PopSift: a faithful SIFT implementation for real-time applications

Carsten Griwodz
University of Oslo
and Simula Research Laboratory
griff@ifi.uio.no

Lilian Calvet
University of Clermont-Auvergne
lilian.calvet@gmail.com

Pål Halvorsen
Simula Research Laboratory
and University of Oslo
paalh@simula.no

ABSTRACT

The keypoint detector and descriptor Scalable Invariant Feature Transform (SIFT) [8] is famous for its ability to extract and describe keypoints in 2D images of natural scenes. It is used in ranging from object recognition to 3D reconstruction. However, SIFT is considered compute-heavy. This has led to the development of many keypoint extraction and description methods that sacrifice the wide applicability of SIFT for higher speed. We present our CUDA implementation named PopSift that does not sacrifice any detail of the SIFT algorithm, achieves a keypoint extraction and description performance that is as accurate as the best existing implementations, and runs at least 100x faster on a high-end consumer GPU than existing CPU implementations on a desktop CPU. Without any algorithmic trade-offs and short-cuts that sacrifice quality for speed, we extract at >25 fps from 1080p images with upscaling to 3840x2160 pixels on a high-end consumer GPU.

CCS CONCEPTS

•**Computing methodologies** → **Interest point and salient region detections;** *Image processing;* •**Computer systems organization** → *Single instruction, multiple data;*

KEYWORDS

GPGPU, Feature Extraction, Interest Point Detection, Keypoint Extraction, SIFT

ACM Reference format:
Carsten Griwodz, Lilian Calvet, and Pål Halvorsen. 2018. PopSift: a faithful SIFT implementation for real-time applications. In *Proceedings of 9th ACM Multimedia Systems Conference, Amsterdam, Netherlands, June 12–15, 2018 (MMSys'18),* 6 pages.
DOI: 10.1145/3204949.3208136

1 INTRODUCTION

Image matching aims at establishing correspondences between similar objects appearing in different images. It is one of the fundamental steps in many applications such as image recognition, three-dimensional reconstruction, image registration and object tracking. Among image matching methods, the ones relying on keypoints are widely used. They consist of two main parts: *keypoint detection* and *descriptor extraction.* The first part comprises the detection of *keypoints*, also called *local region* or *interest point*, and the selection of a region surrounding each of them. A descriptor associated with a selected region is then extracted and used later on to find keypoint correspondences across a collection of images representing same objects.

One of the most used methods for keypoint detection and descriptor extraction, if not the most used, is SIFT [8]. The method combines a Difference of Gaussian (DoG)-based keypoint detector that is invariant to rotation, translation and scale[1] with a descriptor based on the gradient orientation distribution in the region.

In the context of our POPART and LADIO projects, we aim at real-time natural feature-based camera tracking from 3D reconstructed environments. To achieve this, keypoint-based image matching algorithm are essential in both reconstruction and tracking, as they provide, based on raw input images, keypoint correspondences used both in Structure-from-Motion (SfM) and image-based camera localization pipelines. Keypoint detection and descriptor extraction remain among the most time consuming steps of the SfM pipeline. The required accuracy of the tracking demands the use of an HD (1080*p*) live video stream.

There are already some GPU implementations of SIFT. SiftGPU [13] is close to reaching the speed goal, but we could not build on it due to our need for a more flexible license. CudaSift [3, 4] is capable of keypoint detection and description in real-time, but it does not actually behave like SIFT, as shown in Section 5. We therefore need a new open source SIFT implementatins that is both fast enough for real-time applications and implements the SIFT algorithm faithfully.

This paper describes our open source implementation of SIFT as proposed by Lowe [8] on CUDA following the descriptor normalization proposed by Arandjelovic and Zisserman [1]. Our implementation is called PopSift. PopSift as submitted to MMSys'18 can be found at **https://github.com/acmmmsys/2018-PopSift**[2]. To illustrate its performance compared to a very good CPU implementation, Table 1 shows computing times of VLFeat on an i5-4590 and PopSift on a GTX 1080 for 3 individual frames of resolution 1920x1080.

