Workload Analysis and Efficient OpenCL-based Implementation of SIFT Algorithm on a Smartphone

Guohui Wang, Blaine Rister, and Joseph R. Cavallaro
Department of Electrical and Computer Engineering, Rice University, Houston, Texas 77005
Email: {wgh, blaine.rister, cavallar}@rice.edu

***Abstract*—Feature detection and extraction are essential in computer vision applications such as image matching and object recognition. The Scale-Invariant Feature Transform (SIFT) algorithm is one of the most robust approaches to detect and extract distinctive invariant features from images. However, high computational complexity makes it difficult to apply the SIFT algorithm to mobile applications. Recent developments in mobile processors have enabled heterogeneous computing on mobile devices, such as smartphones and tablets. In this paper, we present an OpenCL-based implementation of the SIFT algorithm on a smartphone, taking advantage of the mobile GPU. We carefully analyze the SIFT workloads and identify the parallelism. We implemented major steps of the SIFT algorithm using both serial C++ code and OpenCL kernels targeting mobile processors, to compare the performance of different workflows. Based on the profiling results, we partition the SIFT algorithm between the CPU and GPU in a way that best exploits the parallelism and minimizes the buffer transferring time to achieve better performance. The experimental results show that we are able to achieve 8.5 FPS for keypoints detection and 19 FPS for descriptor generation without reducing the number and the quality of the keypoints. Moreover, the heterogeneous implementation can reduce energy consumption by 41% compared to an optimized CPU-only implementation.**

***Index Terms*—SIFT, GPU, mobile SoC, CPU-GPU algorithm partitioning, OpenCL.**

I. Introduction

Recent advances in the computational capabilities of mobile processors have made possible a wide range of computer vision applications on smartphone and tablet platforms, such as image stitching, object recognition, and augmented reality. Efficient feature detection and extraction are essential building blocks for all of these tasks. The scale-invariant feature transform (SIFT) algorithm can produce distinctive keypoints and feature descriptors [1], and has been considered one of the most robust local feature extraction algorithms [2]. Although many alternative or high speed approximation algorithms to SIFT have been proposed, such as SURF [3], the SIFT algorithm remains in wide use and is attracting more attention.

Many research efforts have been made in prior works to design and optimize high speed SIFT implementations. Most of them use customized hardware or high performance workstations due to the high complexity of the SIFT algorithm [4]. General-purpose computing on graphics processing units (GPGPU) has also been used to speed up the processing of SIFT [5, 6]. However, most of the design and optimization techniques in these prior works are not practical for mobile devices, due to their high computational complexity and memory usage. There are only a few works targeting SIFT implementations on mobile platforms [7, 8], in which only part of the SIFT algorithm is accelerated, or feature accuracy is traded for processing speed.

Throughout the past several years, the power of mobile processors for parallel computation has improved, by the integration of a GPU utilizing multiple programmable graphics pipelines. Except for in the fields of mobile gaming and 3D texture rendering, many computationally-intensive algorithms can be accelerated by modern mobile GPUs by means of emerging parallel programming models such as the Open Computing Language (OpenCL) [9, 10]. Due to the special hardware architecture of mobile processors, many techniques previously applied to desktop GPUs are not suitable for mobile applications. To achieve high performance, we need to analyze the algorithms and workloads specifically on mobile platforms and find an efficient mapping to embedded hardware.

With this goal, we present in this paper an OpenCL-based SIFT implementation using heterogeneous computing techniques on a mobile processor. The paper is organized as follows. Section II briefly introduces the architecture of mobile processors and the OpenCL programming model. Section III describes the SIFT algorithm and its workflow. Section IV analyzes the different workloads of the SIFT algorithm and partitions the workflow efficiently onto the CPU and GPU. Experimental results are presented in Section V. We conclude our findings in Section VI.

II. OpenCL on Mobile Processors

Modern mobile processors typically consist of a multi-core CPU, a GPU with multiple programmable graphics pipelines, and image processing accelerators. There are several differences between mobile GPUs and desktop GPUs, making it challenging to achieve an efficient, high-performance implementation. Firstly, the memory bandwidth of mobile GPUs, at several GB/s, is much lower than the several hundred GB/s of desktop GPUs. Secondly, mobile GPUs have significantly fewer compute units, typically 32 to 128, than desktop GPUs, which usually have between 1024 and 3072 discrete compute units. Thirdly, the clock frequency of mobile GPUs is typically between 200 and 400 MHz, which is much slower than their desktop counterparts, which typically reach 1GHz clock frequency. For these reasons, it is critical that targeted algorithms are carefully analyzed to find a good mapping to mobile GPUs. On the other hand, the low clock frequency of mobile GPUs suggests the possibility of efficient low power designs, in which offloading some parallelizable computationally-intensive algorithms to GPUs may reduce power consumption.

