UNIVERSITÄT KOBLENZ · LANDAU
Fachbereich 4: Informatik

Institute for Computer Graphics and Vision

Extraction of Natural Feature Descriptors on Mobile GPUs

Diplomarbeit

zur Erlangung des Grades eines Diplom-Informatikers
im Studiengang Computervisualistik

vorgelegt von

Robert Hofmann

Erstgutachter:	Prof. Dr.-Ing. Stefan Müller (Institut für Computervisualistik, AG Computergraphik)
Zweitgutachter:	Dipl.-Ing. PhD Hartmut Seichter (ICG, Technische Universität Graz, Österreich)

Koblenz, im Juni 2012

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☐	☐

. . . . . .
(Ort, Datum) (Unterschrift)

Institute for Computational Visualistics
Computer Graphics Center
Prof. Dr. Stefan Müller
Postfach 20 16 02
56 016 Koblenz, Germany
phone.: 0049-261-287-2727
fax: 0049-261-287-2735
email: stefanm@uni-koblenz.de

UNIVERSITÄT KOBLENZ · LANDAU

Department 4: Computer Science

Thesis Research Plan (Aufgabenstellung Diplomarbeit)

Robert Hofmann
(Student ID: 204210214)

Title: **Extraction of Natural Feature Descriptors on Mobile GPUs**

Since the advent of smartphones and tablet computers, augmented reality (AR) became increasingly popular. The comprehensive hardware of those mobile devices enable handheld AR researchers and developers to advance AR from desktops to large-scale indoor and outdoor environments. Prerequisites for such mobile AR applications are markerless tracking methods using natural features already present in the scene to estimate the mobile device's pose. But due to the hardware and software restrictions of today's smartphones, natural feature tracking (NFT) for mobile AR is still a heavy-duty task that requires thorough optimization.

Fortunately smartphones have become increasingly powerful and are now even equipped with multi-core CPUs and GPUs. Additionally, with the introduction of programmable shaders to OpenGL ES 2.0 (the graphics API for embedded systems) general purpose computations on GPUs (GPGPU) became practical for mobile devices. Although mobile GPUs are by far not as powerful as their desktop counterparts, it still seems beneficial to investigate how the potential of mobile GPUs can be exploited for NFT on smartphones.

The goal of this thesis is to investigate how the expensive extraction of local descriptors for natural features can be optimized with a GPGPU approach. A feature descriptor suitable for GPU-accelerated NFT is chosen and a prototype to extract such descriptors from an image through GPGPU is implemented. The prototype is tested and evaluated on different mobile devices and compared to a CPU implementation in terms of runtime and matching performance.

Main focus:

1. Research different feature descriptors and choose one suitable for NFT and GPGPU
2. Familiarize with GPGPU through OpenGL ES 2.0
3. Design and implementation of a prototype to extract feature descriptors on a mobile GPU
4. Evaluation of the runtime of the implementation and the robustness of the extracted descriptors
5. Documentation and presentation of the results

The thesis is accomplished in cooperation with the Christian Doppler Laboratory for Handheld Augmented Reality, Institute for Computer Graphics and Vision, Graz University of Technology.

Thesis advisors: Prof. Dr. Stefan Müller
Hartmut Seichter, PhD, Dipl.-Ing.

Koblenz, 03/13/2012 – Prof. Dr. Stefan Müller –

Acknowledgments

The author of this thesis would like to thank Hartmut Seichter for his constant support during writing this thesis as well as during creating and evaluating the prototype program, Gerhard Reitmayr for his valuable input and ideas, and the ICG at the Graz University of Technology for the friendly welcome. Research in this thesis has been conducted as part of the Christian Doppler Laboratory for Handheld Augmented Reality.

Zusammenfassung

In dieser Arbeit wird der Nutzen von GPGPU (Allzweckberechnungen auf Grafikprozessoren) zur robusten Deskription von natürlichen, markanten Bildmerkmalen mit Hilfe der Grafikprozessoren mobiler Geräte bewertet. Dazu wurde der SURF-Deskriptor [4] mit OpenGL ES 2.0/GLSL ES 1.0 implementiert und dessen Performanz auf verschiedenen mobilen Geräten ausgiebig evaluiert. Diese Implementation ist um ein Vielfaches schneller als eine vergleichbare CPU-Variante auf dem gleichen Gerät. Die Ergebnisse belegen die Tauglichkeit moderner, mobiler Grafikbeschleuniger für GPGPU-Aufgaben, besonders für die Erkennungsphase von NFT-Systemen (Tracking mit natürlichen, markanten Bildmerkmalen), die in Augmented-Reality-Anwendungen genutzt werden. Die nötigen Anpassungen am Algorithmus des SURF-Deskriptors, um diesen effizient auf mobilen GPUs nutzen zu können, werden dargelegt. Weiterhin wird ein Ausblick auf ein GPGPU-gestütztes Tracking-Verfahren gegeben.

Abstract

In this thesis the feasibility of a GPGPU (general-purpose computing on graphics processing units) approach to natural feature description on mobile phone GPUs is assessed. To this end, the SURF descriptor [4] has been implemented with OpenGL ES 2.0/GLSL ES 1.0 and evaluated across different mobile devices. The implementation is multiple times faster than a comparable CPU variant on the same device. The results proof the feasibility of modern mobile graphics accelerators for GPGPU tasks especially for the detection phase in natural feature tracking used in augmented reality applications. Extensive analysis and benchmarking of this approach in comparison to state of the art methods have been undertaken. Insights into the modifications necessary to adapt and modify the SURF algorithm to the limitations of a mobile GPU are presented. Further, an outlook for a GPGPU-based tracking pipeline on a mobile device is provided.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

Since smartphones and the more recent tablet computers hit the mass-market, augmented reality (AR) became increasingly popular. Their inexpensive price and comprehensive hardware make them the favorable platform to handheld AR researchers and developers. Because such devices usually feature a back side video camera and a relatively large display on the front, they correspond very well to the intuitive "Magic Lens" metaphor [5]. In this context the device acts as a "window" into the virtually augmented real world. By tracking the device's pose (i.e. its position and orientation) relative to the world, virtual objects can be rendered perspectively correct and superimposed on the video camera stream, which is then finally displayed on the device's screen. The correct alignment of those virtual objects with the view of the real world (registration) is the key to convincing and immersive AR applications.

1.1 Problem Statement

In order to achieve a correct registration of the virtual augmentations, accurate tracking techniques are essential. Since smartphones are usually equipped with a video camera, it is reasonable to utilize it for that purpose. Past research on this so called visual tracking has spawned several tracking libraries including the popular ARToolKit [22], ARTag [13], ARToolKitPlus [46] and the Studierstube Tracker [45].

Those libraries track the camera pose by searching for fiducial landmarks with known size and shape (markers) in the camera images. As a result, AR applications are restricted to the environment where such markers have been placed, which makes the AR device rather portable than mobile. However, the goal is to drive handheld AR towards mobility in large-scale outdoor environments.

Prerequisites for truly mobile AR are *markerless* visual tracking methods, which do tracking from distinctive features already present in the scene. Since such natural features are seldom as distinctive as markers, their detection is computationally more intensive. Furthermore, the local area around a feature needs to be described in a way that the feature can be matched against other features detected in a different camera frame. Extracting such a feature descriptor from the image data also is computationally expensive when robustness against image transformations and illumination changes is required. This so called natural feature tracking (NFT) therefore poses a challenge especially to the restricted computational capabilities of smartphones.

1.2 Motivation

Despite its complexity and computational costs, it has been shown that NFT is feasible on smartphones as long as the algorithms are carefully tailored to the limitations of the device and the requirements of the AR application [26, 44].

The computational power of a mobile device is in large part restricted by its battery capacity. Yet, smartphones have become increasingly faster: not only their clock rates grew, but also more complex and dedicated processing units were introduced. The Nvidia Tegra 3, for example, features a quad-core CPU, a twelve-core GPU and a companion core for low-power modes [12]. Also programmable DSPs (digital signal processors) are featured on many platforms to process multimedia data. In order to spread the load of an AR application it seems therefore advantageous to not only utilize the available cores of a mobile CPU, but also the additional processing units. In particular, mobile GPUs are becoming more and more powerful in order to support the ever larger screen resolutions and demanding visual appearance of mobile operating systems.

Bringing GPUs into use for problems outside the scope of computer graphics has provided a leap of computation power on the desktop. Since the standard graphics pipelines of OpenGL and Direct3D are fully programmable through so called shader programs, those APIs reached a level of flexibility that made it possible to harness the specialized hardware of GPUs for more general computations beyond rendering 3D computer graphics. Such general-purpose computing on GPUs (GPGPU) can benefit from the large number of processing units available on a GPU. As long as the applied algorithm is data-parallel, i.e. there are no dependencies between individual data elements and all of them are transformed by the same operation, it is possible for the GPU to process as many data elements in parallel as there are processing units on the GPU. Mobile GPUs undergo the same restrictions in terms of power efficiency as mobile CPUs. Consequently, they are more restricted than their desktop counterparts. These restrictions are reflected in computer graphics APIs such as OpenGL ES (OpenGL for embedded systems), which is a subset of OpenGL with a few additions to alleviate the limitations of embedded systems. Since programmable shaders have been introduced in OpenGL ES 2.0, GPGPU became practical on many mobile devices supporting this API.

It seems beneficial to investigate how the potential of mobile GPUs can be exploited for NFT on smartphones. Especially considering that the majority of operations required to describe trackable features in an image are highly data-parallel pixel operations. However, due to the limitations of mobile GPUs, research on GPGPU for mobile applications has not gained as much attention as for desktop GPUs yet. This is particularly true for mobile application scenarios such as AR requiring interactive or real-time

response speeds. With this work we therefore intent to contribute to the research of GPGPU in the context of mobile NFT.

1.3 Contributions

Specifically, we want to evaluate whether it is feasible to off-load the expensive generation of feature descriptors from the mobile CPU onto the mobile GPU of a smartphone. Therefore, we are addressing the following research questions in this thesis:

- What feature descriptors are suited for mobile GPGPU?
- How efficiently can feature descriptors be extracted from an image by means of mobile GPGPU?
- Can CPU-based mobile NFT applications benefit from GPU-accelerated feature descriptor extraction?
- How does a GPU-accelerated feature descriptor extraction perform across different mobile devices?
- How does the reduced floating-point precision of mobile GPUs affect the descriptor quality in terms of matching performance?

Mobile NFT systems often rely on the extraction of feature descriptors for tracking initialization and recovery in the event of tracking failures. Applying such tracking-by-detection methods as a general means is usually not feasible on mobile devices and therefore restricted to special cases. Improving the efficiency of the descriptor extraction would allow more robust feature descriptions and more tracking stability. Furthermore, utilizing the mobile GPU disburdens the CPU of the mobile device, which frees resources for other AR related computations[1]. GPGPU programs are also very scalable and can easily profit from future mobile GPUs featuring more shader units without further modifications.

Because smartphones are wide-spread and relatively inexpensive, mobile AR experienced a significant popularization. The research presented in this thesis is therefore relevant for a wide range of mobile AR applications. Note that the proposed GPGPU optimizations are also useful for other mobile applications that rely on feature extraction and matching (e.g. panorama stitching and object recognition).

[1] One still has to take into account that the mobile GPU may be occupied with rendering the virtual augmentations. Whether a mobile AR application actually can benefit from GPGPU therefore depends on its general GPU load as well.

1.4 Methodology

In order to assess the feasibility of our mobile GPGPU approach to feature descriptor extraction, we proceeded according to the following plan of action:

1. Evaluate several state of the art feature descriptors and choose one suited for mobile GPGPU.
2. Implement a method to extract such descriptors from images on mobile GPUs.
3. Compare our implementation with a reference CPU implementation of the descriptor in terms of runtime speed and matching performance.
4. Compare our implementation's performance across different mobile devices.

1.5 Outline of the Thesis

Section 1 of this thesis briefly introduced the challenges of NFT on mobile devices and why we consider them to be remediable by means of mobile GPGPU. Section 2 reviews previous work related to NFT on mobile phones, GPGPU with mobile GPUs, as well as CPU- and GPU-based approaches to natural feature description. Section 3 first introduces deeper background knowledge of the algorithms and tools we used and then proceeds with explicating our design considerations as well as presenting our design itself. Section 4 explains our final implementation in detail. Our experimental results along with a discussion of those are presented in section 5. Section 6 summarizes the results based on our experiments and provides an outlook towards the application of our implementation in an NFT pipeline.

2 Related Work

2.1 Natural Feature Tracking on Mobile Phones

To this time only a few NFT systems accomplish interactive or faster frame rates for AR applications on mobile phones.

Wagner et al. [44] combined active search and tracking-by-detection to achieve both real-time performance and robustness against tracking failures. Given a coarse pose of the mobile phone, their system efficiently estimates the pose change between successive frames by cross-correlating image patches with known features, which have been affinely warped and back-projected into the current image frame. For initial pose estimation

and reinitialization in case the tracking fails, the system switches to a much slower, yet more robust tracking-by-detection using a modified version of SIFT [29].

In [33] Oberhofer et al. adapted the approach of combining two tracking techniques with orthogonal characteristics, but replaced SIFT with the more lightweight BRIEF descriptor [8] for more efficient descriptor extraction and matching. Their NFT system is solely based on web technologies like JavaScript, HTML5 and WebGL. It runs with any HTML5 compatible browser and achieves real-time performance on desktop PCs and interactive frame rates on the Samsung Galaxy S II smartphone.

Herling et al. [17] employ a two-phase approach in their NFT system as well. It also relies on invariant descriptors of point features (in this case a heavily modified version of SURF [4]) for initial pose estimation of the mobile phone. But instead of switching to active search with normalized cross-correlation (NCC) afterwards, their system matches un-oriented SURF features between the local areas around predicted feature positions and a feature map. The authors have shown that they were able to detect, describe and match SURF-36 features in real-time but gave no further evaluation of their tracking system.

Lima et al. [28] use lines instead of point features to track three-dimensional targets on the Windows Mobile Pocket PC platform. Depending on the target complexity, their system runs at interactive frame rates or close to real-time. To detect the visible edges of the tracking target, the graphics hardware is utilized through OpenGL ES 1.0 by rendering a hypothetical pose of the tracking target's wireframe model.

A different approach took Klein et al. [26] with their NFT system coined PTAM (parallel tracking and mapping). Instead of tracking a designated target, their system builds a dynamic map of arbitrary visual features already present in the environment and searches for them in subsequent camera frames for pose estimation. This concept is based on simultaneous localization and mapping (SLAM), which is common in autonomous robots research. Although PTAM was originally designed for non-handheld setups, the authors have also modified it to work on the Apple iPhone 3G [27].

Generally speaking, current NFT systems track the camera pose in two different ways: extracting and matching either dense low-quality features (e.g. corners as in PTAM [26]) or sparse high-quality features (e.g. SIFT features as in [14]). The main difference being that the former approach actively addresses the invariance problem (i.e. robustness against image transformations, illumination change and perspective distortion) efficiently during correspondence search, whereas the latter achieves better robustness by extracting invariant feature descriptions in the first place. Although such robust descriptors are computationally much more expensive, they also exhibit more data-parallelism, since correspondence search becomes decoupled from system state. Wagner et al. [44] exploited the orthogonality

of both approaches and fused them into a flexible two-lane NFT pipeline, capable of switching between either of them depending on tracking stability. However, with the advent of increasingly powerful mobile graphics hardware, it is to investigate how GPGPU can alleviate the shortcomings of tracking-by-detection and invariant feature descriptors, by exploiting their inherent parallelism.

2.2 General-Purpose Computing on GPUs

When programmable shaders were introduced to the graphics libraries Direct3D and OpenGL, their fixed-function pipelines to render three-dimensional geometry became flexible and allowed more complex visual effects. This flexibility of GPU programming awoke the interest of researchers in diverting those high-performance many-core processors from their intended use and employ them to solve problems beyond the scope of computer graphics but with similar data-parallel characteristics.

However, GPGPU programming through shaders is complicated [16]: algorithms have to be cast in terms of a graphics API, which sometimes enforces severe restrictions on the data format and the computational accuracy of the operation. To make matters worse, the underlying GPU architecture was often not fully exposed or documented. Nevertheless, by carefully remodeling a problem to fit the parallelization concepts of GPUs, it is possible to achieve large speed-ups over a sequential CPU implementation.

With the release of Nvidia CUDA (compute unified device architecture) and ATI Stream (formerly Close to Metal), GPGPU took a huge step forward. GPUs were now programmable with more general APIs that disburdened GPGPU developers from struggling with the peculiarities of graphics libraries. However, each of those two GPGPU APIs is tied to graphics hardware of the same vendor, thus making cross-platform development cumbersome.

To provide more abstraction to parallel computing, the Khronos Group later released the open standard OpenCL (open computing language). This framework is designed for parallel programming on heterogeneous platforms consisting of multi-core CPUs, GPUs and other processors like DSPs. Consequently, OpenCL abstracts vendor-specific hardware details, which means that an OpenCL program is executable on every hardware that implements the OpenCL specification.