2 THE SIFT ALGORITHM

The SIFT algorithm follows several steps that are illustrated in Figure 1. They perform the following sequence:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MMSys'18, Amsterdam, Netherlands
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
978-1-4503-5192-8/18/06…$15.00
DOI: 10.1145/3204949.3208136

[1] A mathematical proof is given in [6].
[2] New developments can be found at https://github.com/alicevision/popsift.

	VLFeat		PopSift	
	runtime	descriptors	runtime	descriptors
maine	7.195 sec	44666	0.043 sec	44930
cap	7.232 sec	44966	0.043 sec	45179
boston	6.060 sec	30262	0.037 sec	30473

Table 1: VLFeat on an i5-4590 at 3.3Ghz vs PopSift on a GTX 1080. The time spans keypoint detection and descriptor extraction (and CPU-GPU transfers for PopSift) but no image decoding or disk operations. Figures had resolution 1920x1080, upscaled to 3840x2160, using the default parameters of VLFeat.

Upscaling. Upscale the input image by a factor of 2 in both X and Y dimension. Generally, implementations of SIFT do also allow the user to skip this step to sacrifice accuracy for speed.

Creating a Gaussian pyramid. SIFT attempts scale-free matching by emulating a freely scale pyramid of resolutions. This pyramid is emulated by computing groups of same-resolution images that are increasingly blurry, each group called an *octave*, and the same-resolution images of an octave are referred to as *levels*. The first octave has the resolution of the scaled input image, every subsequent group halves the resolution. Downscaling is performed from the third-last level of an octave. SIFT uses 2D Gaussian blurring with a blur factor that is traditionally called σ. The next lower resolution octave is computed by first downscaling the image by a factor of 2 whose resulting accumulated blur factor is 2σ.

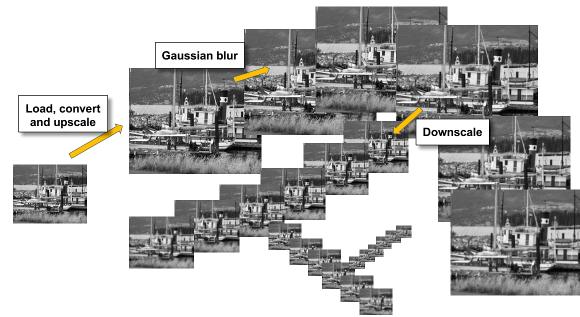
Computing the DoG. The DoG is computed to approximate to the scale-normalized Laplacian-of-Gaussian as studied by Lindeberg [7].

Detecting keypoints. Every pixel in the DoG layers that is an extremum (absolute minimum or maximum) in its neighbourhood in the same, previous and next level (26 neighbouring pixels) initiates a keypoint search close to it in scale-space (X, Y and blur level direction). The potential keypoint location is found at the location of the maximum or minimum of a quadratic function through the extremum in 3D space. The candidate is accepted as a keypoint when it passes a contrast and "edgeness" test.

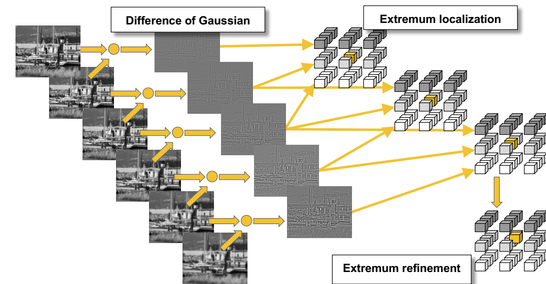
Computing the keypoint orientations. Each keypoint may be described by one or more SIFT Descriptors. For each descriptor, the feature point has an associated dominant orientation indicating a 2D direction of the strongest luminance change within at least a 30° arc from the feature point. The dominant orientation of a SIFT feature vector is a floating point value that exists at least once for each accepted extremum.

