SPEEDING UP SPEEDED UP ROBUST FEATURES

Team 42 - Carla Jancik, Valentin Wolf, Laurin Paech, Sebastian Winberg

Department of Computer Science
ETH Zurich, Switzerland

ABSTRACT

We propose multiple optimizations for implementing the Speeded Up Robust Features (SURF) [1] algorithm a widely used scale-invariant feature detector and descriptor. Our optimized implementation gives significant speed-ups and achieves state-of-the-art single-core performance despite using the more sophisticated M-SURF descriptor [2]. As a major contribution, we propose a novel way of exploiting the symmetry of rounding and separability of Gaussian kernels that significantly reduces the computation required for the double Gaussian weighting scheme of the descriptor. In extensive experiments, we analyze our optimizations and show their impact not only on runtime but also on memory and cache performance.

1. INTRODUCTION

Efficient feature detection and extraction algorithms are of great importance for many applications in Computer Vision. Robot navigation requires real-time performance at low power consumption [3, 4] and large scale structure-from-motion pipelines [5] or reverse image search engines [6, 7] need to extract features from millions of images in reasonable time. While SURF's [1] efficient yet robust algorithm greatly reduced the computational burden - making it the go-to option for performance-critical applications - it's existing implementations are not well optimized and do not make full use of modern processors.

Contribution. In this paper we present an optimized implementation of Speeded-Up Robust Features [1] that is, to the best of the authors' knowledge, the fastest single-core implementation by a factor of $2.8\times$, despite using the more sophisticated Modified-Upright-SURF (M -SURF) descriptor from [2].

Related work. Apart from SURF [1] various other efficient scale-invariant feature extraction algorithms have been proposed in recent years, e.g. ORB [8], FREAK [9], BRISK [10] or (A)KAZE [11, 12]. However, despite the apparent need for speed, no highly optimized open source implementation exists for any of these algorithms. We show that there is significant room for improving the runtime of the SURF algorithm without changing the underlying algorithm.

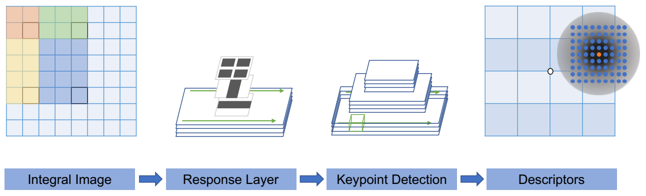


Fig. 1: Overview of the SURF algorithm.

2. BACKGROUND ON SURF

The SURF algorithm is an efficient scale-invariant feature detection and extraction algorithm. Its novelty is the exploitation of efficient box integral computations with integral images. As shown in Figure 1, SURF can be split into the following four parts: (1) computing the integral image, (2) computing the response layers i.e. approximations of second-order derivatives using box integrals at multiple scales, (3) non-maximum suppression to find keypoints (4) constructing a descriptor of the local neighborhood around each keypoint. In the following, the parts are described in more detail.

Integral Image. Computing the integral image J of an image I is straightforward. Instead of storing pixel values at a location $J(x, y)$ one stores the partial sum over pixel values of the rectangle that starts in the upper left corner and has its lower right corner at coordinate (x, y), i.e.

$$J(x,y) = \sum_{i \le x, j \le y} I(i,j)$$

Once the integral image is computed, we can compute the sum of pixel values of the original image in an arbitrary sized rectangle in constant time. Simply evaluate $D - C - B + A$ where A, B, C and D are the values of the integral image J at the top left, top right, bottom left and bottom right corner coordinates of the rectangle.

Fast-Hessian Detector. Following past approaches [13, 14, 15, 16], feature detection is based on the Hessian matrix, which consists of second-order derivatives in each direction and can be used to determine the value of change. Their local extrema can then be regarded as potential keypoints. Following [1], consider a point $\mathbf{x} = (x, y)$ in the image and

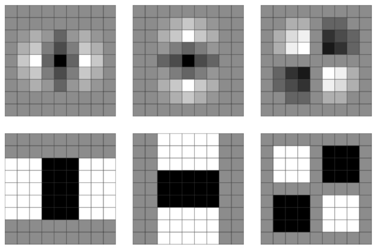


Fig. 2: Figure taken from [17]. Top: Laplacian of Gaussians. Bottom: D_{xx}, D_{yy} and D_{xy} boxfilter approximations.

scale σ, the Hessian matrix $\mathcal{H}(\mathbf{x}, \sigma)$ is defined as:

$$\mathcal{H}(\mathbf{x}, \sigma) = \begin{bmatrix} L_{xx}(\mathbf{x}, \sigma) & L_{xy}(\mathbf{x}, \sigma) \\ L_{xy}(\mathbf{x}, \sigma) & L_{yy}(\mathbf{x}, \sigma) \end{bmatrix}$$

where $L_{xx}(\mathbf{x}, \sigma)$ is the convolution of the Gaussian second-order derivative (Laplacian of Gaussians) with the image I in point $\mathbf{x}$. The determinant of the Hessian provides the scalar interest measure and thus determines the value of change at point $\mathbf{x}$. The SURF algorithm proposes to approximate the Laplacian of Gaussians with box filters, which can be efficiently computed using the integral image. The approximated filters are denoted by D_{xx}, D_{yy} and D_{xy}, referring to the filter size normalized derivatives in the corresponding directions (see Fig. 2). Resulting in the determinant of the Hessian as:

$$\det(H_{approx}) = D_{xx}D_{yy} - (0.9D_{xy})^2$$

With this formula the response of the box filters is computed for every pixel, stored in so-called response layers and used to find keypoints.

