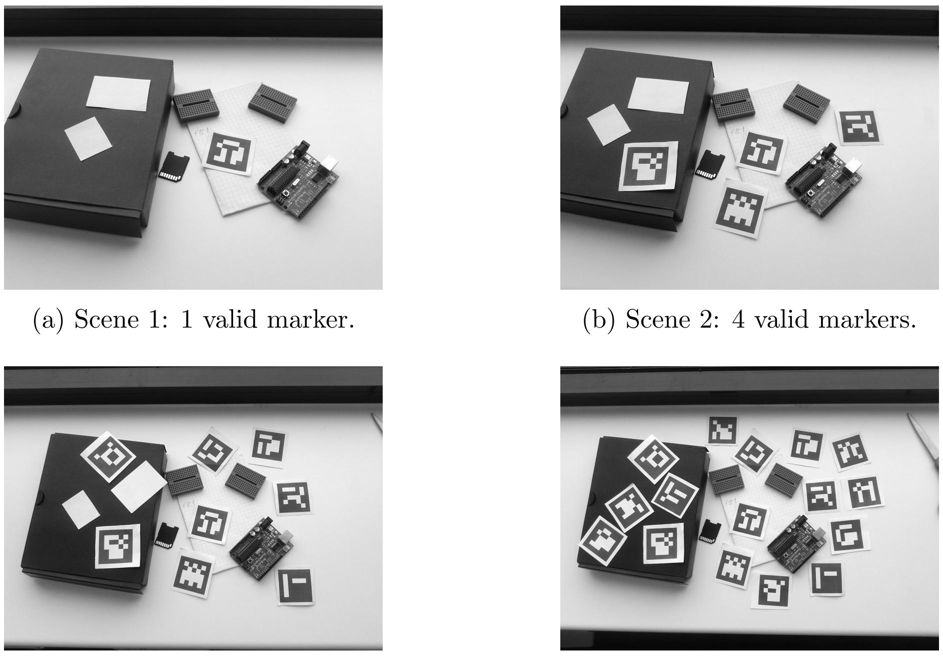
The same Android device as described in section 4.2.1 was used for development and testing. Time measurements for six steps in the marker detection process were conducted: Preprocessing, adaptive thresholding, shape extraction (finding contours), shape approximation *and* filtering for quadrilaterals, marker identification, and marker pose estimation. If image format conversions or copy operations were necessary, they are included in the time measurements (this also applies for the CPU-GPU memory transfers in the later described accelerated version). The results always represent the average of 100 measurements that were taken with still images to provide a precise and exact

[9]The inverse needs to be calculated because the marker pose in relation to the camera needs to be found and not vice versa. See [Bag+12, pp. 78-81] for more details.

Scene	Valid markers	Contour points	Possible markers
1	1	452	6
2	4	484	17
3	8	507	17
4	16	493	22

Table 5.1.: Table describing the different test scene images and their properties. The images themselves can be seen in figures 5.2.

scenario for all measurements. The still images have the same resolution as camera frames (1280x720) to give realistic results. Four different scenes – each successive image increasing the amount of markers – are tested. The images are depicted in figures 5.2 and their properties are summarized in table 5.1. Having 16 markers in one frame as in scene 4 is of course rather unlikely in a real environment, but offers better understanding of how big the impact of the number of markers is on the overall application speed. Additionally, an overall frame rate in frames per second (fps) was measured for real-time performance evaluation using the camera.


(a) Scene 1: 1 valid marker.

(b) Scene 2: 4 valid markers.

(c) Scene 3: 8 valid markers.

(d) Scene 4: 16 valid markers.

Figure 5.2.: Test scene images.

Performance Results

All markers were correctly detected, identified and visualized in the four test scenes. The execution times of the individual processing steps can be seen in figure 5.3. As is obvious, the first three processing steps perform in constant time since they are only dependent on the input image and downscaled image resolution, which stay the same. Preprocessing and adaptive thresholding contribute most to the overall processing time, with a total of about 41 ms added. Shape extraction using `findContours` executes quite fast – within 3 to 5 ms. Also, shape approximation and filtering for quadrilaterals only rises slightly with an increased number of contour points. Adding 3 to 5 ms to the execution time, it is not as significant for the overall speed as the marker identification step, which takes about 4 to 18 ms depending on the number of quadrilaterals that represent possible markers. The last step, marker pose estimation, does contribute to total execution time when an unrealistically high number of markers are present in the scene. The overall marker detection process executes in between 53 and 83 ms.

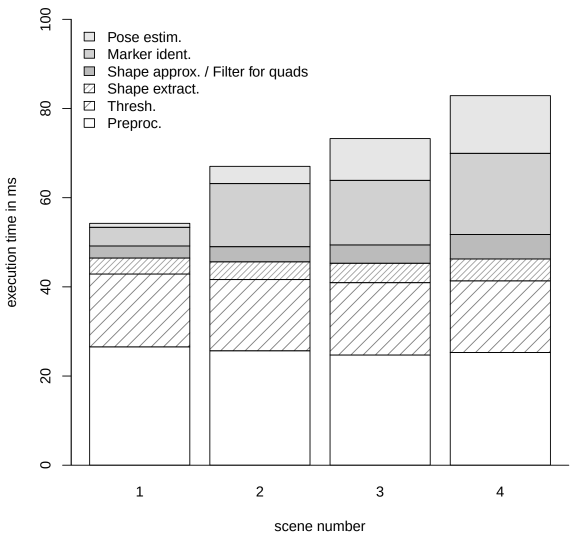


Figure 5.3.: Execution times for marker detection with the reference implementation using OpenCV. Individual processing steps are represented in the stacked bars.

The overall application frame rate is approximately 7 fps, changing only slightly when more than about 10 markers should be detected. One reason for this rather poor performance seems to be slow camera frame rates of the Nexus 10 device in conjunction with OpenCV (as reported in [xaf13]). Evaluation showed that only 16 fps can be achieved for a pure camera frame preview (when disabling marker detection or using an OpenCV “native camera” example project). Improving the camera frame output rate is not part of this thesis, so the following optimizations will focus on steps in the marker detection process.

5.2. Accelerating the Marker Detection Process with GPGPU

5.2.1. Evaluation of Possible GPU-based Optimizations

In order to optimize the performance of the marker detection process using GPGPU, evaluation of what parts might be efficiently sourced out on the GPU should be performed. Which steps take notable time to complete on the CPU and could potentially perform better on a GPU – taking into account the additional memory transfer overhead? Which of the presented technologies and algorithms could be used? In order to answer these questions, the major detection steps should be examined in more detail:

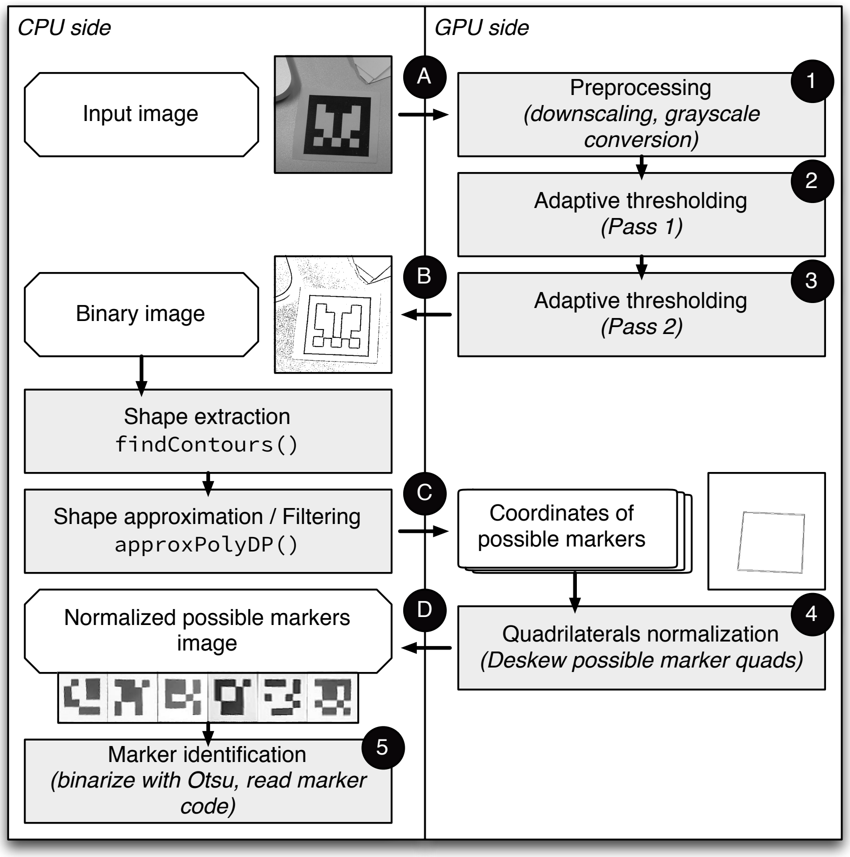
Preprocessing and image binarization: As mentioned in the previous section, these steps make up about 50% to 80% of the total processing time. Since the involved algorithms work in a data-parallel fashion, they can be efficiently implemented for GPU computing. Especially image scaling can be easily accelerated with OpenGL. Employing image binarization via adaptive thresholding on a graphics unit is feasible since it is a kind of linear filter operation. The averaging operations of an adaptive thresholding filter are separable (see section 4.1.1) so this part could be implemented as two-pass filter with OpenGL fragment shaders.