Extracting descriptors. To compute the SIFT descriptor for each dominant orientation of a keypoint, 16 vectors of 8 floats are computed. Each group of 8 represents a gradient histogram in one of 16 square regions arranged around the keypoint. The symmetrical grid of 16 squares is oriented according to the keypoint's dominant orientation and scaled according to the Z-coordinate. A detail that is rarely represented in SIFT illustrations is that these square regions overlap (the red square in Figure 1(d) shows the actual coverage).

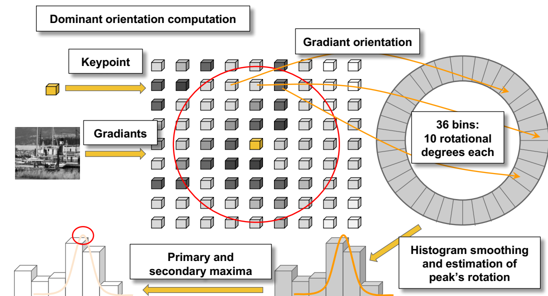
Normalizing descriptors. Descriptor vectors are normalized to unit length to ensure invariance to luminance changes. The classical norm is an L2 norm with preprocessing for thresholding. RootSIFT normalization proposed in [1] has shown to improve significantly the matching performance.



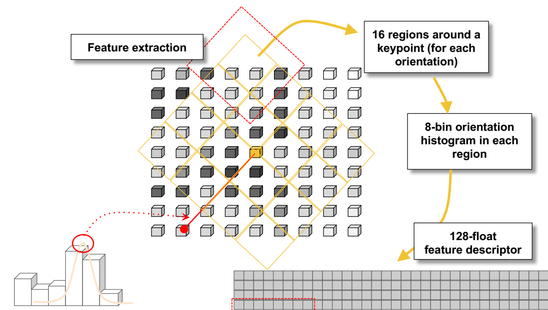
(a) Pyramid creation



(b) Difference of Gaussian and extrema localization



(c) Finding dominant orientations



(d) Descriptor creation

Figure 1: The stages of SIFT.

3 POPSIFT IMPLEMENTATION

PopSift follows strictly the steps prescribed by Lowe [8]. It is meant to be used as a drop-in replacement for VLFeat [12], and implements a non-blocking dataflow that uses an input queue and double buffered transfer to the GPU, while still allowing calling programs to interact with PopSift as if they were using blocking calls. To achieve this, we make use of `futures` from the C++11 standard library. The SIFT extraction in the background uses two threads. The first one moves queued jobs from the open-ended first queue into a double buffer of CUDA-allocated host memory. CUDA-allocated host memory is pinned, page-aligned memory that is suitable for DMA onto the GPU, and the two stages are used to minimize the amount of pinned memory. The second thread starts all required CUDA kernels until the SIFT features are extracted. Since the number of SIFT features is varying considerably even for images of the same resolution, the returning host buffer is allocated dynamically, using page-aligned memory. For the download operation, it is pinned to allow DMA, unpinned, and stored in the `promise`, thereby potentially unblocking the `future` in case the calling program has already called `get()`.

Image upload and upscale. After transferring an input image to the GPU, we access it through a CUDA texture using hardware-supported normalized access and interpolating access. Consequently, we are not restricted to original-size or double-size input images, but can choose arbitrary scale factors.

Gaussian blurring and DoG computation. PopSift exploits the separability property of the Gaussian filter. Filters are loaded into CUDA `constant` memory. Multiplication operations are the bottleneck in the Gaussian filter computation, and PopSift uses symmetry and, optionally, interpolation to reduce memory access and multiplications. Still, wider filters require more time, and PopSift supports several filter width computation. The default width is $\lceil 4\sigma \rceil + 1$, which considers all Gaussian terms above 10^{-8}. Alternatives are $(\lceil 16\sigma + 2 \rceil / 4)|1$ and fixed 15. For a default $\sigma = 1.6$ and default filter width, the full dual-sided filter widths for incremental blurring range from 11 for blur level 1 to 27 for level 5. For downscaling, PopSift defaults to the prescribed downscaling of the third-to-last level of an octave, but it is also possible to downscale the first level of every octave directly from the upscaled input image. To increase parallelism, it is also possible to downscale to all levels of an octave directly from the scaled input image. However, this means larger σ values, which leads to default filter widths ranging from 15 to 43. The DoG layers require a simple pixel-wise subtraction.