The lack of good programming tools is another problem for mobile GPGPU applications. In the past few years, the OpenGL Embedded System (OpenGL ES) application programming interface (API) has been widely used for GPGPU programming on mobile devices. However, OpenGL ES was designed for graphics, so it is difficult to use for general-purpose computing, leading to inflexible designs. Recently, emerging mobile processors have begun to support the OpenCL API for heterogeneous computing. With OpenCL, parallel workloads are mapped to work groups, each containing work items that can share local memory and be synchronized together, allowing for greater flexibility and performance in GPGPU applications. Our previous work has shown the capability of mobile GPUs to accelerate complicated computer vision algorithms using OpenCL [10].

To appear at IEEE Global Conference on Signal and Information Processing (GlobalSIP), December 2013, Austin, Texas, USA

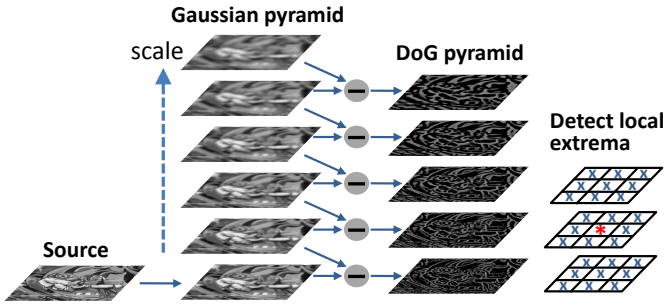

Fig. 1. Diagram of SIFT keypoint detection algorithm showing one octave with 6 Gaussian image layers.

III. Overview of SIFT

A local feature keypoint represents an image pattern which differs from its neighboring pixels. The properties of keypoints can be extracted and described using keypoint descriptors. Keypoint descriptors are highly distinctive, enabling image matching or object recognition with high probability in a large database of features [1, 2]. The SIFT algorithm consists of the following stages to detect keypoints and extract feature descriptors [1]. The major steps for keypoint detection are shown in Fig.1

Gaussian pyramid. First, an input image is convolved by a series of Gaussian smoothing kernels $G(x, y, \sigma)$ with different σ to get the first set of images, called the first octave. We downsample the input image to form the base of a smaller set of images, called the next octave, which will in turn be smoothed to an even higher scale factor. Each octave includes a few layers, each corresponding to a different scale factor σ. In this paper, we use typical algorithm parameters suggested by Lowe's paper: 5 octaves and 6 layers in each octave [1].

Difference of Gaussian (DoG) pyramid. We subtract every two adjacent layers to get DoG pyramids. For each octave, 5 DoG images are generated.

Keypoint detection and refinement. We compare each pixel in a DoG image with its surrounding 3×3 neighbor pixels from the current layer, a lower layer and a higher layer. If the value of a pixel is a maxima or minima among the total 26 neighbors, it is identified as a keypoint candidate. Then we perform a quadratic interpolation in scale space around the candidate to refine the accuracy of keypoint location to subpixel level. Finally, keypoints with low contrast are considered non-distinctive, so they are rejected.

Gradient orientation assignment. For each keypoint, we compute the gradient magnitude and orientation angle of the Gaussian pyramid images for all pixels in a region around the keypoint. We build an orientation histogram, which partitions 360 degrees of angular orientation into 36 bins. Each pixel in the KP-region contributes to one of the 36 bins based on its gradient orientation angle. The contribution score is the gradient magnitude value weighted by a Gaussian function $G(x_0, y_0, \sigma)$, where (x_0, y_0) is the pixel's coordinate relative to the keypoint. We then choose the bins with highest scores as dominant orientations for the keypoints. Each keypoint can have multiple orientations, and each orientation is used to generate a unique feature descriptor.