2.2.1 General-Purpose Computing on Mobile GPUs

Although advanced GPGPU APIs like OpenCL and CUDA are available on many hardware devices for desktop PCs and scientific computing systems,

they are still lacking on mobile devices[2]. In order to ease adherence to the OpenCL standard on embedded platforms, its specification also features an "embedded profile" which relaxes the OpenCL compliance requirements for handheld and embedded devices. Similar to the OpenCL "full profile" specification, to date there are no end-user mobile devices exposing this feature.

This means that GPGPU on mobile devices is currently only possible through OpenGL ES. Since OpenGL ES 1.0 and 1.1 only support a fixed-function pipeline, OpenGL ES 2.0 is the only reasonable choice for mobile GPGPU. Fortunately, most newer devices support the 2.0 version, so we have a wide-spread API at hand to develop for a wide range of embedded systems (e.g. smartphones, tablets and development boards like the PandaBoard).

Recent research work has shown that GPGPU can accelerate image processing even on mobile devices. As a result of their limited potential for parallelization compared to desktop GPUs, the most noticeable increase in performance is achieved for pixel-wise operations with low complexity (i.e. low shader instruction count and rare dynamic branching). Fortunately, GPU implementations are highly scalable, so current systems will likely profit from future mobile GPUs with more shader processors, increased clock rates, and higher memory bandwidth.

A few attempts have been made to implement feature descriptors with OpenGL ES 2.0. For reasons of coherence we will address those works in section 2.3.1 after the feature descriptors in question have been introduced.

López et al. [6] employed NFT-related techniques, such as feature extraction, matching, and camera motion estimation, to implement a panorama builder for mobile phones. They applied GPGPU methods to construct the final image panorama in a post-processing step. Even though the used OpenGL ES 1.1 API is restricted to the fixed-function pipeline, which only allows general purpose computations through texture combiners, the authors achieved noticeable speed-ups for some parts of their system. For the most parts, using GPGPU was not beneficial, since the overheads introduced by the necessary data uploads and read-backs swallowed much of the performance gain.

Using the OpenGL ES 2.0 API, several image processing algorithms, including the Harris corner detector, have been ported to the GPU of a mobile device in [41]. Depending on the input image size, the achieved performance ranges from interactive to real-time. The authors did not provide a comparison to implementations running on the mobile device's CPU.

In [7] LBP (local binary pattern) feature extraction for face tracking has

[2]Even though some manufacturers of mobile GPUs already implemented OpenCL capabilities for their products (e.g. as on the PowerVR SGX535 by Imagination), their customers so far chose not to expose this feature to the end-user. This is probably due to reasons of energy efficiency.

been implemented with GPGPU through OpenGL ES 2.0. Although their GPU implementation is more energy efficient than an equivalent CPU implementation, it's runtime performance suffered from the GPU's restrictions not allowing efficient binary arithmetic. Nevertheless, the authors were able to improve their system's performance by concurrently processing two successive frames with the mobile CPU and GPU.

Wang et al. [47] implemented a face recognition system for Android smartphones. Exploiting the graphics hardware to extract Gabor-based face features, they could speed up their application's response time by a factor of four.

2.3 Image Features

The performance of a NFT system strongly depends on the method used to detect trackable image features and the way those are described for robust matching. To this day, numerous feature detectors and descriptors have been proposed. Many of them largely differing in their invariance against image transformations (translation, rotation, and scale), viewpoint change, illumination variation, blur, noise, and image compression artifacts. Especially in the context of NFT on mobile devices, the computational cost of detecting, describing, and matching features is another important factor that must be taken into account when choosing the right method for feature extraction.

Edges or, more generally, lines are a class of image features with interesting properties (such as structural abstraction of the scene, resilience against motion blur and suitability for three-dimensional tracking targets), but also with significant drawbacks that make them impractical in our context. Efficient line tracking for AR (e.g. as in [49] and [28]) requires at least a wireframe model of the tracking target. Such a model, however, is considerably harder to acquire than an image of the target's surface. Additionally, lines are not distinctive in a local sense, which makes one-to-one comparisons against match candidates unfeasible. Hence, matching is performed globally by taking the geometric relations of the detected lines to one another into account. From the perspective of GPGPU, features are preferred to be fine-grained with no dependencies between them. Line features do not match that preference: they are an abstract representation and introduce dependencies between the elements they are composed of (the points on the line) as well as between each other during matching (the aforementioned geometric relations). Point-like features, such as corners and blobs (image regions with a center point), are more suitable for GPGPU, as they are restricted to a local area around their position in the image and are independent of one another.

Extracting positions of point-like features from an image is done with a feature detector. In the simplest case such a detector filters the image with

a certain operator, deciding for each image position whether it is "interesting" or not. More advanced detectors also assign a measure of "interestingness" (detector response strength) to the feature position and operate on an image pyramid (i.e. multiple, differently sized versions of the input image) to capture features of different size (scale). Such point-like features are often referred to as keypoints.

Once all features have been detected, they are usually collected in a list-like structure. Although this is naturally easy in a CPU program, it is a significantly more challenging task to do on the GPU. Given the output of a GPGPU feature detector as a sparse matrix in video memory, [11] and [52] proposed two different ways to generate a point list from it on the GPU: Cornelis et al. using CUDA and Ziegler et al. using a multi-pass technique with programmable shaders. However, both approaches cannot be used on a mobile GPU, because there is so far no support for high-level GPU APIs like CUDA and multi-pass operations that require a very large number texture fetches are subject to significant slow-downs on mobile GPUs. The remaining options are processing the detector output on the CPU or doing feature detection on the CPU in the first place. Because the first option introduces the bottleneck of downloading the detector output from video into system memory, we therefore decided to use a CPU feature detector.

Although many different feature detectors have been proposed so far, we will only briefly introduce the FAST detector here for further understanding, because of its interesting properties for mobile NFT and the focus of this work on feature descriptor extraction.

FAST (features from accelerated segment test) [36] is a high-speed corner detector based on simple intensity comparisons between a center pixel and the neighbor pixels in a Bresenham circle of given radius around it. Later, those comparisons were optimized using machine learning [37, 35]. FAST is currently the fastest corner detector available. Originally it does not support scale invariance, but it has been shown that FAST can efficiently be applied to an image pyramid for scale-invariant features (e.g. in [26], [44], and [48]). Furthermore, FAST has also been extended to calculate reproducible feature orientations in [38].

Given a list of feature positions (ideally accompanied by a scale factor for each feature), one needs to compare this set of features to a set of different features (e.g. from another camera frame or a feature map of the tracking target), to find feature correspondences between images. This matching is done by analyzing the local area around the feature positions in question. One can then either directly compare intensity values of local image patches or try to determine matches between more abstract representations of those patches (so called descriptors). The former is usually the faster method, while the latter approach allows more robust matching (e.g.

in terms of invariance against different lighting conditions or viewpoint change) at the cost of computing such a feature descriptor beforehand.

Below we will give a brief overview of feature descriptors that have either been published recently or already been applied in the context of mobile NFT.

SIFT (scale-invariant feature transform) [29] combines a feature detector and descriptor, designed to be invariant to image transformations and partially invariant to changes in perspective and illumination. To detect features, it searches for local extrema in a difference of Gaussians (DoG) scale-space pyramid. For each such feature position a main orientation is calculated and a gradient-based descriptor is extracted from a local patch around the feature. Despite its high computational cost is SIFT very popular in computer vision for its excellent matching performance. When applied for mobile NFT, SIFT has to undergo major modifications to become computationally feasible (e.g. as in [44]).

The fact that SIFT relies on gradient histograms, complicates an efficient GPGPU implementation with OpenGL ES 2.0. Implementing a histogram on a mobile GPU requires branching to sort the samples into the correct bins, which is inherently disadvantageous for GPU parallelism. Other GPGPU methods to generate histograms require a high number of render passes with numerous texture fetches or atomic write operations to prevent multiple threads from incrementing a histogram bin at the same time. But the former is not feasible and the latter not available on mobile GPUs yet. In [39] race conditions between GPU threads are circumvented by scattering point primitives and blending them to accumulate the histogram values. But floating-point textures are required to avoid over-saturating histogram bins due to lack of precision.

SURF (speeded up robust features) [4] tackles SIFT's deficiencies by extensively using integral images [43] to avoid the costly construction of a scale-space pyramid during a preprocessing step. The integral image enables SURF to apply box filters of arbitrary scale to the input image in almost constant time, rather than repeatedly down-sampling and Gaussian filtering it altogether. Furthermore, the descriptor is based on gradient sums instead of gradient histograms like in SIFT, which effectively makes it smaller in size, faster to extract, and less sensitive to noise. Studies have shown that SURF features are indeed much faster to compute, while preserving a matching performance comparable to SIFT [3]. Even so, SURF is not fast enough for real-time NFT applications, not to mention on mobile devices, and therefore had to be approximated to be feasible for mobile NFT in [17].

SURF's gradient sums are a better match for GPU architecture than

SIFT's gradient histograms are. Although they also require a costly gathering operation (i.e. one fragment fetches and accumulates a relatively large number of samples) in a mobile GPGPU implementation, they do not depend on branching and can alternatively be approximated with mipmaps (e.g. as in [11]). The construction of an integral image, however, is an expensive multi-pass operation for mobile GPUs. But as in [11], this step can be avoided by constructing the scale-space pyramid through hardware-accelerated mipmapping.

BRIEF (binary robust independent elementary features) [8] is a binary string feature descriptor based on intensity difference tests of randomly distributed point pairs around a feature position. To form 256 of such pairs proved to be a good compromise between computational effort and matching performance, thus yielding a 256 bit descriptor vector. Those descriptors can be matched efficiently with the Hamming distance as a similarity measure. The authors showed that BRIEF performs comparable to SURF in terms of recognition rate, while being significantly faster to extract and match. However, BRIEF does not offer any invariance against scale or rotation.

Two things appear worth mentioning about BRIEF in the context of mobile GPGPU. First, its dependence on randomly generated numbers hampers efficient GPGPU parallelization, since in OpenGL ES 2.0 random number functions are non-standard and may only be exposed through extensions. The common workaround would be to sample from Gaussian noise textures generated by the CPU and copied into video memory. However, this requires an additional preprocessing step for each frame. Second, the BRIEF descriptor heavily relies on binary arithmetic for fast matching, but since mobile GPUs usually do not support this feature, that advantage would be lost in a GPU implementation. Although a GPGPU parallelization of BRIEF may still be beneficial, it does not seem as advantageous as with other, more robust feature descriptors judging from its GPU-unfriendly nature.

ORB (oriented FAST and rotated BRIEF) [38] is combination of the FAST detector and the BRIEF descriptor. The FAST detector has been extended to calculate a measure for response strength and a reproducible orientation for each feature positions. The orientation is determined as the vector between the feature's center and the intensity centroid of a circular patch around it. Further has the BRIEF descriptor been modified, in that it does not rely on randomly generated tests, since they become correlated when distributed along feature orientation and therefor reduce descriptiveness of the feature vector. Instead the authors use a set of uncorrelated tests determined by a machine learning algorithm. It has been shown that ORB features are

significantly faster to compute than SIFT or SURF features, but are not as robust as those two prominent descriptors [25].

Like BRIEF, does the ORB descriptor trades robustness for very fast matching based on binary arithmetic. But, as mentioned above, a GPGPU implementation of ORB with OpenGL ES 2.0 would not benefit from this.

OSID (ordinal spatial intensity distribution) [9] is descriptor designed to be invariant to more complex illumination changes than the usually assumed intensity shift or affine brightness changes. It is based on the idea to use relative orderings of pixel intensities instead of their absolute values, since this ordering remains unchanged between local patches of corresponding image locations as long as the brightness change function is monotonically increasing. For descriptiveness also the spatial information of the local patches are captured by subdividing them into pie segments. Both clusterings form a two-dimensional histogram, which is rasterized into a 128 byte descriptor vector. The authors have shown that OSID outperforms other descriptors under presence of nonlinear brightness change functions like square or square root and applied it for automatic color correction and rectification of uncalibrated stereo image pairs.

OSID's two-dimensional histogram and the necessity of sorting intensities pose two particularly hard challenges for mobile GPGPU. An implementation with OpenGL ES 2.0 therefore seems not feasible, when runtime performance is as important as in the context of NFT.

Normalized cross-correlation (NCC) is a distance measure used to compare intensity values between an image and a search template directly. For this purpose the intensities around a feature position constitute a very simple descriptor vector of that feature. Invariance against changes in illumination is achieved by normalizing the descriptor to unit length. However, correlating such a descriptor with a whole image results in a high computational complexity. Therefore, the search region has to be narrowed down by predicting correspondence locations. Sub-sampling the template and the search window to reduce the dimensionality of the correlation can further decrease the complexity of the comparison. Also an integral image can be used to speed-up the calculation of the denominator for normalization.

Under certain circumstances NCC can be a very efficient tool to determine correspondences between frames. For example, in [44] NCC is used when the mobile device's pose has been estimated with tracking-by-detection and only small pose changes between frames are assumed. A corresponding feature is then predicted to be located in a small image region, which reduces the complexity of NCC significantly. But because NCC is by definition not invariant to scale and perspective distortions, these issues have to be actively addressed with additional measures like applying

it on image pyramids and affinely warping the search template. This basically means that the generation of invariant feature descriptors and feature matching are merged together in an inseparable way, which introduces dependencies that complicate a GPGPU implementation. In our context, it is therefore more reasonable to rely on descriptors addressing the invariance problem during a separate descriptor generation step prior to matching, in order to increase the parallelism of the task.

SIFT and SURF are the de facto standards for invariant feature detection and description, with SURF being an efficient approximation of SIFT. Their unmatched robustness comes at the price of a high computational complexity. Despite from being faster to compute as SIFT, the SURF descriptor also has the advantage of being a better fit for GPU architecture: instead of being histogram-based, SURF accumulates feature characteristics with sums, which are easier to implement especially on the restrained GPUs of mobile devices. From our survey on feature descriptors we therefore found the SURF descriptor to have the most favorable properties in terms of robustness and GPGPU compatibility. SIFT is also an interesting candidate, but is ranked behind SURF due to the higher computational complexity of its descriptor extraction part.

2.3.1 Implementations of Feature Descriptors

In this section we briefly review notable CPU and GPU implementations of SURF and SIFT for both desktop PCs and mobile devices.

Implementations for Desktop PCs

- The original SURF[3] and SIFT[4] libraries are closed-source and cannot be used for in-depth comparison to our implementation.
- The OpenCV library[5] includes implementations of SIFT and SURF, is open-source, and pre-compiled libraries are also available for Android and ARM-based systems.
- OpenSURF[6] is one of the earliest open-source implementations of SURF and widely used.
- Pan-o-matic[7] is an open-source software to automatize the generation of panorama images. It includes a SURF implementation that yields results which are identical to the original SURF library [15].

[3] `http://www.vision.ee.ethz.ch/~surf/download.html`
[4] `http://www.cs.ubc.ca/~lowe/keypoints/`
[5] `http://opencv.willowgarage.com/wiki/`
[6] `http://www.chrisevansdev.com/computer-vision-opensurf.html`
[7] `http://aorlinsk2.free.fr/panomatic/`

- Parallel SURF[8] is based on Pan-o-matic, but uses OpenMP to parallelize SURF and spread its workload among the available CPU cores.
- SIFT++[9] is an open-source C++ implementation of SIFT.

Due to the high computational cost of SURF and in particular SIFT, many works accelerated both algorithms by means of GPGPU with traditional shaders, high-level GPGPU APIs or a mixture of both.

- In [11] a GPU implementation of SURF using the Cg shading language and CUDA is proposed. In order to take full advantage of the graphics hardware the authors made heavy use of mipmaps and other methods closely related to graphics processing. For instance, their implementation constructs a scale-space pyramid of the input image with a mipmap instead of an integral image, which is a natural fit for graphics hardware. The implementation extracts SURF feature positions and descriptors in real-time and is available for download[10] but closed-source.
- Terriberry et al. [42] implemented SURF using only OpenGL and Cg. Their implementation features a heavily optimized 2D-parallel prefix-sum algorithm to compute the integral image. To generate a list of the detected features the authors used Ziegler et al.'s HistoPyramids [52]. Both algorithms require a very large number of render passes to complete, but due to their high level of parallelization they are still very efficient on desktop GPUs.
- Several works implemented SURF using APIs like CUDA (CUDA SURF[11], OpenCV also features a CUDA implementation of SURF) and OpenCL (clsurf[12], OpenSurfCL[13]). Because such high-level GPGPU APIs are not available on mobile devices yet, those works are mentioned here only for completeness and will not be evaluated further.
- Heymann et al. [18] implemented SIFT for GPUs with programmable shaders. To maximize parallelization and to utilize modern GPUs' SIMD[14] architecture the authors remodeled many parts of the algorithm, carefully restructuring the input data of computations to fit into OpenGL's texture formats.

