MAKING SPEEDED UP ROBUST FEATURES (SURF) FASTER

Adrian Hoffmann, Luohong Wu, Marc Styger, Max Eichenberger

Department of Computer Science
ETH Zurich, Switzerland

ABSTRACT

Computer vision is increasingly important in today's world. A common task in computer vision is feature extraction with some applications even requiring this to happen in real-time. An example of a well-known algorithm for this problem is Speeded Up Robust Features (SURF) which detects points of interest in an image and extracts its features. We divide the algorithm into multiple parts on which we apply optimizations to make SURF even faster. We perform run time experiments on optimizations like mathematical reforming, loop unrolling, blocking, inlining and vectorization. In the end, we analyze their behaviour with methods like roofline analysis and run time plots, which allow us to show that we achieve significant speed up.

1. INTRODUCTION

Feature extraction is the first step in many computer vision tasks such as object tracking and object detection. It plays an important role since those tasks are based on these extracted features.

Motivation. Feature extraction is a challenging problem since it needs to be robust to factors such as light and orientation. In order to be robust, different filters and methods are required, which are computationally expensive in some computer vision tasks. For example, the computer vision system in autonomous driving needs to detect different cars, people, and traffic signs, which requires extracting features in different environments from high frame-rate videos. This is possible only if the feature extraction algorithm is robust and can be applied in real-time.

Related work. In order to extract robust features efficiently, the Speeded Up Robust Features algorithm (SURF) has been proposed by Bay et al.[1]. In this algorithm, the integral image is defined and used for applying some filters such as Haar wavelet filters, which reduces the number of flops dramatically. Because of its robustness and efficiency, it has been applied in many vision systems such as Emgu CV [2]. The initial implementation of this work is based on SURF. For a more efficient computation of integral images, a new formulation is proposed where some values such as cumulative row sums are reused [3]. Furthermore, a row-wise parallel method is also proposed in [3]. This work combines this parallel method with other optimization methods such as vectorization and inlining for further performance improvement. On the other hand, box filters can be applied to approximate first order and second order derivative Gaussian filters efficiently [4]. This work tried different horizontal and vertical box filters of different sizes.

Contribution. The contribution of this work is twofold: first, we determined that most modules of SURF are memory bound and the bottleneck module is estimating the main orientation of interest points. Second, we applied standard C optimization methods and vectorization to make SURF faster. For some modules such as interest point detection, we achieve a speed up of more than 10 for some problem sizes. For the bottleneck module, we achieve a speed up of more than 3.

2. BACKGROUND

In this section we describe the SURF algorithm [1] and the necessary context. We then conclude with a cost analysis. Broadly speaking, the SURF algorithm comprises of four stages: (1) preparation, (2) locating points of interest, (3) computing the orientation of said points of interest, and (4) computing a descriptor for each point of interest. We proceed by describing each part separately.

Preparation. First of all, SURF works on greyscale images, hence a conversion to greyscales might be needed. We assume a greyscale input from here on. Secondly, an *integral image* $I_\Sigma(x, y) = \sum_{i=0}^{i \leq x} \sum_{j=0}^{j \leq y} I(i, j)$ is generated, where I is the input image and $I_\Sigma(x, y)$ and $I(i, j)$ denote a pixel in the integral and input image respectively. This allows us to compute the sum of all pixel values within an up-right, rectangular block in I with only three flops using the integral image. This optimization is core to SURF.

Locating Points of Interest. SURF's approach to finding points of interest is based on the determinant of the Hessian at all locations at different scales in the image. However, these values are only approximated to ease the computational burden. This is achieved by using block approxima-

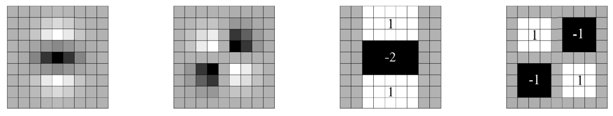

Fig. 1. Left: the discretised and cropped Gaussian second-order partial derivatives (in y- and xy-direction respectively), right: their approximations (taken from [1])

tions of the Gaussian second-order partial derivatives (see Fig. 1). Say their responses (normalized by kernel size) are D_{xx}, D_{yy}, and D_{xy} (for second-order derivatives in x-, y-, and xy-direction respectively) for a specific location in I. Then the approximate Hessian determinant at said position is $D_{xx} \cdot D_{yy} - (0.91 \cdot D_{xy})^2$. Performing this for the entire image yields a map of approximate Hessian determinants.
In order to find points of interest at different scales, SURF creates multiple of these maps but using Gaussians with increasing standard deviation σ. Specifically, σ is increased such that the kernels increase their side lengths by a predetermined step size. For example, if the first kernel size is 9×9 (as in Fig. 1) then the next larger sizes are 15×15, 21×21, and so on. All these sizes (resulting from a common step-size) are grouped into an *octave*, of which SURF uses multiple. Each octave has as its smallest kernel size the second smallest side length from the previous octave and double the step size. In our example, the next octave would contain side lengths 15×15, 27×27, 39×39, and so on. In SURF, choosing the number of octaves and sizes per octave is left to the user (likely values are 3 and 4).
For the remaining steps, SURF stacks all maps of approximate Hessian determinants from one octave yielding a 3D-tensor (cropping maps where necessary). Each of these tensors is then max-windowed with a $3 \times 3 \times 3$ window. Lastly, SURF interpolates through image and scale-space (as in [5]) around each found maximum to get a more refined measure of where and at what scale the point of interest lies.
Consequently, this step of the SURF algorithm produces a set of (x, y, s) triples with each describing a point of interest, x and y are the coordinates and s is the so-called *scale* of the point of interest (i.e. the standard deviation of the Gaussian which leads to the maximum in image-scale space).