Descriptor generation. A feature descriptor is a vector which represents the local properties of a keypoint. In this step, a keypoint region (KP-region) surrounding a keypoint is extracted and rotated to account for the keypoint orientation. Then, the KP-region is partitioned into 4×4 subregions. In each subregion, we build an orientation histogram with 8 bins, or 45 degrees per bin. Then, in a process similar to the gradient orientation assignment step, pixels in each subregion contribute to the histogram by adding the Gaussian-weighted gradient magnitude to the corresponding bins. Repeating this for all 4×4 subregions results in a feature vector of 128 values, representing 8 bins for each of the 16 subregions. Finally, we truncate and normalize this vector to form a keypoint descriptor.

IV. Algorithm Profiling and Workload Partitioning

To better understand the workload and performance of the SIFT algorithm on mobile devices, we implemented the SIFT algorithm using both serial C++ and parallel OpenCL computation kernels. The SIFT algorithms are roughly partitioned into computation functions based on the workflow described in Section III. By running our SIFT implementations over a benchmark dataset on a mobile processor, we measure the processing time for each function for both the CPU and GPU implementations. The data transfer time between the host CPU and the device GPU is also counted for the GPU implementation. Carefully analyzing the profiling results finally leads to a heterogeneous implementation, which efficiently maps the SIFT algorithm onto the CPU and GPU in a mobile processor.

A. Experimental Setup

We use 48 images from a benchmark dataset designed for local feature performance evaluation [2, 11]. The dataset covers transformations such as blurring, change of viewpoint, zoom, rotation, and compression. All our test images have widths of 320 pixels, and image height varies from 214 to 256. We use a mobile device with a Qualcomm Snapdragon S4 Pro APQ8064 chipset, which includes a 1.5GHz quad-core CPU and an Adreno320 GPU, with four compute units containing an array of ALUs running at 325MHz. We compiled our optimized C++ code using the Android Native Development Kit (NDK) version r8c and OpenCL 1.1. In this section, profiling is performed on an image called "graf" in the test dataset, as shown later in Fig.4. The "graf" image has more feature keypoints than other images, revealing the worst case processing time for the dataset.

B. Data Structure

We first did experiments to determine the necessary data precision. When using a 32-bit floating point format, 269 keypoints are extracted, while switching to an 8-bit fixed point format reduces the number of keypoints to 137. Specifically, we observed that most of the keypoints for larger scales disappeared with the loss in precision. In addition, we lose sub-pixel level keypoint accuracy. These facts could significantly degrade the image matching performance, so we choose to use the floating point format.

For our OpenCL implementation, we define the data buffers as the Image2D type, taking advantage of the GPU's high-performance texturing hardware, which provides image interpolation, border pixel handling and texture caching. By doing this, access to image data is accelerated and branching instructions in the kernel are reduced, since we do not need to pay special attention to handling the border pixels.

Packing several grayscale data into an RGBA texel is a popular technique to reduce the memory bandwidth for GPGPU applications [5, 8]. In our case, four 32-bit floating point data in a 2×2 square are packed into a CL_RGBA|CL_FLOAT format data. Data packing is applied to the Gaussian pyramid, DoG pyramid, and gradient pyramid. We can benefit from reduced device memory accesses, as well as utilization of the GPU's vector processing units. With this data packing approach, we observed a 28% reduction in execution time for Gaussian pyramid generation, and about a 40% reduction for gradient pyramid generation.

TABLE I
PROFILING RESULTS FOR THE MAJOR PROCESSING STEPS OF SIFT, USING THE 320 × 256 "GRAF" IMAGE. 269 KEYPOINTS ARE DETECTED.

	Time results[a] (ms)		
	CPU	GPU	GPU readback
Gaussian pyramid	98.5	**52.01**	**14.38**
DoG	**16.08**	22.68	16.01
Gradient pyramid	80.73	**21.00**	**10.65**
Local extrema	**14.13**	23.50	N/A[b]
Keypoint refinement	**0.85**	7.91	N/A[b]
Orientation assignment	**30.17**	140	N/A[b]
Descriptor generation	193.87	**148.33**	N/A[b]

a. Bold fonts indicate our choices for each step.
b. Data transfer time is included in GPU time, due to small data sizes.