Shape extraction, approximation and filtering for quads: Shape extraction (i.e. extracting contours from the binary image) does not overtly hinder the detection process. Furthermore, parallelization is difficult since the original algorithm employs border following that involves a map of border states (see limits for parallel computing in section 2.4.2) and many branches. However, implementing the PC lines variant of Hough transform (see section 4.1.2) could be beneficial since a shape approximation step could be omitted and a continuous processing pipeline on the GPU is assured. On the other hand, possible quadrilaterals in the line intersections must be detected in an additional step. With respect to the issues that are involved with implementing the Hough transform variant using OpenGL ES 2.0 or OpenCL (see sections 4.5.3 and 4.3.3), this approach should be set aside.

Marker identification: This step makes up 7 to 21% of the total processing time, depending on how many possible markers have been detected. Improving the execution speed of this step could contribute to faster overall performance. Transforming and normalizing the detected quadrilaterals to form equally sized squares can be efficiently mapped to a graphics unit. Using OpenGL seems appropriate since it is a task that is in alignment with its main purpose – altering a shape and its texture according to a given transformation. Otsu thresholding could also be accelerated by calculating the image histogram of each normalized square image on the GPU, where the thresholding process itself could also be performed. Reading the marker code from the binarized square image could be sped up by downscaling each portion of it to pixel dimensions of 7x7 (according to the number of marker code cells). Linear interpolation can be used for downscaling. Using this method, the grayscale value of each pixel of the downscaled image represents the proportion of black and white pixels of each marker cell. Retrieving the marker code can then be done very quickly on the CPU side.

Marker pose estimation: This step does not have a big impact on the overall performance, hence GPGPU-based optimization for it should be set aside.

5.2.2. Implementation Approach



CPU side
GPU side
Input image
A
Preprocessing
(downscaling, grayscale conversion)
1
Adaptive thresholding
(Pass 1)
2
Binary image
B
Adaptive thresholding
(Pass 2)
3
Shape extraction
findContours()
Shape approximation / Filtering
approxPolyDP()
C
Coordinates of possible markers
Normalized possible markers image
D
Quadrilaterals normalization
(Deskew possible marker quads)
4
Marker identification
(binarize with Otsu, read marker code)
5

Figure 5.4.: Marker detection pipeline using the GPU for certain processing steps. All images but the first are inverted.

Based on the previous observations, an implementation approach was devised. The aim was to design a continuous processing pipeline that harnesses both the strengths of the GPU (efficient computation of data-parallel tasks) and the CPU (efficient execution of complex algorithms that involve branching and dynamic loops) while avoiding unnecessary memory transfer operations. Hence, the preprocessing and image binarization steps should be performed on the graphics unit, while shape extraction and approximation are better suited for CPU processing. Marker identification includes parts that can take advantage of GPU computing (deskewing of possible markers for normalization, calculating the histogram and thresholding), but other parts like Otsu threshold value calculation and reading the marker code are shortened when performed on the CPU side.

Along with these requirements comes the question of which GPGPU technology to use. A conclusion about the situation of available GPGPU technologies on mobile devices has already been drawn in section 4.6 and applies in this context: Since OpenGL ES 2.0 is the only portable solution with widespread support (in contrast to OpenCL) and decent documentation (in contrast to RenderScript), all other technologies are not worth further consideration. Furthermore, it was shown that data-parallel image processing tasks can be implemented very efficiently with it (see section 4.5.2).

During prototype development several problems occurred, which prevented implementation of all the previously mentioned processing steps on the GPU. First, it was found out that it is not possible to use texture lookups in vertex shaders when the texture to be read from is previously created dynamically as a color attachment to an FBO. In practice this means that it is not possible to chain the output of one rendering pass to another rendering pass that tries to read this output (a texture attached to an FBO) in a vertex shader. However, using a fragment shader works. Since it is necessary to perform the texture lookup in a vertex shader for the histogram calculation, this step could not be implemented on the GPU with OpenGL ES 2.0. To provide a continuous processing pipeline all following operations (Otsu thresholding and reading the marker code) remain calculated by the CPU.

A working approach was devised with respect to the aforementioned problems and is depicted in figure 5.4. As can be seen, four rendering passes are performed on the graphics unit to process the input camera image. The individual steps are denoted by numbers in the mentioned figure and are explained in more detail:

Preprocessing (1): This step is implemented in a fragment shader that only calculates the scalar product of the RGB color vector of each pixel with a constant vector of weights for each color channel in order to get a grayscale value based on the luminance Y:[10]

$$Y = \begin{bmatrix} R \\ G \\ B \end{bmatrix} \cdot \begin{bmatrix} 0.299 \\ 0.587 \\ 0.114 \end{bmatrix} \tag{5.4}$$

[10]The channel weights are derived from the luminance calculation in the YUV color model [Rus07, p. 47] and are also implemented in OpenCV's `cvtColor` function.

To perform the downscaling of the input image, the output size for the FBO rendering is set by using `glViewport` with half of the original image dimensions. Trilinear sampling (via `GL_LINEAR_MIPMAP_LINEAR`) is employed on the GPU, interpolating between the pixels of the two subsequent mipmap levels [MGS09, p. 192]. For as yet undetermined reasons, this setting performed slightly faster than bilinear filtering (`GL_LINEAR`), at least on the given hardware.

Image binarization (2 and 3): Adaptive thresholding is implemented via performing two rendering passes on the GPU as shown in listings 5.1 and 5.2. An averaging operation over a 5x5 or 7x7 two-dimensional block size is performed, which can be separated into two orthogonal one-dimensional operations. This approach reduces the number of texture read and division operations for a block size N from N^2 to $2N$. The method was introduced in section 4.1.1 and implemented for a Gauss kernel in section 4.5.2. It also involves accessing the neighboring pixels in the texture with a pixel delta value passed in as uniform vector `uPxD`, which describes the distance between single pixels in normalized texture coordinate (s, t) space. In case of adaptive thresholding, the implementation uses slight differences in order to binarize the image. In the first rendering pass, the grayscale values of the center pixel at x, y (original pixel value `centerGray`) and two neighboring pixels in each horizontal direction are fetched from the texture that was created in the preprocessing step (lines 3-9). The average of all five pixels is calculated and the result is saved in the "red" color channel of the final fragment color (`gl_-FragColor`) (line 13). The "green" channel is set to `centerGray` in order to pass the original grayscale value of the center pixel to the next rendering pass. The other two color channels are not used. In the second rendering pass the same averaging operations are performed in vertical direction on the output of the previous pass (lines 7-13).[11] It yields a final averaged value `avg` from which a constant `bigC` is subtracted in order to form the thresholding value `avg - bigC` according to the original algorithm also implemented in OpenCV. By using the `step`, function the original grayscale value `centerGray` (taken from the "green" channel of the center pixel) is compared to the thresholding value which results in the binarized output image (lines 15-17). This is copied back to main memory for further processing.

Deskewing of possible markers (4): After the binarized image has been processed on the CPU side to detect quadrilaterals that form possible markers, these need to be deskewed and normalized for later marker code recognition. The GPU does this by mapping the original coordinates of each found quadrilateral to the texture coordinates of squares of equal size. These original coordinates describe the corners of the quadrilaterals in the downscaled image of size $W \times H$. In order to deskew them, the efficient texture mapping functionalities of a graphics unit are exploited. The marker deskewing stage in the rendering pipeline is informed about the number of found quadrilaterals. It then calculates the output texture size so that each

[11]For efficient memory access, the input texture is rotated by 90° as also already implemented for the Gauss kernel in section 4.5.2.

normalized square fits in an area of 64x64 pixels. Therefore, this size equals $64 * N$ pixels for N possible markers.[12] After that, it constructs the vertex coordinate buffer for each square so that they are rendered next to each other (see listing A.2 in the appendix). The texture coordinate buffer, which is also calculated in this step, essentially contains the information for finally deskewing the found quadrilaterals. Here, the normalized texture coordinate pair $[s, t]$ for each vertex of each square is calculated as $[s, t] = [x/W, y/H]$ in order to map the corners of the found quadrilaterals to the equal-sized squares, as shown in figure 5.5. Once again, the output is rendered into a texture that is attached to an FBO. It is copied back to the main memory for the final marker code recognition process.

```glsl
// (vTex, uPxD, sTex are defined before)
void main() {
  float centerGray = texture2D(sTex, vTex).r;  // get center pixel value
  // sum of horizontal pixel neighborhood
  float sum = texture2D(sTex, vTex + vec2(uPxD.x * -2.0, 0.0)).r +
              texture2D(sTex, vTex + vec2(uPxD.x * -1.0, 0.0)).r +
              centerGray +
              texture2D(sTex, vTex + vec2(uPxD.x *  1.0, 0.0)).r +
              texture2D(sTex, vTex + vec2(uPxD.x *  2.0, 0.0)).r;
  // get the average
  float avg = sum / 5.0;
  // Result stores average pixel value (R) and original gray value (G)
  gl_FragColor = vec4(avg, centerGray, 0.0, 1.0);
}
```

