A SIFT DESCRIPTOR FOR FEATURE MATCHING

Nicolas Hafner, Costanza Maria Improta, Zsombor Kalotay, Jan Leutwyler

Department of Computer Science
ETH Zürich
Zürich, Switzerland

ABSTRACT

The Scale Invariant Feature Transform is used to detect important regions in an image and allows tracking of such regions across transformed variants of the same image. We examine an existing implementation of the SIFT algorithm in C++ and compare it to our own in C. Our own implementation presents a cleaner interface, and manages to outperform the original C++ implementation in every substep of the algorithm. Using AVX2 vectorisation and other optimisation techniques we achieve speedups of up to 10x in select parts of the algorithm.

1. INTRODUCTION

Motivation

The SIFT algorithm can be used for object recognition and object tracking, both important aspects of computer vision. The features it recognises in an image are robust against affine transformations, and even against variations in lighting. This is useful to recognise objects in video and track them as they move over time. Among other applications it can also be used for image stitching by aligning features with similar descriptors. Especially for the use in robotics and other real-time environments such as camera tracking, an efficient implementation of SIFT is important.

Optimisation of SIFT is non-trivial as it is a rather complicated algorithm to begin with. Many of its steps also require a large amount of data access, sometimes over many different sets of data at once, complicating access locality.

In this paper we first design a reasonable architecture for SIFT that is amenable for future optimisations. We then write a straightforward implementation of SIFT in this architecture, including an automated test suite for verification and performance measurement. From there we perform both standard C optimisations as well as manual single-core vectorisation using AVX2, measuring and comparing along the way.

Related work

The original SIFT algorithm is outlined in a paper by **?**][**?**].

Our implementation is based on the ezSIFT implementation by **?**] as well as the implementation in OpenCV[**?**]. Our own implementation is standalone like ezSIFT, but offers a pure C interface that can be used by any other project or language that supports C calling convention. We also do not depend on any particular image format, but instead leave the loading of image data up to the user.

2. BACKGROUND

In this section we will outline the steps of the SIFT algorithm as detailed in the original paper[**?**], as well as our overall analysis of the algorithm in terms of asymptotic complexity and cost.

The SIFT Algorithm

Our implementation of the SIFT Algorithm is based on the work of Lowe et al.[**?**] and EZSift[**?**]. Their algorithm can be split into four stages.
The first stage is the *Scale-space extrema detection*. Here the scale space of an image is defined as a function $L(x, y, \sigma)$, that is produced from a convolution of a Gaussian kernel $G(x, y, \sigma)$ with an input image $I(x, y)$

$$L(x, y, \sigma) = G(x, y, \sigma) * I(x, y) \tag{1}$$

To efficiently detect extrema in scale space, they generate a difference-of-gaussian pyramid, where k represents a constant multiplicative factor

$$D(x, y, \sigma) = L(x, y, k\sigma) - L(x, y, \sigma) \tag{2}$$

To detect local minima and maxima in $D(x, y, \sigma)$, they compare the pixel to all 8 neighbours in the same scale, as well as to each 9 neighbours in the scale above and below of the pyramid.

The second stage is *Keypoint Localization*. In this stage they use the Taylor Expansion of the scale-space function $D(x, y, \sigma)$ to determine the interpolated location and the scale of the extrema candidates. Also, final keypoints are selected based on measures of their stability. Hence, they eliminate edge responses to guarantee a high measure of stability.

The third stage is the *Orientation Assignment*. Here, one or more orientations are assigned to each keypoint location based on local image gradient directions. They use pixel differences to calculate gradient magnitude $m(x, y)$ and orientation $\Theta(x, y)$ from the Gaussian pyramid.

$$P(x, y) = L(x + 1, y, \sigma) - L(x - 1, y, \sigma) \tag{3}$$

$$Q(x, y) = L(x, y + 1, \sigma) - L(x, y - 1, \sigma) \tag{4}$$

$$m(x, y) = \sqrt{P(x, y)^2 + Q(x, y)^2} \tag{5}$$

$$\Theta(x, y) = \arctan \frac{Q(x, y)}{P(x, y)} \tag{6}$$

Then, they form an orientation histogram from gradient orientations of sample points within a region around the keypoint. This histogram has 36 bins covering the 360° range of orientations. Each sample added to the histogram is weighted by its gradient magnitude. Peaks correspond to dominant directions of local gradients.