A scale space of three octaves is introduced, where each octave consists of four response layers. Each response layer corresponds to a different filter size. Typically, images are down-scaled and then convolved with Gaussian kernels to achieve scale invariance. However, this is rather inefficient and introduces dependencies between layers. Instead, for SURF filters are up-scaled and applied to the original image, exploiting that box integral computation, using the integral image, takes constant time. After computing the response layers, non-maximum suppression in a $3 \times 3 \times 3$ neighborhood is applied, discarding all non-maximal values. The maxima are then interpolated to sub-pixel locations yielding the keypoints and respective scales.

Keypoint Descriptor. With keypoints and their scales identified, a constant size feature vector describing the local patch around each keypoint, the so called descriptor, is computed. We implemented and optimized the Modified-Upright-SURF (M-SURF) [2] descriptor which yields improved performance compared to the original upright descriptor proposed in [1], by using a more elaborate weighting scheme. The computation of the M-SURF descriptor

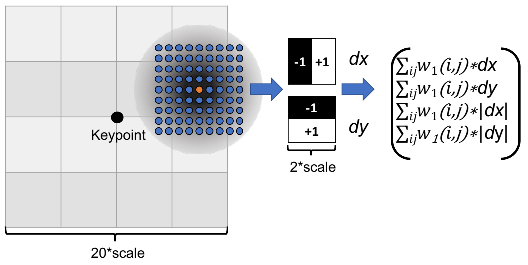

Keypoint
-1
+1
dx
-1
+1
dy
2*scale
20*scale
$\sum_{ij} w_1(i,j) * dx$
$\sum_{ij} w_1(i,j) * dy$
$\sum_{ij} w_1(i,j) * |dx|$
$\sum_{ij} w_1(i,j) * |dy|$

Fig. 3: Computation of 9×9 Haar wavelet responses and the inner Gaussian weighting scheme for one sub-patch.

for one keypoint with scale σ can be grouped into three steps. First, a $20\sigma \times 20\sigma$ patch around the keypoint is taken and then further split into 4×4 equal-sized square sub-patches. Second, as shown in Figure 3 box filter approximations of the first-order derivative, the so called Haar wavelet responses, in both x- and y-direction are computed at 9×9 sample locations centered around each sub-patch center. For each sub-patch these responses in x- and y-direction, as well as their absolute values are weighted by a Gaussian centered on the corresponding sub-patch (later referred to as the inner Gaussian) and summed up, yielding four values describing the sub-patch. Third, the resulting values of the sub-patches are weighted by a second Gaussian that is centered on the keypoint (referred to as the outer Gaussian). Finally, the weighted features from each of the 16 sub-patches are concatenated and normalized to a unit vector resulting in a 64-dimensional descriptor for the keypoint.

Cost Analysis. As a cost measure, we chose the FLOP count, taking into account single-precision additions, multiplications, FMAs, as well as casts, rounds and the exponential function. At first, we counted the FLOPs and memory movement manually for each function. However, due to the complexity of the algorithm, we partially relied on data from *Intel Advisor* for the roofline plot, after carefully cross-validating the results with our own findings. Note that Drews et al.[18] investigated the computational complexity of SURF. Given an input image with height m and width n their results show that the integral image has an asymptotic runtime of $\theta(mn)$. Computing the response layers and extracting the keypoints is in $O(mn - \frac{mn}{\min(m,n)^2})$. The runtime for the descriptor creation is constant, i.e. $O(k)$, in the number of detected keypoints k.

3. METHODOLOGY

In this section, we first describe the baseline implementation, followed by a detailed explanation of our proposed optimizations for the computation of the integral image, response layers and descriptors. As input we assume squared images divisible by 8. However, our implementation is easily extendable to arbitrary sized images.

3.1. Baseline Implementation

Our baseline implementation is an adaptation of the OpenSURF C++ library [17], which also uses M-SURF descriptor. Note, that due to the length of the algorithm we only implemented the upright version of the descriptor, which is not rotation-invariant. However, we see no hindrance in applying the proposed optimizations to the rotation-invariant descriptor.

3.2. General Optimizations

We noticed that functions from the `math.h` header were used with `float` arguments despite working with `double`. Due to the implicit back and forth casting this is inefficient. Hence, we replaced the functions with their correct counterparts.

Next, we applied scalar replacement, removed unnecessary and simplified computations by making use of associativity. Also, we reduced redundancies by moving computation to the outer most loop.

To reduce cache misses, we experimented with different memory layouts for the buffers (e.g. integral image, response layers, etc.). However, cache and page aligning and even laying them out "flat" in one big buffer with different offsets in between, gave no significant performance improvements. This is not surprising, as irregular and strided memory access patterns dominate throughout the algorithm and sequential accesses are rare.

3.3. Integral Image

Even though the integral image computation is only a minor contributor to the overall runtime, we pursued two interesting optimizations. First, we implemented the algorithm proposed in [19] which increases instruction level parallelism (ILP), by computing multiple rows with a shifted offset and thereby fully utilizing multiple floating point addition ports.