[8] `http://sourceforge.net/apps/mediawiki/parallelsurf/`
[9] `http://www.vlfeat.org/~vedaldi/code/siftpp.html`
[10] `http://homes.esat.kuleuven.be/~ncorneli/gpusurf/`
[11] `http://www.d2.mpi-inf.mpg.de/surf`
[12] `http://code.google.com/p/clsurf/`
[13] `http://sourceforge.net/projects/opensurfcl/`
[14] Single instruction, multiple data—a parallelization concept often referring to data-parallel vector computations. See section 3.1.2 for a more detailed explanation.

- SiftGPU[15] is a project offering two GPU implementations of SIFT: one using OpenGL's GLSL shading language and the other using CUDA. The GLSL version uses an efficient GPU/CPU mixed method based on the HistoPyramids algorithm to build a compact list of feature positions.

None of those GPGPU implementations can be used in our case. As already mentioned are high-level GPGPU APIs like CUDA and OpenCL currently not available for mobile GPUs, which renders such implementations unusable in our context. The implementations based on programmable shaders often use OpenGL features not supported by mobile GPUs or are written in a shading language incompatible to GLSL ES.

Implementations for Mobile Devices

- Chen et al. [10] optimized the original SURF algorithm and achieved an increase in performance by 30%. Their implementation is targeting mobile CPUs and has been tested on the Nokia N95, on which it runs about 22 times slower compared to a desktop PC.

- In [44] an optimized version of SIFT is used for tracking initialization. The authors dropped the expensive creation of a DoG scale-space pyramid and applied the FAST detector on an image pyramid to extract scale-invariant feature positions. The authors included their SIFT variant into a tracking-by-detection pipeline running at approximately 25 Hz on the Asus P552w. On a desktop PC the system runs about 10 times faster.

- In [17] an optimized variant of SURF targeting mobile devices has been proposed. The authors mainly improved and simplified the orientation assignment, added multi-core support and employed the smaller SURF-36 descriptor. This optimized SURF is able to detect and extract oriented SURF features on the Toshiba TG01 in less then 60 ms. On a desktop PC it runs about 15 times faster than the original SURF library.

- In [48] Weimert et al. improved the runtime performance of SURF on mobile devices by replacing its feature detector with a 3D-adaption of the FAST detector. For that FAST has been adjusted to work on scale-space pyramids of gradient images, in order to detect scale-invariant blob features rather than corners lacking scale information. An additional measure of detector response strength has been implemented to allow for non-maximum suppression. The regular SURF

[15] `http://cs.unc.edu/~ccwu/siftgpu/`

algorithm was employed for orientation assignment and feature description. The authors could speed up the detection and extraction of SURF features by 30%, but were not able to achieve real-time performance on the Nokia N95-6. The construction of the image pyramid and extraction of features turned out to be the most computationally intensive parts.

The above four systems were all targeting mobile CPUs. To this date, only very few approaches are known to utilize the mobile GPU for feature detection and description..

- A GPU/CPU mixed implementation of the SURF algorithm is presented in [50]. The authors developed for a Nvidia Tegra platform using OpenGL ES 2.0. Due to limitations of OpenGL ES, the implementation reads back results from video memory multiple times to perform compaction and search operations with the CPU. The authors did not report runtime measurements, but because of the readbacks we expect it to be below the runtime requirements for NFT.

- The feasibility of mobile GPGPU is evaluated in [23]. The author developed a GPU/CPU mixed implementation of the SIFT feature detector using OpenGL ES 2.0 on smartphones with Qualcomm Adreno 20x GPUs. The implementation requires over 900 ms to extract SIFT feature positions from a 200×200 image. The author determined the creation of the scale-space pyramid and a readback to generate the keypoint list with the CPU to be the main cost drivers of the operation.

- SIFT for iPhone[16] is an open-source GPU implementation of SIFT with OpenGL ES 2.0. This system also reads back the feature detector result to create a compact list of keypoints with the CPU.

3 Design

From our review of known feature descriptors and their implementations, we chose SURF as our candidate for implementation. Both SIFT's and SURF's robustness is superior to those of other feature descriptors. But, SURF is computationally less complex and lends itself towards an easier GPU implementation.

For the comparison of runtime and matching performances between the desktop and mobile devices as well as between the mobile CPU and GPU, we chose the SURF implementation featured in OpenCV as a reference, because of its availability for Android, iOS and other mobile platforms.

[16] `http://github.com/Moodstocks/sift-gpu-iphone`

Our implementation uses OpenGL ES 2.0 and the GLSL ES 1.0 shading language as they are available for many mobile platforms and is referred to as SURF-ES throughout the rest of this thesis. In order to support as much devices as possible, our implementation avoids utilizing OpenGL ES extensions unless they are absolutely necessary.

3.1 Background

This section introduces the background knowledge necessary to understand the implications that led to our design and implementation.

3.1.1 SURF: Speeded Up Robust Features

SURF is a combination of a feature detector and feature descriptor. Extrema are extracted from a Hessian matrix-based image pyramid to obtain scale-invariant blob features. To accelerate the construction of the scale-space pyramid, the second-order Gaussian derivatives are approximated with simple box filters, which can be computed very efficiently with the help of integral images [43]. Using an integral image, the calculation of a filter sum becomes independent of the filter's scale, i.e. arbitrarily sized box filters can be computed in almost constant time. Since this thesis focuses on SURF descriptor extraction, please refer to [4] for a more detailed explanation of SURF's "Fast Hessian" feature detector.

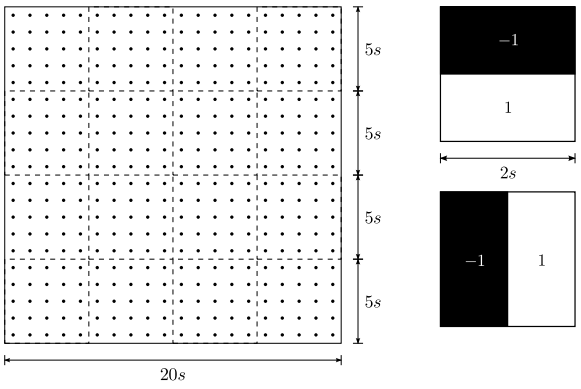

Figure 1: Depiction [42] of SURF's sample grid and the Haar wavelets computed at each samples position. Both are proportional to feature scale s.

In order to capture the image intensities surrounding a feature, the SURF descriptor spans a sample grid proportional to the feature's scale s around it. In total, the image is sampled at 400 regularly spaced sample positions with a sample step of s yielding a descriptor window of size $20s \times 20s$. Further, the descriptor window is split into 4×4 subregions, each covering 5×5 sample positions, to capture spatial information as well. At

each sample position two gradients in orthogonal directions are computed with Haar wavelets also proportional to the feature's scale. Figure 1 shows both the sample grid and wavelets used to compute the gradient information. The Haar responses are weighted with a Gaussian ($\sigma = 3.3s$) centered at the feature position to improve robustness against feature localization errors and geometric deformations due to perspective changes.

The weighted gradients of a subregion are collected in four different sums: $\sum d_x$, $\sum d_y$, $\sum |d_x|$, and $\sum |d_y|$, with d_x and d_y being the responses of each Haar wavelet at a sample position. Accumulating absolute sums enables the SURF descriptor to capture alternating gradients (e.g. repeating texture patterns as illustrated in figure 2) as well. Four such sums for each subregion constitute a 64-bin descriptor vector, which is normalized to gain contrast invariance. Note that the Haar wavelets are box filters that can be computed very efficiently with the help of a pre-computed integral image.

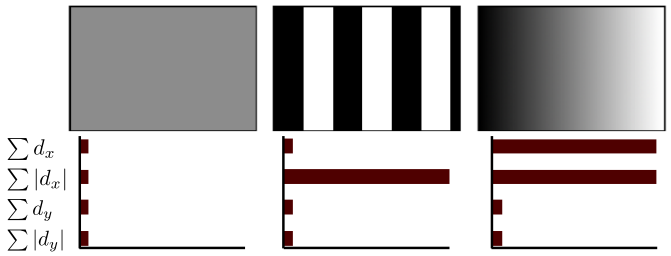

Figure 2: Illustration [4] how the SURF descriptor reacts to different intensity patterns. Left: homogeneous regions leave only low signatures in all descriptor sums. Middle: frequencies tend to become canceled out in the regular sums, but are captured by the absolute sums. Right: gradually increasing intensities are captured by both sum types.

Bay et al. also proposed a smaller descriptor window with 3×3 subregions and a total of 15×15 sample positions yielding a 36-bin descriptor vector. This so called SURF-36 descriptor is faster to compute and match than the regular SURF-64 descriptor while preserving enough distinctiveness to ensure an acceptable matching performance [4].

In order to achieve invariance against in-plane rotations, the sample grid is aligned to a feature orientation determined beforehand. Instead of rotating the Haar wavelets only their responses in x- and y-direction are rotated to save computation time. The dominant orientation of a feature is determined by computing Gaussian-weighted Haar wavelets proportional to feature scale around the feature. Summing those Haar responses in a sliding window yields a maximal response vector that lends its orientation to the feature. Please refer to [4] for a more detailed explanation, as this approach to orientation assignment is not feasible on mobile GPUs due to its reliance on computationally complex sort operations. The unoriented SURF descriptor (coined upright SURF or U-SURF, for short) is invariant to

rotations of about $\pm 15°$.

3.1.2 GPGPU with OpenGL ES 2.0

OpenGL ES 2.0 introduced the programmable graphics pipeline to mobile devices. Before that, only a fixed-function pipeline was available, which offered no (OpenGL ES 1.0) or only little possibilities (through texture combiners in OpenGL ES 1.1) to perform general-purpose computations on GPUs. Figure 3 briefly illustrates the programmable pipeline of OpenGL ES 2.0. The shaded boxes indicate the programmable stages of the pipeline. The vertex shader executes user-defined per-vertex operations on the vertices send to the graphics hardware. These operations include transformation of a verticis position and texture coordinates as well as generating new vertex attributes like shading. After vertex processing the vertices are assembled into drawable primitives such as triangles, lines, or points. The rasterizer converts each primitive from the previous stage into a set of two-dimensional fragments, which represent the pixels to be drawn on the screen. This stage also interpolates the vertex attributes between the vertices of a primitive and assigns the results to the fragments that fill this primitive. The next stage is again programmable and allows user-defined per-fragment operations. The fragment shader might use the interpolated output of the vertex shader as well as additional data from textures and uniforms (constant data) to compute the fragment color. This RGBA vector is the only output value of the fragment shader. After various additional per-fragment operations (e.g. depth test, alpha test, and blending) the fragment's color is finally written to the framebuffer.

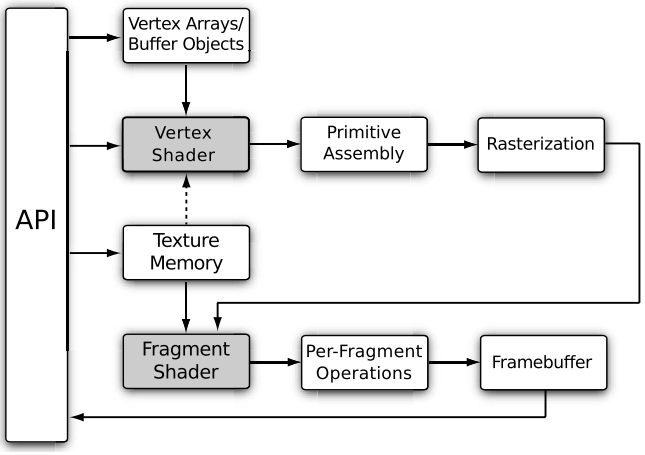

Figure 3: Illustration [32] of the OpenGL ES 2.0 graphics pipeline.

GPU architectures are designed to process the geometry fed to the graphics hardware in parallel. Exploiting the data-parallelism of the rendering

task, each data element can be processed independently from other elements. This means, the more shader units a GPU has, the better it can parallelize vertex and fragment processing. Usually, much more fragments than vertices need to be processed during rendering. Additionally, vertex shader units become idle during fragment processing, due to the pipelined architecture. GPU vendors therefore introduced unified shader units, which unify the instruction sets of vertex and fragments shaders, in order to process fragments with idle vertex shader units as well.

From a GPGPU point of view, the fragment shader offers the most potential for parallelizing general-purpose computations. In the simplest case one fragment is generated for each element of an input data array and each element is processed by a user-defined fragment shader working as the operator. Of course the result of such an operation must not depend on the results of other elements, i.e. the operation must be data-parallel. The input data must be uploaded into video memory where textures serve as two-dimensional containers. To let the graphics hardware generate a fragment for each texel (i.e. each input data element) one would render a textured quad orthogonal to the view axis of the camera while using an orthographic projection and setting the viewport to the size of the input array. The rasterizer will then generate a fragment for each viewport pixel and interpolate the texture coordinates of the vertices defining the quad. The interpolated texture coordinates serve as a unique index of each fragment to address elements in the input data.

Texture coordinates are normalized in OpenGL ES 2.0, i.e. they are floating-point values between 0 and 1. When reading from a texture either the value of the texel which center is closest to the coordinate or an interpolation of the four closest texels is returned. Note that this bilinear interpolation allows for subpixel-accurate sampling of the input data.

More complex operations often need to be separated into multiple render passes, where intermediate results of previous passes are reused in subsequent passes. This can be accomplished by rendering to textures, which then serve as containers for those intermediate results. To this end, OpenGL ES 2.0 offers so called framebuffer objects (FBOs), which allow for efficient switching between different render targets. OpenGL ES 2.0 does not provide support for MRT (multiple render targets). Hence, OpenGL ES applications cannot render to multiple textures at the same time.

Beside the parallel processing of vertices and fragments, GPU architectures offer another level of parallelization: because most computations in computer graphics are performed on four-dimensional vectors (e.g. RGBA colors and homogeneous coordinates), GPU vendors included SIMD (single instruction, multiple data) instruction sets into their hardware. Those SIMD instructions can perform operations on each element of a 4D vector in parallel. Therefore, packing data into textures accordingly not only reduces the input texture size by factor 4, but also parallelizes a GPGPU task

even more.

GLSL ES 1.0 introduced the precision qualifiers `lowp`, `mediump`, and `highp`. These qualifiers let the shader author specify the precision with which computations for a shader variable are performed. Declaring variables with low precision might result in faster execution and/or better power efficiency of a shader program. Supporting `highp` variables is optional. Note that it is not mandatory to actually implement different precisions. Implementations might also ignore those qualifiers and perform all computations with the highest precision.

Mipmapping is an important computer graphics tool to tackle aliasing artifacts during texturing, hence improving the visual quality of a rendered scene. A mipmap is defined as a chain of a texture image, where each subsequent level is half as large in both dimensions as the one before it. The top end of this mipmap chain is the originally specified texture, and the bottom end a texture having a size of 1×1 tx. The texels in lower levels are usually computed as the mean of the four corresponding texels in the next highest level. OpenGL ES 2.0 offers automatic mipmap generation, which is often implemented with graphics hardware acceleration. The texel values in lower mipmap levels can also be regarded as normalized sums of different regions in the top level (i.e. the input data). Mipmaps can therefore allow for efficient box filtering. The discretized structure of mipmaps can be efficiently circumvented by exploiting hardware-accelerated interpolation between mipmap levels (trilinear interpolation).

Before any rendering can take place, OpenGL ES 2.0 requires a rendering context and a drawing surface. Both can be created with the EGL API. EGL serves as a platform-independent link between graphics libraries like OpenGL ES and the native windowing system of the platform.

3.2 Performance Considerations

Mobile GPUs are far more restrained than their desktop counterparts: they comprise only a fraction of the shader processors desktop GPUs have and run at lower clock speeds to save battery power. The OpenGL ES 2.0 standard accounts for these restrictions by having significantly relaxed compliance requirements. Consequently many features common on desktop GPUs are not available on mobile GPUs. Therefore one has to consider several aspects when implementing GPGPU applications targeting mobile devices.

Texture formats: OpenGL ES only requires support for low-precision texture formats that store a color channel with one byte per value. Therefore each channel of an RGBA texel is represented as a 8-bit fixed-point value between 0 and 1. But, such a low precision is not sufficient to store SURF descriptors or intermediate results of the SURF algorithm accurately. In

order to store four floating-point values in a texel one has to use extensions that are not supported by all mobile GPUs. On devices with an ARM Mali-400 MP GPU is the corresponding extension GL_OES_texture_float not available. Because we aim to test our implementation on a wide range of mobile devices we chose to employ a fixed-point format with higher precision, which uses four bytes (i.e. all color channels) to store a floating-point value.

SIMD instructions: The fact that SURF requires high-precision floats for most parts of its algorithm also has implications on how we can exploit the GPU's SIMD architecture. PowerVR SGX5xx GPUs, for example, can only vectorize operations for `mediump` and `lowp` floats [21]. Squeezing four `highp` floats into a `vec4` will therefore not increase runtime performance on those GPUs.

Texture fetch latency: Texture reads are significantly slower on mobile GPUs than on desktop GPUs. It is therefore even more important to spatially group texture fetches for optimal texture cache efficiency. Memory latency can also be "hidden" with arithmetic operations on some GPUs. PowerVR SGX5xx GPUs, for example, are capable of scheduling operations from a pool of fragments when a fragment waits for a texture read to finish [20]. The ratio of texture fetches to arithmetic operations is referred to as the arithmetic intensity of a shader program.