Listing 5.1: Fragment shader for adaptive thresholding (rendering pass 1).

```glsl
// (BLOCKSIZE, vTex, uPxD, sTex are defined before)
void main() {
  // centerPx stores values from pass 1: horizontal avg and orig. gray value
  vec4 centerPx = texture2D(sTexture, vTexCoord);
  const float bigC = (BLOCKSIZE + 4) / 255.0;
  // sum of vertical pixel neighborhood (input image is rotated by 90deg!)
  float sum = texture2D(sTex, vTex + vec2(uPxD.y * -2.0, 0.0)).r +
              texture2D(sTex, vTex + vec2(uPxD.y * -1.0, 0.0)).r +
              centerPx.r +
              texture2D(sTex, vTex + vec2(uPxD.y *  1.0, 0.0)).r +
              texture2D(sTex, vTex + vec2(uPxD.y *  2.0, 0.0)).r;
  // get the average
  float avg = sum / 5.0;
  // create inverted binary value
  float bin = 1.0-step(avg-bigC, centerPx.g); // centerPx.g is orig. gray value
  // store thresholded values
  gl_FragColor = vec4(bin, bin, bin, 1.0);
}
```

Listing 5.2: Fragment shader for adaptive thresholding (rendering pass 2).

[12] When the maximum texture width (usually 4096 pixels) is exceeded (with $N > 64$), another row is added and the texture height is increased by 64 pixels.

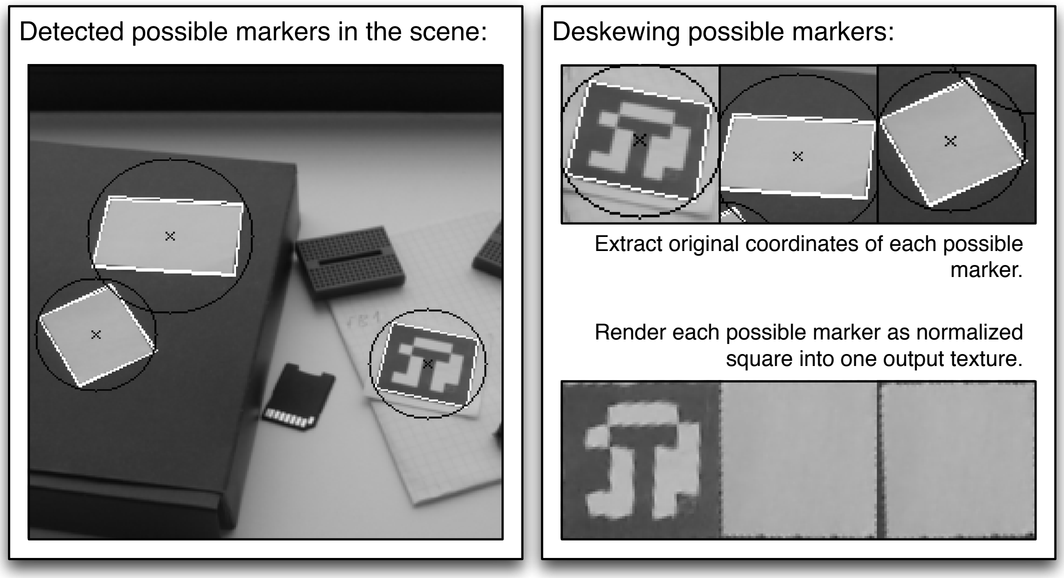

Detected possible markers in the scene:
Deskewing possible markers:
Extract original coordinates of each possible marker.
Render each possible marker as normalized square into one output texture.

Figure 5.5.: Deskewing of possible markers on the GPU.

Four copy operations between the CPU and GPU are necessary during the marker detection process. The first (denoted as A in figure 5.4) is to copy the camera image to the GPU texture memory.[13] Since the input image is not yet downscaled, it represents the most expensive copy operation. Copying back the binary image (operation B) is faster since the image is already downscaled to half of its original size. For normalizing possible markers (C), only the coordinates of the found quadrilaterals in the image are passed as vertex and texture buffers to the GPU. Thus, this operation is very fast. The time for the last copy operation (D) is dependent on the number of possible markers that need to be deskewed, as explained in step 4.

Problems

The suggested approach of dynamically resizing the output frame in step 4 (deskewing possible markers) to keep necessary data transfer to a minimum caused some problems. Every time the number of detected quadrilaterals in a frame changed (which happens nearly every frame), the output texture attached to the FBO of this process needed to be recreated and set to a new size. This led to recurring random empty output frames for this step, resulting in unstable marker detection. To circumvent this, the fix applied

[13] Since the camera image must be present on the SoC memory, there could be a way to directly access this data and hence speed up the process, similar to the Android *gralloc* or *GraphicsBuffer* functions described in section 4.5.2. However, optimizing camera image access is not in the focus of this work.

at first was to render each square in a loop one by one to a 64x64 pixels sized framebuffer and fetch the result. Unfortunately, with this approach only the first square was rendered and all subsequent render requests failed. This is because it is only possible to draw a result into an FBO once within an overall rendering cycle. The solution was to set the output frame size to a fixed minimum of 1024x64 pixels, allowing up to 16 possible markers to be deskewed per rendering cycle, which is sufficient in most cases. If it is not, the output size would be increased temporarily to make place for a bigger amount of detected possible markers. With this solution, highly frequent recreation of the FBO-attached texture could be circumvented. The drawback, of course, is that unnecessarily often a big amount of data needs to be copied from the GPU memory space – in the worst case 16 times more than would be necessary. To avoid this the solution was extended so that instead of calling `memcpy` to copy the whole graphics buffer, only the necessary area is copied by using OpenCV's `Mat::copyTo` in conjunction with a region of interest (ROI).

The adaptive thresholding on the GPU was observed to produce finer contours and is a little more prone to image noise than the OpenCV equivalent. This may be because the preprocessed (downscaled) image is calculated differently in a prior step. It has no influence on the stability of the marker detection process. However, since there are more contour points generated in the shape extraction process, this and the approximation step on the CPU run slightly slower. Different approaches were tried to reduce the number of generated contour points (optimizing the C parameter for thresholding, Gauss filtering, dilation). Each of these either did not provide satisfactory results or added additional processing time, impeding possible performance improvements.

5.2.3. Results

According to the described implementation concept, a partly GPU-accelerated prototype was developed. All markers in the test images were successfully detected and the same marker codes were extracted. The execution times for the different scenes and individual processing steps are presented in figure 5.6. The results include the necessary copy operations and were measured using `glFinish` before and after each step to retrieve exact timings for these operations. As can be seen, the preprocessing step can be executed in significantly less time on the GPU than on the CPU. About 12 ms are measured for the input images for size 1280x720 pixels. The largest time increment (about 9 ms) is caused by the image upload to the GPU. Adaptive thresholding on the GPU takes about the same time and is performed about 4 ms faster than on the CPU, including copying back the binary result image. The marker identification step could be sped up significantly by employing parallel quadrilateral deskewing on the GPU – it can be reduced to between 4 and 6 ms, saving up to 10 ms. This also results in the biggest speedup rate per processing step as shown in figure 5.7b. However, since preprocessing and adaptive thresholding are the most time consuming tasks of the overall marker detection process, their speedup