Second, we vectorized the integral image computation. This is not straightforward as one needs to break the inherent data dependency of the algorithm. Inspired by the SIMD library [20], we leverage the AVX2 *integer* intrinsic `_mm256_sad_epu8`. The intrinsic adds up chunks of eight consecutive unsigned 8-bit integers and stores the result in the low 16 bits of these 64-bit chunks. By loading the byte-sized pixel values in a specific pattern the inter-iteration dependency can be partially broken, allowing us to compute multiple integral image terms in parallel. As this trick only works with integers, the result has to be converted to single-precision floating points in the end, to be compatible with the rest of the algorithm.

Apart from the increased performance, using integers is numerically more stable due to the lack of rounding errors. However, it comes with the downside of input images being limited to at most 2^{23} (e.g. 2048×1024) pixels. Otherwise the 32-bit *signed* integers could overflow.

3.4. Compute Response Layer

The computation of the response layers is is a major contributor to the overall runtime. Consisting mostly of box filter convolutions with the image, a significant number of memory accesses are performed while little actual computation occurs. Hence, like most of SURF, it is inherently memory bound. Further profiling showed that bound checks, when computing box filters, are a major bottleneck.

Cache Locality. To somewhat alleviate memory boundedness we experimented with various ways to reduce cache misses and thereby memory accesses. First, we applied blocking with scalar replacement to the computation of the response layer. We assumed that computing filter responses of adjacent rows would improve cache locality. However, close by filters do not access the same memory locations, and only share cache lines when they are in the same row of the integral image.

As a next approach, we computed all response layers simultaneously for filters centered at the same location. This removes the need to iterate over the image multiple times . Intuitively, one would expect reuse of cache lines. However, due to the different filter sizes memory accesses are not in the same row and thus never in the same cache line when computing multiple response layers for one pixel location. This insight invalidates the approach as, in addition to not gaining any locality, we further enlarge the working set of the cache and thereby only increase cache misses.

Conditional Removal. As stated above, profiling showed that the repeated bound checks in the box filters are costly. Our approach was to remove them by analyzing how box filters traverse the integral image, particularly the inner and outer corners of the filters. The conditionals were replaced by for-loops, by defining the ranges that are seen in Figure 4a as differently colored areas that are only dependent on the filter layout. Thus we know the positions of the filter corners with respect to the image and their corresponding value. For D_{yy} box filters this case distinction applies:

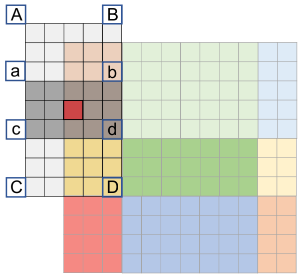
```
if $filter \leq image$ then
    Separate into 9 cases (see Fig. 4a)
else
    (a) Filter is longer but narrower than image
    (b) Filter is longer and wider than image (i.e.
      completely covering the image)
end
```

Additionally, the inner part of the filter also needs to be taken into account leading to a total of 15 cases if filter $\leq$ image. Following the example of Figure 4a: the filter is in the top-left corner, since *A*, *B* and *C* (outer filter-corners) are out-of-bounds to the top and left of the integral image, they correspond to zero. The sum of the outer-part of the filter is the value of the integral image at *D*. For the inner part of the filter, *a* and *c* correspond to zero, while *b* and *d* correspond to their respective values in the integral image.

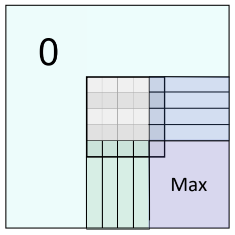
Furthermore, we analyzed whether the computations of the D_{xx} box filter could be reused for the inner-part of the larger D_{yy} box filter. Unfortunately, due to the scaling of the box filters, this is only possible in the case of D_{xx} box filter of size 9 and inner-part of D_{yy} box filter of size 15. Otherwise, there is a size mismatch. However, we found that the D_{xy} box filter makes redundant computations by sequentially covering the same area with equally sized squares. Computing only the upper squares over the whole image and reusing them prevents this.

Padding. Due to the complexity of optimizing every box filter separately, we padded the integral image instead. In particular, we introduced a boundary area around the integral image such that every access outside results in the appropriate value. To be precise, values above and to the left of the image are zero, and values to the right/bottom are simply the last element in each row/column (see Fig. 4b. Although all conditionals can be removed, this leads to more memory accesses.

Padding & Boxfilter Cases. To reduce memory accesses and floating point operations further, we combined our optimized D_{yy} box filter with the padded integral image. With the latter being used as a fallback mechanism to remove conditionals for the unoptimized D_{xx} and D_{xy} box filters. Additionally, this leads the way to apply scalar replacement for index calculations and reduces floating point operations.



(a) D_{yy} **subcases if filter $\leq$ image.**



(b) **Padding**

Fig. 4: (a) Image is split into 9 cases, depending on the filter location. I.e. as shown if the filter is in the top-left corner $A, B, C = 0$. (b) Original 5×5 integral image in the middle. Left and top filled with zeros. Right and bottom filled with the last element in each row and column respectively. Bottom right corner filled with max element.

3.5. M-SURF Descriptor

Basic Improvements. We inlined the Gaussian function used for the weighting and dropped its normalizing constant due to redundancy (the resulting descriptor is normalized in the end). Further, we changed the loop order to compute Haar wavelet responses row- instead of column-wise for improved locality.

Haar x- and y-direction at once. We noticed that during computation of the Haar wavelet responses in x- and y-direction the box filter function redundant memory accesses occur. By doing the computation for both directions at once and using scalar replacement we were able to decrease the memory accesses by 50%.