Keypoint detection. The number of CUDA threads used by PopSift is straightforward, one thread checks a single pixel. They are arranged in groups of 32x4, where 32 achieve the optimal load operation of 128 contiguous bytes and 4 groups of 32 compensate for load latency. Extremum computation is performed by filling a bitmask that evaluates to true if the pixel is either an absolute minimum or maximum. These bit operations avoid all code branches and allow groups of 32 threads to execute in lockstep. Testing for presence of an extremum is a single boolean operation.

To implement subpixelic refinement of the keypoint position in three dimensions (X position, Y position and blur level), PopSift uses a closed-form solution instead of Gaussian eliminiation. The specific refinement test differs between SIFT implementations. PopSift defaults to an own variant, but implements also OpenCV and VLFeat variants. Importantly, OpenCV and PopSift variants can move in the third dimension (between blur levels), whereas VLFeat cannot do this.

Dominant orientation computation. PopSift uses the fast CUDA intrinsics `__shfl*()`, `__popc()` and `__ballot()` in dominant orientation computation. This requires the use of 2^n threads, but no more than the maximum warp size of 32. This prevents the use of one thread for each of the prescribed 36 histogram bins (each representing 10°) for collecting an orientation histogram of gradiants around the extremum, but pays off by reducing memory latency. For best speed, we use 16 threads per keypoint. To compute secondary dominant orientations effectively, we implemented a very specific 32-cell bitonic sort.

Descriptor extraction. PopSift implements several approaches for sampling images and create SIFT descriptors. In the "loop" approach, 512 threads cooperate in 16 groups of 32, where each group of 32 compute 8 values of a descriptor, representing histogram for one of 16 rotated squares surrounding a keypoint. They define a box aligned with the image that contains their assigned square entirely, and scan every pixel in the box. Weighted gradient information is computed for those pixels that are also inside the square and added to the 8-bin histogram that makes up the descriptor.

In the "grid" approach, interpolating textures are used to sample each of the 16 squares following a 16x16 grid patterns that is aligned to a domiment orientation of the keypoint. 256 threads cooperate for each keypoint orientation, in groups of 16 for every 16x16 grid. Weighted gradients are computed and inserted in the histogram.

The third approach (called "notile") makes use of the overlap between the squares. Whereas outer regions of corner squares contribute only to this square's histogram, pixels closer to the center may actually contribute to the histograms of 4 squares. This approach uses 32 threads per keypoint to sample a 64x64 grid of the entire area covered by the 16 squares, and computes the gradient for each sample, and subsequent weight and add it to the one, two or four relevant histograms.

The final step requires normalization of the descriptors. PopSift implements L2 and RootSIFT normalization [1]. The resulting descriptors can optionally be scaled by any power of 2 in the same step, simplifying conversion to byte descriptors instead of float descriptors before transfer to the CPU. Typical multipliers are 2^8 and 2^9.

4 COMPILING AND USING POPSIFT

PopSift is developed and tested on Ubuntu 16.04 and MacOS X. Third parties report having built and used it on Windows. It has three mandatory dependencies, CUDA 7.0+ as GPU programming framework, CMake 3.4+ for the build system and Boost 1.53.0+ for simple support functions.

LibDevIL is an optional dependency of the demo application `popsift-demo`. It adds the ability to load images that are not in PGM or PPM format, as well as recursively reading entire directories. However, libDevIL looses so much precision due to its internal intermediate representation that it is clearly observable in repeatability tests, and it is therefore mainly useful for speed demonstrations.