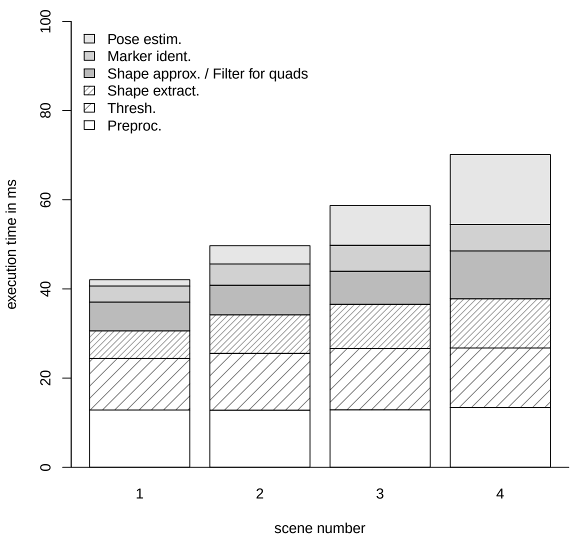
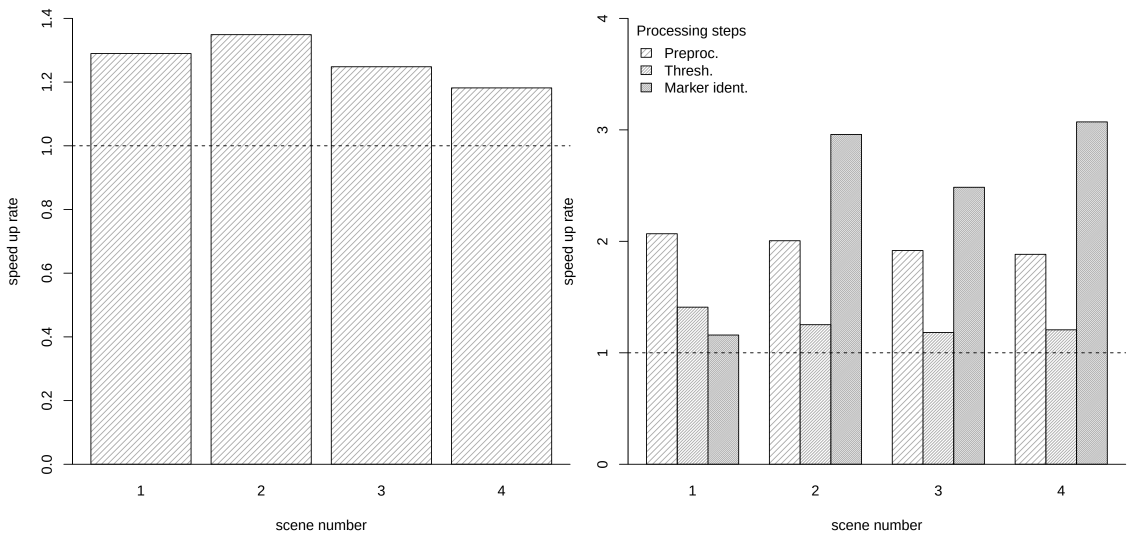


Figure 5.6.: Execution times for marker detection with the OpenGL accelerated approach. Individual processing steps are represented in the stacked bars.

has stronger influence on the overall performance. An altogether speedup of between 1.2x and 1.35x was achieved and is depicted in figure 5.7a.

To measure the overall performance in a realistic scenario, the application prototype was again tested using the camera stream in the same environment as the CPU reference implementation. At first, the observed frame rate of about 8 fps was disappointing, but occasional spikes occurred in which the frame rate increased to as much as 14 fps. This is correlated to the marker identification step. When no possible markers are detected, the whole process of deskewing possible markers and copying back the result image was skipped and hence the frame rate rose. The point of time when to call `glFinish` is of great importance to performance and detection stability. Removing all `glFinish` calls increased the framerate to about 14 fps, but also introduced unstable detection results. The reason for this was that the framebuffer contents, which are copied back from the GPU, would be sometimes empty or outdated when omitting `glFinish`, resulting in faulty marker detection. Adding `glFinish` before each framebuffer read operation (i.e. once after the adaptive thresholding and the deskewing step) solves this problem and leads to a frame rate of about 10 fps, which corresponds as expected with the measured speedup rates of about 1.3x.



(a) Total speedup rates.

(b) Speedup rates per accelerated step.

Figure 5.7.: Speedup rates for marker detection with the OpenGL accelerated approach as compared to the reference implementation.

5.3. Summary of GPU-accelerated Marker Detection Results

GPU-accelerated marker detection could be successfully implemented. By sourcing out some calculations to the graphics unit, a speedup in performance could be achieved. However, this practical project also illustrates the problems and limits of GPGPU on mobile devices in a common application scenario such as marker detection for AR. Many possible speed improvements are seriously reduced by mandatory memory copy operations. This problem would be negligible were more processing steps computed on the GPU side, but implementing a continuous processing pipeline there is hard. Some algorithms are impossible to implement efficiently by means of OpenGL ES 2.0 shaders. Because of this, only three steps of the marker detection processing pipeline could be fully or partly accelerated by the GPU. The development efforts for this are admittedly high in comparison to the achieved frame rate improvement, but nevertheless it shows the potential of the basic concept of distributing work within a mobile platform to specialized hardware such as a GPU. With future improvements, especially in the area of graphics and heterogenous computing APIs, this potential could be unleashed.

6. Summary, Conclusions, and Future Work

This chapter will provide an overview about the obtained results. In addition, the current situation of GPGPU on mobile device platforms will be summarized. Based on these conclusions, future work and potential in this field of research are outlined.

6.1. Summary of Results and Evaluation

The hardware design of mobile devices exhibits distinctive differences when compared to desktop or laptop systems. Mobile device development options suffer from limited hardware resources and more restrictive APIs, as was explained in the beginning of chapter 3. Nevertheless, three different technologies that provide GPU computing support on such devices could be found, explored and evaluated on their merits as described in section 3.2. Previous research exists regarding the performance potential for mobile devices when being used to accelerate certain algorithms on the GPU, but most of them focused only on OpenGL ES 2.0.

In comparison to prior research in this field, this thesis did not limit itself to examining the GPGPU potentials of OpenGL ES 2.0, but also evaluated OpenCL and RenderScript (RS) in chapter 4. The latter underwent only limited examination due to missing elements of documentation and functionality. Several prototypes could be developed in order to study the potential of each technology in different branches of DSP, especially image processing. This proved to be an important branch with lots of potential for GPU computing, yielding speedup rates of up to 8x on the testing device. Although the technology is still in its infancy on mobile platforms, it is nevertheless worthwhile to perform highly data-parallel computations on the GPU.

This is not to say all problems with parallelization on mobile GPUs are negligible. Heterogenous computing in general, and GPGPU on mobile devices in particular, present varying problems to the developer. Portability is the main issue – at the time of writing it is only possible to support a wide range of devices by using OpenGL ES 2.0, which introduces many limitations in terms of GPU computing support. Furthermore, even with this technology it is necessary to implement certain platform- or even device-specific tricks to circumvent GPU-CPU memory transfer bottlenecks, as explained in section 4.5.2. Depending on the graphics rendering performance and memory transfer speed of the device, it is still unclear for the developer if an application would perform faster with or without GPU acceleration. Unfortunately only one testing device was available during research for this thesis, so the performances of different hardware could not be compared. Related research such as [EH11; HSR12; Ris+13] shows heterogenous results for different devices.

All in all the effort of implementing GPU accelerated algorithms that not only run on a wide range of mobile devices but really *accelerate* the overall performance is very high and could quickly exceed the resulting performance gain. A wider support of OpenCL could help with this problem, because its API allows querying hardware characteristics in order to optimize the algorithm accordingly in addition to providing automatic work scheduling. RS, on the other hand, hides all hardware characteristics from the developer and tries to determine the most efficient work scheduling approach by itself. Unfortunately the technology was still not very evolved at the time of writing. Therefore, a final evaluation cannot be given.

6.2. Future Prospects

Harnessing the power of the GPU for general-purpose computing will definitely become more important in the future, especially with upcoming hardware designs that reduce the strict separation between different kinds of processing units. Many hardware manufacturers have realized that only increasing the clock rate of their CPUs does not maintain value as long, especially not for embedded systems that require low heat generation and power consumption. After the strategy of adding more and more processing cores to a CPU die reaches its peak, the future for mobile device hardware likely will belong to heterogenous computing. AMD promotes its new *heterogenous systems architecture (HSA)* based on accelerated processing units (APUs) that "[combine] the benefits of a CPU and a GPU into a single chip" [AMD14]. The "heterogenous queuing" model promises to provide close cooperation between the CPU and GPU, enabling each to distribute workload without involvement on the operating system level. With this, members of the HSA Foundation hope to "boost performance and battery life on mobile devices" [Hal13].

A seamless integration of different, specialized processing units is not yet widely available either for hardware or software. GPGPU on mobile devices requires a lot of effort to provide portability, stability, and improved performance.