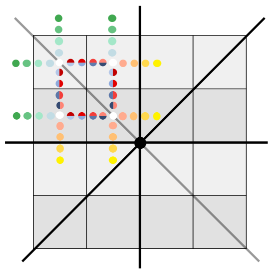
Reducing exponential function calls. A bottleneck of the M-SURF descriptors are repeated calls to the exponential function from the Gaussian weighting scheme. The 16 weights of the outer Gaussian are the same for all keypoints and can simply be hard coded. However, the weights from the inner Gaussian are responsible for over 99% of the `exp()` calls. Unfortunately, these can not simply be precomputed as they depend on the sample point locations, which themselves are dependent on the rounded multiples of the keypoint's (non-integer) scale. Thus these weights are non-identical for different keypoints and not even for sub-patches of the same keypoint.

However, we noticed the following: (1) when computing the Gaussian weight only the absolute (i.e. unsigned) distance to the center is required, (2) with rounding towards zero the rounded multiples of the (non-integer) scale are point symmetric around the keypoint (apart from their sign) and (3) by the separability of the Gaussian kernel, only the weights of the sample points that lie on the same x- or y-coordinate as the sub-patch center are needed to compute all other values. E.g. for a 2×2 Gaussian it holds that

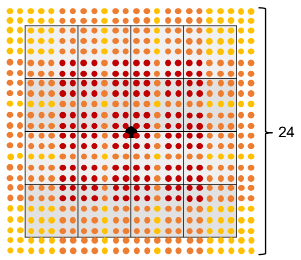
$$\frac{1}{4}\begin{bmatrix}1\\2\\1\end{bmatrix} \cdot \frac{1}{4}\begin{bmatrix}1 & 2 & 1\end{bmatrix} = \frac{1}{16}\begin{bmatrix}1 & 2 & 1\\2 & 4 & 2\\1 & 2 & 1\end{bmatrix}$$

As explained in Figure 5a, we can exploit these three insights and reduce the number of `exp()` calls per descriptor from 1296 to just 16. Note, that due to not having a normalization constant the weight of the sample points at sub-patch centers (white points in Fig. 5a) are equal to 1, making it unnecessary to compute.

Precompute Haar wavelets. The 9×9 sample points of a sub-patch, where we computed the Haar wavelet responses, overlap with adjacent sub-patches. This results in responses of the same location being computed up to four times (on average 2.2 times). In Figure 5b this pattern is visualized. To reduce redundant computation, we compute all Haar responses for a keypoint at once, store the results in



(a) **Distinct weights for inner Gaussian**



(b) **Haar wavelet recomputation pattern**

Fig. 5: (a) 17 distinct weights of the inner Gaussian - each depicted by a unique color - required to compute all others by exploiting separability of Gaussian kernels and point symmetry of rounding around a keypoint (black lines represent axes of symmetry). Note that sample points with two colors are used for different sub patches and weighted with differently in each. (b) Responses for yellow points are computed once, orange points twice and red points four times.

a 24×24 array and then access them as needed. This drastically reduces memory accesses and computation, greatly compensating the overhead from the array.

Just like in the computation of the response layers we noticed that the excessive bound checks from the box filters are an impediment when computing Haar wavelet responses. As a simple, yet effective workaround, we first check if any part of the descriptor will be out of bounds by checking its corner points. If not, no further boundary checks will be performed. Additionally, we implemented a version that makes use of padding, as introduced in 3.4, where boundary checks are no longer needed. While the latter performs better on small images the difference becomes negligible on larger images, as the percentage of descriptors that lie partly out of bounds decreases with image size.

Efficient Rounding. Surprisingly, profiling reported the frequently used `(int)roundf(x)` (to obtain indices from sub-pixel aligned coordinates) as a major cycle sink. Thus, we experimented and found that `(int)(x+0.5f)` for $x \geq 0$ and `(int)(x-0.5f)` for $x < 0$ is considerably faster, especially if the sign of x is known. We assume this is due to `roundf()` checking for border cases e.g. x = `NaN`, which are unnecessary in our case.

Blocking with Autotuning. Even though computing the Haar wavelets responses at once drastically reduced the computation, they remain the bottleneck of the descriptor. To improve ILP and cache locality we applied blocking with scalar replacement. As initial experiments showed promising gains we built a code generator capable of generating code with scalar replacements for all blocking variants including the corresponding timing infrastructure using Python. This is used to autotune the block size to a specific CPU.

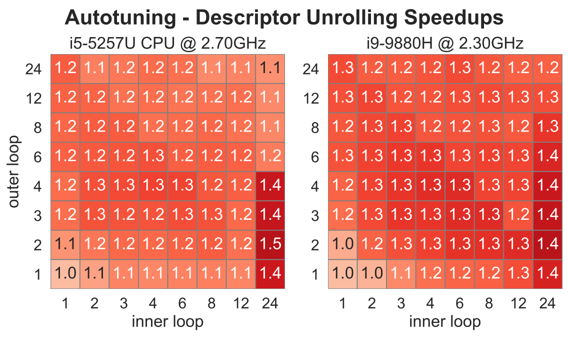


Fig. 6: Speedups of different block sizes for two different CPUs using all mentioned optimizations in 3.5 except for vectorization. Speedup is relative to no blocking (i.e. 1x1).

Due to limited resources and time, we shrunk the search space from 24^2 variants to 8^2 by limiting it to sizes that are true divisors of 24. Figure 6 shows the speedups obtained with different block sizes for two different CPUs.