To compile PopSift, clone the git repository **https://github.com/acmmmsys/2018-PopSift**[3]. Set up the local build system using CMake. Before compiling, it is a good idea to check the CMake variables named `PopSift_*`, for example using `ccmake`. Compile speed can be increased considerably by restricting the CUDA Compute Capability list in `PopSift_CUDA_CC_LIST` to the relevant platform, and by disabling `PopSift_USE_GRID_FILTER` if the grid filter functionality is not required.

The main output of the compilation process is the library `libpopsift.a`, which is meant for linking by other programs.

A demo program called `popsift-demo` is provided, whose multitude of parameters allows switching between the alternative implementation options described in Section 3. Default values of these parameters target precision over speed. The only mandatory parameter of `popsift-demo` is the choice of an image or of a directory containing images, indicated by `-i`.

Library, headers and demo program as well as CMake configuration files can be installed using `make install` and removed using `make uninstall`.

PopSift is available under the Mozilla Public License Version 2.0.

An executable for Ubuntu 16.04 that is statically linked to CUDA and Boost and does not use libDevIL can be found at https://github.com/alicevision/popsift/releases/tag/os_01_2018.

5 COMPARISON WITH CUDASIFT

The fastest code that claims to be an implementation of SIFT is CudaSift, and it does outperform PopSift. However, CudaSift does not actually behave like a faithful SIFT implementation in terms of keypoint detection. It implements an approximation of the Laplacian-of-Gaussian for all levels directly from the input image, which should work well, but it uses narrow filters with the argument that locality dominates in keypoints detection. In our understanding, this implies that the lower levels of an octave cannot be faithful estimates of a downscaled image, since distant pixels are not considered. It behaves very differently from the other SIFT implementations in this test, as shown in Figure 2, and is apparently not feasible as a drop-in replacement for a CPU-based SIFT implementation.

6 OTHER COMPARISONS

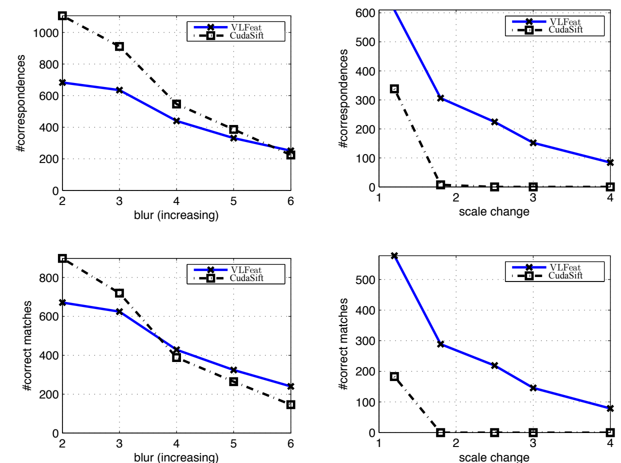
We compared PopSift with three other open-source SIFT implementations (VLFeat, OpenCV and SiftGPU) using datasets that were published for the evaluation of keypoint extraction and matching (VGG [9], Hannover [5] and CVLAB [11]). Since none of these datasets contains images at 1080p resolution, we have additionally downloaded 12 images from Flickr with 1920x1080 resolution.

6.1 Qualitative performance

To compare quality with the other implementations, we used the repeatability evaluation using true correspondances based on region overlap as defined in [10] and the corresponding publicly available code [9].

PopSift performs very similar or slightly better than VLFeat and performs better than SiftGPU and OpenCV. A subset of our results can be seen in Figure 4.

[3]Or more recent versions at https://github.com/alicevision/popsift



(a) Example of CudaSift performing better than VLFeat

(b) Example of CudaSift performing worse than VLFeat

Figure 2: Illustration of the deviation of CudaSift from a by-the-book SIFT implementation on (a,c) *Bikes* and (b,d) *Bark*. Evaluation according to [10].