A. Appendix

A.1. Additional Source Code Listings

A.1.1. Vertex Shader Listing for Section 4.5.3

```
attribute vec2 aTexCoord;     // texture lookup coordinate
attribute float aTSCoordType; // type of line point: T or S space, begin or end
varying float vFragAddVal;    // value that will be added in the fragment shader
uniform sampler2D sTexture;
// Define the coordinate factors and summands for
// line beginnings and endings in T and S space, respectively.
// mat2x4 is not available in OGLES 2.0, hence a 4x4 matrix is used.
const mat4 tsCoordFactors = mat4(
    0, -1, 0, 0,   // t begin
    0,  1, 0, 0,   // t end
    0,  1, 0, 0,   // s begin
    0,  1, 0, 0    // s end
);
const mat4 tsCoordAdd = mat4(
   -1,  0, 0, 0,   // t begin
    0,  0, 0, 0,   // t end
    0,  0, 0, 0,   // s begin
    1,  0, 0, 0    // s end
);
void main() {
  // get the binary value at aTexCoord
  float bin = texture2DLod(sTexture, aTexCoord, 0.0).r;
  // get the provided index that selects a coordinate factor and summand
  int idx = int(aTSCoordType);
  vec2 coordFact = tsCoordFactors[idx].xy;
  vec2 coordAdd = tsCoordAdd[idx].xy;
  // normalize the coordinates and create a flipped version
  vec2 normCoordStd = -1.0 * aTexCoord + 2.0 * aTexCoord;
  vec2 normCoordFlipped = normCoordStd.yx;
  // select the standard or the flipped version depending on TS coord type
  vec2 finalCoord = vec2(0, 0);
  if (idx == 0 || idx == 3) finalCoord = normCoordStd;
  else finalCoord = normCoordFlipped;
  // calculate one line point in TS space
  vec2 linePoint = bin * (coordFact * finalCoord + coordAdd);
  // set the value that will be added in the fragment shader
  vFragAddVal = bin * (1.0 / 256.0);
  // set the position of the line point
  gl_Position = vec4(linePoint, 0.0, 1.0);
}
```

Listing A.1: Vertex shader for generating PC lines.

A.1.2. Marker Deskewing Function Listing for Section 5.2.2

```
#define NORM_COORD(c) (-1.0f + 2.0f * (c))
// (...)
void PipelineProcMarkerWarp::addMarkerOriginCoords(vector<cv::Point2f> coords) {
  // generate vertex and texture coordinates
  unsigned int vertBufOffset = lastAddedMarkerNum * QUAD_VERTEX_BUFSIZE;
  unsigned int texBufOffset = lastAddedMarkerNum * QUAD_TEX_BUFSIZE;
  // coords contains 4 skewed vertex coordinates (as absolute pixel coordinates)
  const cv::Point2f v0 = coords[0];
  const cv::Point2f v1 = coords[1];
  const cv::Point2f v2 = coords[2];
  const cv::Point2f v3 = coords[3];
  // calculate cell in which the marker will be rendered
  const unsigned int cellX = lastAddedMarkerNum % maxMarkersPerRow;
  const unsigned int cellY = lastAddedMarkerNum / maxMarkersPerRow;
  const float vertXLeft   = NORM_COORD((float)cellX / (float)maxCellX);
  const float vertXRight  = NORM_COORD((float)(cellX+1) / (float)maxCellX);
  const float vertYBottom = NORM_COORD((float)cellY / (float)maxCellY);
  const float vertYTop    = NORM_COORD((float)(cellY+1) / (float)maxCellY);
  // vertex 1: bottom left
  setVertBufCoord(vertexBuf+vertBufOffset, vertXLeft , vertYBottom);
  setTexBufCoord(texCoordBuf+texBufOffset, v0.x/inFrameW, v0.y/inFrameH);
  // vertex 2: bottom right
  setVertBufCoord(vertexBuf+vertBufOffset+3, vertXRight, vertYBottom);
  setTexBufCoord(texCoordBuf+texBufOffset+2, v1.x/inFrameW, v1.y/inFrameH);
  // vertex 3: top left
  setVertBufCoord(vertexBuf+vertBufOffset+6, vertXLeft, vertYTop);
  setTexBufCoord(texCoordBuf+texBufOffset+4, v3.x/inFrameW, v3.y/inFrameH);
  // vertex 4: top right
  setVertBufCoord(vertexBuf+vertBufOffset+9, vertXRight, vertYTop);
  setTexBufCoord(texCoordBuf+texBufOffset+6, v2.x/inFrameW, v2.y/inFrameH);
  // increment number of added markers
  lastAddedMarkerNum++;
}
```

Listing A.2: Calculating vertex and texture coordinates of possible markers.

A.2. Provided Files on the Enclosed CD

Test Images and Results for Evaluations of the Prototypes

The image sets that have been used throughout this thesis for evaluating the performance and quality of certain implementations are located in the directory `test_images` on the enclosed CD. The results of the performance measurements were saved in CSV format in the `results` folder.

Source Code of Prototype Applications

The source code of the programs developed during this thesis are included as separate ZIP-files in directory `sources` on the enclosed CD. Each of the programs has been tested on a Google Nexus 10 device with Android 4.2.1 (see section 4.2.1 for detailed device specifications), but apart from the applications that use OpenCL (see section 3.2.1 for OpenCL device support) the included programs should run on all Android devices with an OS version of at least 3.0.

The source code can be imported into integrated development environments such as *Eclipse*. The image processing projects require the OpenCV Java package be imported.[1] Many of the provided projects use the Android NDK, which requires separate compilation of the C/C++ part of the source code. Because some projects use *SWIG*[2] to generate the JNI source parts automatically, the command listed in the file `swigcmd` in the project root directory has to be invoked beforehand.[3] Most applications are written to provide statistical information about their performance and unless otherwise noted, they do not provide a sophisticated graphical user interface. Furthermore, the majority of these programs can only be configured at compile-time by adjusting some constants in the source code or in the compile settings. More information about this is given for each project in the following records:

ClAudio implements the OpenCL audio synthesis project described in section 4.3.4. Uses the NDK. Buffer size and sample rate can be configured in file `jni/cl_audio.c`.

ClImageProc implements OpenCL-based image convolution and Hough transform from sections 4.3.2 and 4.3.3. Uses the NDK. Image processing algorithm and other options can be configured in `src/net/mkonrad/climageproc/MainActivity.java` and `jni/Android.mk`.

CvImageDroid is the CPU-based reference implementation for image convolution and Hough transform as described in section 4.2.3. Configuration options can be set in `src/net/mkonrad/imagedroid/MainActivity.java`.

[1]See `http://opencv.org/opencv-java-api.html`.
[2]See `http://www.swig.org/`.
[3]If the project is imported into Eclipse, the complete build chain is already provided and no separate SWIG invocation and NDK-based library compilation is necessary.

CvMarkerDetect contains the full AR project described in chapter 5. The AR pipeline is implemented in the self developed `cv_accar` library, whose sources are located at `jni/cv_accar`. It can be configured by editing `jni/cv_accar/common/conf.cpp` to enable or disable GPU acceleration among other options. The Android application shows an options panel with which it is possible to view the output at different AR pipeline stages.

GlAudio is an attempt to generate audio with OpenGL ES 2.0 shaders, which failed because of several reasons described in section 4.5.4.

GlImageProc is a prototype that implements the initial approach for image convolution with OpenGL ES 2.0 shaders as depicted in section 4.5.2. It makes use of the OpenGL Java API and employs `glReadPixels` for memory transfer. Options can be edited in the file `src/net/mkonrad/glimageproc/GLRenderer.java`.

GlImageProcNative is an NDK-based port of the above project and furthermore implements an optimized approach as described in 4.5.2 (including the usage of EGL image extensions instead of `glReadPixels`). It additionally implements Hough transform. The comprehensive source code is mainly implemented in C++ classes under `jni/gl_gpgpu`. Options can be set in the file `src/net/mkonrad/-glimageprocnative/MainActivity.java`.

RsImageProc implements RenderScript-based image convolution as depicted in section 4.4.2. Configurations can be edited in file `src/net/mkonrad/rsimageproc/-MainActivity.java`.

A.3. Comparison of Related Work

See table A.1 on next page.

Article	Use Case	Used API	Used test device(s)	Speedup[1]	Energy[2]
[Ris+13]	SIFT	OpenGL ES[3]	Snapdragon S4, Nexus 7, Galaxy Note II, Tegra 250	4.7x-7.0x	87%
[Wan+13]	Visual object removal	OpenCL EP[3]	Snapdragon S4	4x-92x[4]	–
[Pul+12]	Stitching, video stabilization	OpenGL ES 2.0	Tegra 3 dev. board	1.5x-2x, 5x-6x[11]	–
[HSR12]	SURF	OpenGL ES 2.0	Snapdragon S4, iPad 4G, iPhone 4S, Tegra 3, HTC Evo 3D, Galaxy S II, Desire Z, Nokia N9	2x-14x	–
[Bor+11]	Image processing[5]	OpenGL ES 2.0	Beagleboard rev. C, Zoom AM3517 EVM, Nokia N900	0.6x-0.9x, 1.3x-1.4x[6,7]	115%[7]
[EH11]	Canny edge detection	OpenGL ES 2.0	Nexus One, Nexus S, Galaxy S, Galaxy S II, iPhone 4, Desire HD, Nokia N8	0.4x-2.4x	-
[CW11]	Face recognition (Gabor)	OpenGL ES 2.0	Tegra 2 dev. board	4.3x, 1.8x[6]	26%, 55% [6]
[SPC10]	Image processing[8]	OpenGL ES 2.0	TI OMAP 3430 dev. board	-[9]	-
[Kay10]	SIFT	OpenGL ES 2.0	AMD Z400 family (Adreno 200)	-[9]	-
[Bor+09]	Image warping	OpenGL ES 1.1	Nokia N95	4x	-
[LNS09]	Image processing[10]	OpenCL EP 1.0	TI OMAP 3430 dev. board	3.6x, 3.4x[6]	14%, 33% [6]

Table A.1.: Comparison of related work on the topic of GPGPU on mobile devices

[1] Speedup for implementation (co-)utilizing the GPU as compared to the implementation running on the CPU alone, unless otherwise noted.
[2] Energy consumption improvement/loss per image; less is better for GPU implementation.
[3] Version not specified.
[4] Only the duration of 393.8s for the OpenCL implementation running on the CPU is provided, compared to 96.7s down to 4.3s on the GPU (depending on algorithm configuration). There are is no CPU-optimized implementation.
[5] Includes image scaling and LBG algorithm.
[6] First results for GPU-only implementation, latter for heterogenous CPU/GPU solution.
[7] Results for combined algorithm (scaling and LBG).
[8] Video scaling, cartoon-style non-photorealistic rendering, Harris corner detector.
[9] No comparative results to CPU-only implementation reported.
[10] Geometry correction, Gaussian blur, color adjustment.
[11] First result for panorama stitching, second for video stabilization.