Vectorization. We implemented a vectorized version with AVX2, where Haar wavelets for 8 sample points (in both x- and y-direction) are computed in parallel. As sample points are (irregularly) strided, loading from memory is done with the less efficient `_mm256_i32gather_ps` intrinsic. A version that uses the optimal unrolling for our CPU as determined from the previous autotuning was also implemented. The speedup gained from vectorization, compared to blocking is noticeable but far from the ideal $8\times$. This is partly because not the whole function is vectorized, but more so due to memory boundedness.

4. EXPERIMENTAL RESULTS

Experimental setup. For all experimental results a MacBook Pro 13" Early 2015 Intel Core i5-5257U CPU with 32KB, 256KB, 3MB caches and macOS Catalina 10.15.2 was used. We investigated different compiler flag optimizations. Adding the FMA flag did not appear to have any noticeable effect. For `-ffast-math`, we ran into numeri-

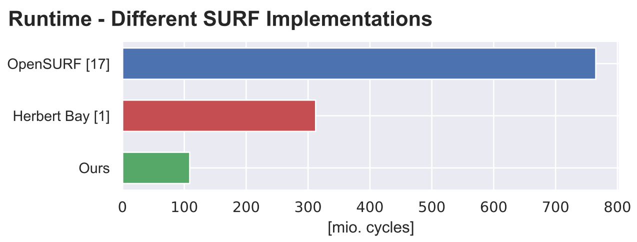


Fig. 7: Comparison of runtime between OpenSURF C++ implementation [17], the original implementation [1] and our fastest implementation.

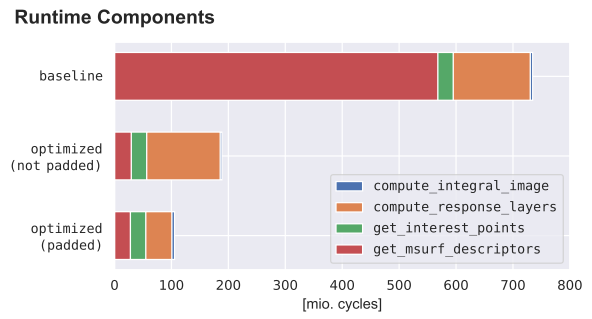


Fig. 8: Runtime comparison between three different versions for the 1024×1024 sunflower image, showing the contributions of the different subsections of the algorithm. Note that the interest points function is identical for all three versions, as basic optimizations no significant improvement.

cal instabilities when computing the M-SURF descriptors. Ultimately, we chose neither of them and remained with the Apple clang 11.0.0 compiler with the optimization flags `-O3` and `-march=native` for all our experiments. Furthermore, we also compared different compilers, more precisely gcc 9.3.0 and MSVC 19.14.26431.0 with clang. For gcc there were slight fluctuations in runtime for different functions. Regardless, the overall runtime stayed within a $\pm 3\%$ margin in comparison to clang. With Microsoft's compiler we got significantly worse results across the board with even up to 40% higher runtime, even with equivalent optimizations flags.

In the plots, we report results for the square sunflower image[1] of varying sizes 32, 64, ..., 2048. We are highly aware that power-of-two inputs (as used) can cause cache aliasing. Yet local benchmarking with different input sizes 40, 80, ..., 2560 reports the same performance for all functions and optimizations. This shows that cache aliasing does not create a bottleneck in our case.

Overall runtime and comparison. Figure 7 shows the runtime of our most optimized version, which combines the fastest version of each function, in comparison with other open source implementations. Our implementation is 7.5× faster than OpenSURF and 2.5× faster than the original SURF implementation [1] on the test image. The latter is even more noteworthy, considering that [1] uses the simpler SURF descriptors and not the computationally more involved M-SURF descriptors used by our implementation.

Figure 8 shows that padding with our proposed optimizations give a 7.0× speedup. The majority of it comes from the 21.9× speedup in the M-SURF descriptor computation. While the optimizations for the response layer give only a slight speedup in the unpadded version, the 2.8× speedup with the padding also significantly contributes to the reduction of the overall runtime of the program.

[1]see `/images/sunflower/` in our git repository

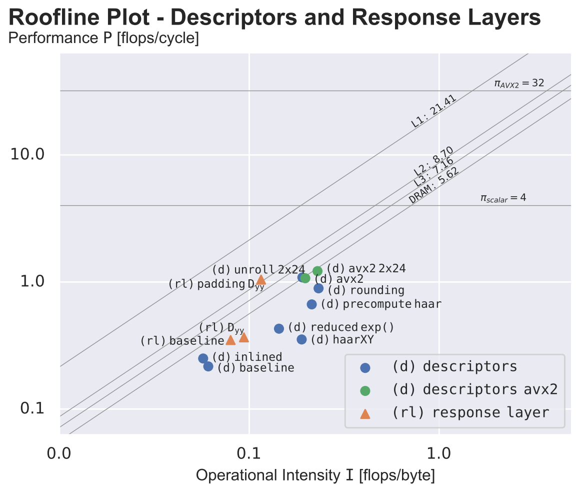


Fig. 9: Roofline plot for descriptor, as well as response layer improvements on the 1024 × 1024 image. Note that we distinguish AVX2 optimizations with a different color.

4.1. Integral Image

Integral Image Runtime	Unpadded	Padded
baseline	4.06	4.85
improved ILP	2.70	3.50
integer AVX2	2.40	3.15

Table 1: Different runtimes in mio. cycles of integral image optimizations on the 1024 × 1024 sunflower image. The padded version have a padding of 128 pixels on each side.