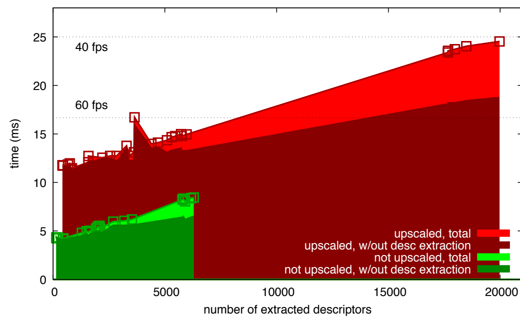
6.2 Time performance

The implementation decisions that we documented in Section 3 have in general enabled us to achieve our goal of processing 1080p frames and extracting descriptors in real-time. We measure only the time from the calling application's handover of the loaded luminance image to the PopSift until the buffer containing the extracted descriptors can be read by the application. The time includes all transfers between CPU and GPU as well as buffer allocation. The first image in a sequence does always have additional 20 msec delay for CUDA memory allocation. It is therefore recommended to create the PopSift object once and queue frames. Typically, in spite of non-blocking operation, PopSift processes images faster than the host can load compressed images from disk.

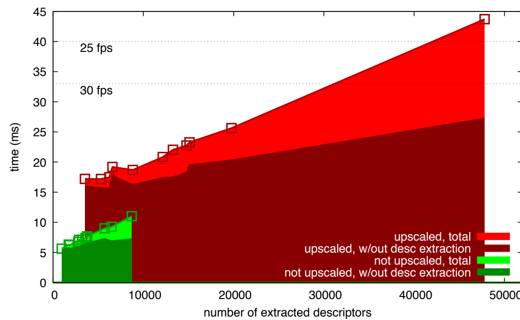
Figure 3 illustrates the processing time for 3 sets of differently sized images (1536x1024, 1920x1080 and 2560x1920 pixels). The specific datapoints are marked by boxes. Although the construction of the Gaussian pyramid is typically the most time-consuming step, it is constant for a given image size. The dominant dynamic part is the descriptor extraction step. It is possible, as Figure 3(b) shows, that an extremely feature-rich image requires so much processing time in the descriptor extraction step that the real-time goal cannot be achieved.

However, good camera pose estimation and other applications that would benefit from real-time operation do generally work well when on the order of 2000 features are extracted from all images. In our example images, this goal could mostly be achieved without the image upscaling step.

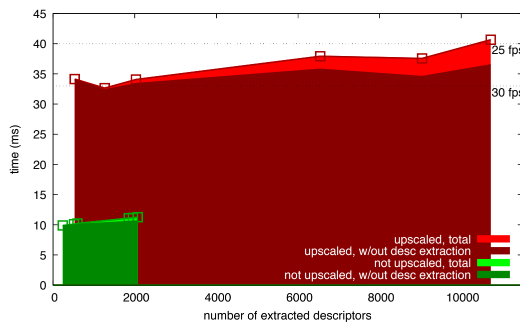
To reduce the number of keypoints to a managable number, PopSift implements also an optional grid filtering method that sorts keypoints by their scale within the cells of a regular grid that is overlaid the image. Only up to a desired number of keypoints is then handed over to descriptor extraction, uniformly distributed over the grid cells. This optional step makes use of CUDA Thrust. It is



(a) Resolution 1536x1024 (Hannover dataset)



(b) Resolution 1080p, ie. 1920x1080 (Flickr images)



(c) Resolution 2560x1920 (pyramids in Holidays dataset)

Figure 3: Speed of PopSift. Red colors for default mode with upscaling, green without upscaling. The darker tones show the speed without the final descriptor extraction step. Dataset images are sorted by increasing number of extracted features.

expensive and not recommended unless more than 1000 descriptor extractions can be avoided by performing it.

The pyramids subset of the Holiday dataset (Figure 3(c)) is an exceptional example. Besides the high resolution, these photographs are very feature-poor, and partly taken at night and with considerable blur. As a consequence, the number of keypoints is very small, not even reaching the desirable 2000-descriptor mark at an upscaled resolution of 5120x4840. The required computing time is therefore dominated by the constant elements, construction of the pyramid, computation of the DoG and keypoint search. For images this large, we are so far just reaching 25 fps. For a feature-rich image of this resolution, we would not achieve real-time speeds yet (but neither can other implementations).

