A GPGPU Accelerated Descriptor for Mobile Devices

Robert Hofmann†
University of Koblenz-Landau
Hartmut Seichter‡
Graz University of Technology
Gerhard Reitmayr‡
Graz University of Technology

Abstract

We present a modified upright SURF feature descriptor for mobile phone GPUs. Our implementation called uSURF-ES is multiple times faster than a comparable CPU variant on the same device. Our 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.

Index Terms: H.5.1 [INFORMATION INTERFACES AND PRESENTATION]: Multimedia Information Systems—Artificial, augmented, and virtual realities; K.7.m [IMAGE PROCESSING AND COMPUTER VISION]: Segmentation—Edge and feature detection

1 Introduction

Local feature detection and description in model or SLAM-based tracking methods, commonly referred to as natural feature tracking (NFT), are essential for modern Augmented Reality (AR) applications. 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 [5, 8].

The computational power of a mobile device is mainly restricted by its battery capacity. Yet, smartphones have become increasingly faster. Beside the CPUs, FPGA and SIMD units such as ARM NEON, more and more powerful GPUs are necessary to support the ever larger screen resolutions and demanding visual appearance of mobile operating systems. This provides AR application with an opportunity to leverage this additional processing power.

In natural feature based AR applications most processing time is spent on object detection and only a small fraction on rendering. Therefore it seems beneficial to investigate the potential of mobile GPUs for detection to free up resources. More so, algorithms for describing trackable features in an image are highly data-parallel pixel operations. However, due to the limitations of mobile devices compared to the desktop, research on GPGPU targeting these has not gained much traction. This is particularly true for mobile application scenarios such as AR, requiring interactive or real-time performance.

Our work is based on the the upright variant of SURF[1] as we intend to extend the algorithm with a alternative rotation assignment. Research about NFT can be found in [2, 3, 8, 7] ranging from CPU bound implementations that run in realtime to implementations that are far from realtime or not applicable to AR scenarios.

We contribute to this body of work with insights into GPGPU for local feature description with focus on mobile AR applications.

2 Implementation

uSURF-ES is written in C/C++ and targeting OpenGL ES 2.0. It uses a shader generator to adjust for variations in GPU capabilities. Features in the input image are extracted with SURF in OpenCV 2.3.1. Then uSURF-ES is initialized with the desired number of descriptor bins and a fixed image size and generates textures and shader programs once and reuses them each time new input data is handed in.

2.1 Sampling of Haar Responses

The SURF descriptor is based on 400 regularly spaced samples of Haar wavelets as described in [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 tex (i.e. 51×20 sample blocks). Each texels corresponds to a fragment generated by rendering an equally sized quad.

[1] suggests an integral image for efficient computation of arbitrarily sized box filters. This approach is not feasible on mobile GPUs due to multiple dislocated texture operations. We opted for a mipmap-ing approach instead as proposed in [4]. For that we divide the area covered by a Haar wavelet into four equally sized quadrants. The intensity sum of each quadrant can then be read from a lower mipmap level. Another advantage is that bilinear interpolation between texels provides sub-pixel accuracy. Fetching from the correct mipmap level and trilinear interpolation are all done automatically by the graphics hardware and no additional shader instructions are required.

2.2 Descriptor Formation

uSURF-ES generates one fragment for each SURF bin to sum up the Haar responses of a subregion. This is achieved by rendering a quad of size $n \times 64$ pix, $n$ 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.

<table>
<thead>
<tr>
<th>Device</th>
<th>GPU, CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Ubuntu 10.10 “Maverick Meerkat”)</td>
<td>Intel Core 2 Quad (4 cores, 2.66 GHz)</td>
</tr>
<tr>
<td>Qualcomm Snapdragon S4 MSM8960 (Android 4.0.3 “Ice Cream Sandwich”)</td>
<td>Qualcomm Adreno 225</td>
</tr>
<tr>
<td>Apple iPad 4G (iOS 5.1)</td>
<td>PowerVR SGX543MP4</td>
</tr>
<tr>
<td>NVIDIA Tegra 3 (Android 3.2.1 “Honeycomb”)</td>
<td>ARM Cortex-A9 (4 cores, 1.4 GHz)</td>
</tr>
<tr>
<td>Apple iPhone 4S (iOS 4.0.1)</td>
<td>PowerVR SGX543MP4</td>
</tr>
<tr>
<td>Samsung Galaxy S II (Android 2.3.3 “Gingerbread”)</td>
<td>Qualcomm Snapdragon (2 cores, 1.2 GHz)</td>
</tr>
<tr>
<td>Samsung Galaxy Nexus (Android 2.3.3 “Gingerbread”)</td>
<td>ARM Cortex-A9 (2 cores, 1.2 GHz)</td>
</tr>
<tr>
<td>HTC Evo 3D (Android 4.0.2 “Ice Cream Sandwich”)</td>
<td>ARM Cortex-A9 (2 cores, 1.2 GHz)</td>
</tr>
<tr>
<td>Nokia N9 (MeeGo 1.2 “Harmattan”)</td>
<td>PowerVR SGX543MP4</td>
</tr>
</tbody>
</table>

Table 1: Specifications of the tested devices.
instead of, on mobile GPUs, slow reductions we implemented a simpler 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 use an efficient method proposed in [4] 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 $[-1671.8, 1671.8]$, however in practice the mean is relatively close to 0 with very low variance. To cover such a wide range of magnitudes accurately we employ a more sophisticated encoding. Therefore the sum values are encoded into all four channels of a texel as described in the next section 2.3.

### 2.3 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, they can propagate results to subsequent shader passes 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.

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 with an 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 it to the shaders as a uniform.

### 3 Performance

To assess our GPGPU approach to feature descriptor extraction, we tested our implementation on a wide range of mobile devices and on a desktop PC using the PowerVR OpenGL ES 2.0 emulation. We used images from the testset in [6] to make results comparable.

Measurements include uploading the input image and the list of keypoints to video memory, but exclude initial image loading and keypoint detection with OpenCV as well as downloading the resulting descriptors from video memory. Just as expected newer mobile devices with multi-core GPUs perform better than older devices.

Compared to the same implementation on a desktop GPU the performance difference ranges from less than $10 \times$ (MSM8960) to almost $120 \times$ (N9).

In order to determine the speed-up our GPU implementation we compared uSURF-ES to the CPU implementation of upright SURF in OpenCV. The runtime differences between the GPU and CPU of each mobile device, when extracting 1020 SURF descriptors from a $512 \times 384$ image are shown in figure 1. Devices with a Cortex-A9 CPU 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 approximately $10 \times$. The Scorpion GPUs on the HTC Evo 3D and Desire Z are significantly slower, which leads to an almost $8 \times$ speed-up. Same for the iPhone 4S (over $7 \times$) and the iPad 4G (14$\times$) with comparatively slow CPUs but faster GPUs. It should be noted that OpenCV SURF is not multithreaded on ARM CPUs and numbers therefore reflect a per-core performance.

### 4 Conclusion and Future Work

We presented a GPGPU implementation of the SURF descriptor, called uSURF-ES, that is 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-awareness, subpixel-accurate Haar wavelet sampling and the implementation of a fixed-point encoding dynamically adjusting to the GPU’s floating-point precision.

Our 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. Overall, exploiting the mobile GPU for descriptor extraction and matching is a promising approach to the detection problem of mobile AR.

### Acknowledgements

This work was conducted at the Christian Doppler Laboratory for Handheld Augmented Reality.

### References