As a proof of concept, we wanted to show possible optimizations for the creation of the integral image. However, considering the negligible proportion of the overall runtime, we decided against pursuing this to its full extent. The improvement of ILP, by breaking dependencies, led to a speedup of 1.5× which is short of the expected 2×. We believe that with e.g. an autotuning infrastructure this can be further improved. The vectorized integer AVX2 version gives a slightly faster speedup of 1.69×. When padding is used the overall runtime increases slightly, as more memory needs to be accessed. This performance overhead, however, grows smaller for larger images, as can be seen in the performance plot in Figure 11. This small overhead is outweighed by the great runtime enhancements padding incurs for other functions.

4.2. Compute Response Layer

Cache locality. We benchmarked the cache locality and general optimizations (scalar replacement, inlining) and com-

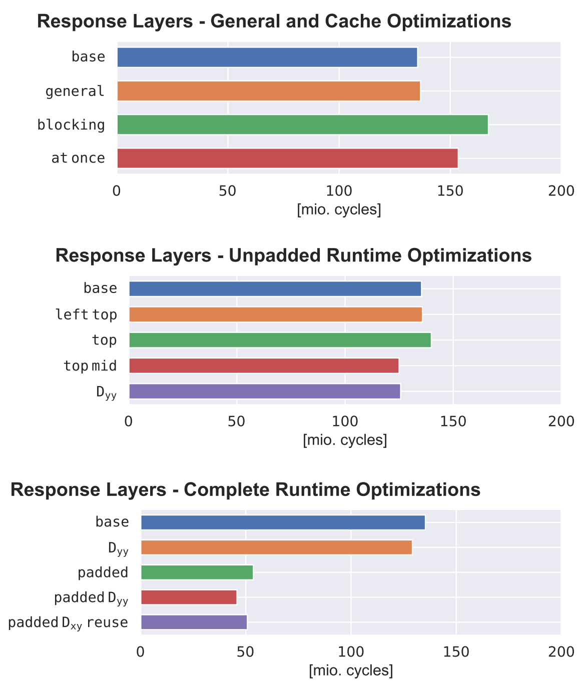


Fig. 10: (a) Cache locality optimizations.
(b) Progressive optimizations of D_{yy} box filter.
(c) Padded runtime optimizations of main optimizations. All runtime plots use sunflower image of size 1024×1024 and padding of 128 pixels.

pared them to gain insight into whether they yield runtime improvements. Unfortunately, all improvements failed to reduce the runtime as can be seen in Figure 10a. According to our expectations, most general optimizations were already done by the compiler and the difference between them is negligible. While we first assumed that the other two optimizations should increase spatial locality, any kind of blocking resulted in a significant runtime increase, even though we tested various automatically generated blocking sizes. We speculate that it inhibits compiler optimizations. Likewise, computing all response layers at once increases the runtime (see Section 3.4 for an explaiantion).

Conditional Removal. To reduce conditionals, we first optimized the D_{yy} box filter as we assumed that optimizations made can be easily adapted to the D_{xx} box filter due to similarities (them being rotated versions of each other). In Figure 10b we show the runtime of progressing optimizations as we went from the top-left corner to the fully optimized D_{yy} box filter. Especially for larger images the central area of the middle part (green in Fig. 4) is greater than boundary areas. For this reason, removing conditionals from that part leads to the biggest comparable speedup. By additionally reducing memory accesses the operational intensity is increased as can be seen in the roofline plot. The

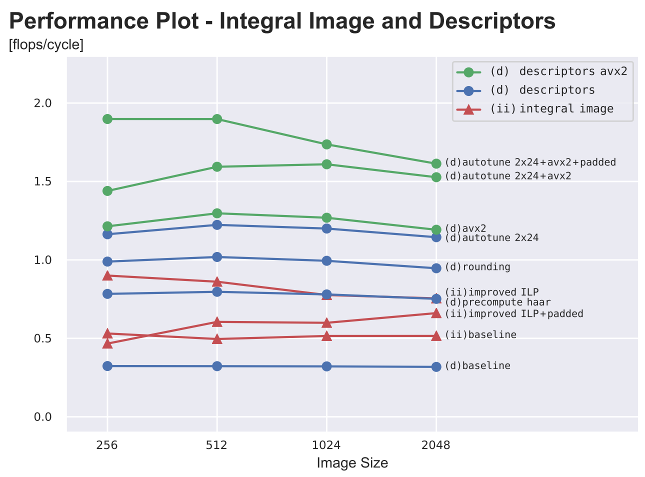


Fig. 11: Performance plot for integral image computation with floating points and M-SURF descriptors.

box filter optimizations gave a speedup of around 15%.

Padding. The addition of padding achieved by far the most significant decrease in runtime. The improvements are in line with our profiling results and confirm our assumptions even better than anticipated. An overview over our main implementations can be seen in Figure 10c. Without further optimizations padding yields a speedup of $2.5\times$. This includes the elimination of all boundary conditions, albeit slightly increasing memory accesses and adding costs by construction. However, the overall runtime improvements more than outweigh the additional costs (c.f. Fig. 7). Furthermore, combining padding and box filter optimizations results in a speedup of $3\times$, reaching the L2-bound in the roofline plot. We not only slightly increased operational intensity but decreased the runtime and therefore improved performance. We also implemented a padded optimization that removes redundant D_{xy} box filter computations. Unfortunately, this is limited to only the first four filters due to the sampling intervals. The speedup of this approach is $2.7\times$. Combining this with other box filter optimizations seems promising but is left as future work.