7 CONCLUSION

The goal of writing an implementation of SIFT that is faithful to the original paper [8] on modern GPU hardware has been achieved with PopSift. We have demonstrated the performance of PopSift both in terms of keypoint detection and extraction results. It competes with the best known open source implementations. In terms of speed, it fulfils the promise of real-time feature extraction from 1080p frames on an NVidia GTX 1080 card in all but the most exceptionally feature-rich cases. In most scenarios, in particular those that are relevant for real-time applications, it overperforms with considerable computing time to spare. Specifically for our own use case, it extracts up to 10000 descriptors from non-upscaled 1080p videos at 100 fps or better.

Publication of PopSift does not mean that our work is done. We implemented PopSift by the book because only the well-studied, repeatedly implemented original algorithm allows us to verify with high confidence the correctness and quality of our implementation. In the future, we will add fast matching methods to PopSift (so far, only brute force is available), and extend PopSift with variations of SIFT that are known to increase its qualitative performance. We have been asked by project partners to add Upright SIFT [2], and we are looking at other advanced variants.

8 ACKNOWLEDGEMENTS

PopSift was developed within the Horizon 2020 projects POPART (#644874) and LADIO (#731970) and support from Norwegian national project *PCIe* (#235530)

REFERENCES

[1] Relja Arandjelovic and Andrew Zisserman. 2012. Three things everyone should know to improve object retrieval. In *CVPR*. 2911–2918.
[2] Georges Baatz, Kevin Köser, David Chen, Radek Grzeszczuk, and Marc Pollefeys. 2010. Handling Urban Location Recognition as a 2D Homothetic Problem. In *ECCV*. 266–279.
[3] Mårten Björkman. 2014. Celebrandil/CudaSift. (2014). https://github.com/Celebrandil/CudaSift
[4] Mårten Björkman, Niklas Bergström, and Danica Kragic. 2014. Detecting, segmenting and tracking unknown objects using multi-label MRF inference. *Comput. Vis. Image Underst.* 118 (2014), 111–127.
[5] Kai Cordes, Bodo Rosenhahn, and Jörn Ostermann. 2013. *High-Resolution Feature Evaluation Benchmark*. Springer, 327–334.
[6] Morel Jean-Michel. 2011. Is SIFT Scale Invariant? *Inverse Problems and Imaging* 5, 1 (2011), 115–136.
[7] Tony Lindeberg. 1994. Scale-space theory: A basic tool for analyzing structures at different scales. *Journal of Applied Statistics* (1994).
[8] David G Lowe. 2004. Distinctive Image Features from Scale-Invariant Keypoints. *Int. J. Comput. Vis.* 60, 2 (2004), 91–110.
[9] Krystian Mikolajczyk and Cordelia Schmid. 2005. A performance evaluation of local descriptors. *Trans. Pattern Anal. Mach. Intell.* 27, 10 (2005), 1615–1630.
[10] Krystian Mikolajczyk, Tinne Tuytelaars, Cordelia Schmid, Andrew Zisserman, Jiri Matas, Frederik Schaffalitzky, Timor Kadir, and Luc J. Van Gool. 2005. A Comparison of Affine Region Detectors. *Int'l J Comp. Vis.* (2005), 43–72.
[11] Christoph Strecha, Alex Bronstein, Michael Bronstein, and Pascal Fua. 2012. LDAHash: Improved Matching with Smaller Descriptors. *Trans. Pattern Anal. Mach. Intell.* 34, 1 (2012), 66–78.
[12] Andrea Vedaldi and Brian Fulkerson. 2010. Vlfeat: An Open and Portable Library of Computer Vision Algorithms. In *ACM MM*. 1469–1472.
[13] Changchang Wu. 2013. A GPU implementation of David Lowe's Scale Invariant Feature Transform. (2013). https://github.com/pitzer/SiftGPU