C. Implementation of OpenCL Kernel Functions

After analyzing the workflow described in Section III, the following major steps were implemented using separate functions: Gaussian pyramid generation, DoG pyramid generation, local extrema detection, keypoint refinement, orientation assignment, and descriptor generation. It is worth mentioning that both orientation assignment and descriptor generation require the gradient information, or the gradient magnitude and orientation angle, of the pixels in the KP-region. The gradient calculation needs the computationally expensive dot-product, square root and $atan2()$ operations. Although for each keypoint, we only compute the gradient for a small KP-region, experiments show that the gradient information of each pixel is requested an average of more than one time during the whole SIFT computation. Therefore, when both the SIFT detector and descriptor are computed for the same image, it is more efficient to pre-compute the gradient information for all pixels to avoid redundant computations. In fact, all keypoints are located on layers 1∼3, so the orientation and descriptor are only computed on those three layers. Therefore, we only build the gradient pyramid for layers 1∼3.

In order to take advantage of the parallel architecture of the mobile GPU, we use an image tiling technique. We divide an input image into tiles, where each tile is assigned to a work group consisting of multiple work items, each of which processes one or more pixels in a tile. Due to the properties of the hardware, the typical size of a work group is 8×8. Image tiling is applied to the Gaussian pyramid generation, DoG pyramid generation, local extrema detection, and keypoint refinement kernels, to exploit the parallelism of the algorithms. However, for orientation assignment and descriptor generation, computations are only performed in the KP-regions for a list of keypoints. For these two kernels, we map all work groups to one KP-region, using atomic instructions to compute the histograms.

D. Profiling Results and Workload Partitioning

In Table I, the processing times are shown for each of the major steps for both the C++ and OpenCL implementations. The readback time also includes the time for data unpacking from RGBA format to the normal image format, since this must happen after memory readback. In the steps of local extrema detection, keypoint refinement, orientation assignment and descriptor generation, the GPU readback time is negligible due to the very small data size.

Since the algorithms for Gaussian pyramid generation, gradient pyramid generation and descriptor generation provide a sufficient amount of parallelism, we can benefit from the GPU's parallel architecture. The GPU processing time plus data transfer overhead is shorter than the CPU processing time, indicating that we can achieve better performance by offloading these functions to the GPU. On the other hand, the CPU version outperforms the GPU version for the other kernels. Obviously, keeping them on the CPU results in higher performance. Among them, orientation assignment has much longer processing time on the GPU than the CPU, since the main operation in this kernel is histogram computation, which is inherently serial and becomes a major bottleneck limiting the performance. For DoG pyramid generation, local extrema and keypoint refinement, the GPU implementation is slower since the relatively simple computations cannot fully utilize the GPU's resources. In these cases, the higher clock frequency of the CPU leads to better performance. Based on the profiling results, we partition the SIFT algorithm to a CPU-GPU heterogeneous implementation to minimize the total processing and memory transfer time, as shown in Fig.2.

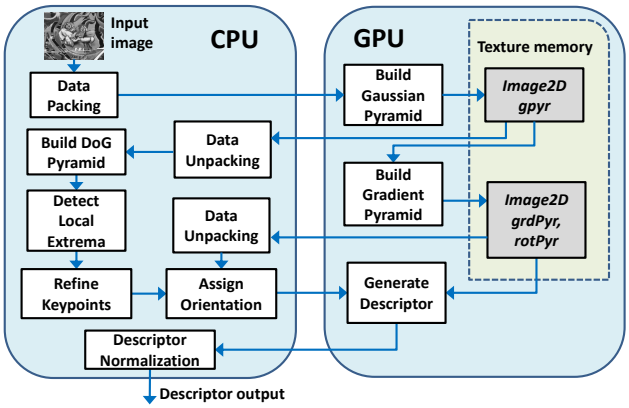

Fig. 2. Algorithm partitioning for heterogeneous implementation.

E. Optimization of Parallel Gaussian Pyramid Generation

Generation of the Gaussian pyramid is the most time consuming step in keypoint detection, so it is necessary to explore optimization techniques. For desktop GPUs, due to the large number of cores, we may compute Gaussian blur for all 6 layers in an octave (or even multiple octaves) in a single parallel kernel. In mobile devices, the limited number of compute units and onchip memory make the fully parallel approach infeasible. Following our previous work, in this implementation, we use a separable two-pass Gaussian filtering method. We also apply dynamic runtime code generation to produce branchless OpenCL code [8]. To be specific, we use standard I/O library functions of C++ to generate formatted OpenCL kernel code incorporating filter parameters at runtime, so that loops in OpenCL kernel functions are fully unrolled without branch operations.