In a first attempt we parallelized the creation of an integral image as proposed in [19]. This algorithm was disproportionately slower on mobile GPUs than on desktop GPUs due to its large number of texture fetches ($4\times$ number of image pixels) and its bad cache efficiency. Sengupta et al. [40] proposed another parallelization of the problem with optimized texture cache usage. Although it is $4\times$ faster on desktop GPUs, we expect it to be even slower on mobile GPUs, because it requires more than twice the number of texture fetches than Horn's parallelization does. Therefore we refrained from using an integral image at all and employed a mipmap of the input image instead. Mipmaps are faster to generate while offering similar scale-space properties. Using a mipmap also has the advantage of an inherently better cache efficiency when sampling at arbitrary scales.

Note that high memory latencies also affect other multi-pass algorithms that require too many texture reads. For instance, the HistoPyramids algorithm [52], which could be used to generate a compact keypoint list from the output matrix of a GPU feature detector, is also unfeasible on mobile GPUs.

Data read-backs: Due to the reduced memory bandwidth of mobile GPUs it is particularly problematic to copy data from video memory into system

memory. Such operations are blocking and therefore stall the rendering pipeline for a disproportionately long time on mobile GPUs. It is important to avoid such read-backs amidst a GPGPU processing pipeline in order to compute intermediate results on the CPU, when fast frame rates are crucial.

Floating-point accuracy: How accurately a floating-point value can be stored depends on the precision of a GPU's floating-point format. The precision can be queried through the OpenGL ES 2.0 API and is represented as an integer value. The Mali-400 MP, for example, calculates with a precision $p = 10$. I.e. the smallest representable value ϵ that satisfies $1 \neq 1 + \epsilon$, is calculated as $\epsilon = 2^{-p} = 1/1024$. This ϵ varies greatly across mobile GPUs and must be taken into account when running computations requiring high precision.

3.3 Parallelizing the SURF Descriptor

The obvious inherent parallelism of the SURF algorithm is that it treats each detected feature independently of other features. Multi-threaded implementations of SURF targeting multi-core CPUs (e.g. [15] or [17]) exploit this data-parallelism by assigning the tasks of describing different SURF features to different threads. As a result, the more cores a CPU has, the more descriptors can then be extracted in parallel. This coarse parallelization of SURF is reasonable on multi-core CPUs as they are actually designed for task parallelism and their individual cores are therefore loosely coupled. Consequently, multi-threaded CPU implementations do not gain any extra advantage from executing the same code on different data elements. Additionally, thread creation and scheduling induce a significant overhead on CPUs, which is reflected in the speed-ups multi-threaded CPU implementations can achieve relative to the number of available CPU cores. Although the speed-up factor theoretically should be equal to the number of cores, in practice it is slightly lower and degrades when the number of utilized cores increases [15]. Because of their limited number of parallel processing units and thread scheduling overheads, a more fine-grained parallelization is not profitable on multi-core CPUs.

In contrast, GPUs are designed for data parallelism. The execution units of a GPU core are tightly coupled and can run code concurrently as long as it does not diverge due to branching. This restriction on the executed code enables GPUs to dispatch a fetched instruction to multiple concurrent threads at once. Additionally, the threads of a GPU's execution unit are light-weight and can be scheduled very efficiently. A GPU implementation of the SURF descriptor has to account for this highly parallelized execution model to achieve maximum efficiency. Because the output of a shader unit is limited to a 4D vector (the fragment's color), the task of extracting a SURF descriptor has to be separated into several parallelized subtasks that reuse

intermediate results. This introduces an additional level of parallelization, beyond the inherent keypoint-level parallelism, to the implementation. A SURF descriptor is therefore extracted by several fragments and in multiple render passes. The SURF descriptor algorithm can be separated into the following subtasks (i.e. render passes):

1. Calculate weighted Haar responses at each sample position.
2. Accumulate descriptor sums.
3. Normalize descriptor.
   (a) Calculate descriptor length (norm).
   (b) Divide each descriptor bin by the norm.

Each of these subtasks might be parallelized, depending on the degree of data-parallelism they exhibit.

The first subtask of calculating the Haar responses is highly data-parallel. Each sample position can be handled by an individual fragment which calculates the Haar responses in both directions and stores them in its output color. The execution path of all fragments does not diverge in case no border handling is necessary. Because the Haar wavelets of neighboring sample positions overlap, data reuse is not optimal. However, in case neighboring fragments also fetch from neighboring texels in the input image texture, reuse is automatically optimized by the texture cache.

Accumulating the descriptor sums allows for a parallelization at subregion level. One fragment can accumulate all Haar responses of a subregion to compute its four different descriptor sums. This parallelization only works if the four sums can be stored in the four channels of the output color. Otherwise, only a parallelization at descriptor-bin level is possible, which quadruples the number of fragments. Letting one fragment accumulate all the 25 Haar responses on its own is suboptimal, because it results in a shader with low arithmetic intensity. However, parallelizing this subtask with a more sophisticated reduction operation is not feasible on mobile GPUs. A more promising optimization might be to align the sampling grid to power-of-two dimensions and use mipmaps to compute the sums of the Haar responses.

The third subtask is again not very data-parallel and only allows for a parallelization at keypoint level: one fragment iterates over a whole descriptor and computes its length. As this subtask is similar to the previous one, the same optimizations might be considered.

The last subtask allows for a parallelization at descriptor-bin level due to its good data-parallelism. As many fragments are generated as there are descriptor bins and each fragment divides a bin value by the descriptor norm.

3.4 Wrapping the OpenGL ES 2.0 API for GPGPU

In order to ease the cumbersome cross-platform GPGPU programming with OpenGL ES 2.0, we abstracted its API to have a more simplified interface comprising only three classes. This minimal interface is roughly inspired by MinGPU [2] and therefore referred to as MinGPU-ES throughout this thesis. Figure 4 shows a class diagram of MinGPU-ES.

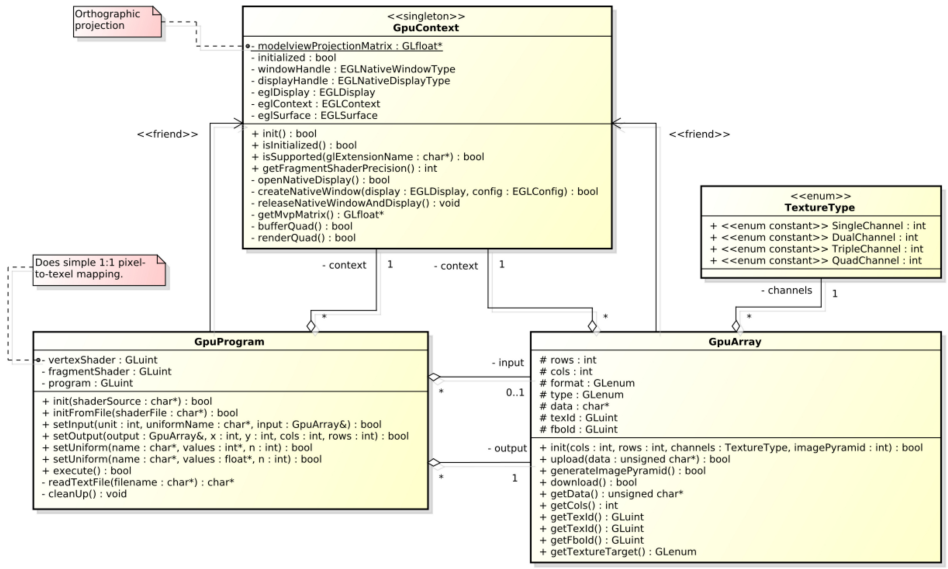

Figure 4: Class diagram of MinGPU-ES.

The class `GpuContext` hides platform-specific details regarding the creation of EGL and OpenGL ES contexts. It is implemented following the singleton pattern to prevent the user from creating multiple contexts. Additionally, a `GpuContext` serves as a simple renderer drawing a screen-aligned textured quad. `GpuProgram` is responsible for reading, building and linking shaders. The vertex shader is pre-defined as its sole purpose is to pass through texture coordinates and ensure a 1:1 pixel-to-texel mapping by transforming each vertex with the orthographic projection matrix defined in `GpuContext`. The fragment shader is provided by the user and defines the operator applied to the input data. A `GpuProgram` instance also triggers rendering when it is executed by the user. Finally, `GpuArray` wraps the different texture types of OpenGL ES 2.0 for simplified use as input and output data containers.

3.5 Interface of SURF-ES

The interface to SURF-ES is straightforward: only one class is accessed by the user to initialize and execute our implementation. A class diagram of

SURF-ES can be seen in figure 5.

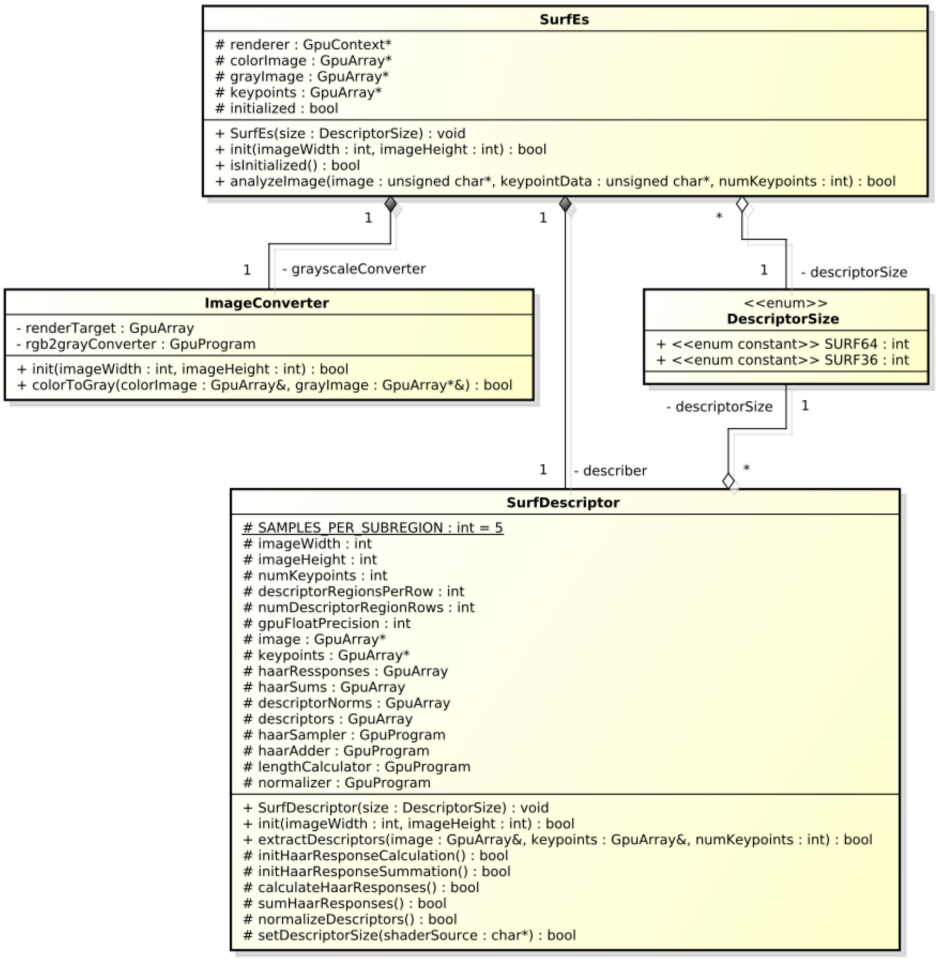

Figure 5: Class diagram of SURF-ES.

The main class `SurfEs` provides an initialization method to trigger the allocation of video memory and shader creation prior to execution. Since SURF-ES reuses its textures, it is fixed to the image size provided during initialization. A second method executes the descriptor extraction and expects an RGB image as well as a list of feature positions where SURF descriptors should be extracted. The two additional classes shown in the figure 5 are not accessed by the user and only encapsulate the different underlying tasks of SURF-ES.

4 Implementation

SURF-ES has been implemented with C/C++ and runs on virtually all systems that support OpenGL ES 2.0. For the detection of features in the input image we use the CPU implementation of SURF in OpenCV 2.3.1. The implementation realizes the interface presented in section 3.5. Additional Java and Objective-C classes were necessary to embed SURF-ES in Android and iOS apps. Note that SURF-ES does not assign orientations to features, thus extracts U-SURF descriptors, which are not rotationally invariant.

When SURF-ES is being initialized with the desired number of descriptor bins and a fixed image size, it creates all the necessary textures and shader programs once and reuses them each time new input data is handed to it. The upload of an RGB image to video memory works straightforward when its data type is `unsigned char`: OpenGL ES automatically copies each value to the correct texel and channel in an RGB texture and interprets it as an 8-bit fixed-point value between 0 and 1. To store the list of n keypoints with a subpixel position (x, y) and a scale s in a texture, we employed a simple encoding scheme. Each keypoint is stored in two RGBA texels, yielding a texture of size $2 \times n$ tx. The x-position of keypoint i is stored in the red, green and blue channel of texel $(1, i)$ with the integral part of the position stored in the red and green channel and the subpixel offset stored in the blue channel. The keypoint's y-position is stored analogously in texel $(2, i)$. The scale s of the keypoint is also split into its integral and fractional part and stored separately in the two remaining alpha channels of both texels.

The rest of this sections explains in detail how we implemented each part of the SURF descriptor with OpenGL ES 2.0 and GLSL ES. The explanations apply for the SURF-64 descriptor, but are of course analogous to the smaller SURF-36 descriptor.

4.1 Color to Grayscale Conversion

Color-converting between simple color spaces like RGB and grayscale is a particular easy task for GPUs: all image pixels are transformed by a linear operator and therefore have no dependencies between each other. Hence this task is implemented with a fairly simple fragment shader. Each fragment fetches a RGB value from the texture holding the input image and weights each color channel to obtain the pixel's luminance l according to the formula

$$l = \begin{pmatrix} 0.299 \\ 0.587 \\ 0.114 \end{pmatrix} \cdot \begin{pmatrix} r \\ g \\ b \end{pmatrix},$$

where $\cdot$ denotes the dot product of two vectors. To generate a fragment

for each image pixel, we render a textured quad with image dimensions. OpenGL ES 2.0 stores a RGB value as a triplet of 8-bit fixed-point values between 0 and 1. To increase the computational accuracy of the operation, we convert them to integer values between 0 and 255 beforehand, round the resulting l to the closest integer value, and convert it to a float within the range $[0, 1]$. Finally the luminance is stored in the red channel of the output texture. Because this output texture will be mipmapped later, it has to have dimensions aligning to powers of two. In order to support input images that do not meet this requirement, the color conversion shader also pads the input image with black pixels on the right and bottom if necessary.

4.2 Sampling of Haar Responses

The SURF descriptor is based on 400 regularly spaced samples of Haar wavelets as shown in figure 1. This sample grid is represented as a quadratic 2D block of texels in an output texture so that each texel maps to one sample position. A formation of multiple such texel blocks in the output texture constitutes the set of descriptors to extract. Since texture dimensions have an upper boundary, we limit the number of processable keypoints to 1020 requiring a texture of size 1020×400 tx (i.e. 51×20 sample blocks). For each of those texels a fragment is generated by rendering a quad of the same size. If there are less than 1020 keypoints, the viewport is scaled down accordingly to avoid the generation of fragments that will not do any work.

Each fragment then first calculates the numeric ID of the keypoint it is associated with using its texture coordinates. Since the accuracy of floating-point texture coordinates degrades with increasing texture sizes, they are converted to integer coordinates beforehand[17]. The keypoint ID is converted into a texture coordinate to fetch the keypoint's position and scale from the keypoint texture. With that data the fragment's texture coordinates are transformed into image space to obtain the subpixel-accurate sample position. The keypoint's scale also determines the size of the two Haar samples (x- and y-direction) the fragment will compute.

In their original work, Bay et al. proposed to use an integral image for efficient computation of arbitrarily sized box filters. However, since it is unfeasible to create and read from integral images on mobile GPUs, we opted for a mipmap of the input image instead (as proposed in [11]). For that we divide the area covered by a Haar wavelet (figure 1 on the right) into four equally sized quadrants. The intensity sum of each quadrant can then be read from a lower mipmap level. Calculating both Haar responses from those four sums is trivial. Using a mipmap has several advantages:

[17] The inbuilt fragment coordinates (`gl_FragCoord`) could not be used, because their y-axis is flipped on Adreno GPUs.

instead of having to fetch eight widely spaced texels from an integral image to calculate the wavelet responses for both directions, we only have to fetch four neighboring texels with greatly improved spatial locality. The filter results of both methods are identical as long as the Haar wavelets have power of two dimensions. Of course this is seldom the case, but responses for different wavelet sizes can be approximated by interpolating between mipmap levels. Furthermore, bilinear interpolation between texels provides subpixel accuracy. The cost of additional texture fetches for trilinear interpolation is tolerable due to good texture cache usage. Fetching from the correct mipmap level and trilinear interpolation are all done automatically by the graphics hardware and no additional shader instructions are required.