The fourth and final stage is the *extraction of the keypoint descriptor*. In the previous stages, they have assigned an image location, scale, and an orientation to each keypoint. As a last step they compute a descriptor for the local image region that is highly distinctive, but also as invariant as possible to remaining variations (e.g., illumination). They achieved this by taking a 16x16 pixels patch around a keypoint, subdividing this patch into 16 4x4 blocks, and creating for each of these smaller blocks an 8-bin orientation histogram. This gives a total of 128 bin values, which they represented as a vector to form the keypoint descriptor.

Cost Analysis

We analysed the algorithm precisely using a counting system. We separately considered floating point adds, muls, divs, and the amount of memory transferred. You can find the counts for a sample image tabulated in Table 1. We make use of this counting scheme in order to automatically generate the performance and roofline plots shown in section 4. The counting facility is described in more detail in section 3.

A precise asymptotic analysis of the algorithm is very involved, and can be found in a paper by **?**].

3. OUR METHOD

With the background of the algorithm covered, we will now detail our implementation, which we call "ethSIFT" for short.

Testing and Measurement Framework

In order to verify the continued correctness of ethSIFT over the optimisation steps, and in order to conveniently measure the performance of the individual algorithm steps, we implemented a custom testing and measurement framework. In this framework, a new test can be defined using a special macro, `define_test`. The macros `with_measurement` and `with_repeating` allow convenient definition of regions that should be considered for measurement within the test, and finally the macro `fail` can be used to signal a test failure and give an appropriate description. An example test definition is shown in 1.

```
define_test(SampleTest, 1, {
    if(!prepare_things())
      fail("Test setup failed");
    with_repeating(compute_something());
  })
```

Listing 1: A sample measurement test definition.

`with_repeating` automatically sets up a warmup loop followed by a loop of measurements of its body, allowing very convenient definition of repeat measurements. Depending on the presence of the `USE_RDTSC` compiler flag, a measurement section will either measure the runtime using C++' `std::chrono`, or the cycle count using the `rdtsc` instruction. This allows us to measure both in separate runs with minimal noise, without having to manually change or duplicate any of our code.

The tester framework automatically picks up any test definitions and will run them in order of definition. For tests that incur measurement, the measurement values are automatically recorded. The tester will continuously compute and display the median, as well as the median absolute deviation when the tests are run, and finally output the last recorded values to a CSV file for plot generation.

In order to capture accurate counts of flops and memory use, we also defined a custom set of macros that can be used to increase a relevant counter. We then manually invoke these macros throughout our code base whenever we explicitly perform a flop or access memory. These counters are only active when compiled with the `IS_COUNTING` flag, to avoid disturbing normal operations of the library. The counter values are output to a separate CSV file for processing in our plot system.

Overall Architecture

The overall architecture of ethSIFT was designed to allow allocations to be factored out of the primary loops as much as possible, and to allow the user more control over the individual steps where necessary. To this end every step in the SIFT algorithm has a corresponding function in ethSIFT. To minimise heap access, all function arguments that are used as input are passed by value, including image structures. The return value of all ethSIFT functions is normalised to be a success value, though this is only really meaningful for functions that can fail, such as allocations.