Orientation. To compute the orientation of a point of interest $p = (x_p, y_p, s_p)$, SURF applies Haar wavelet filters as in Fig. 3 ($4s_p$ side length) for both x- and y-direction at different sample points in a neighbourhood ($6s_p$ radius) around (x_p, y_p). The filter responses are weighted by a Gaussian ($\sigma = 2s_p$) centered at (x_p, y_p). SURF then reinterprets the weighted filter responses as points in 2D space and slides a window (a cone with angle $\frac{\pi}{3}$) over them, see Fig. 4. The value of a window is the sum of the Euclidean norms of all points within the cone. We get the orientation of p by summing the points within the window with the largest value. Hence, p is augmented to $p = (x_p, y_p, s_p, o_p)$. The same is performed for every point of interest.

Descriptor. To compute the descriptor for a point of interest $p = (x_p, y_p, s_p, o_p)$ SURF defines a 4×4 grid ($20s_p$ side length) centered at (x_p, y_p) and oriented according to o_p. Within each grid cell there are 5×5 equally spaced sample points. SURF applies Haar wavelet filters ($2s_p$ side length) in both x- and y-direction at each sample point within the cells (x- and y-direction is here defined according to o_p[1]). We denote the responses of the filters as d_x and d_y for a specific sample point and the two considered directions. As a second to last step, d_x and d_y are weighted by a Gaussian centered at (x_p, y_p) with $\sigma = 3.3s_p$. Before finally combining all d_x and d_y of a cell into four values: $\sum d_x$, $\sum |d_x|$, $\sum d_y$, and $\sum |d_y|$. These four values of each of the 16 cells are concatenated to the final 64-entry descriptor of p.

Cost Analysis. We consider here our initial implementation and the number of flops thereof. h and w denote the height and width of the input image respectively.
Generating the integral image takes $3hw$ flops and is very straight forward. Computing the 3D-tensors of approximate Hessian determinants depends on the parameters chosen by the user and we here present the case for 3 octaves with 4 scales each. The cost is then $4 \cdot 61((h-26)(w-26) + (h-50)(w-50) + (h-98)(w-98))$ flops.
Max-Windowing the 3D-tensors requires incredibly few flops, merely $2 \cdot (\#scales)(\#octaves)$. However, this is logical by the nature of the maximum operation whose main computations are boolean and index computations.
The cost of computing the orientation of an individual point of interest $p = (x_p, y_p, s_p)$ splits up into (a) $4,608{s_p}^2$ flops for computing the Haar wavelet responses at the sample points and (b) $14,976{s_p}^2$ flops for the sliding window. Note here that s_p is not that big of a number (e.g. ≤ 4 for the first octave with 4 scales).
Lastly, computing the descriptor for an individual point of interest costs a constant $400 \cdot (59 + 2 \cdot \texttt{div} + \texttt{exp}) + 4 + \texttt{sqrt} + \texttt{div}$ flops, where `div`, `exp` and `sqrt` denote floating point division, the exponential function, and the square-root function respectively.
As is apparent from the cost analysis: the major factor for the first two parts (integral image and locating points of interest) is the size of the image while this then shifts for the later parts to how many points of interest get detected.

3. METHODOLOGY

The SURF algorithm nicely splits into parts which is excellent for a team to optimize. Furthermore, these different parts are remarkably diverse leading to various chal-

[1]In SURF an interpolation is used for speed.

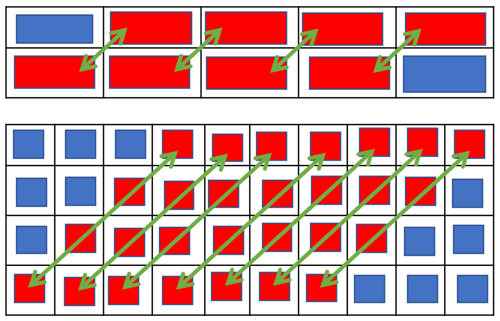

Fig. 2. Delay parallel computing of integral images. The upper sub-figure is the 2-row parallel and the lower sub-figure is the 4-row parallel.In both sub-figures, the blue elements are computed individually and the red elements with the same green arrow are computed in the same single iteration.

lenges. Our initial implementation contains two simplifications compared to the algorithm described in the previous section. (1) We omit the interpolation step from [5] and (2) we implement the sliding window by moving the window at a constant step size of $10°$. We proceed by investigating each part separately.