Bibliography

[AAB12] A. Acosta, F. Almeida, and V. Blanco. “ParallDroid: A Framework for Parallelism in Android”. In: *Actas XXIII Jornadas de Paralelismo (JP2012)*. Universidad Miguel Hernández, 2012.

[AMD14] Advanced Micro Devices Inc. *AMD and HSA: A revolutionary, new architecture pioneered by AMD*. 2014. URL: `http://www.amd.com/us/products/technologies/hsa/Pages/hsa.aspx` (visited on 02/06/2014).

[Amd67] Gene Amdahl. “Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities”. In: *AFIPS Conference Proceedings*. Vol. 30. 1967, pp. 483–485.

[AOS] Android Open Source Project. *Android Graphics*. URL: `http://source.android.com/devices/graphics.html` (visited on 01/15/2014).

[App13] Apple Inc. *iOS Device Compatibility Reference - OpenGL ES Graphics*. 10/22/2013. URL: `https://developer.apple.com/library/ios/documentation/DeviceInformation/Reference/iOSDeviceCompatibility/OpenGLESPlatforms/OpenGLESPlatforms.html` (visited on 01/13/2014).

[ARM13] ARM Ltd. *Mali-T604*. 2013. URL: `http://www.arm.com/products/multimedia/mali-graphics-plus-gpu-compute/mali-t604.php` (visited on 01/06/2014).

[AS08] T. Akenine-Möller and J. Ström. “Graphics Processing Units for Handhelds”. In: *Proceedings of the IEEE* 96.5 (2008), pp. 779–789.

[Azu97] Ronald T. Azuma. “A Survey of Augmented Reality”. In: *Presence: Teleoperators and Virtual Environments* 6.4 (08/1997), pp. 355–385.

[Bac72] John Backus. “Can programming be liberated from the von Neumann style?” In: *Communications of the ACM* 21 (08/1972), pp. 613–641.

[Bag+12] Daniel Lelis Baggio et al. *Mastering OpenCV with Practical Computer Vision Projects*. Packt Publishing, 2012.

[Bhu09] Moreshwar R. Bhujade. *Parallel Computing*. 2nd ed. New Age Science, 2009.

[Bim13] Bim. *Reading the OpenGL backbuffer to system memory*. 01/28/2013. URL: `http://lektiondestages.blogspot.de/2013/01/reading-opengl-backbuffer-to-system.html` (visited on 01/15/2014).

[Bla13] Barney Blaise. *POSIX Threads Programming*. 01/2013. URL: `https://computing.llnl.gov/tutorials/pthreads/` (visited on 10/18/2013).

[BML08] David Blythe, Aaftab Munshi, and Jon Leech, eds. *OpenGL ES Common and Common-Lite Profile Specification Version 1.1.12 (Full Specification).* The Khronos Group Inc. 04/2008.

[Bor+09] Miguel Bordallo López et al. *Graphics hardware accelerated panorama builder for mobile phones.* 2009.

[Bor+11] Miguel Bordallo López et al. "Accelerating image recognition on mobile devices using GPGPU". In: *Proceedings of the SPIE, Parallel Processing for Imaging Applications.* Vol. 7872. 2011.

[Bor12] Patrick Borgeat. *GLSL Audio Sandbox.* 2012. URL: `http://www.cappel-nord.de/b/projects/` (visited on 01/15/2014).

[Bre09] Clay Breshears. *The Art of Concurrency.* 1st ed. O'Reilly, 05/2009.

[BTG06] Herbert Bay, Tinne Tuytelaars, and Luc Gool. "SURF: Speeded Up Robust Features". In: *Computer Vision – ECCV 2006.* Vol. 3951. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, pp. 404–417.

[Bur+] Phil Burk et al. *The Synthesis of Sound by Computer - Additive Synthesis.* URL: `http://music.columbia.edu/cmc/musicandcomputers/chapter4/04_02.php` (visited on 01/07/2014).

[Can86] John Canny. "A Computational Approach to Edge Detection". In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* PAMI-8.6 (1986), pp. 679–698.

[com13] compubench.com. *OpenCL Compute Performance of (GPU) Sony Xperia Z.* 2013. URL: `https://compubench.com/device-info.jsp?config=14950526&test=CLB10101` (visited on 11/09/2013).

[com14] compubench.com. *OpenCL Compute Performance of (GPU) Google Nexus 10.* 2014. URL: `https://compubench.com/device-info.jsp?config=14669863` (visited on 01/09/2014).

[CPT03] A. Criminisi, P. Perez, and K. Toyama. "Object removal by exemplar-based inpainting". In: *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition.* Vol. 2. 2003, pp. 721–728.

[CSG98] David E. Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel Computer Architecture.* 1st ed. Morgan Kaufmann, 08/1998.

[CW11] Kwang-Ting Cheng and Yi-Chu Wang. "Using mobile GPU for general-purpose computing - A case study of face recognition on smartphones". In: *International Symposium on VLSI Design, Automation and Test (VLSI-DAT).* 2011, pp. 1–4.

[DAS12] Michel Dubois, Murali Annavaram, and Per Stenström. *Parallel Computer Organization and Design.* 1st ed. Cambridge University Press, 10/2012.

[DH72] Richard O. Duda and Peter E. Hart. "Use of the Hough transformation to detect lines and curves in pictures". In: *Communications of the ACM* 15.1 (1972), pp. 11–15.

[DHH11] Marketa Dubska, Jiri Havel, and Adam Herout. “Real-time detection of lines using parallel coordinates and OpenGL”. In: *Proceedings of the 27th Spring Conference on Computer Graphics*. ACM. 2011.

[Dia07] Franck Diard. *GPU Gems 3 - Using the Geometry Shader for Compact and Variable-Length GPU Feedback*. 2007. URL: `http://http.developer.nvidia.com/GPUGems3/gpugems3_ch41.html` (visited on 10/13/2013).

[DP73] David H Douglas and Thomas K Peucker. “Algorithms for the reduction of the number of points required to represent a digitized line or its caricature”. In: *Cartographica: The International Journal for Geographic Information and Geovisualization* 10.2 (1973), pp. 112–122.

[Edd06] Steve Eddins. *Separable convolution: Part 2*. 11/28/2006. URL: `http://blogs.mathworks.com/steve/2006/11/28/separable-convolution-part-2/` (visited on 01/14/2014).

[EH11] Andrew Ensor and Seth Hall. “GPU-based Image Analysis on Mobile Devices”. In: *CoRR* abs/1112.3110 (2011).

[Fly72] Michael J. Flynn. “Some Computer Organizations and Their Effectiveness”. In: *IEEE Transactions on Computers* 21 (09/1972), pp. 948–960.

[FO08] Leandro A.F. Fernandes and Manuel M. Oliveira. “Real-time line detection through an improved Hough transform voting scheme”. In: *Pattern Recognition* 41.1 (2008), pp. 299–314.

[Fun13] Cedric Fung. *Faster Alternatives to glReadPixels and glTexImage in OpenGL ES*. 11/04/2013. URL: `https://vec.io/posts/faster-alternatives-to-glreadpixels-and-glteximage2d-in-opengl-es` (visited on 01/15/2014).

[Gar13] Rahul Garg. *Android 4.3 update for Nexus 10 and 4 removes unofficial OpenCL drivers*. 08/01/2013. URL: `http://www.anandtech.com/show/7191/android-43-update-for-nexus-10-and-4-removes-unofficial-opencl-drivers` (visited on 11/09/2013).

[Gas+13] Benedict R. Gaster et al. *Heterogeneous Computing with OpenCL*. Revised OpenCL 1.2 Edition. Morgan Kaufmann, 2013.

[Göd06] Dominik Göddeke. *GPGPU::Basic Math / FBO Tutorial*. 2006. URL: `http://www.seas.upenn.edu/~cis565/fbo.htm` (visited on 10/13/2013).