In addition to the above approaches, we utilize a recursive Gaussian blur method which is a good fit for mobile GPUs. The recursive Gaussian blur is based on the idea that one pass of the Gaussian blur with σ_a is equivalent to applying two passes with σ_b and σ_c sequentially if $\sigma_a^2 = \sigma_b^2 + \sigma_c^2$. Therefore, a higher layer Gaussian blur image with σ_1 can be generated by applying an extra amount of blurring ($\sqrt{\sigma_1^2 - \sigma_0^2}$) to the previous layer with σ_0. One major benefit of the recursive Gaussian blur is that the subsequent convolution is done using small scale factor $\Delta\sigma = \sqrt{\sigma_1^2 - \sigma_0^2}$, therefore, the required Gaussian filter kernel can be shorter (which is typically $6\sigma + 1$). A shorter filter kernel means fewer memory accesses, which allows for an efficient mobile GPU implementation, since low memory bandwidth is one of the major bottlenecks for the GPU.

Furthermore, layers $0 \sim 2$ in each octave n are directly generated simply by downsampling the layers $3 \sim 5$ in octave $(n-1)$ by one half according to scale space theory, without doing any Gaussian filtering. Downsampling by one half can be implemented very efficiently by accessing $1/4$ of the pixels without the need for

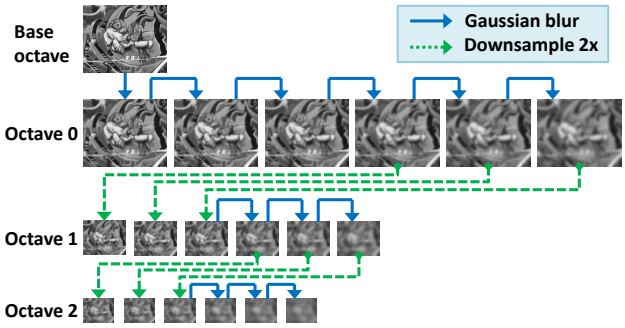

Fig. 3. Efficient Gaussian pyramid generation. The procedure for octaves $0 \sim 2$ is shown. For octaves $3 \sim 4$, the process is similar to octave 2.

Fig. 4. Keypoint detection results for "graf", "ubc", and "boat" images. Each circle represents a keypoint, its radius indicating the scale of the feature. The lines in the circle represent orientation of the keypoints. Numbers of keypoints are 269, 133, and 192, respectively.

arithmetic operations. Our efficient Gaussian pyramid workflow is shown in Fig.3, in which for each octave other than the first, we only need 3 downsample operations and 3 Gaussian blur operations. The Gaussian blur and downsample blocks are implemented using OpenCL kernels with the image tiling technique.

V. Experimental Results

We benchmark the complete SIFT implementation with keypoint detection and descriptor generation on 48 images using the whole dataset [11]. The keypoint detection results for three images in the dataset are shown in Fig.4. Thanks to the full precision we used, there is no performance loss in our heterogeneous implementation in terms of keypoint number and accuracy, compared to some popular SIFT implementations for the desktop CPU, such as Lowe's implementation [1] and VLFeat [12].

Table II reports the average time per image. For the CPU-only implementation, we compile the NDK code with both the ARM-v5 and ARM-v7a instruction set architectures. The average number of keypoints detected per image is 95. First, we notice that ARM-v7a generates much more efficient code than ARM-v5. For the detection portion, we observe a 1.69X speedup with the heterogeneous implementation, compared to the CPU-only implementation. With the heterogeneous implementation, we can achieve 8.5 frames per second (FPS) for keypoint detection, and 19 FPS for descriptor generation. We note that a nonnegligible part of time for the heterogeneous solution comes from memory transfers (24.6ms, which is 21% of the total detection time). If the next generations of mobile GPUs can include newer memory technology and faster memory bandwidth between the CPU and GPU, we foresee greater speedup using heterogeneous computing techniques.

To measure energy efficiency, we compute SIFT on the same dataset for several minutes, measure the average system power, and then subtract the idle system power. We measured 1490mW and 1429mW power consumptions for the CPU-only implementation and the heterogeneous implementation, respectively. The average energy consumption per image is 413.0mJ for the CPU-only implementation, and 242.3mJ for the heterogeneous one. Therefore, a 41% reduction in energy consumption is achieved.