Integral Image. The integral image is computed recursively in the initial implementation, $I_\Sigma(\mathbf{x},\mathbf{y}) = I_\Sigma(\mathbf{x},\mathbf{y}-1) + I_\Sigma(\mathbf{x}-\mathbf{1},\mathbf{y}) - I_\Sigma(\mathbf{x}-\mathbf{1},\mathbf{y}-\mathbf{1}) + I(\mathbf{x},\mathbf{y})$ where I is the original image. For an image with height h and width w, the total number of flops is $3hw$. In order to reuse some intermediate values, the integral image is computed by $I_\Sigma(\mathbf{x},\mathbf{y}) = I_\Sigma(\mathbf{x},\mathbf{y}-\mathbf{1}) + row\_sum(\mathbf{x},\mathbf{y})$, with $row\_sum(\mathbf{x},\mathbf{y}) = \sum_{i=0}^{i\leq x} I(i,y)$ [3]. With this reformulation, the number of flops is reduced from $3hw$ to $2hw$, which should give a speed up of about 1.5. We have great spatial locality since each pixel from the original image is accessed sequentially. Temporal locality can not be improved since we access each element of I only once anyways. In addition, due to the dependency between elements in the same row, vectorization is not helpful either. However, the elements in different rows are not fully dependent. More specifically, the element $I_\Sigma(\mathbf{x},\mathbf{y})$ is only dependent on the element $I_\Sigma(\mathbf{x}-\mathbf{1},\mathbf{y})$. In the case of 2-row parallel, if $I_\Sigma(\mathbf{x}-\mathbf{1},\mathbf{y})$ is computed first, $I_\Sigma(\mathbf{x},\mathbf{y})$ and $I_\Sigma(\mathbf{x}-\mathbf{1},\mathbf{y}+\mathbf{1})$ can be computed. This is called *delay parallel* in this work [3]. This process is illustrated in Fig. 2

Approximate Hessian Determinants. The initial implementation of this part is very simple since each octave, each map within an octave, and each entry within a map can be computed independently. The first optimization arises immediately from this since some scales appear in consecutive octaves. Hence, the corresponding maps can be reused between octaves, lowering data movement and flops count. Next, the mathematical operations in the innermost loop (which approximate an individual Hessian determinant) can be optimized by moving some outside the loop and divisions can be replaced by multiplications. The third optimization is inlining functions to save the overhead of the function calls. Moreover, this allows saving some mathematical operations. Using multiple accumulators then promises better instruction-level parallelism (ILP) which constitutes the fourth optimization. While spatial locality is great in the initial implementation, temporal locality can be improved. For example by computing multiple rows of a map at the same time in the innermost loop such that there is overlap between values which kernels need to access. Lastly, we can take advantage of vectorization (AVX, AVX2, and FMA) to further speed up this part.

Max-Windowing. To find points of interest the initial implementation simply iterates over each of the octaves of the Hessian determinants separately. Every entry is compared to its surrounding 26 elements, this can be imagined as a 3x3x3 cube with the tested element at the centre. As one can already see there are nearly no floating point operations and only loading and comparing of elements, which creates a lot of conditional branches. The only mathematical optimization is replacing a division with a multiplication, but we can increase the efficiency of loading. For the first optimization step, we divide the 3x3x3 cube into 3 slices in direction of the width. The first slice D_L contains the 3x3x1 square left of the centre element, D_M are the 9 elements around the centre along the same width and D_R is the slice to the right of the centre. In a first step, the loop over the width is unrolled to reuse D_M and D_L for the next iteration, such that $D_M \rightarrow D_{L_{new}}$ and $D_L \rightarrow D_{M_{new}}$ while $D_{R_{new}}$ has to be loaded from memory. This will save us a load from memory for $2 \cdot 9$ elements. Because elements are accessed consecutively spatial locality is already good. To further improve temporal locality we implemented a blocking mechanism in the next step. Given that the number of scales in an octave is mostly small (e.g. 3 or 4) a block spans over all scales in an octave. To find the optimal block-size we used the formula depending on the constant cache-size γ and the $number\_of\_scales$, we yield: $number\_of\_scales \cdot block\_size^2 \ll \gamma \Rightarrow block\_size \ll \sqrt{\frac{\gamma}{number\_of\_scales}}$. As a last iteration of the optimization process vector instructions were introduced to the Max-Windowing problem. In this step, we tried to take advantage of the fact that we can compare and load multiple elements at once. For 4 consecutive elements, the surrounding four 3x3x3 cubes could be loaded together in 18 vectors. From these vectors, we can extract the maximum by using compare instructions and store a point of interest if one occurs.