The resulting Haar responses are weighted with an unnormalized Gaussian ($\sigma = 3.3s$) centered at the feature position. Afterwards both responses are stored in the four channels of the fragments output color (i.e. one texel in the output texture). This means that there are 16 bits (two color channels) available to store each floating-point Haar response value. In theory, the Haar responses are ranging from -2 to 2, but in practice are distributed around 0 with very low variance. We chose a simple scheme to encode this values into two 8-bit floats: splitting the values into their integral and fractional part and storing both in separate color channels. To use all bits of the channel storing the integral part, we first scale the Haar responses to the range $[-128, 128]$. The fractional part can be stored directly, but the integral part must be converted to a float value between 0 and 1 before writing. This conversion introduces a negligible error, because, with 8 bits, we can "only" represent the range $[-128, 127]$.

4.3 Descriptor Formation

For every bin of each SURF descriptor a fragment is generated to sum up the Haar responses of a subregion. This is achieved by rendering a quad of size $n\times$64 px, n being the number of keypoints. Each fragment is then associated to a descriptor window's subregion by its texture coordinates and computes one of the four sums that describe this subregion.

Because reduction operations are not feasible on mobile GPUs, we implemented a simple gather operation to calculate the descriptor sums. Each fragment iterates over a subregion with a nested loop, fetching the Haar responses and adding them together. This means that each fragment does 25 texture fetches and only very few arithmetic operations resulting in a low arithmetic intensity of this shader. We also implemented the more efficient method proposed in [11], which uses mipmaps of float textures (as provided with the extension GL_OES_texture_float). Unfortunately, we encountered problems on mobile devices when we tried rendering to float textures or mipmapping them.

Theoretically the sum values are in the range $[-1671.8, 1671.8[$ (the sum of 25 *Gaussian-weighted* Haar responses within the range $[-128, 128[$), however in practice they are again distributed around a mean relatively close to 0 with very low variance. To cover such a wide range of magnitudes accurately we had to employ a more sophisticated encoding than we used to store the Haar responses in the previous step. Therefore the sum values are encoded into all four channels of a texel as described in section 4.5. Before encoding, the sums are scaled down to be within the range $[-128, 128[$.

In OpenGL ES 2.0, fragments have only one output color. Because all four channels of a texel are required to store a single descriptor sum, we need to generate four fragments to compute and write each of the four sums of a subregion. This means that each subregion is visited four times without the possibility to reuse fetched data. Again, this could be circumvented with the availability of renderable float textures.

The resulting output texture stores an unnormalized SURF descriptor in each of its columns and is 1020 columns wide. Unused columns will be undefined, because we adjust the viewport according to the number of keypoints before rendering.

4.4 Descriptor Normalization

Normalizing the SURF descriptors is important to gain contrast invariance. This step has been implemented in two straightforward render passes. The first pass calculates the length of each descriptor vector and stores it in an 1D texture. The second pass fetches the lengths from that texture and normalizes the descriptor bins with it.

In the first pass we generate a fragment for each descriptor vector by rendering a quad of size $n \times 1$ px, where n is the number of keypoints. Each fragment then loops over the descriptor it is associated with by its texture coordinates and calculates the descriptor's vector length. This is done by fetching from the texture generated in the previous step, squaring each fetched bin value, adding them all together and calculating the square root of that sum. Because the number of bins in the SURF-ES descriptor can either be 64 or 36, but GLSL ES only supports constant loop expressions [24], the shader source string is configured at runtime according to the desired descriptor size. The computed vector length is in the theoretical range $[0, 522.68[$. Since the resulting values are much lower in practice, we omit an extra down-scaling and store the length in the fragment's output color with the encoding described in section 4.5.

The second render pass divides each descriptor bin with the calculated length of the descriptor it belongs to. This final step can be implemented very efficiently with GPGPU, due to its strong data-parallel nature. By rendering a quad with the dimensions $n \times 64$ px, one fragment is generated for each descriptor bin. Each fragment then fetches an unnormalized bin

value and the corresponding descriptor length from the output textures of the two previous steps, normalizes the bin value and stores it in the final output texture using our encoding for high-precision floats.

The final descriptors reside in an RGBA texture of size 1020×64 tx in video memory, with each column holding one SURF-64 descriptor. In case less than 1020 keypoints have been passed to SURF-ES the unused columns will be undefined.

4.5 Encoding of High-Precision Floats in Textures

Regular textures in OpenGL ES 2.0 can store floating-point values only with an 8-bit fixed-point format. Although GLSL ES fragment shaders are required to perform floating-point computations with a precision of at least 10 [24], they can propagate their results to subsequent shader passes only through low-precision textures. Textures supporting higher precision are available as extensions but not on all mobile GPUs. By aiming to develop for a wide range of devices, we implemented an encoding scheme to store a single high-precision float in the four channels of an RGBA texel. The storage precision of floating-point values differs largely across GPUs of different vendors. The Adreno 220 GPU by Qualcomm, for example, has a precision of 24, whereas the ARM Mali-400 MP only offers a precision of 10. In order to exploit the full capabilities of high-precision GPUs, but also support GPUs with lower precision, our encoding scheme takes the GPU's precision dynamically into account.

Essentially a floating-point value is stored in a texel by splitting it into four parts: the integral part of the float is stored in the red channel and the fractional part in the remaining three channels. Having only one byte to store the integral part restricts our encoding to floats within the range $[-128, 128[$. Larger values need to be scaled to this range before encoding them. The encoding of the fractional part depends on the GPU's floating-point precision p, which we retrieve with `glGetShaderPrecisionFormat()` and pass to the shaders as a uniform.

Red	Green	Blue	Alpha
$r - 128$	$g \cdot 2^{-p}$	$b \cdot 2^{8-p}$	$a \cdot 2^{16-p}$

Table 1: The Encoding of a floating-point value in the color channels of a texel taking the GPU's available floating-point precision p into account. The integral part is stored in the red channel and the fractional part is split over the remaining channels. The values r, g, b and a are all integers between 0 and 255.

Table 1 demonstrates how the fractional part of floating-point number is represented as a sum of different powers of two, which are one byte apart. The actual exponents and the number of summands are determined by the

available floating-point precision p of the GPU. If the GPU's precision is below 16, the value of the alpha channel a will always be set to 0 in order to avoid superfluous computations. This data layout derives from the formal expression of a floating-point value f in the range $[-128, 128 - 2^{-p}]$ as

$$f = r - 128 + \frac{g \cdot 2^0 + b \cdot 2^8 + a \cdot 2^{16}}{2^p},$$

where r, g, b, and a are the color values of a texel converted to integers in the range $[0, 255]$ under the condition that $g \cdot 2^0 + b \cdot 2^8 + a \cdot 2^{16} < 2^p$ holds true.

Listing 1 shows how we implemented the encoding of floats in texels with GLSL ES. It can be seen that the decision, whether the alpha channel of the texel will be used or not, is actually not based on the retrieved precision p, but whether the macro `GL_FRAGMENT_PRECISION_HIGH` is defined or not (i.e. `highp` floats are available or not). This could lead to an overflow of the blue channel in case the GPU does not support `highp` floats but its precision p exceeds 16. However, because the GLSL ES specification makes a precision of 16 the minimum requirement for `highp` floats [24], we consider this very unlikely to be the case. The advantage of an `#ifdef`-directive over a regular `if`-statement is that it yields less shader instructions.

This format can accurately store floating-point values with a precision up to 24. However, note that the encoding fails due to an overflow of the alpha channel if the GPU supports a higher precision and the `precision` value passed to the shader as uniform has not been clipped accordingly. Decoding an encoded float value from a fetched texel is straightforward and has been implemented by simply reversing the operations applied to encode it.

5 Experimental Results

To assess the feasibility of our GPGPU approach to feature descriptor extraction, we have tested our implementation on a wide range of state of the art mobile devices. Table 2 lists all those devices along with their incorporated GPUs and CPUs. Mobile devices generally do not feature a separate GPU and CPU, but so called SoCs (system on a chip), which integrate both processing units (and others) into a single chip. Unfortunately, their architecture is often undisclosed and not accurately specified. Note that we did not perform the full spectrum of tests on the iPhone 4S, the iPad 4G, and the MSM8960 as these became only available recently. For comparison we have also tested SURF-ES on a desktop PC using the PowerVR OpenGL ES 2.0 emulation libraries[18].

[18] `http://www.imgtec.com/powervr/insider/sdkdownloads/`

```
1  /** @precondition: precision = 2^p ∧ −128 ≤ value ≤ 128 − 2^−p */
2  vec4 float2vec4(float value)
3  {
4    vec4 container;
5
6    // Convert fraction into an integer in range [0, 2^p[.
7    float fraction = fract(value);
8    fraction = floor(fraction * precision);
9
10   #ifdef GL_FRAGMENT_PRECISION_HIGH
11     // Store multiples of 2^16.
12     container.a = floor(fraction / 65536.0);
13     fraction = fraction - container.a * 65536.0;
14   #else
15     container.a = 0.0;
16   #endif
17
18   // Store multiples of 2^8.
19   container.b = floor(fraction / 256.0);
20   fraction = fraction - container.b * 256.0;
21   // Store multiples of 2^0.
22   container.g = fraction;
23   // Store integral part accounting for sign.
24   container.r = floor(value) + 128.0;
25
26   // Convert 8-bit integers to 8-bit floats between 0 and 1.
27   return container / vec4(255.0);
28 }
```

Listing 1: Function to encode a float value into a texel. The variable `precision` is passed as a uniform to the shader and is set to 2^p.

Device (OS)	GPU, CPU
Desktop PC (Ubuntu 10.10 “Maverick Meerkat”)	Nvidia GeForce GTX 260 Intel Core 2 Quad (4 cores, 2.66 GHz)
Qualcomm Snapdragon S4 MSM8960 (Android 4.0.3 “Ice Cream Sandwich”)	Qualcomm Adreno 225 Qualcomm Krait (2 cores, 1.5 GHz)
Apple iPad 4G (iOS 5.1)	PowerVR SGX543MP4 Apple A5X (2 cores, 1 GHz)
Nvidia Tegra 3 (Android 3.2.1 “Honeycomb”)	ULP GeForce ARM Cortex-A9 (4 cores, 1.4 GHz)
Apple iPhone 4S (iOS 5.1)	PowerVR SGX543MP2 Apple A5 (2 cores, 800 MHz)
HTC Evo 3D (Android 2.3.4 “Gingerbread”)	Qualcomm Adreno 220 Qualcomm Scorpion (2 cores, 1.2 GHz)
Samsung Galaxy S II (Android 2.3.3 “Gingerbread”)	ARM Mali-400 MP ARM Cortex-A9 (2 cores, 1.2 GHz)
HTC Desire Z (Android 2.3.3 “Gingerbread”)	Qualcomm Adreno 205 Qualcomm Scorpion (800 MHz)
Samsung Galaxy Nexus (Android 4.0.2 “Ice Cream Sandwich”)	PowerVR SGX540 ARM Cortex-A9 (2 cores, 1.2 GHz)
Nokia N9 (MeeGo 1.2 “Harmattan”)	PowerVR SGX530 ARM Cortex-A8 (1 GHz)

Table 2: Specifications of the tested devices. Please note that both the MSM8960 and the Tegra 3 are development devices.

5.1 Runtime Performance

To measure the runtime performance of SURF-ES, we executed it multiple hundred times in a loop and determined the average duration of an iteration for each mobile device. The device was reset before each test run and we used the OS specific methods to stop unnecessary background processes. In order to have comparable results to other papers in the domain we used down-sampled versions of the first two images of the "graffiti" test set [30]. Figure 6 shows the average time each device took to extract 1020 feature descriptors from an image of size 512×384 px. Those measurements include uploading the input image and the list of keypoints to video memory, but exclude image loading and keypoint detection with OpenCV as well as downloading the resulting descriptors from video memory. Just as expected, novel mobile devices with up to date multi-core GPUs perform significantly faster than older devices. Compared to the same implementation running on a desktop GPU the slowdown ranges from less than 10× (MSM8960) to almost 120× (N9).

GPU vendors may choose to off-load render tasks to the mobile CPU. Therefore, we also measured the CPU utilization of the SURF-ES process during a GPGPU run. Those measurements are integrated in figure 6. For the most devices CPU utilization is higher as expected. The majority of the devices indeed seem to spread the workload of SURF-ES between their GPU and CPU. Particularly, the Galaxy S II has a very high CPU load when executing GPU tasks. We can also observe here that very similar GPUs can yield different CPU utilizations. This originates in the handset producers' choices for drivers, features on the SoC, and the like. When running OpenCV SURF, CPU utilization easily reached 80–90% across all devices.

In order to determine the speed-up our GPU implementation can deliver on mobile devices, we compared SURF-ES to the CPU implementation of SURF featured in OpenCV. We used release builds of OpenCV with device specific optimizations such as NEON intrinsics or VfP turned off. Since SURF-ES is not rotationally invariant we configured OpenCV SURF to extract U-SURF descriptors, which ignore feature orientation as well. The runtime differences between the GPU and CPU of each mobile device, when extracting 1020 SURF descriptors from a 512×384 image, are shown in figure 7. Devices with a Cortex-A9 CPU, viz. the Tegra 3, the Galaxy S II, and the Galaxy Nexus, perform similar and yield a GPGPU acceleration by factor 2 to 5. The MSM8960's Krait CPU is only slightly faster but due to its much faster GPU the achieved speed-up is almost 10×. The Scorpion CPUs on the two tested HTC devices (Evo 3D and Desire Z) are significantly slower, which leads to an almost 8× speed-up on both of them. Same can be said for the iPhone 4S (over 7×) and the iPad 4G (14×) with comparatively slow CPUs but faster GPUs. OpenCV performed extremely slow on the N9. We suspect a problem with the ARM EABI (armel) libraries

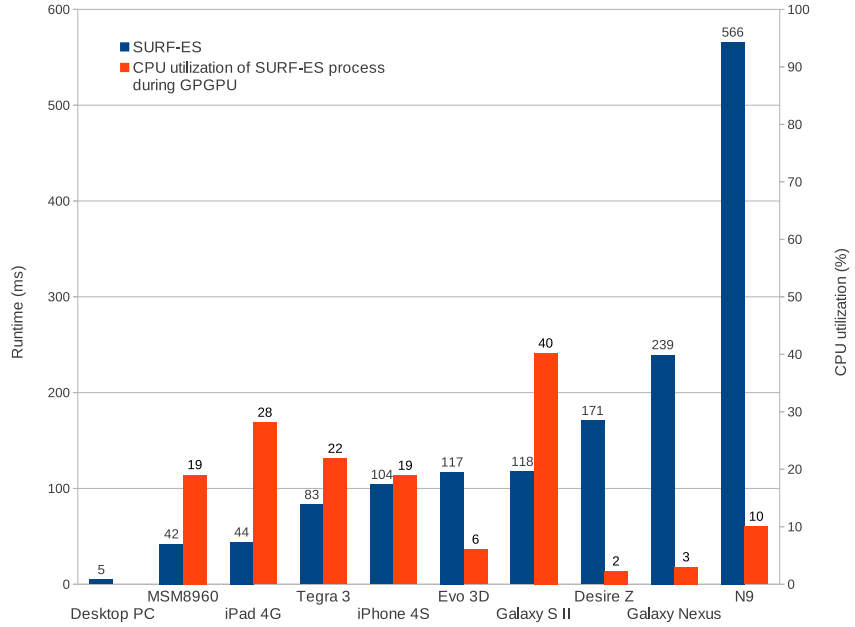

Figure 6: Runtime of SURF-ES to extract 1020 feature descriptors from a 512×384 image. Red bars denote the CPU utilization while running SURF-ES.

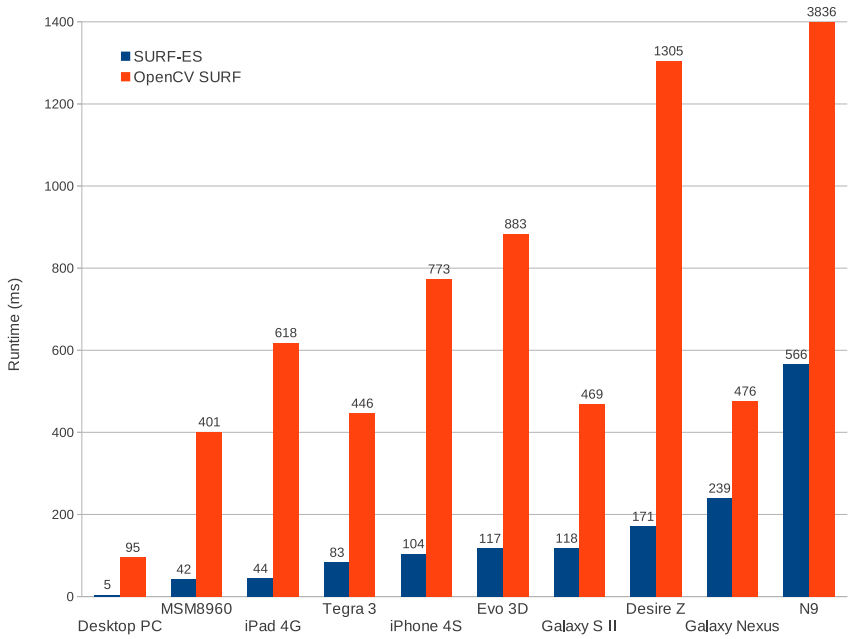

Figure 7: Runtime of OpenCV SURF to extract 1020 feature descriptors from a 512×384 image. Note that the red bar representing the OpenCV SURF runtime on the N9 has been cut off.

of OpenCV for Meego 1.2 that we were not able to identify. On a desktop PC SURF-ES runs 20× faster than OpenCV's SURF implementation. Note that OpenCV SURF is not multithreaded. Hence, the measurements rather reflect a CPU's per-core performance than its full potential.