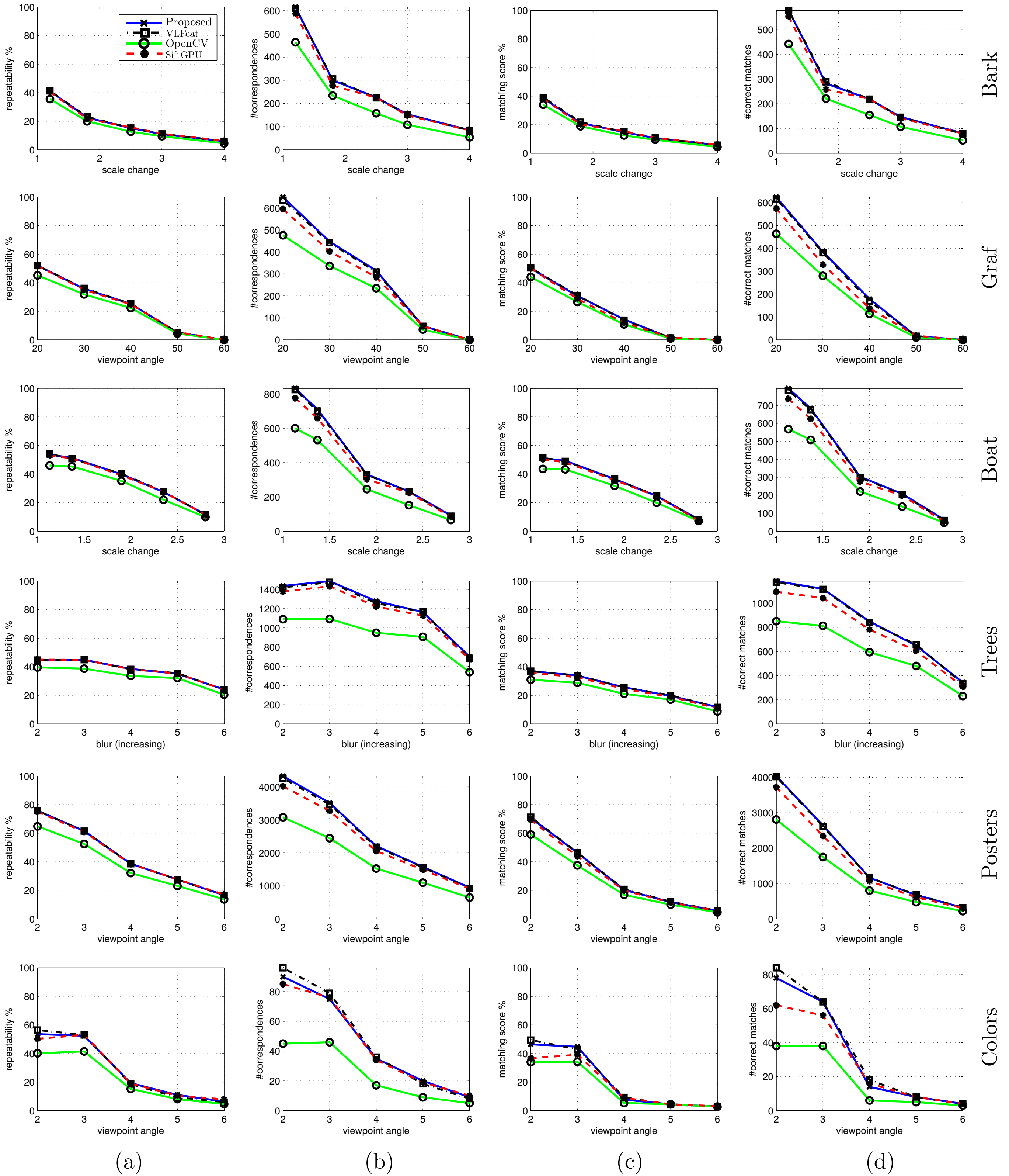


Figure 4: Qualitative performance: (a) Repeatability rate, (b) number of correspondances, (c) matching rate and (d) number of correct matches evaluated on excerpt from VGG and Hannover datasets. Higher values are better.