Sampling Interest Region. The orientation of the interest point is computed by applying a horizontal and vertical Haar wavelet filter for each image point in the $6 \cdot s_p$ radius around a point of interest. The program iterates through

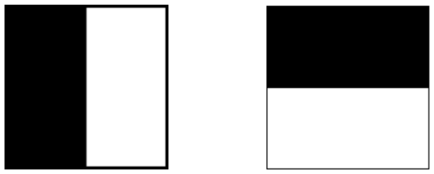

Fig. 3. Haar wavelet filters for computing gradient dx and dy, The dark parts have weight -1 and the white parts have weight 1 (taken from [1])

the region, and for each iteration, the Haar wavelet functions are called and weighted with a Gaussian. As a first optimization, the Gauss function was simplified by eliminating divisions and hardcoding values. The Haar wavelet functions were accelerated by manually inlining the block loading and wavelet-addition function for the integral image to reuse as much index computation as possible, which constitutes the first optimization. To reduce the overhead of calling a function, the Gauss and Haar wavelet functions were inlined by the compiler into the loop. The problem is again profiting from perfect spatial locality because the interest region is accessed consecutively over the innermost elements. We tried to improve locality by introducing blocking. The working set is a square block that yields the formula $block\_size \ll \sqrt{\gamma}$. We do not expect many benefits from blocking because we are already iterating from innermost to outermost dimension and not traversing in y-direction, thus we already use spatial locality at its best. In the last step of scalar optimization, all functions were manually inlined into the main loop. This opened the opportunity to move out constants and recurring computations out of all loops, executing them only once. The divisions from the Gauss function could be precomputed once before the loop as well as the π constants, leaving us with only 2 multiplications and a call to the `exp` function inside the loop. Vectorizing the Sampling part was straightforward since we access the image elements consecutively, with vector instructions we could now access 4 at once. For the square functions in the Gaussian weighting, FMAs could be used. The `exp` and `atan2` function were still called on scalars by loading the vector into a small array, calling the function on each element and then storing the array back into the vector. With the ICC compiler, the Intel SVML functions can be used for these two problems.

Orientation. The orientation of the interest point is estimated by the dominant orientation in the interest area around that point. After all neighbour points' gradients are computed, they are put into a circle graph (Fig. 4), and a window ω of $\frac{\pi}{3}$ slides around the circle. Let Ω be the set of all points falling into ω. Each window's gradient is computed by $dx_{window} = \sum_{p\in\Omega} dx_p$ and $dy_{window} = \sum_{p\in\Omega} dy_p$, where dx_p and dy_p is the horizontal and vertical gradient respectively. And the magnitude of each window's gradient is computed by $l = ||[dx_{window}, dy_{window}]||^2$. Finally, the orientation of the interest point is determined by the window whose gradient has the largest magnitude (the red arrow in Fig. 4). Estimating the main orientation involves many kinds of flops and various optimization technologies can be applied to improve the performance. The first one is scalar replacement and scalar reusing. Computations such as $\frac{\pi}{3}$ can be replaced by its value 1.047197551196, and the computation of some common values in each sliding window is moved out of the loop. The second optimization method applied is increasing ILP. However, there are already 4 comparison instructions issued in one iteration, and hence ILP does not improve much since the number of the same kind of instruction issued in one cycle is limited. In this part, since each neighbour point is compared sequentially to each window, it has perfect spatial locality. However, since the longest window's gradient is compared and recorded in each iteration, the temporal locality is worst for a large scale s. For each sliding window, there are about $144s^2$ points that need to be loaded. If s is big enough, the cache is not able to store all data, which leads to poor temporal locality. The blocking method is not feasible without modifying the implementation since a recording method is used. In addition, vectorization can be used because each comparison and summation is independent and vectorization should improve the performance. Finally, inlining should be helpful since functions such as the Gaussian are called often in the implementation, and function calling is expensive.

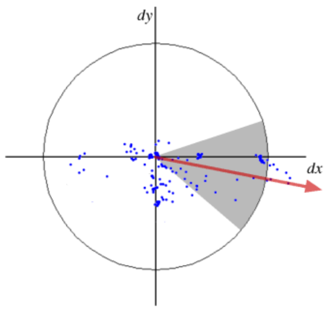

Fig. 4. All neighbour points' gradients are put into a circle. A window (the gray part) of $\frac{\pi}{3}$ slides around the circle. Each window's gradient is computed as sum of gradients of points that fall into the window. The interest point's orientation is determined by the longest window's gradient. (taken from [1])