Apart from the device SURF-ES is running on, two additional factors influence its runtime: the size of the input image and the number of feature descriptors to extract. Both are connected because a larger image can yield more features. Figure 8 shows SURF-ES' runtime plotted against the number of keypoints. Among the mobile devices, the MSM8960 and the iPad 4G perform best also in terms of scalability. Overall it can be seen that newer GPUs with more shader units scale better than older ones. The Galaxy S II however scales badly, having an almost constant runtime independent of the number of keypoints.

To evaluate the influence of the image size, we have tested SURF-ES on images with several different sizes (256×256, 320×240, 512×512, 640×480, 800×600 and 1024×1024 px). The measurements of this test are presented in figure 9. Keep in mind that the dimensions of the input image get padded to powers of two during the RGB-to-grayscale conversion step (e.g. an image of size 320×240 px is padded to 512×256 px), which influences the performance of the subsequent mipmap generation and Haar wavelet sampling step. The results are comparable to the ones presented in figure 8: newer GPUs outperform older ones, due to more shader units increasing their parallel processing power. Once again, the Galaxy S II lacks scalability and requires disproportional more runtime for larger images. The Tegra 3 on the contrary scales exceptionally well. The MSM8960 scales slightly worse but is faster when comparing absolute runtimes. The N9 was excluded, because SURF-ES crashed on it when working with certain image sizes for reasons we were not able to track down.

In order to gain insight into how each mobile GPU copes with the individual tasks of the SURF algorithm, we also measured the runtime for the main computational steps of SURF-ES separately. Figure 10 shows the runtime of each step relative to the absolute runtime of SURF-ES. Each bar represents one device and each colored segment a part of the algorithm. Although there are significant differences between the devices, one can still note that the tasks with low arithmetic intensity (summation of the Haar responses and calculation of the descriptor length) generally perform worst. This is not true for the Galaxy S II: the generation of the image mipmap requires an extremely large part of the runtime, whereas the calculation of the descriptor sums and length are performed very fast.

Additionally we measured the performance of the smaller but less robust SURF-36 descriptor. A comparison of those measurements in relation to the runtime of the SURF-64 descriptor is shown in figure 11. Generally, using the SURF-36 descriptor decreases the runtime by about 30%, except for the Galaxy S II where only a speed-up by 8% is noticeable.

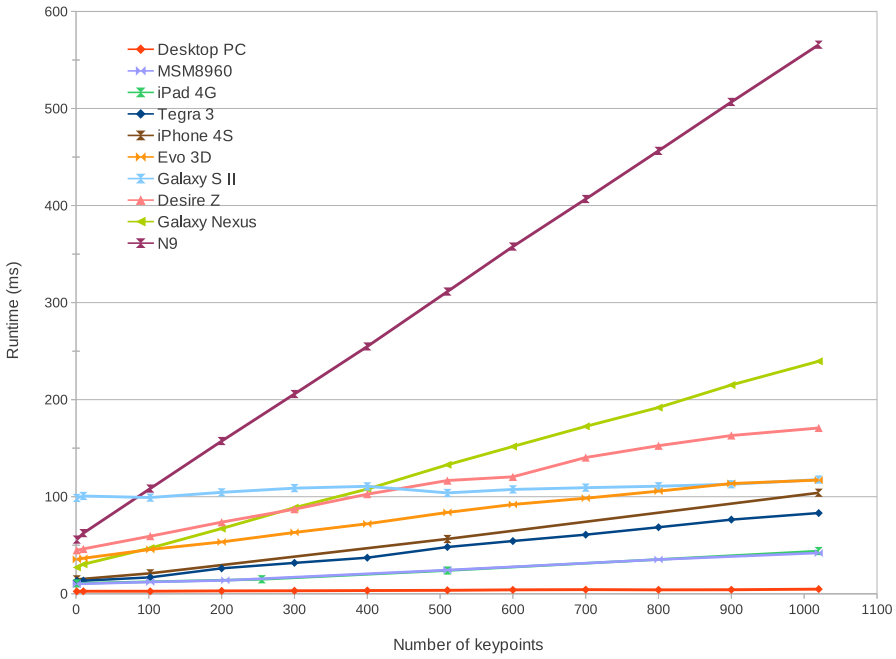

Figure 8: Runtime of SURF-ES relative to the number of keypoints in a 512×384 image. Note that the curves for the MSM8960 and the iPad 4G almost lie on top of each other.

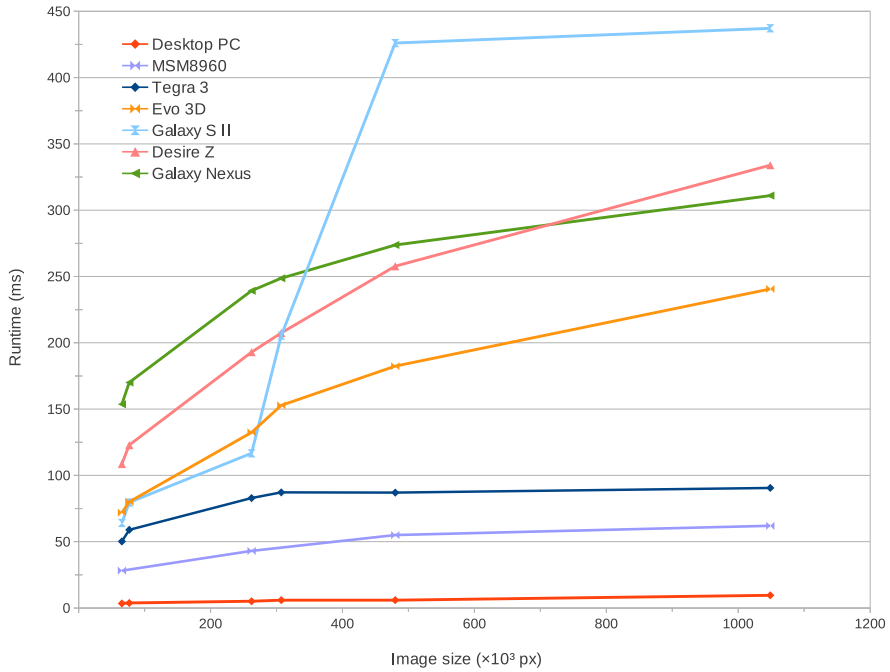

Figure 9: Runtime of SURF-ES relative to the image size.

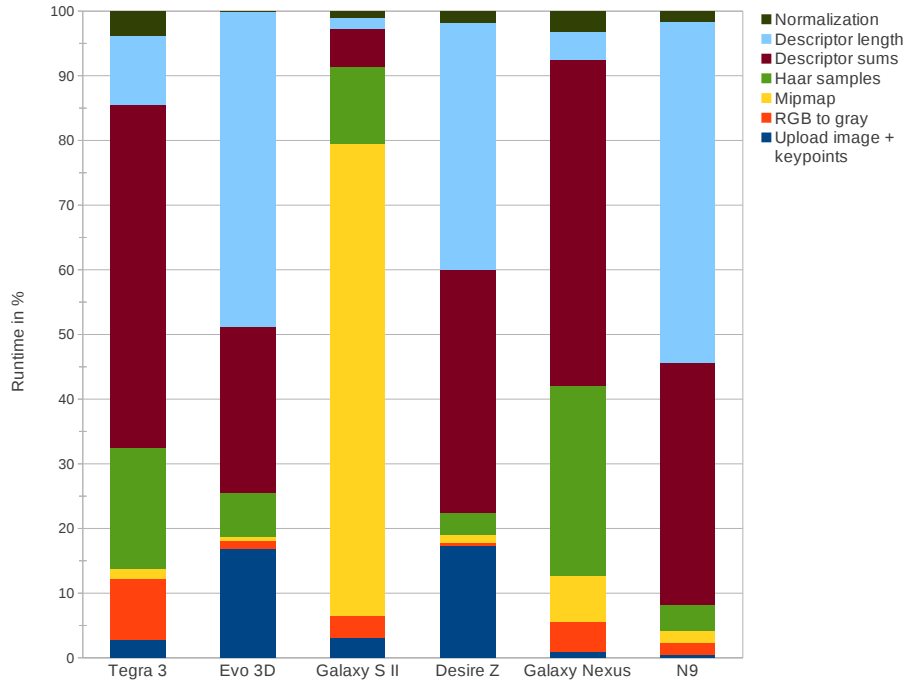

Figure 10: Relative runtime of SURF-ES for each part of the algorithm.

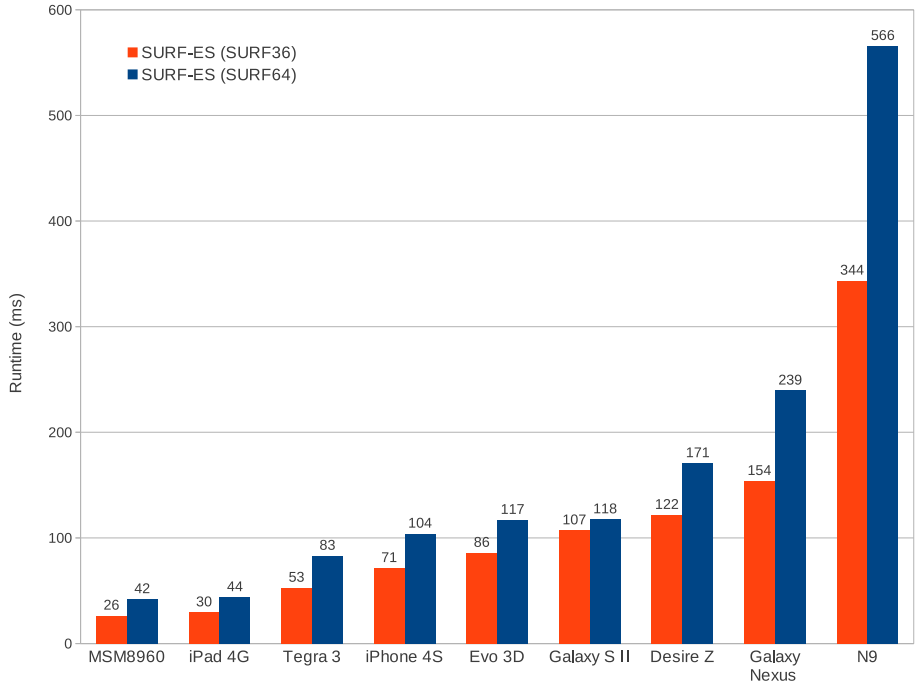

Figure 11: Runtime of SURF-ES to extract 1020 SURF-36 descriptors from a 512×384 image.

5.2 Matching Performance

We evaluated the correctness of our SURF implementation with the descriptor evaluation framework by Mikolajczyk et al. [31] using OpenCV's upright SURF implementation as reference. The evaluation has been run on the test images[19] provided by the authors. This image set comprises eight image sequences addressing different changes in image conditions. Each image sequence consists of one reference image, five images with increasing degradation and five homography matrices, which serve as ground truths for the image pairs. Because our SURF implementation is not rotationally invariant we excluded images from the evaluation that were transformed by a rotation greater than ~15°. Figure 12 shows the sequences we used in our evaluation. The "wall" sequence focuses on viewpoint changes up to ~60°. The "leuven" sequence consists of images of the same scene but with decreasing illumination. The "bikes" sequence addresses image blur and the "ubc" sequence JPEG compression artifacts.

The main purpose of this evaluation was to quantify the impact of reduced floating-point accuracy on the overall descriptor quality. Although floating-point accuracy varies significantly across the tested devices, it is the same for devices featuring a GPU of the same vendor. Therefore we did not evaluate all mobile devices, but only a subset with actually differing floating-point precisions. Table 3 shows the different precisions among the tested GPUs. The candidates for the matching performance evaluation were: the HTC Evo 3D (featuring a Qualcomm Adreno 220 GPU with a maximum precision of $p = 24$), the Samsung Galaxy Nexus (PowerVR SGX 540, $p = 23$), the Nvidia Tegra 3 (ULP GeForce, $p = 13$), and the Samsung Galaxy S II (ARM Mali-400 MP, $p = 10$).

GPU	Floating-point precision		
	`lowp`	`mediump`	`highp`
Required minimum	8	10	16
Qualcomm Adreno 2xx	24	24	24
PowerVR SGX 5xx	8	10	23
Nvidia Tegra 3	8	13	n/a
ARM Mali-400 MP	10	10	n/a

Table 3: The fragment shader floating-point precision of different GPUs as returned by `glGetShaderPrecisionFormat()`. The top row shows the GLSL ES 1.0 compliance requirements [24]. Supporting `highp` floats is optional.

For the evaluation we used the threshold-based similarity matching provided by the framework. There the Euclidean distance d_e between two descriptor vectors a and b works as the distance measure and is defined as

[19] `http://www.robots.ox.ac.uk/~vgg/research/affine/`

$$d_e = \sqrt{(a-b) \cdot (a-b)},$$

where $\cdot$ denotes the dot product. Two features are considered matching when their distance in descriptor space is below a threshold t. If both descriptors were extracted around corresponding feature positions then they constitute a correct match and otherwise a false match. The maximum number of correct matches is referred to as the number of correspondences. The possible correspondences are calculated from the output of the employed feature detector and the provided ground truth homographies by the evaluation framework.

Like Mikolajczyk et al. we used recall-precision as evaluation criterion. Recall is defined as the ratio between the number of correctly matched features and the number of possible matches:

$$\textit{recall} = \frac{\textit{number of correct matches}}{\textit{number of correspondences}}.$$

Recall is measure of descriptor *robustness*, i.e. how well corresponding features can be matched under different image conditions. The precision those correct matches are attained with is expressed as 1-precision, which relates the number of false matches to the total number of matches:

$$1 - \textit{precision} = \frac{\textit{number of false matches}}{\textit{number of correct matches} + \textit{number of false matches}}.$$

Precision is a measure of descriptor *distinctiveness*, i.e. how well the unique intensity structure of an image region is captured. Recall-precision is visualized by varying the distance threshold t during matching and plotting recall against 1-precision for each t. Increasing t therefore also increases recall but at the cost of decreasing matching precision.

Feature positions have been detected with the OpenCV SURF detector. Because SURF-ES only supports a limited number of features, we selected the 1020 features with the greatest detector response strength. To obtain one graph for each image sequence we measured recall and 1-precision across all five image pairs of a sequence and averaged the results. This process has been repeated for each mobile device and the descriptors generated by OpenCV's SURF implementation.

Figures 13 to 16 show that SURF-ES generally performs comparable to OpenCV SURF. Recall and 1-precision are distributed similarly across all test sequences with 1-precision being only slightly lower for SURF-ES on most devices.

The Tegra 3 has very low matching rates for all image sequences except the least challenging "ubc" sequence. This is due to a descriptor extraction error we were not able to identify exactly. During the summation of the

Figure 12: The first and the last image of each image sequence we used in our evaluation. From top to bottom: "wall" sequence (viewpoint), "leuven" sequence (illumination), "bikes" sequence (blur), "ubc" sequence (JPEG compression).

Haar responses parts of the descriptor sums become corrupted with random values. Since the Tegra 3 is a pre-production development kit and is the only device exhibiting this behavior among all nine tested devices, we suspect a driver-related problem.

Devices featuring GPUs with high floating-point precisions perform best as expected. The Evo 3D and the Galaxy Nexus have recall-precisions very close to OpenCV SURF and even slightly outperform the reference in some sequences ("bikes" and "ubc"). The significantly lower floating-point precision of the Mali-400 MP GPU has only a minor influence on the matching performance of the Galaxy S II for most test sequences. The "wall" sequence, however, poses a much greater challenge to the device. Its low-precision floats are less sufficient to capture the very self-similar features of this textured scene as accurately as OpenCV SURF does. Overall it can be seen that floating-point precision correlates with 1-precision, i.e. the distinctiveness of the generated descriptors. On GPUs with low floating-point precisions smaller intensity differences in the image region around a feature cannot be represented accurately enough and result in more false positives during matching.

5.3 Power Consumption

Complex applications challenge mobile devices not only in terms of their computational capabilities but also their energy efficiency. Minimizing the power consumption of smartphone applications is therefore important to prolong battery life. We tested if mobile GPGPU has an effect on energy efficiency by comparing the power consumption of SURF-ES and OpenCV SURF. Measurements have been taken with PowerTutor [51], which is available only for Android. Unfortunately, Xcode Instruments Energy Monitoring for the iPhone 4S and the iPad 4G did not yield reliable readings. OpenCV SURF's extremely long runtime on the N9 also inhibited a fair comparison. Hence, we excluded those three devices and limited the test to the six remaining Android devices.