4.3. M-SURF Descriptor

Basic Improvements. The first improvements, referred to *inlined* in the plots, include general improvements (see Section 3.2) and inlining the computation of the Gaussian weights. This resulted in a $1.5\times$ speedup. The roofline plot shows increased performance and reduced operational intensity.

Haar x- and y-direction at once. The joined Haar wavelet computation for both directions yielded another relative speedup of approx. $1.5\times$. Since this optimization nearly halves the memory accesses the overall data move-

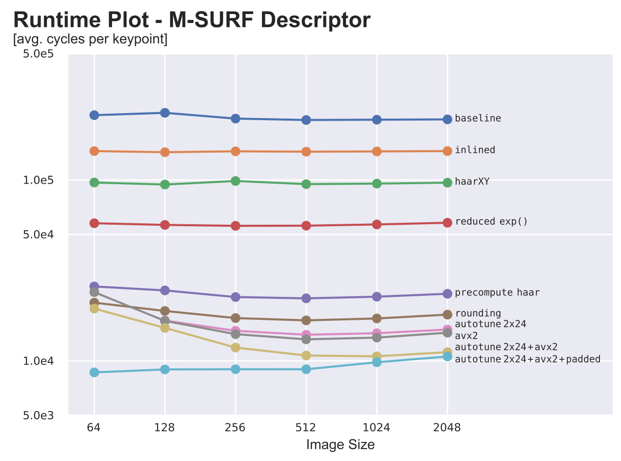


Fig. 12: Incremental improvements of M-SURF descriptor optimization. Plot shows average runtime to compute a *single* M-SURF descriptor for different image sizes.

ment and thus operational intensity I increases significantly, which can be observed in the roofline plot.

Reduction exponential calls. Next, we look at the optimization that heavily reduced the `exp()` calls by more than 99% making use of rounding symmetry and Gaussian kernel separability. In addition to the reduced `exp()` calls, it further reduces overall computation and thereby significantly reduces the FLOP count. Consequently both performance P and operational intensity I are decreased. The runtime, however, improves noticeably by another $1.68\times$ compared to the previous optimization.

Precompute Haar wavelets. The next step of our optimization used precomputation of the Haar wavelet responses, reducing the number of responses computed by a factor of $2.2\times$. This results in 40% fewer memory accesses, a nearly 30% decreased FLOP count and a speedup of $2.5\times$. This, as shown in the roofline plot, gives a drastic increase of both the performance P and the operational intensity I.

Efficient Rounding. With all these optimizations in place, rounding became a bottleneck. Replacing it with a simple casting and addition resulted in a $1.32\times$ runtime improvement and a performance increase to around 1 flop/cycle.

Blocking with Autotuning. For the next experiment we used blocking with a 2×24 block size, determined as the fastest by our autotuning procedure (see Fig. 6) for the used benchmarking setup. Blocking reduces data dependencies and improves locality. This is supported by the performance plot reporting more than one flop/cycle. Note, that this optimization directly touches the memory bound in the roofline plot suggesting that it is strictly memory bound.

Vectorization. The naive vectorized version outperforms the autotuned version only slightly but is far from an $8\times$ speedup of AVX2 due to the empirically shown memory bound. We are certain that in this case blocking and vectorization both reduce data dependencies, the former by scalar replacement giving rise to ILP and the latter by its inherent parallelism. Thus, it is not surprising that both approaches are in the same ballpark runtime and performance-wise. However, combining blocking (also with 2×24 block sizes) with vectorization further increases performances to over 1.5 flops/cycle and reduces runtime by nearly 30% compared to vectorization without blocking. We suppose this stems from better cache locality from blocking and compiler optimizations made possible by scalar replacement. Note that the block size was not tuned together with vectorization and other block sizes could perform even better. Lastly, the lowest line of the runtime plot shows the gain when using a padded image in combination with all previous optimizations. Due to the complete removal of boundary checks, small images – where for most keypoints part of the descriptor lies out of bounds – have a comparable (even faster) runtime per keypoint as well, while for large images the difference becomes negligible.

5. CONCLUSION

In this work, we proposed and successfully combined a variety of different optimization techniques for implementing the SURF algorithm. The optimizations range from meticulous hand-crafted conditional removal of boundary checks to autotuning for block sizes. Our novel approach of exploiting the symmetry of rounding and separability of Gaussians for the M-SURF descriptors deserves a special mention. Further, we analyzed the impact of the proposed optimizations not only with respect to their runtimes but also their cache and memory usage. Experiments show that our optimizations result in a $2.8\times$ speedup (or more) compared to currently available implementations.

As a final outlook, we believe there remains room for future work. Regarding the integral image it would be interesting to see if peak performance can be reached by making full use of ILP through further (and possibly autotuned) unrolling. Non-maximum suppression was barely touched on in this paper. Yet, it might still bear potential. For the response layer we expect that combining striding of the D_{xy} filter with conditional removal will yield a further speedup. Lastly, adapting the proposed optimizations of the M-SURF descriptor for its rotational-invariant counterpart is left as future work.

6. CONTRIBUTIONS OF TEAM MEMBERS

Carla. Implemented the algorithm for ILP improvement for the integral image. Worked on the optimizations of compute response layers with Laurin. Implemented case splitting to reduce conditionals, implemented the padding and scalar replacement.