Descriptor. The initial implementation computes the descriptors for each point of interest in a straightforward manner, i.e. compute the sample points and apply the operations described in Section 2 to each sample point. We start

the optimization with basic scalar replacement and separate accumulators to improve ILP.
Next, we observe that through the way the rest of the code is implemented, we receive the points of interest in an array where neighboring points of interest are highly probable to have the same scale. This results in the idea of reusing values that only depend on the scale. Most importantly, the 4x4 sample grid only depends on the scale and orientation. We, therefore, calculate the sample grid only once for a distinct scale and store it for subsequent points of interest with the same scale. Rotating the sample grid according to the orientation of the point of interest remains an issue for each individual point of interest since the orientation varies significantly. Other smaller values (e.g. simple multiples of the scale of a point of interest needed for the computations) are precomputed as well. Reducing the storage usage of the sample grid is the next step. In the initial implementation, we store the 400 coordinates describing the sample points, however, this can be reduced to 10 scalars. Given the 10 scalars, we just need to combine the values correctly to describe all sample points all the while reducing the number of computations. The second to last optimization is inlining functions. We finish with vectorization of the code.
It does not make sense to apply blocking for the computation of a single point of interest as we do not experience capacity cache misses and values were only used once per point of interest. Furthermore, managing cache locality between separate points of interest is near impossible as memory accesses are highly unstructured. Hence, any attempt would incur a large computational overhead erasing any locality benefits.

4. EXPERIMENTAL RESULTS

The optimizations described in the previous section were implemented incrementally on top of our initial implementation. In this section we present the effect these improvements had on the run time (measured via the Time Stamp Counter of x86 architectures). Moreover, we analyze why some optimizations failed to improve the run time. We proceed again by part as in the previous section.

Integral Image. This part is benchmarked using AMD Ryzen 3550H, whose base clock is 2.1GHz, L1 cache is 384KB, L2 is 2MB, and L3 is 4MB. The used compiler is G++ 9.3 with flags `-O3 -fno-tree -vectorize -mavx2`. The result is shown in Fig. 5. In this chart, the inputs are square images and the horizontal axis depicts the side length of an image. The vertical axis depicts the run time in million cycles. From this chart, it can be found that standard C optimization methods provide a speed up of about 2. However, vectorization is not helpful, which meets our expectation due to the data dependency between each iteration. It can also be seen that 2-row parallel provides a

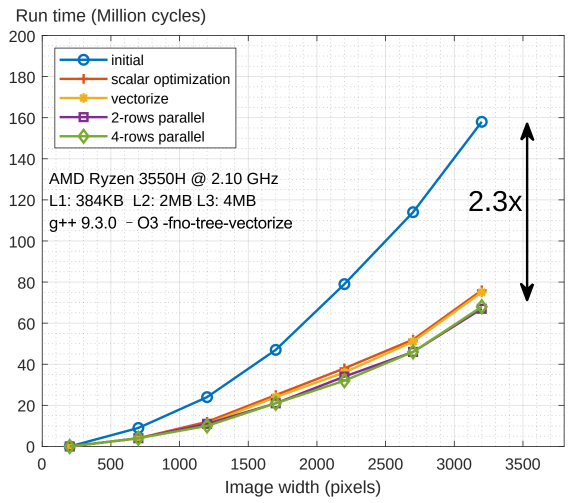

Fig. 5. Run time of computing integral images.

speed up of only 2.3, and 4-row parallel provides the same speed up as the 2-row parallel. This is not as expected. In order to find out the reason, a roofline analysis was conducted. We have $W(h,w) \approx 2hw$ flops, $Q(h,w) \approx 16hw$ bytes and operational intensity $I = \frac{W}{Q} \approx 0.125 \frac{\text{flops}}{\text{byte}}$. For AMD Ryzen 5 3550H whose peak performance[2] is $\pi = 6 \frac{\text{flops}}{\text{cycle}}$ and bandwidth $\beta = 17 \frac{\text{bytes}}{\text{cycle}}$, $I \approx 0.125 \frac{\text{flops}}{\text{byte}} < \frac{\pi}{\beta} \approx 0.35 \frac{\text{flops}}{\text{byte}}$, which means this part is memory bound. This explains why the 2-row parallel and 4-row parallel provide only little speed up.

Approximate Hessian Determinants. Fig. 6 shows the run times for the different optimizations from the previous section that improved the run time (ignore "Inlining (compiler)" for the moment). For these measurements, we consider random images with a height of $1,600$ pixels and variable width (x-axis in the plot). Five measurements were taken (cold cache) per width and optimization. The problem sizes are designed such that even the most optimized version can still take a few seconds (and not merely milliseconds). Consequently, the initial implementation has high run times and is therefore susceptible to costly system calls. Hence, instead of averaging the five measurements, we choose the median for its better robustness.
As is visible, most of the described optimizations improved the run time to an overall speed up of approximately 9. However, improving the temporal locality does not lower run time. The reason is that we try to optimize a compute bound code as a quick roofline plot analysis confirms. The machine has a measured $\beta = 13.56 \frac{\text{bytes}}{\text{cycle}}$ [6], its peak performance is $\pi = 4 \frac{\text{flops}}{\text{cycle}}$, and the code we tried to op-

[2]Note that all used peak performance values π were found in homework 1 of the course.