Measurements were taken for both SURF-ES and OpenCV SURF when extracting 1020 SURF-64 features from a 1024×1024 image. Since energy measurements are not very accurate, we repeatedly measured power consumption in a loop (1000 iterations) and averaged the results. To make the measurements comparable across devices and more readable in the plot, we subtracted an averaged base line that has been measured for each device when only standard Android features were running, network connections switched off and screen on brightest level to avoid effects of other power saving techniques. Please note that these values are still approximations as we used the total power consumption and therefore cannot make any judgment of the percentual influence of CPU or GPU.

The results of our tests are shown in figure 17. It can be seen that SURF-

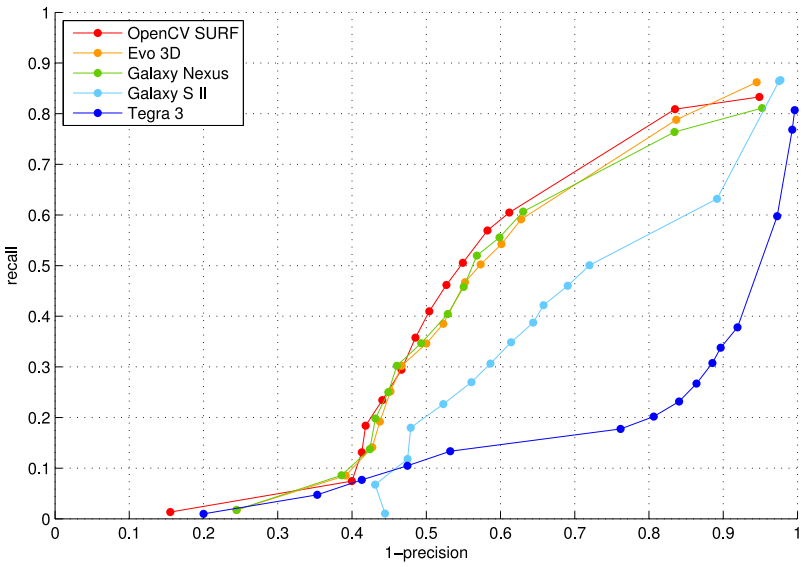

Figure 13: Matching performance of SURF-ES for the "wall" sequence (viewpoint change) on different mobile devices compared to OpenCV SURF. Note that SURF-ES produces erroneous descriptors on the Tegra 3.

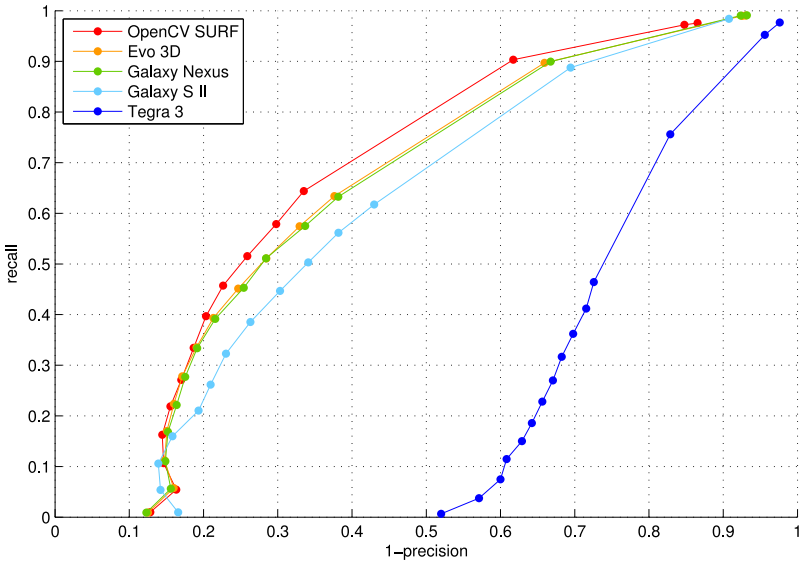

Figure 14: Matching performance of SURF-ES for the "leuven" sequence (illumination change) on different mobile devices compared to OpenCV SURF.

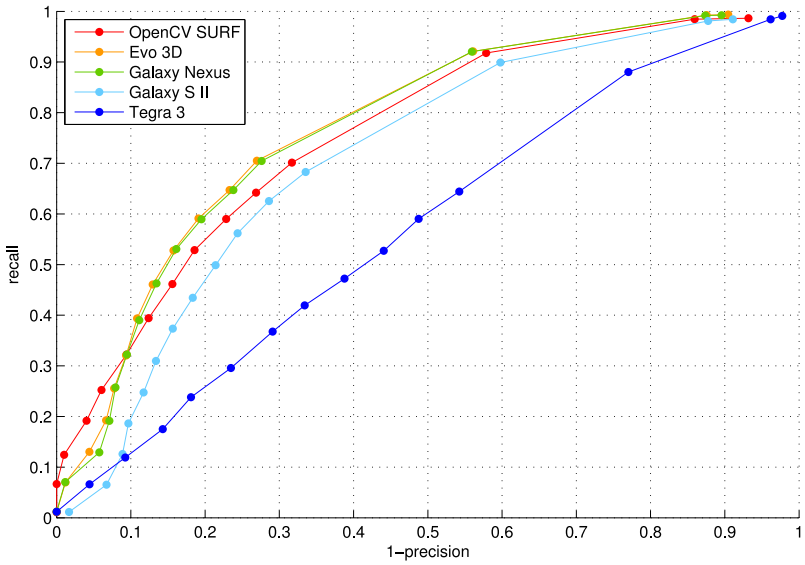

Figure 15: Matching performance of SURF-ES for the "bikes" sequence (blur) on different mobile devices compared to OpenCV SURF.

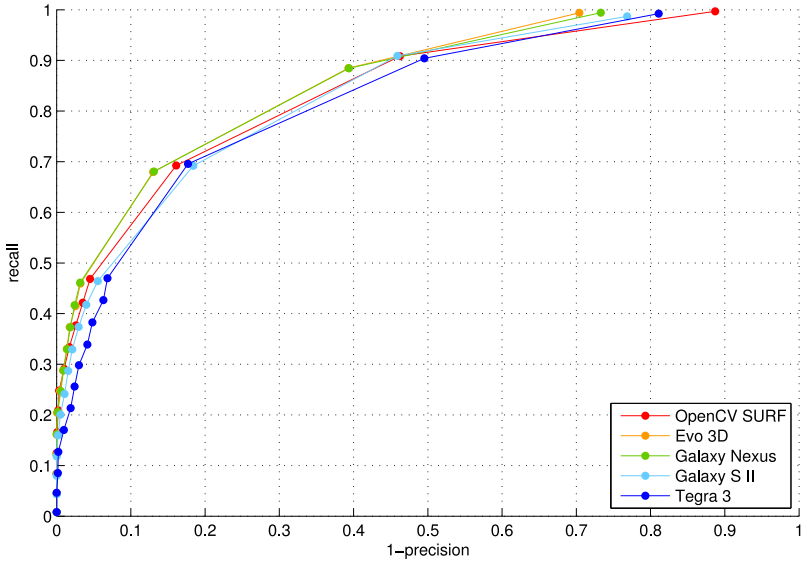

Figure 16: Matching performance of SURF-ES for the "ubc" sequence (JPEG compression) on different mobile devices compared to OpenCV SURF.

ES generally has a slightly reduced power consumption. Only on the Tegra 3 and the Galaxy Nexus SURF-ES exhibits significant power savings.

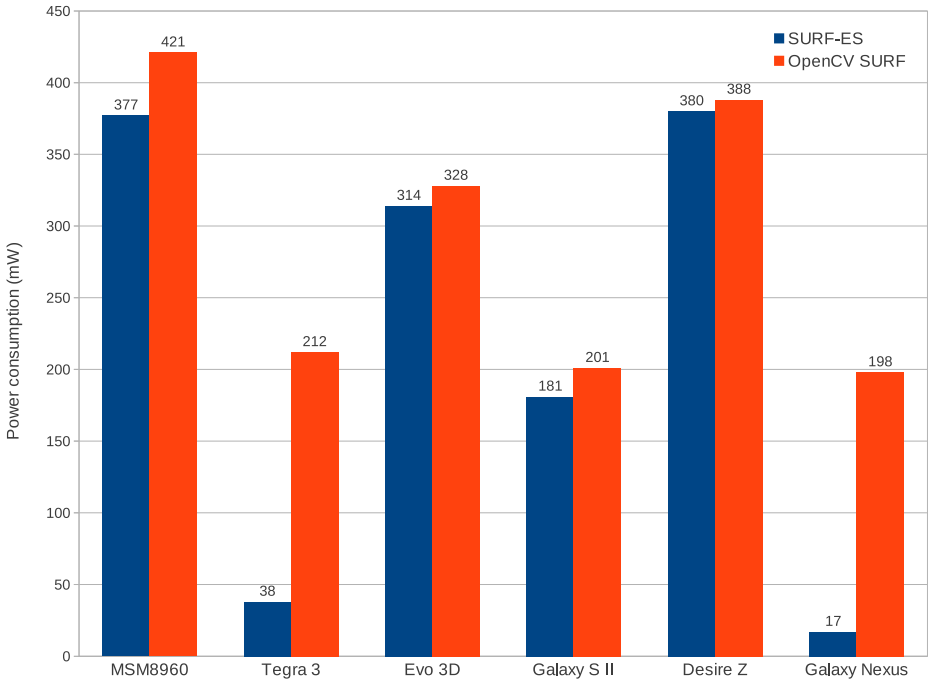

Figure 17: Power consumption of SURF-ES and OpenCV SURF. Note that only Android devices have been included in this test to ensure a fair comparison.

5.4 Discussion

From the results of the runtime performance tests we can see that Moore's law also applies to mobile GPUs, indicating their increasing GPGPU fitness. SURF-ES is able to extract about 1000 SURF descriptors from small to medium sized images with 2 to 20 Hz, depending on the mobile device. When applied in a practical context like the detection phase of a mobile AR system with NFT, usually much less feature positions need to be described. About 300 to 400 features, for example, can be described at 10 to 50 Hz on all tested devices except the N9. Note that working on small images and using the faster SURF-36 descriptor further decreases the runtime of SURF-ES. We therefore assume that SURF-ES leaves enough frame time for other NFT-related tasks like image acquisition, feature detection, matching, and pose estimation, as long as they are also tailored to mobile platforms. This is especially true for state of the art multi-core GPUs as featured in the MSM8960 and the iPad 4G.

Our comparison of SURF-ES against a standard implementation of the SURF descriptor has shown that GPGPU techniques can indeed improve the performance of complex operations like SURF descriptor extraction even on mobile GPUs. Although the attained speed gain is not as significant as on desktop PCs (20×) and varies across devices (2–14×), the trend is clearly for mobile GPUs to offer increasing acceleration as they mature. We are aware that our comparison to a sequential CPU implementation is to some degree unfair since many mobile devices already feature multi-core CPUs. This is due to the fact that the only open-source multi-core implementation of SURF known to us (namely Parallel SURF [15]) uses OpenMP, which is neither supported by Android nor by iOS.

Another advantage of SURF-ES over a CPU implementation is that it can greatly reduce CPU utilization by offloading work onto the GPU. Note that there is usually a low GPU load during the detection phase of an NFT system as the virtual augmentations are not being rendered until a pose has been estimated. SURF-ES therefore would not compete with other GPU tasks when applied in such a context. The measured CPU loads indicate that several mobile devices assign some render tasks to the mobile CPU. Particularly the Galaxy S II appears to create mipmaps with the help of its CPU. Because SURF-ES triggers mipmap generation for a texture that has been rendered to, this would explain the disproportionately long runtime of this step on the Galaxy S II. The texture in question has to be read-back from video memory, mipmapped and then uploaded again. This would also explain the Galaxy S II's lack of scalability as the process becomes memory bound rather than processing unit bound. The number of keypoints hardly influences performance because the real bottleneck is mipmap creation accounting for over 70% of the overall runtime. On the contrary, increasing the size of the input image drastically increases runtime, because downloading large segments of video memory causes severe performance hits.

On the other tested mobile devices we observed texture fetches as the main bottleneck. Consequently, shader programs with low arithmetic intensity perform worst. The shaders to compute a descriptor bin entry and length take up 50 to 90% of the runtime, due to the relatively large number of fetches (25 and 64) and few arithmetic operations that can be scheduled during texture fetch latency. Desktop GPGPU overcomes such bottlenecks by optimizing those gather operations with parallel reductions. But because reduction operations significantly increase the number of fragments, they are not feasible on mobile GPUs, which feature only a fraction of the amount of shader units available on desktop GPUs. Other methods use mipmaps to compute such sums [11]. Because of the high dynamic range of SURF's descriptor bin values, this is only sensible with float textures. However, the OpenGL ES 2.0 specification for the extension GL_OES_texture_float does not require support for rendering to float

textures. Therefore, we encountered problems on mobile devices when we tried to implement this optimization. This alone would be a great addition for mobile GPGPU in future OpenGL ES APIs.

The matching tests have shown that the SURF descriptors generated by OpenCV SURF and by GPUs with lower floating-point precision provide comparable distinctiveness as long as the scene's features are not self-similar. Among all test sequences does the "wall" sequence, addressing invariance to perspective changes, expose the weakest matching performance. Lowe [29] pointed out that this is caused by gradient samples lying close to subregion borders, as they are likely to shift positions between descriptor bins on perspective changes. Both SURF-ES and OpenCV do not account for that. Agrawal et al. [1] proposed a GPGPU-friendly way to cope with unstable sample-to-bin associations. By overlapping the subregions and applying a second Gaussian weighting centered at the subregion center, border samples also leave a signature in neighboring descriptor bins depending on their distance to the subregion border. This method is easier to implement on GPUs than a bilinear interpolation approach (e.g. as in SIFT and Pan-o-matic SURF). On the downside it drastically increases the workload of the sample summation step by tripling the number of texture fetches and cannot be optimized with mipmaps. As this is a trade-off between computational complexity and robustness of the descriptor, we opted in favor of runtime performance and did not implement this method.

As our tests have shown, GPGPU generally only provides little advantages in terms of energy efficiency for the time being. Significant power savings can only be achieved when mobile devices feature power save modes. To this date, this is not a common feature on current SoCs.

6 Conclusion

In this thesis we presented a GPGPU implementation of the SURF descriptor specifically tailored to the limitations of mobile devices. Our main adjustments of the SURF descriptor extraction in respect to the original algorithm were the utilization of mipmaps for scale-aware, subpixel-accurate Haar wavelet sampling and the implementation of a fixed-point encoding dynamically adjusting to the GPU's floating-point precision. We extensively evaluated our implementation across different hardware and software platforms in terms of runtime behavior, matching performance, and energy efficiency.

Our experimental results show that using GPGPU for feature description is a feasible way to speed up computation even on a mobile device. Especially, with newer devices sporting more and more computational parallelism these methods will become an important advancement for doing AR with ever larger image sizes achieving better performance and robust-

ness. We also have shown that the limitations of mobile GPUs in terms of computational accuracy and storage precision can be alleviated in such a way as to enable a feature matching performance comparable to that of a CPU implementation. Additionally, we could show that GPGPU potentially yields significant power savings when manufacturers and vendors incorporate more advanced power save modes into their SoCs.

During development we experienced cross-platform GPGPU on mobile devices as a cumbersome undertaking as we were battling bugs in premature drivers and incompatibilities between devices arising from differing interpretations of the OpenGL ES 2.0 standard. The need for high-level GPGPU APIs like OpenCL becomes increasingly pressing when future mobile GPUs should be utilizable to their full potential and in a productive way.

6.1 Future Work

Our prototypic implementation still leaves room for further optimization and extension. In order to improve the efficiency of the Haar sample summation and descriptor normalization, the textures holding the unnormalized descriptors could be reshaped to store each descriptor in a quadratic texel block rather than a column. This would spatially localize texture fetches and therefore increase texture cache utilization at the cost of more complex index calculations.

When rendering to float textures becomes reliable (i.e. a requirement of the extension or at least common practice between vendors) the summation of the Haar samples could be re-implemented with the help of mipmaps. We expect the higher storage requirement and bandwidth load of float textures as well as the cost of generating an additional mipmap to be amortized by several improvements this optimization provides:

- The 4-bin descriptor of a subregion could be stored in a single texel and only a quarter of the fragments need to be generated for the summation step.
- Mipmapping would drastically reduce the number of necessary texture fetches per fragment from 64 to 1 at optimal cache efficiency.
- As pointed out in [11], the recursive generation of mipmaps would also increase the numerical stability of the summation.
- Our floating-point encoding would become obsolete, thus reducing the instruction count in several shaders.

Additionally, the calculation of the descriptor lengths for normalization could be implemented with mipmapping. However, it is not yet foreseeable when float textures actually will be renderable on mobile devices.