We use a global initialisation function to precompute shared values such as the Gaussian

Step	Adds	Mults	Divs	Bytes Transferred
Downscale	0	0	0	6220800
Convolution	37324800	37324800	0	286891200
Gaussian Pyramid	438110040	438110040	0	2838231226
DOG Pyramid	13820160	0	0	176898096
Gradient & Rotation Pyramids	66288005	24876180	8292060	265345968
Histogram	4600	3576	2	13010
Extrema Refinement	92	102	3	1285
Keypoint Detection	977237	774713	2184	1724135010
Descriptor Extraction	4524956	2249352	300	10798360
Full Run	529942750	469143365	8296348	5030656822

Table 1. Recorded flop counts for our implementation, for a sample 1080p image.

kernels and temporary memory regions that are used during kernel convolution. Finally we provide an optimised image pyramid allocation function that allows allocating all images of a pyramid in a single allocation call. A user may use this function to pre-allocate the pyramids and re-use them during multiple SIFT analysis runs. With pre-allocated pyramids, the SIFT algorithm does not require any further heap allocations during execution. This makes it much cheaper for use in continuous image feeds or videos.

Finally we fix many SIFT parameters in place as constants in order to allow better static memory allocation and constant folding.

Gaussian Kernel Convolution

The Convolution function is one of the main building blocks of our SIFT implementation. It gets called several times to generate the Gaussian Pyramid, which is, computationally and time-wise, the main bottleneck in our implementation. And therefore, optimizations to this function promise the biggest payoff.

The main goal of the function is to blur an image by applying a 2D-convolution using 2D Gaussian kernel. In our baseline implementation, we made use of the separability of a Gaussian filter. Due to this separability, we could split the 2D-convolution in two 1D-convolutions using 1D Gaussian kernels by first filtering the image horizontally and in a second step vertically. In our implementation, the 1D-convolution for the horizontal row filter can be described as

$$g_{i,j} = \sum_{k=0}^{K-1} p_{i-\frac{K}{2}+k,j} * ker_k \tag{7}$$

here p is the input image, and ker is the 1D Gaussian kernel. This gets applied to every pixel in the input image. The vertical col filter can be described as

$$f_{i,j} = \sum_{k=0}^{K-1} g_{i,j-\frac{K}{2}+k} * ker_k \tag{8}$$

In our implementation we used the horizontal row filter only and just transposed its output, this allowed us to reuse the same filter function for the vertical part. After the second row filter, the results gets transposed again.

For our standard-C optimizations, we increased ILP by unrolling our loops wherever possible. And, we also applied some tricks, e.g., scalar replacement, to overcome compiler limitations. In our best version, we managed to achieve an up to 2 times speed-up in performance.

We further optimized our implementation using AVX intrinsics. Our best performing optimization applies the 1D convolution to 8 pixels at once. Further, we made use of FMAs to calculate the sums in equation 7 and 8. This way we managed to achieve a speed-up of more than 4 times compared to the baseline.

Gradient and Rotation Computation

One of the bottlenecks we found during profiling was the calculation of the Gradient & Rotation Pyramids.
The calculation of the two pyramids involves differences on given row r and column c in the form of two temporary variables $t1 = g(r-1,c) - g(r+1,c)$, $t2 = g(r,c-1) - g(r,c+1)$ where g is some Gaussian filtered image from the Gaussian pyramid and $g(r,c)$ a pixel at row r and column c. Due to the fact that for each row and column we look at the row (and column) above and below (resp. to the left and to the right) of it, we had in our first implementation an inline method for getting the pixel at a given position and checking for the border. In case we would have exceeded the border in some form, it returned the value at the border. The naive implementation of this method included branching and a worst case of eight conditional checks.
In a first step of our Standard-C optimization, we were able to get a performance boost by reducing the worst case checks from eight to a constant amount of four. Also we unrolled an outer loop of three iterations and prevented recalculation of variables which were recyclable. This lead to a more than 2x performance boost.
For the AVX2 implementation we then had to come up for a solution regarding the border issue. We decided to calculate the border as we have done before and only parallelize the middle of the image. Since the calculations also required the atan2 function and there was no intrinsics function for that, we also implemented an inline method using AVX2 intel intrinsics which we called "eth_mm256_atan2_ps". Thanks to this implementation, we were able to get an approximate speed-up of factor 11.