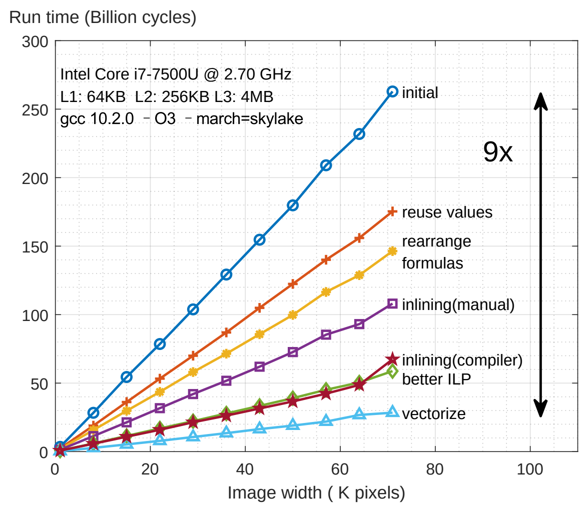

Fig. 6. Run times for computing approximate Hessian determinants. Image height is always 1,600 pixels. Image width in thousand pixels. "Inlining (compiler)" uses additional flags for compiling (see Approximate Hessian Determinants in Section 4). Speed up of 9 for largest problem size.

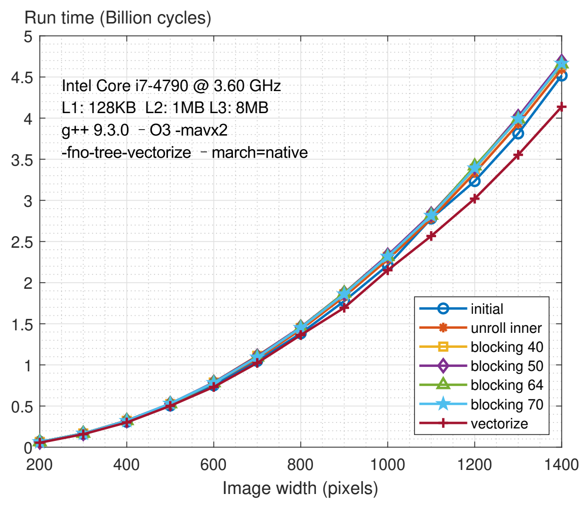

Fig. 7. Run time measurements of different Max-Windowing implementations.

timize ("better ILP" in Fig. 6) has $W(n) = (58,800n - 28,815,160)$ flops for $n \geq 99$ and $Q(n) \leq 1,024,000n$ bytes where n denotes the image width in the experiments. For the problem sizes in our experiments this means $I(n) \geq 0.54\,\frac{\text{flops}}{\text{byte}}$ while $\frac{\pi}{\beta} \approx 0.295\,\frac{\text{flops}}{\text{byte}}$ i.e. the computational intensity falls into the compute bound region of the roofline plot.

We furthermore investigate if there is a difference between manually inlining compared to with compiler flags. To test this, we take the code from "rearrange formulas" and compile it with additional flags (`finline-functions`, `findirect-inlining`, `finline-limit=1000`) to inline functions. Unfortunately, this does not yield any speed up at first. But, we gain a lot once we help the compiler by moving all code we want to inline into a single file. As is visible in Fig. 6, setting these compiler flags improves the run time by about as much as manually inlining and improving ILP together.

Unfortunately, we could not find any compiler flags that improved the run time of the vectorized code by a noteworthy amount. Notably, we tried the inlining flags from above and `-Ofast` which includes `-ffast-math`.

Lastly, we note that we do not show a performance plot as the operations count of different optimizations differs and sometimes by a lot. Still, the peak performance observed by the vectorized code is 1.70 flops per cycle while the peak for the initial implementation was 0.32 flops per cycle (note that the initial implementation has about twice as many flops as the vectorized code) for different measurements.

Max-Windowing. The optimizations are benchmarked on a Haswell architecture Intel Core i7-4790 with a base clock of 3.6GHz. The L1 cache has a size of 128KB which corresponds to $16 \cdot 2^{10}$ doubles, L2 of 1MB and the L3 cache 8MB. For the compiler the G++ compiler 9.3.0 is used with the flags `-O3 -fno-tree-vectorize -mavx2`. The inputs are random square images with their side length depicted on the horizontal axis. The run time of the different optimizations can be seen in Fig. 7. From the block-size formula shown in Section 3 the optimal block-size on this machine with a number of scales of three, would be: $block\_size \ll \sqrt{16/3 \cdot 2^{10}}$ doubles ≈ 74 doubles. In the experiments, we try block sizes of 40, 50, 64 and 70 elements to see if even lower blocks can improve the cache use. It stands out that no optimization yields a big improvement. This follows from the Max-Windowing problem. With a computational intensity of $I(h, w) = \frac{1}{108 \cdot h \cdot w}\,\frac{\text{flops}}{\text{bytes}}$ the problem is heavily memory bound. In the formula, h and w denote the height and width of a map in the Hessian Determinants octaves. From the architecture we know that the bandwidth is 25.6GB/s which yields at the given base clock rate $\beta = 7\,\frac{\text{bytes}}{\text{cycle}}$. The peak performance is $\pi = 4\,\frac{\text{flops}}{\text{cycle}}$. Now we can deduce from $\frac{\pi}{\beta} \approx 0.57\,\frac{\text{flops}}{\text{byte}} \gg I(h, w)$ that the problem falls extremely low in the memory bound area of the roofline plot. Also, the Haswell architecture has only two branch execution units but the Max-Windowing problem schedules 26 comparisons per iteration, thus only a limited amount can be scheduled per cycle leaving us with another bottleneck.