TABLE II
Processing time on the whole dataset with 48 images.

	Time results (ms)		
	CPU only		CPU+GPU
	ARM-v5	ARM-v7a	ARM-v7a
Gaussian pyramid	710.96	81.78	40.15
Readback Gaussian	N/A	N/A	8.32
DoG pyramid	30.28	15.30	12.56
Gradient pyramid	232.37	78.81	16.22
Readback Gradient	N/A	N/A	16.75
Local extrema (Incl. refinement, orientation)	42.94	21.40	20.37
Keypoint detection total	1015.65	197.43	117.02
Descriptor generation total	261.43	79.74	52.52
Complete SIFT	1278.09	277.18	169.54

VI. Conclusion

This paper presents workload analysis and an efficient implementation of the SIFT algorithm on a mobile processor. We discuss efficient algorithm mapping to the GPU architecture based on profiling results and optimization techniques, with emphasis on optimized Gaussian pyramid generation. Experiments show that we can achieve a 1.69X speedup for keypoint detection compared to an optimized C++ reference design. The frame rates for keypoint detection and descriptor generation are improved to 8.5 FPS and 19 FPS, respectively. Meanwhile, we reduce the energy consumption by 41%. However, we notice that the limited number of compute units and the low memory bandwidth between the CPU and GPU are still bottlenecks for a high speed mobile computer vision application.

Acknowledgment

This work was supported in part by Samsung, Qualcomm, Renesas Mobile, Texas Instruments, and by the US National Science Foundation under grants CNS-1265332, ECCS-1232274, and EECS-0925942.

References

[1] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *Int. J. Computer Vision*, vol. 60, no. 2, pp. 91–110, 2004.

[2] K. Mikolajczyk and C. Schmid, "A performance evaluation of local descriptors," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 27, no. 10, pp. 1615–1630, Oct. 2005.

[3] H. Bay, A. Ess, T. Tuytelaars, and L. V. Gool, "Speeded-Up Robust Features (SURF)," *Comput. Vis. and Image Underst.*, vol. 110, no. 3, pp. 346–359, 2008.

[4] F.-C. Huang, S.-Y. Huang, J.-W. Ker, and Y.-C. Chen, "High-performance SIFT hardware accelerator for real-time image feature extraction," *IEEE Trans. Circuits and Syst. for Video Technol.*, vol. 22, no. 3, pp. 340–351, 2012.

[5] S. Heymann, B. Fröhlich, F. Medien, K. Müller, and T. Wiegand, "SIFT implementation and optimization for general-purpose GPU," in *Winter School of Computer Graphics*, 2007. [Online]. Available: wscg.zcu.cz/wscg/Papers_2007/Full/G03-full.pdf

[6] S. Sinha, J.-M. Frahm, M. Pollefeys, and Y. Genc, "Feature tracking and matching in video using programmable graphics hardware," *Mach. Vis. and Appl.*, vol. 22, pp. 207–217, 2011.

[7] D. Wagner, G. Reitmayr, A. Mulloni, T. Drummond, and D. Schmalstieg, "Pose tracking from natural features on mobile phones," in *Proc. IEEE/ACM Int. Symp. Mixed and Augmented Reality (ISMAR)*, 2008, pp. 125–134.

[8] B. Rister, G. Wang, M. Wu, and J. R. Cavallaro, "A fast and efficient SIFT detector using the mobile GPU," in *Proc. IEEE Int. Conf. Acoustics, Speech, and Signal Process. (ICASSP)*, May 2013, pp. 2674–2678.

[9] K. Pulli, "OpenCL in handheld devices," *4th annual multicore EXPO*, 2009.

[10] G. Wang, Y. Xiong, J. Yun, and J. R. Cavallaro, "Accelerating computer vision algorithms using OpenCL framework on the mobile GPU - a case study," in *Proc. IEEE Int. Conf. Acoustics, Speech, and Signal Process. (ICASSP)*, May 2013, pp. 2629–2633.

[11] Visual Geometry Group, University of Oxford, *Affine covariant features*. [Online]. Available: http://www.robots.ox.ac.uk/~vgg/research/affine/

[12] A. Vedaldi and B. Fulkerson, "VLFeat: An open and portable library of computer vision algorithms," http://www.vlfeat.org/.