[Goo] Google Inc. *RenderScript / Android Developers*. URL: `http://developer.android.com/guide/topics/renderscript/compute.html`.

[Goo13] Google Inc. *Nexus 10 Tech Specs*. 2013. URL: `https://www.google.com/nexus/10/specs/` (visited on 01/06/2014).

[Goo14] Google Inc. *Android Developer Device Dashboards - OpenGL*. 01/08/2014. URL: `http://developer.android.com/about/dashboards/index.html#OpenGL` (visited on 01/13/2014).

[Gro13] Khronos Group. *Khronos Releases OpenCL 2.0.* 06/22/2013. URL: https://www.khronos.org/news/press/khronos-releases-opencl-2.0 (visited on 11/09/2013).

[Gui12] Herve Guihot. *Pro Android Apps Performance Optimization.* Apress, 01/2012.

[Gus88] John L. Gustafson. "Reevaluating Amdahl's law". In: *Commun. ACM* 31.5 (05/1988), pp. 532–533.

[Hal13] Gareth Halfacree. *AMD announces Heterogeneous Queuing tech.* 10/22/2013. URL: http://www.bit-tech.net/news/hardware/2013/10/22/amd-hq/1 (visited on 02/06/2014).

[Har13] Mark Harris. *CUDA for ARM Platforms is Now Available.* 06/18/2013. URL: https://developer.nvidia.com/content/cuda-arm-platforms-now-available.

[Hej11] Martin Hejna. *Using Eclipse for Android C/C++ Debugging.* 01/23/2011. URL: http://mhandroid.wordpress.com/2011/01/23/using-eclipse-for-android-cc-debugging/ (visited on 01/10/2014).

[HHG99] John Hennessy, Mark Heinrich, and Anoop Gupta. "Cache-coherent distributed shared memory: perspectives on its development and future challenges". In: *Proceedings of the IEEE* 87.3 (1999), pp. 418–429.

[Hin13] Vincent Hindriksen. *Google blocked OpenCL on Nexus with Android 4.3.* 08/01/2013. URL: http://streamcomputing.eu/blog/2013-08-01/google-blocked-opencl-on-android-4-3/ (visited on 11/09/2013).

[Hou59] Paul V. C. Hough. "Machine analysis of bubble chamber pictures". In: *International Conference on High Energy Accelerators and Instrumentation.* Vol. 73. 1959.

[HS88] Chris Harris and Mike Stephens. "A combined corner and edge detector". In: *4th Alvey Vision Conference.* Vol. 15. 1988, pp. 147–151.

[HSR12] R. Hofmann, H. Seichter, and G. Reitmayr. "A GPGPU accelerated descriptor for mobile devices". In: *IEEE International Symposium on Mixed and Augmented Reality (ISMAR).* 2012, pp. 289–290.

[Jos13] Andrew Josey. *POSIX(R) 1003.1 Frequently Asked Questions (FAQ Version 1.14).* 10/2013. URL: http://www.opengroup.org/austin/papers/posix_faq.html (visited on 10/18/2013).

[Kay10] Guy-Richard Kayombya. "SIFT feature extraction on a Smartphone GPU using OpenGL ES 2.0". MA thesis. Massachusetts Institute of Technology. Dept. of Electrical Engineering and Computer Science, 2010.

[KB99] Hirokazu Kato and Mark Billinghurst. "Marker Tracking and HMD Calibration for a video-based Augmented Reality Conferencing System". In: *Proceedings of the 2nd International Workshop on Augmented Reality (IWAR '99).* IEEE. 1999, pp. 85–94.

[Kem+13] Roelof Kemp et al. "Using RenderScript and RCUDA for Compute Intensive tasks on Mobile Devices: a Case Study". In: *First European Workshop on Mobile Engineering*. 2013.

[KH12] David B. Kirk and Wen-Mei W. Hwu. *Programming Massively Parallel Processors: A Hands-On Approach*. 2nd ed. Morgan Kaufmann, 2012.

[Khr14] Khronos Group. *EGL - Native Platform Interface*. 2014. URL: `http://www.khronos.org/egl/` (visited on 01/15/2014).

[Knu76] Donald E Knuth. "Big omicron and big omega and big theta". In: *ACM Sigact News* 8.2 (1976), pp. 18–24.

[Kom13] Jari Komppa. *Instancing in OpenGL*. 01/17/2013. URL: `http://sol.gfxile.net/instancing.html` (visited on 01/24/2014).

[Kur13] Jürgen Kuri. *Marktforscher: Tablet-Markt wächst rasant, PC-Markt bricht weiter ein*. 10/21/2013. URL: `http://www.heise.de/newsticker/meldung/Marktforscher-Tablet-Markt-waechst-rasant-PC-Markt-bricht-weiter-ein-1982814.html` (visited on 10/25/2013).

[Lab03] National Physical Laboratory. *Acoustics*. 2003. URL: `http://www.npl.co.uk/educate-explore/factsheets/acoustics/` (visited on 01/06/2014).

[Lar12a] Brad Larson. *Faster alternative to glReadPixels in iPhone OpenGL ES 2.0*. 04/14/2012. URL: `http://stackoverflow.com/a/9704392` (visited on 01/15/2014).

[Lar12b] Brad Larson. *Introducing the GPUImage framework*. 02/12/2012. URL: `http://www.sunsetlakesoftware.com/2012/02/12/introducing-gpuimage-framework` (visited on 01/24/2014).

[Laz12] Victor Lazzarini. *Android audio streaming with OpenSL ES and the NDK*. 03/03/2012. URL: `http://audioprograming.wordpress.com/2012/03/03/android-audio-streaming-with-opensl-es-and-the-ndk/` (visited on 01/09/2014).

[LBG80] Yoseph Linde, Andres Buzo, and Robert M. Gray. "An algorithm for vector quantizer design". In: *IEEE Transactions on Communications* 28.1 (1980), pp. 84–95.

[LH07] D. Luebke and G. Humphreys. "How GPUs Work". In: *Computer* 40.2 (2007), pp. 96–100.

[Lip11] Benj Lipchak, ed. *GL_OES_texture_half_float, GL_OES_texture_float*. Khronos Group Inc. 11/2011.

[Lip13] Benj Lipchak, ed. *OpenGL ES Version 3.0.3*. The Khronos Group Inc. 12/2013.

[Llo82] Stuart Lloyd. "Least squares quantization in PCM". In: *IEEE Transactions on Information Theory* 28.2 (1982), pp. 129–137.

[LNS09] J. Leskela, J. Nikula, and M. Salmela. “OpenCL embedded profile prototype in mobile device”. In: *IEEE Workshop on Signal Processing Systems (SiPS).* 2009, pp. 279–284.

[Low99] D.G. Lowe. “Object recognition from local scale-invariant features”. In: *Proceedings of the Seventh IEEE International Conference on Computer Vision, 1999.* Vol. 2. 1999, pp. 1150–1157.

[MGK00] Jiri Matas, Charles Galambos, and Josef Kittler. “Robust detection of lines using the progressive probabilistic hough transform”. In: *Computer Vision and Image Understanding* 78.1 (2000), pp. 119–137.

[MGM11] Aaftab Munshi, Benedict Gaster, and Timothy G. Mattson. *OpenCL Programming Guide.* Addison-Wesley Longman, 07/2011.

[MGS09] Aaftab Munshi, Dan Ginsburg, and Dave Shreiner. *OpenGL ES 2.0 Programming Guide.* Addison-Wesley, 2009.

[ML10] Aaftab Munshi and Jon Leech, eds. *OpenGL ES Common Profile Specification Version 2.0.25 (Full Specification).* The Khronos Group Inc. 11/2010.

[MMT95] B.M. Maggs, L.R. Matheson, and R.E. Tarjan. “Models of parallel computation: a survey and synthesis”. In: *Proceedings of the Twenty-Eighth Hawaii International Conference on System Sciences.* Vol. 2. 1995, pp. 61–70.

[Mon12] Clay D. Montgomery. *Using OpenGL ES to Accelerate Apps with Legacy 2D GUIs.* 02/11/2012. URL: `http://software.intel.com/en-us/articles/using-opengl-es-to-accelerate-apps-with-legacy-2d-guis` (visited on 01/20/2014).

[Moo98] Gordon E. Moore. “Cramming More Components onto Integrated Circuits”. In: *Proceedings of the IEEE.* Vol. 86. 1998, p. 4.

[Moz13] Mozilla Community. *C++ Source code for GLFunctions.* 10/28/2013. URL: `http://hg.mozilla.org/mozilla-central/raw-file/12d3ba62a599/widget/android/AndroidGraphicBuffer.cpp` (visited on 01/15/2014).

[MRR12] Michael McCool, James Reinders, and Arch Robison. *Structured Parallel Programming: Patterns for Efficient Computation.* 1st ed. Morgan Kaufmann, 07/2012.

[ND10] J. Nickolls and W.J. Dally. “The GPU Computing Era”. In: *Micro, IEEE* 30.2 (2010), pp. 56–69.