Sampling Interest Region. The experimental setup is the same as for the Max-Windowing problem. As one can observe in Fig. 8 the scalar optimizations give a speed up

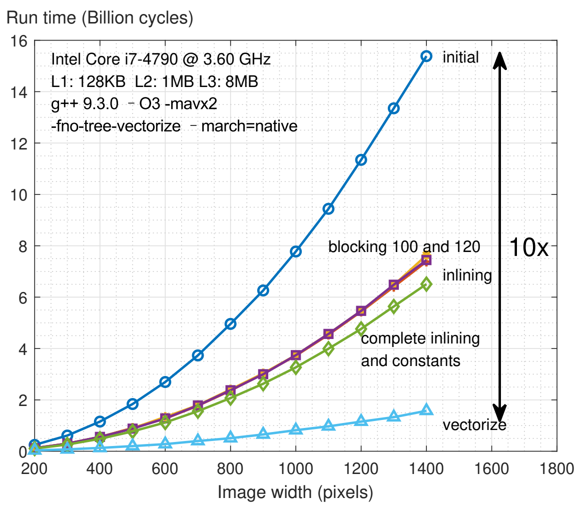

Fig. 8. Run time measurements of different implementations for the Sampling Interest Region part.

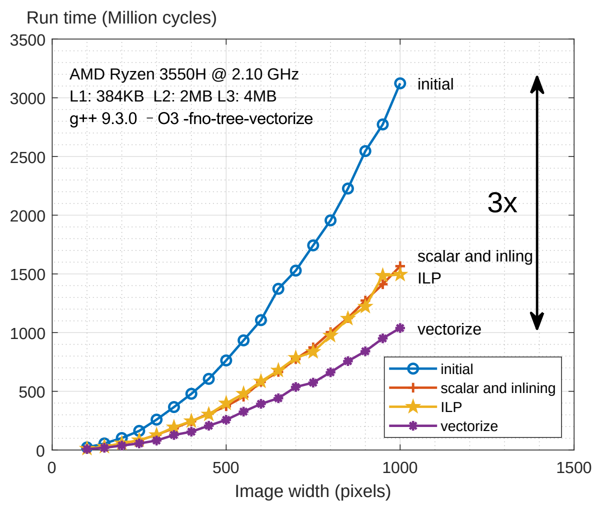

Fig. 9. Run time of estimating the main orientation of all interest points in one image.

of 2. The compiler inline and first mathematical optimizations give the biggest increase. To inline with the compiler the same flags as in Approximate Hessian Determinats are used. As already mentioned blocking is not that productive due to the manner we access the array and yields the same performance as the first inlining experiment. From the formula $block\_size \ll \sqrt{16 \cdot 2^{10}}$ doubles $= 128$ doubles, we try a block size of 100 and 120 so that the working set fits into the L1 cache. The complete manual inlining, better use of constants, and precomputation give the best scalar performance. With vectorization, a speed up of 10 is reached compared to the initial implementation.

Orientation. The benchmark platform is the same as for the Integral Image measurements. The result is shown in Fig. 9. In this chart, the horizontal axis is the side length of the square input images, and the vertical axis is the run time in million cycles. The standard scalar optimization here includes scalar replacement, scalar reuse, etc. In addition, inlining by hands and the compiler are also tried. It turned out that exhaustive inlining by hands provides the fastest speed up. As can be seen, the standard C optimization methods provide a speed up of about 2.06 and the vectorization provide a speed up of about 3. Increasing ILP is not helpful, which meets our expectation.

Descriptor. Fig. 10 shows the run time of three different implementations for different amounts of points of interest. The benchmarks are measured on a random image with height and width of $1,000$ pixels and a changing the number of points of interest.
We use an Intel Core i7-10750H with a base clock of 2.6 GHz. The L1 cache has a size of 384 KB, the L2 cache 1.5 MB and the L3 cache 12 MB. The peak memory bandwidth is 45.8 GB/s according to [7]. The computation of the descriptor is clearly compute-bound for the initial implementation. To show this we assume the best case for data movement i.e. we assume that the grid is horizontal and the scale of the interest point is small. This way we have the best spatial and temporal locality possible. We get $Q \geq 5,632$ bytes. The work of the program is $W = 400 \cdot 63 + 6$ flops $= 25,206$ flops. The bandwidth is $\beta = \frac{45.8\,\text{GB/s}}{2.6\,\text{GHz}} = 17.62\,\frac{\text{bytes}}{\text{cycle}}$ and the peak performance $\pi = 4\,\frac{\text{flops}}{\text{cycle}}$. The approximated operational intensity is given by $I \leq 4.49\,\frac{\text{flops}}{\text{byte}}$. In the worst case with high scale and non-zero rotation of the sample grid we get $Q = 25,641$ bytes. Thus $\frac{\pi}{\beta} \approx 0.23\,\frac{\text{flops}}{\text{byte}} < 0.983\,\frac{\text{flops}}{\text{byte}}$.
Unfortunately, the optimizations do not bring significant change to the run time. The main issue are dependencies we are unable to remove. Additionally, we have expensive boundary checks that are necessary because of the size of the sample grid.
Another issue is the unmanageable cache locality that we want to improve despite the compute bound program. Spatial locality of the image pixels highly depends on the orientation and scale of point of interest which defines the orientation and scale of the sample grid. Improving such run time changes is not possible without increasing the number of computational operations even more as we would have to check the orientation and scale to change the way we iterate over the image pixels. There is no temporal locality for one point of interest regarding the image pixels, however, there might be for different points of interest. Again, exploiting this is very costly in terms of book-keeping.
Note that there is no run time for the vectorization in the plot. The reason being that it does not bring any improvement with it. This comes from the suboptimal specification