Feature detection: When applied in a real-time context, SURF-ES requires a faster feature detector than SURF's Fast Hessian. As already mentioned, we do not consider a GPGPU implementation feasible on mobile devices, since compaction of the detector responses to a list requires multi-pass reduction operations on the GPU or reading back the response matrix for processing with the CPU. Therefore, we suggest employing one of the already existing scale-aware modifications of the very efficient FAST detector, viz. [26], [44], or [48]. Additionally, applying a measure of detector response strength (e.g. as in [48] or [38]) would help to meet the input limitations of SURF-ES by eliminating weak features.

Orientation assignment: SURF-ES does not align its sampling grid to feature orientation and is therefore not rotationally invariant. Once again, we consider the original approach to orientation estimation of SURF unfeasible on both the mobile CPU and GPU, due to its computational complexity. Still, SURF-ES could be extended with a reproducible orientation assignment in different ways. Herling et al. [17] proposed several optimizations for SURF's orientation estimation reducing its runtime while retaining a comparable matching performance. In case the employed feature detector extracts corners, the intensity centroid of the region around a corner can be computed to determine an orientation vector [34]. Intensity centroids are very efficient and have been successfully applied in [38].

Feature matching on mobile GPUs: To get the full advantage of GPGPU feature description on mobile GPUs, one has to perform feature matching on the GPU as well. Reading back a matrix of several hundred SURF-64 descriptors stalls the graphics pipeline for a disproportionately long time, whereas the read-back of a vector containing only the match wins can be performed much more efficiently. To this end, GPGPU matching must be sufficiently fast of course. Thus, we briefly tested a naïve GLSL ES implementation of matrix multiplication on the Tegra 3 and Galaxy S II. The aim was to get a general idea of how efficiently mobile GPUs can compute similarities between two sets of features using the dot product as a similarity measure. Multiplying two 300×64 matrices required an average runtime of about 55 ms on both devices. In addition, each column of the resulting similarity matrix has to be scanned for the maximum value to determine the match win in a second render pass.

This straightforward approach to matching can be optimized by canceling the dot product calculation between two feature descriptors, in case the dot product between their centers (holding most descriptor energy) does not exceed a threshold [17]. Using the smaller SURF-36 descriptor additionally reduces the complexity of matching. If the feature detector determines whether a feature is light or dark (i.e. lighter or darker than the background

surrounding it), then the dot product between a light and a dark feature can be skipped entirely, since they will never match.

The final pass to determine the match wins can also perform efficient outlier removal with little additional cost. In a similar way as in [29], the similarity ratio of the second-closest to the closest match helps distinguishing between "good" and "bad" matches when the ratio exceeds a certain threshold, i.e. the certainty of matching is not sufficient.

Overall, we think that exploiting the mobile GPU for descriptor extraction and matching can be a fruitful approach to the detection problem of mobile AR with natural features. Although older mobile GPUs do not satisfy the real-time requirement of NFT, newer products like the Adreno 225 and the PowerVR SGX543MP4 can measure up to that. This shows that our GPGPU approach is highly scalable and even faster runtimes can be expected on future mobile GPUs.

References

[1] Motilal Agrawal, Kurt Konolige, and Morten Rufus Blas. CenSurE: Center Surround Extremas for Realtime Feature Detection and Matching. In David A. Forsyth, Philip H. S. Torr, and Andrew Zisserman, editors, *Proceedings of the* 10^{th} *European Conference on Computer Vision*, volume 5305 of *Lecture Notes in Computer Science*, pages 102–115. Springer, 2008. 5.4

[2] Pavel Babenko and Mubarak Shah. MinGPU: A Minimum GPU Library for Computer Vision. *Journal of Real-Time Image Processing*, 3(4):255–268, 2008. 3.4

[3] Johannes Bauer, Niko Sünderhauf, and Peter Protzel. Comparing Several Implementations of Two Recently Published Feature Detectors. In *Proceedings of the International Conference on Intelligent and Autonomous Systems*, 2007. 2.3

[4] Herbert Bay, Andreas Ess, Tinne Tuytelaars, and Luc Van Gool. Speeded-Up Robust Features (SURF). *Computer Vision and Image Understanding*, 110(3):346–359, 2008. (document), 2.1, 2.3, 3.1.1, 2, 3.1.1

[5] Eric A. Bier, Maureen C. Stone, Ken Pier, William Buxton, and Tony D. DeRose. Toolglass and Magic Lenses: The See-Through Interface. In *Proceedings of the* 20^{th} *annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '93, pages 73–80, New York, NY, USA, 1993. ACM. 1

[6] M. Bordallo López, J. Hannuksela, O. Silvén, and M. Vehviläinen. Graphics Hardware Accelerated Panorama Builder for Mobile Phones. In *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, volume 7256 of *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, pages 72560D–1–72560D–9, February 2009. 2.2.1

[7] M. Bordallo López, H. Nykänen, J. Hannuksela, O. Silvén, and M. Vehviläinen. Accelerating Image Recognition on Mobile Devices Using GPGPU. In *Proceedings of SPIE. Parallel Processing for Imaging Applications*, volume 7872, pages 78720R–78720R–10, 2011. 2.2.1

[8] Michael Calonder, Vincent Lepetit, Christoph Strecha, and Pascal Fua. BRIEF: Binary Robust Independent Elementary Features. In *Proceedings of the* 11^{th} *European Conference on Computer vision: Part IV*, ECCV'10, pages 778–792, Berlin, Heidelberg, 2010. Springer-Verlag. 2.1, 2.3

[9] Nelson L. Chang, Feng Tang, Suk Hwan Lim, and Hai Tao. A Novel Feature Descriptor Invariant to Complex Brightness Changes. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2009)*, pages 2631–2638. IEEE, June 2009. 2.3

[10] Wei-Chao Chen, Yingen Xiong, Jiang Gao, N. Gelfand, and R. Grzeszczuk. Efficient Extraction of Robust Image Features on Mobile Devices. In *Proceedings of the* 6^{th} *IEEE and ACM International Symposium on Mixed and Augmented Reality (ISMAR 2007)*, pages 287–288, November 2007. 2.3.1

[11] Nico Cornelis and Luc Van Gool. Fast Scale Invariant Feature Detection and Matching on Programmable Graphics Hardware. *Computer Vision and Pattern Recognition Workshop*, pages 1–8, 2008. 2.3, 2.3, 2.3.1, 4.2, 4.3, 5.4, 6.1

[12] Nvidia Corporation. Tegra 3 Specifications, 2011. `http://www.nvidia.com/object/tegra-3-processor.html` (Last accessed: June 3^{rd} 2012). 1.2

[13] Mark Fiala. ARTag, a Fiducial Marker System Using Digital Techniques. In *Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, volume 2 of *CVPR '05*, pages 590–596, Washington, DC, USA, 2005. IEEE Computer Society. 1.1

[14] Iryna Gordon and David G. Lowe. What and Where: 3D Object Recognition with Accurate Pose. *Lecture Notes in Computer Science*, 4170/2006(1):67–82, 2004. 2.1

[15] David Gossow, Peter Decker, and Dietrich Paulus. An Evaluation of Open Source SURF Implementations. In Javier Ruiz-del Solar, Eric Chown, and Paul G. Plöger, editors, *RoboCup 2010: Robot Soccer World Cup XIV. Papers from the* 14^{th} *annual RoboCup International Symposium, Singapore, June 25, 2010*, volume 6556 of *Lecture Notes in Computer Science*, pages 169–179. Springer, 2010. 2.3.1, 3.3, 5.4

[16] Mark Harris and Dominik Göddeke. GPGPU Programming, 2002. `http://gpgpu.org/developer` (Last accessed: June 3^{rd} 2012). 2.2

[17] Jan Herling and Wolfgang Broll. An Adaptive Training-Free Feature Tracker for Mobile Phones. In *Proceedings of the* 17^{th} *ACM Symposium on Virtual Reality Software and Technology*, VRST '10, pages 35–42, New York, NY, USA, 2010. ACM. 2.1, 2.3, 2.3.1, 3.3, 6.1, 6.1

[18] S. Heymann, K. Müller, A. Smolic, B. Fröhlich, and T. Wiegand. SIFT Implementation and Optimization for General-Purpose GPU.

In *Proceedings of the International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision*, pages 317–322, Plzen, Czech Republic, February 2007. 2.3.1

[19] Daniel Horn. Stream Reduction Operations for GPGPU Applications. In Matt Pharr and Randima Fernando, editors, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley Professional, March 2005. 3.2

[20] Imagination Technologies Ltd. *PowerVR Series 5 SGX Architecture Guide for Developers*, November 2011. `http://www.imgtec.com/powervr/insider/docs/PowerVR%20Series%205.SGX%20Architecture%20Guide%20for%20Developers.1.0.13.External.pdf` (Last accessed: June 3^{rd} 2012). 3.2

[21] Imagination Technologies Ltd. *PowerVR Performance Recommendations*, February 2012. `http://www.imgtec.com/powervr/insider/docs/PowerVR.Performance%20Recommendations.1.0.28.External.pdf` (Last accessed: June 3^{rd} 2012). 3.2

[22] Hirokazu Kato and Mark Billinghurst. Marker Tracking and HMD Calibration for a Video-Based Augmented Reality Conferencing System. In *Proceedings of the 2^{nd} IEEE and ACM International Workshop on Augmented Reality*, IWAR '99, pages 85–94, Washington, DC, USA, 1999. IEEE Computer Society. 1.1

[23] Guy-Richard Kayombya. SIFT Feature Extraction on a Smartphone GPU Using OpenGL ES2.0. Master's thesis, Massachusetts Institute of Technology. Dept. of Electrical Engineering and Computer Science, June 2010. 2.3.1

[24] The Khronos Group Inc. *The OpenGL ES Shading Language*, May 2009. `http://www.khronos.org/registry/gles/specs/2.0/GLSL_ES_Specification_1.0.17.pdf` (Last accessed: June 3^{rd} 2012). 4.4, 4.5, 4.5, 3

[25] Eugene Khvedchenya. Feature Descriptor Comparison Report. `http://computer-vision-talks.com/2011/08/feature-descriptor-comparison-report/` (Last accessed: June 3^{rd} 2012), August 2011. 2.3

[26] G. Klein and D. Murray. Parallel Tracking and Mapping for Small AR Workspaces. In *Proceedings of the 6^{th} IEEE and ACM International Symposium on Mixed and Augmented Reality (ISMAR 2007)*, pages 225–234, November 2007. 1.2, 2.1, 2.3, 6.1

[27] G. Klein and D. Murray. Parallel Tracking and Mapping on a Camera Phone. In *Proceedings of the* 8^{th} *IEEE International Symposium on Mixed and Augmented Reality (ISMAR 2009)*, pages 83–86, October 2009. 2.1

[28] João P. Lima, Veronica Teichrieb, Judith Kelner, and Robert W. Lindeman. Standalone Edge-Based Markerless Tracking of Fully 3-Dimensional Objects for Handheld Augmented Reality. In *Proceedings of the* 16^{th} *ACM Symposium on Virtual Reality Software and Technology*, VRST '09, pages 139–142, New York, NY, USA, 2009. ACM. 2.1, 2.3

[29] David G. Lowe. Distinctive Image Features from Scale-Invariant Keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004. 2.1, 2.3, 5.4, 6.1

[30] K. Mikolajczyk, T. Tuytelaars, C. Schmid, A. Zisserman, J. Matas, F. Schaffalitzky, T. Kadir, and L. Van Gool. A Comparison of Affine Region Detectors. *International Journal of Computer Vision*, 65(1-2):43–72, November 2005. 5.1

[31] Krystian Mikolajczyk and Cordelia Schmid. A performance evaluation of local descriptors. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27(10):1615–1630, October 2005. 5.2

[32] Aaftab Munshi, Dan Ginsburg, and Dave Shreiner. *OpenGL ES 2.0 Programming Guide*. Addison-Wesley Professional, Upper Saddle River, NJ, 1^{st} edition, July 2008. 3

[33] Christoph Oberhofer, Jens Grubert, and Gerhard Reitmayr. Natural Feature Tracking in JavaScript. *Virtual Reality Conference, IEEE*, 0:113–114, march 2012. 2.1

[34] Paul L. Rosin. Measuring Corner Properties. *Computer Vision and Image Understanding*, 73(2):291–307, 1999. 6.1

[35] E. Rosten, R. Porter, and T. Drummond. Faster and Better: A Machine Learning Approach to Corner Detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(1):105–119, January 2010. 2.3

[36] Edward Rosten and Tom Drummond. Fusing Points and Lines for High Performance Tracking. In *IEEE International Conference on Computer Vision*, volume 2, pages 1508–1511, October 2005. 2.3

[37] Edward Rosten and Tom Drummond. Machine Learning for High-Speed Corner Detection. In *European Conference on Computer Vision*, volume 1, pages 430–443, May 2006. 2.3

[38] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski. ORB: An Efficient Alternative to SIFT or SURF. In *International Conference on Computer Vision*, Barcelona, November 2011. 2.3, 2.3, 6.1, 6.1

[39] Thorsten Scheuermann and Justin Hensley. Efficient Histogram Generation Using Scattering on GPUs. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, I3D '07, pages 33–37, New York, NY, USA, 2007. ACM. 2.3

[40] Shubhabrata Sengupta, Aaron Lefohn, and John D. Owens. A Work-Efficient Step-Efficient Prefix-Sum Algorithm. In *Proceedings of the 2006 Workshop on Edge Computing Using New Commodity Architectures*, pages 26–27, May 2006. 3.2

[41] N. Singhal, In Kyu Park, and Sungdae Cho. Implementation and Optimization of Image Processing Algorithms on Handheld GPU. In *Proceedings of the* 17^{th} *IEEE International Conference on Image Processing (ICIP 2010)*, pages 4481–4484, September 2010. 2.2.1

[42] Timothy B. Terriberry, Lindley M. French, and John Helmsen. GPU Accelerating Speeded-Up Robust Features. In *Proceedings of the* 4^{th} *International Symposium on 3D Data Processing, Visualization and Transmission (3DPVT 2008)*, pages 355–362, Atlanta, Georgia, June 2008. 2.3.1, 1

[43] Paul Viola and Michael Jones. Rapid Object Detection using a Boosted Cascade of Simple Features. *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 1:511, 2001. 2.3, 3.1.1

[44] D. Wagner, G. Reitmayr, A. Mulloni, T. Drummond, and D. Schmalstieg. Real-Time Detection and Tracking for Augmented Reality on Mobile Phones. *IEEE Transactions on Visualization and Computer Graphics*, 16(3):355–368, May-June 2010. 1.2, 2.1, 2.3, 2.3, 2.3, 2.3.1, 6.1

[45] Daniel Wagner, Tobias Langlotz, and Dieter Schmalstieg. Robust and Unobtrusive Marker Tracking on Mobile Phones. In *Proceedings of the* 7^{th} *IEEE/ACM International Symposium on Mixed and Augmented Reality*, ISMAR '08, pages 121–124, Washington, DC, USA, 2008. IEEE Computer Society. 1.1

[46] Daniel Wagner and Dieter Schmalstieg. ARToolKitPlus for Pose Tracking on Mobile Devices. In *Proceedings of the* 12^{th} *Computer Vision Winter Workshop*, CVWW '07, pages 139–146, February 2007. 1.1

[47] Yi-Chu Wang, Sydney Pang, and Kwang-Ting Cheng. A GPU-Accelerated Face Annotation System for Smartphones. In *Proceedings of the International Conference on Multimedia*, MM '10, pages 1667–1668, New York, NY, USA, 2010. ACM. 2.2.1

[48] Achim Weimert, Xueting Tan, and Xubo Yang. Natural Feature Detection on Mobile Phones with 3D FAST. *The International Journal of Virtual Reality*, 9(4):29–34, December 2010. 2.3, 2.3.1, 6.1

[49] H. Wuest, F. Vial, and D. Strieker. Adaptive Line Tracking with Multiple Hypotheses for Augmented Reality. In *Proceedings of the* 4^{th} *IEEE and ACM International Symposium on Mixed and Augmented Reality (ISMAR 2005)*, pages 62–69, October 2005. 2.3

[50] Christian Wutte and Georg Wagner. Computer Vision on Mobile Phone GPUs, October 2009. Bachelor's thesis, Technische Universität Graz. 2.3.1

[51] Lide Zhang, Birjodh Tiwana, Zhiyun Qian, Zhaoguang Wang, Robert P. Dick, Zhuoqing Morley Mao, and Lei Yang. Accurate Online Power Estimation and Automatic Battery Behavior Based Power Model Generation for Smartphones. In *Proceedings of the* 8^{th} *IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES/ISSS '10, pages 105–114, New York, NY, USA, 2010. ACM. 5.3

[52] Gernot Ziegler, Art Tevs, Christian Theobalt, and Hans-Peter Seidel. On-the-fly Point Clouds through Histogram Pyramids. In *Proceedings of the* 11^{th} *International Fall Workshop on Vision, Modeling and Visualization 2006 (VMV 2006)*, pages 137–144, Aachen, Germany, 2006. European Association for Computer Graphics (Eurographics), Aka. 2.3, 2.3.1, 3.2