[Neu45] John von Neumann. “First Draft of a Report on the EDVAC”. In: (06/1945). Ed. by Herman Goldstine.

[Ope13] *OpenCV Tutorials: Hough Line Transform.* 12/31/2013. URL: `http://docs.opencv.org/doc/tutorials/imgproc/imgtrans/hough_lines/hough_lines.html` (visited on 12/31/2013).

[OPH96] Timo Ojala, Matti Pietikäinen, and David Harwood. “A comparative study of texture measures with classification based on featured distributions”. In: *Pattern recognition* 29.1 (1996), pp. 51–59.

[Ots79] Nobuyuki Otsu. “A Threshold Selection Method from Gray-Level Histograms”. In: *IEEE Transactions on Systems, Man, and Cybernetics* SMC-9.1 (01/1979), pp. 62–67.

[Owe+05] John D. Owens et al. “A Survey of General-Purpose Computation on Graphics Hardware”. In: *Eurographics 2005, State of the Art Reports*. 09/2005, pp. 21–51.

[Owe+08] J.D. Owens et al. “GPU Computing”. In: *Proceedings of the IEEE* 96.5 (2008), pp. 879–899.

[Pic13] Squid Pickles. *How I built an OpenCL binary for the Nexus 4*. 02/28/2013. URL: `http://sweetpea.tentacle.net/blog/opencl-on-nexus-4/` (visited on 11/09/2013).

[Pul+12] Kari Pulli et al. “Real-time computer vision with OpenCV”. In: *Commun. ACM* 55.6 (06/2012), pp. 61–69.

[QZL12] Xi Qian, Guangyu Zhu, and Xiao-Feng Li. “Comparison and Analysis of the Three Programming Models in Google Android”. In: *First Asia-Pacific Programming Languages and Compilers Workshop (APPLC)*. 2012.

[Ris+13] Blaine Rister et al. “A fast and efficient SIFT detector using the mobile GPU”. In: *Proc. of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*. 2013, pp. 2674–2679.

[Roa96] Curtis Roads. *The Computer Music Tutorial*. MIT Press, 1996.

[Roo00] Seyed H. Roosta. *Parallel Processing and Parallel Algorithms: Theory and Computation*. Springer, 2000.

[RR08] John C. Russ and J. Christian Russ. *Introduction to Image Processing and Analysis*. CRC Press, 2008.

[Rus07] John C. Russ. *The Image Processing Handbook*. 5th ed. CRC Press, 2007.

[SA85] Satoshi Suzuki and Keiichi Abe. “Topological structural analysis of digitized binary images by border following”. In: *Computer Vision, Graphics, and Image Processing* 30.1 (1985), pp. 32–46.

[Sam13] R. Jason Sams. *Evolution of Renderscript Performance*. 01/14/2013. URL: `http://android-developers.blogspot.de/2013/01/evolution-of-renderscript-performance.html`.

[Sca13] Matt Scarpino. *OpenCL on the Nexus 10, A Simple Example*. 02/23/2013. URL: `http://www.openclblog.com/2013/02/opencl-on-nexus-10-part-1.html` (visited on 11/09/2013).

[Sha49] C. E. Shannon. “Communication in the presence of noise”. In: *Proc. Institute of Radio Engineers*. 01/1949.

[Shi12] Anand Lal Shimpi. *Apple A6 Die Revealed: 3-core GPU*. 09/21/2012. URL: `http://www.anandtech.com/show/6323/apple-a6-die-revealed-3core-gpu-100mm2` (visited on 10/25/2013).

[SK09] Robert J. Simpson and John Kessenich, eds. *The OpenGL ES Shading Language.* The Khronos Group Inc. 05/2009.

[SPC10] N. Singhal, In Kyu Park, and Sungdae Cho. "Implementation and optimization of image processing algorithms on handheld GPU". In: *17th IEEE International Conference on Image Processing (ICIP).* 2010, pp. 4481–4484.

[Su+09] Yu Su et al. "Hierarchical Ensemble of Global and Local Classifiers for Face Recognition". In: *Image Processing, IEEE Transactions on* 18.8 (2009), pp. 1885–1896.

[Sut05] Herb Sutter. "The Free Lunch Is Over - A Fundamental Turn Toward Concurrency in Software". In: *Dr. Dobb's Journal* (03/2005).

[SW11] Robert Sedgewick and Kevin Wayne. *Algorithms.* 4th ed. Addison-Wesley Longman, Amsterdam, 2011.

[Tal13] Vinay Tallapalli. *Mobile Processors History Week 4: All things about Apple Processors.* 05/05/2013. URL: `http://www.androidhighlights.com/postdetails.aspx?postid=1025&link=Mobile-Processors-History-Week-4-:-All-things-about-Apple-Processors` (visited on 10/25/2013).

[Van02] Ruud Van Der Pas. "Memory hierarchy in cache-based systems". In: *Sun Blueprints* (2002), p. 26.

[Voi13] Alexandru Voica. *GPU compute on PowerVR with Android's Filterscript API.* 04/22/2013. URL: `http://withimagination.imgtec.com/index.php/powervr/gpu-compute-on-powervr-with-androids-filterscript`.

[Wan+13] Guohui Wang et al. "Accelerating computer vision algorithms using OpenCL framework on the mobile GPU - A case study". In: *Proc. of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP).* 2013, pp. 2629–2634.

[Wil11] James Willcox. *using direct textures on android.* 12/16/2011. URL: `http://snorp.net/2011/12/16/android-direct-texture.html` (visited on 01/15/2014).

[xaf13] xaffeine. *How can I make Android Native Camera go faster?* 03/04/2013. URL: `http://answers.opencv.org/question/8482/how-can-i-make-android-native-camera-go-faster/` (visited on 01/30/2014).

[Yan13] Linus Yang. *OpenCL Simple Test.* 09/25/2013. URL: `https://github.com/linusyang/opencl-test-ios` (visited on 11/09/2013).

[Zwa10] David Zwarg. *GPUs and Parallel Computing Architectures.* 06/29/2010. URL: `http://www.azavea.com/blogs/labs/2010/06/parallel-computing-architectures/` (visited on 10/07/2013).

List of Figures

List of Tables

Listings

Glossary

ALU arithmetic/logic unit. 6, 46

API application programming interface. 2, 13, 18, 20–23, 26, 27, 37, 41, 43, 47, 51, 54, 55, 57, 58, 61, 62, 70, 72, 75, 86–88, 92, 93

APU accelerated processing unit. 88

AR Augmented Reality. 71–73, 75, 86, 92

CPU central processing unit. 2, 6, 7, 12, 13, 16–19, 21, 27–30, 39, 40, 42, 46, 49–51, 53, 55, 56, 58, 60, 61, 65, 68, 71, 75, 78, 79, 81, 83–85, 87, 88, 91, 93

CV computer vision. 30

DSP digital signal processing. 1–3, 16, 18, 30, 31, 87

EP Embedded Profile. 20, 27, 93

ES Embedded Systems. 23

FBO framebuffer object. 23, 25, 26, 61, 62, 65, 67, 79, 81–84, 105

FFT Fast Fourier Transform. 29

fps frames per second. 40, 76, 77, 85

GLSL OpenGL Shading Language. 23, 24, 30, 65

GPGPU general purpose computing on graphics processing units. 1–3, 12, 16, 18, 23, 25–31, 38, 39, 57, 60, 65, 69–71, 78, 79, 86–88, 93, 102, 104

GPU graphics processing unit. 1–3, 6–8, 13, 15–18, 21–23, 25–30, 34, 36, 38, 39, 43–48, 50, 53–63, 65, 68, 70–72, 75, 78, 79, 81, 83–88, 92, 93, 102

HSA heterogenous systems architecture. 88

IPC inter-process communication. 12

JNI Java Native Interface. 42, 54, 91

LBG Local Binary Pattern. 28, 93

MIMD Multiple Instruction, Multiple Data. 6

MPI Message Passing Interface. 12

NDK native development kit. 21, 51, 54, 57, 61, 62, 75, 91, 92

OS operating system. 2, 13, 22, 38, 61, 69, 70, 91

PBO pixelbuffer object. 61

PC parallel coordinates. 34, 67, 78

PE processing element. 19

ROI region of interest. 84

RS RenderScript. 21–23, 26, 30, 55, 56, 70, 87, 88

SDK software development kit. 21, 54, 55, 57, 61, 62

SIFT scale-invariant feature transform. 27, 29, 30

SIMD Single Instruction, Multiple Data. 5, 6, 19, 30, 40, 42, 56

SIMT Single Instruction, Multiple Threads. 6

SISD Single Instruction, Single Data. 6

SM streaming multiprocessor. 7

SoC system on a chip. 16, 17, 70, 83

SP streaming processor. 7

SURF Speeded Up Robust Features. 29, 30

TBB Intel Threading Building Blocks. 12, 30

TI Texas Instruments. 28

VM virtual machine. 22

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Ort, Datum *Markus Konrad*