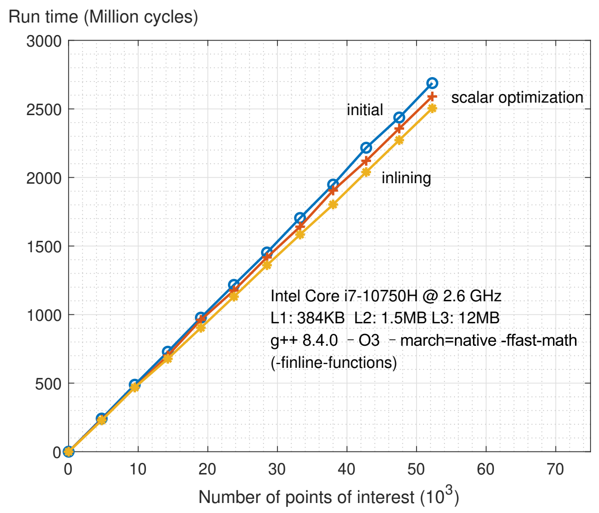

Fig. 10. Run times for the descriptor with a different number of points of interest and different optimizations. Note that the `-finline-functions` compiler flag was only used when benchmarking the inlining optimizations

of the algorithm itself. We have to make computations for sample points within a 5x5 grid per sub-region within the 4x4 grid. This gives us a bad factor of five and leaves us with unvectorized calculations for the remaining five sample points. Additionally, we have too many operations that can not be calculated using vector intrinsics (e.g. `epx`), resulting in multiple loads and stores of vectors.

5. CONCLUSIONS

Using a variety of techniques and analysis, both learned in the lecture and also found online, we improved all the parts of the SURF pipeline in terms of their run time. This helps the applicability of the algorithm in online algorithms where fast execution times are key. Speed ups for some SURF modules of up to 10 for large images are a great result. The most important part, however, is the speed up of three for the found bottleneck of the algorithm which is very welcome. Future work could further improve it by combining all the separated parts together and optimizing the dependencies between them on a specific architecture.

6. CONTRIBUTIONS OF TEAM MEMBERS

We split the SURF algorithm up into different parts which were then assigned to individual members of our team who then did all optimizations in that part. Hence, we list here which part was manged by which team member and then Section 3 together with Section 4 show what optimizations the individual members performed.

Wu Luohong. Made all optimizations for Integral Image as well as Orientation.

Hoffmann Adrian. Made all optimizations for Approximate Hessian Determinants.

Eichenberger Max. Made all optimizations for Max-Windowing as well as Sampling Interest Region.

Styger Marc. Made all optimizations for Descriptor.

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] Corina Pop, Gheorghe-Leonte Mogan, and Razvan Boboc, "Real-time object detection and recognition system using opencv via surf algorithm in emgu cv for robotic handling in libraries," *International Journal of Modeling and Optimization*, vol. 7, pp. 265–269, 10 2017.

[3] Shoaib Ehsan, Adrian F. Clark, Naveed Ur Rehman, and Klaus D. McDonald-Maier, "Integral images: Efficient algorithms for their computation and storage in resource-constrained embedded vision systems," *Sensors*, vol. 15, no. 7, pp. 16804–16830, 2015.

[4] Edouard Oyallon and Julien Rabin, "An Analysis of the SURF Method," *Image Processing On Line*, vol. 5, pp. 176–218, 2015, https://doi.org/10.5201/ipol.2015.69.

[5] Matthew A. Brown and D. Lowe, "Invariant features from interest point groups," in *BMVC*, 2002.

[6] NanoReview.net, "Core i7 7500u: performance and specs," https://nanoreview.net/en/cpu/intel-core-i7-7500u, 2021-06-22.

[7] NanoReview.net, "Core i7 10750h: performance and specs," https://nanoreview.net/en/cpu/intel-core-i7-10750h, Accessed: 23-06-2021.