Vectorising atan2

We implemented an AVX2 vectorized version of the atan2 function, which was mainly used in the Gradient and Rotation Pyramid Computation. Our implementation is able to take ___m256 float vectors x and y as input and compute the eight corresponding values in parallel. Difficulties regarding branching in the atan2 function we overcame by calculating and applying vector masks instead.

4. EXPERIMENTAL RESULTS

In this section we detail our performance measurements and discuss how our changes affected the overall runtime and performance of our implementation. We also take a brief look at how different compilers and compiler flags changed the performance.

Experimental Setup

Our measurements were performed on a machine running an Intel Core i7-8700K CPU with a fixed clock at 4.4GHz. This model of the CPU is part of the Coffee Lake series and has a 6x32 KB 8-way L1-cache, 6x256 KB 4-way L2-cache, and a 12 MB 16-way shared L3 cache.

Unless otherwise stated, we used ICC 19.1.1.217, with `-g -O3 -mfma -mavx2 -march=skylake -flto -ffast-math -fno-unsafe-math-optimizations` for optimisation flags. We used input images in sizes of 240p, 360p, 480p, 720p, 1080p, 2160p, and 4320p.

Results

TODO We achieved huge result. Our results are the best results ever. Noone ever achieved such great results.

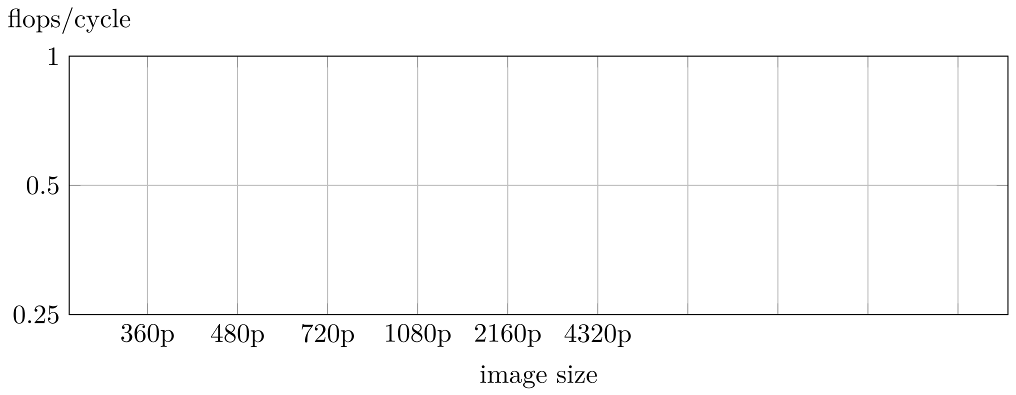

Fig. 1. Performance comparison of our optimisation steps for the whole algorithm

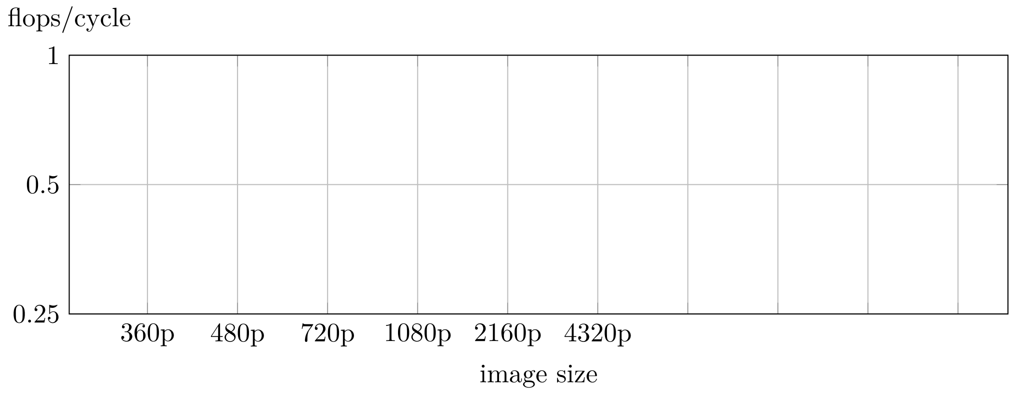

Fig. 2. Performance comparison of our optimisation steps for the gaussian kernel convolution