Valentin. Focused on M-SURF descriptor optimizations and contributed to each of its optimizations. Amongst other things he came up with joining Haar wavelet computation, built the code generator and autotuning framework and worked together with Sebastian on the novel exponential call reduction approach and the AVX2 version of the descriptor. Further contributed to the optimization of the response layers by implementing blocking (also with autotuning but without success), computing response layer at once and finally came up with using strided computation to reuse D_{xy} filter computations.

Laurin. Focused on optimizing the response layer computations. Implemented cache locality optimizations, case splitting for box filters to remove conditionals and `min`-calls in `box_integral` used for boundary checks. Implemented padding with Carla. Improved spatial locality, added scalar replacement, precomputations and increased accuracy of floating point arithmetic that lead to false results in the original OpenSurf implementation.

Sebastian. Focused on M-SURF descriptor optimizations. He worked on precomputing the outer Gaussians, and in collaboration with Valentin discovered the possibility for the Gaussian separability and symmetry trick. Next, he looked at different layouts for memory buffers. Furthermore, he also implemented the integral image integer AVX2 optimization, as well as the padding of integral image computation.

7. REFERENCES

[1] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool, “Surf: Speeded up robust features,” in *Computer Vision – ECCV 2006*, Aleš Leonardis, Horst Bischof, and Axel Pinz, Eds., Berlin, Heidelberg, 2006, pp. 404–417, Springer Berlin Heidelberg.

[2] Motilal Agrawal, Kurt Konolige, and Morten Rufus Blas, “Censure: Center surround extremas for realtime feature detection and matching,” in *Computer Vision – ECCV 2008*, David Forsyth, Philip Torr, and Andrew Zisserman, Eds., Berlin, Heidelberg, 2008, pp. 102–115, Springer Berlin Heidelberg.

[3] Tudor Nicosevici and Rafael Garcia, “Automatic visual bag-of-words for online robot navigation and mapping,” *IEEE Transactions on Robotics*, vol. 28, no. 4, pp. 886–898, 2012.

[4] Adam Schmidt, Marek Kraft, and Andrzej Kasiński, “An evaluation of image feature detectors and descriptors for robot navigation,” in *International Conference on Computer Vision and Graphics*. Springer, 2010, pp. 251–259.

[5] Jared Heinly, Johannes L. Schonberger, Enrique Dunn, and Jan-Michael Frahm, “Reconstructing the world* in six days *(as captured by the yahoo 100 million image dataset),” in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

[6] W. Zhou, H. Li, R. Hong, Y. Lu, and Q. Tian, “Bsift: Toward data-independent codebook for large scale image search,” *IEEE Transactions on Image Processing*, vol. 24, no. 3, pp. 967–979, 2015.

[7] Wengang Zhou, Yijuan Lu, Houqiang Li, Yibing Song, and Qi Tian, “Spatial coding for large scale partial-duplicate web image search,” in *Proceedings of the 18th ACM International Conference on Multimedia*, New York, NY, USA, 2010, MM ’10, p. 511–520, Association for Computing Machinery.

[8] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski, “Orb: An efficient alternative to sift or surf,” in *2011 International conference on computer vision*. Ieee, 2011, pp. 2564–2571.

[9] Alexandre Alahi, Raphael Ortiz, and Pierre Vandergheynst, “Freak: Fast retina keypoint,” in *2012 IEEE Conference on Computer Vision and Pattern Recognition*. Ieee, 2012, pp. 510–517.

[10] Stefan Leutenegger, Margarita Chli, and Roland Y Siegwart, “Brisk: Binary robust invariant scalable keypoints,” in *2011 International conference on computer vision*. Ieee, 2011, pp. 2548–2555.

[11] Pablo Fernández Alcantarilla, Adrien Bartoli, and Andrew J Davison, “Kaze features,” in *European Conference on Computer Vision*. Springer, 2012, pp. 214–227.

[12] Pablo F Alcantarilla and T Solutions, “Fast explicit diffusion for accelerated features in nonlinear scale spaces,” *IEEE Trans. Patt. Anal. Mach. Intell*, vol. 34, no. 7, pp. 1281–1298, 2011.

[13] Richard Szeliski, *Computer vision algorithms and applications*, Springer, London; New York, 2011.

[14] Schmid C. Mikolajczyk K, “Performance evaluation of local descriptors.,” *IEEE Trans Pattern Anal Mach Intell*, vol. 27, no. 10, pp. 1615-1630, 2005.

[15] Mohr R. Bauckhage C. Schmid, C., “Evaluation of Interest Point Detectors,” *International Journal of Computer Vision*, vol. 37, no. 10, pp. 151–172, 2000.

[16] David G Lowe, “Distinctive image features from scale-invariant keypoints,” *International journal of computer vision*, vol. 60, no. 2, pp. 91–110, 2004.

[17] Christopher Evans, “Notes on the opensurf library,” 01 2009.

[18] P. Drews, R. de Bem, and A. de Melo, “Analyzing and exploring feature detectors in images,” in *2011 9th IEEE International Conference on Industrial Informatics*, 2011, pp. 305–310.

[19] Ali Ozturk and Ibrahim Cayiroglu, “Cross-comparison of the performance of sequential summed area table and box filter algorithms with respect to c/c++ compilers,” preprint at https://www.ipol.im/pub/pre/268/.

[20] Yermalayeu Ihar, “Integralsum in simd library,” .