5. CONCLUSIONS

In this paper we implemented a high performance version of the SIFT algorithm that can deliver results for 1080p full HD images in realtime. This allows analysis of direct high resolution video streams to detect and track objects as they move around the image. This tracking can be used for camera focus adjustment, computer vision in robotics contexts, and other automated recognition systems.

The biggest bottlenecks in the SIFT algorithm are by far the Gaussian kernel convolution and the generation of the rotation and gradient pyramids. The Gaussian kernel convolution in particular becomes increasingly dominant as the size of the input image increases.

While a large part of the kernel convolution can be vectorised, a big hindrance is the moving window over the kernel, which causes unaligned memory access and thus many split loads that slow down the loading. We attempted to circumvent this by prefetching and shuffling as required instead, but this resulted in overall worse performance as cross-lane shuffles are expensive.

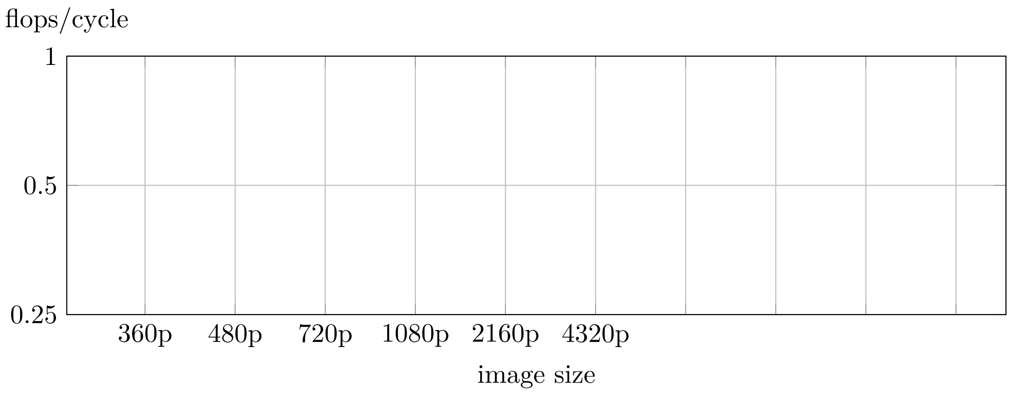

Fig. 3. Performance comparison of our optimisation steps for the gradient and rotation pyramids

Overall, using basic C and AVX optimisations we managed to achieve a speedup of approximately 3x over the baseline implementation, and of up to 10x in select parts of the algorithm.

The source code for ethSIFT can be found under the zlib license on GitHub: https://github.com/shinmera/ethsift.

6. FUTURE WORK

We think it might be possible to replace the Gaussian kernel convolution with an FFT approach to generate the blurred images instead. Whether this would result in improved performance over a fully optimised kernel convolution implementation is not entirely clear to us at this point though.

SIFT steps that are not on the hot path could also be improved further, though we don't see a large benefit to doing so, as the time is massively dominated by the Gaussian convolution and rotation/gradient computations.

Finally, the generation of the pyramid levels could be distributed onto multiple processors relatively easily, as many levels are independent.

7. CONTRIBUTIONS OF TEAM MEMBERS

TODO

Nicolas Implementation of performance and runtime measurement systems, analysis and optimisation of memory allocations and layout, optimised keypoint detection, histogram computation, helped Jan with analysis and optimisation of the gaussian kernel convolution, as well as with the cycle and memory counters.

Zsombor

Jan Implementation of flop and memory counting system, implemented and optimized gaussian kernel convolution, implemented roofline plots, helped writing tests for ethsift functions, helped Zsombor and Costanza with micro and macro analysis.

Costanza