“FAST" CORNER DETECTION PERFORMANCE ANALYSIS AND OPTIMIZATION

Dario Mylonopoulos, Lorenzo Liso, Saidanur Sideif-zada, Sergio Sancho Portolés

Advanced Systems Lab - Project Report FS2021
ETH Zurich, Switzerland

ABSTRACT

We present an optimization study on Features from Accelerated Segment Test (FAST), an algorithm that is designed for fast corner detection. We analyze performance on baseline and existing reference implementations and obtain conclusions about potential bottlenecks and promising optimization paths. Furthermore, we improve performance taking advantage of features in modern processor architectures. Moreover, since a major limitation when optimizing FAST implementations is that total performed work depends on the input values, we also propose strategies to successfully speed up the algorithm by reducing the amount of work required.

1. INTRODUCTION

Corner detection is a computer vision approach used to extract features from an image. A corner is a point which is the conjunction of two edges, where an edge is an abrupt change in image brightness. As described in [1] many vision applications such as tracking, localization and image matching require fast corner detection algorithms, often processing live video signals at full frame rate.

Related work. Many corner detection methods and approaches have been proposed, such as Edge-Based Corner Detectors, Graylevel-Derivative-Based Detectors and Machine Learning Based Detectors.

In this work we focus on the algorithm Features from Accelerated Segment Test (FAST) as proposed in [2], which is designed to be a simple but extremely fast alternative to more complex methods. We analyze the performance of the algorithm on modern processors and propose and implement multiple performance optimizations.

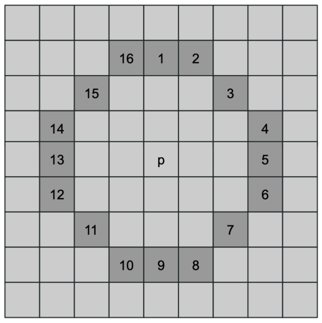


Fig. 1: Neighbouring pixels of p on a ring of radius 3.

2. FAST ALGORITHM

Features from Accelerated Segment Test (FAST) are a class of algorithms characterized by the following steps

- Select a pixel p to be a candidate corner.
- Consider 16 pixels in a ring of radius 3 around p as described in figure 1
- If there are k contiguous neighbours that are all brighter or darker than p for a given threshold t then p is classified as a corner

In this work we focus on the FAST-10 algorithm in which $k = 10$, and we use a threshold value of 25 as used for the examples in [1]. Pixels within a distance of 3 from the image borders are not considered as they don't have a full set of neighbours available.

Implementation. One simple approach to implement the algorithm is, for each candidate pixel, to consider all contiguous sequences of neighbours of length 10 and check if any of these sequences consists of pixels that are all darker or brighter than the current pixel. However checking all possible sequences is unnecessary. Instead, we use the approach used in the FAST-10 implementation from the open source computer vision library libCVD [3]. The main idea is to look at neighbouring pixels in pairs and discard corner candidates that can be ruled out early. The pseudo code of our implementation is outlined in Algorithm 1. Each of the 8 checks consists in loading the neighbouring pixels, comparing their brightness with respect to the current pixel using the given threshold and combining them with the results from the previous checks to decide if the pixel can be a valid corner. As soon as a candidate can be discarded we proceed to the next pixel.

Algorithm 1 – FAST

```
input: $m \times n$ image $I$, threshold $t$
output: array of corners $C$
 1: $C \leftarrow \{\}$
 2: for $y \leftarrow 3$ to $n-3$ do
 3:     for $x \leftarrow 3$ to $m-3$ do
 4:         $p \leftarrow I[x, y]$
 5:         if not check$(p, 1, 9, I, t)$     continue
 6:         if not check$(p, 5, 13, I, t)$    continue
 7:         if not check$(p, 3, 11, I, t)$    continue
 8:         if not check$(p, 5, 13, I, t)$    continue
 9:         if not check$(p, 2, 16, I, t)$    continue
10:         if not check$(p, 8, 10, I, t)$    continue
11:         if not check$(p, 12, 14, I, t)$   continue
12:         if not check$(p, 4, 6, I, t)$     continue
13:         C.append$(x, y)$
14:     end for
15: end for
```

Cost analysis. The asymptotic complexity of the algorithm is linear in the number of pixels in the image, since for each pixel up to 8 checks and an append operation are performed, which are all constant time. However, the exact amount of work depends on the input image since fewer checks are performed if the pixel can be discarded early. Furthermore, each of the 8 checks consist of loading two neighbouring pixels, performing two comparisons and combining the results with the output of the previous checks, thus later checks are more expensive as they need more logical operations to combine the results. Let N_i be the number of times the i^{th} check has to be performed and C_i be its cost, we can approximate the total cost C by summing across all checks and adding the cost C_c of appending the N_c corners

$$C = N_c C_c + \sum_{i=1}^{8} N_i C_i$$

As we will discuss in the next section, different implementations will result in a different number of checks and different associated costs, thus we defer a more in depth cost analysis to the end of section 3 in which we will analyze the generated code to compute performance upper bounds.

3. PERFORMANCE ANALYSIS

Experimental setup. All performance measurements were collected on an Intel(R) Core i5 11400H Tiger Lake CPU @ 2.7 GHz with Turbo Boost disabled, compiling with clang-cl 13.0.1 on Windows 10. We use the "box" data set of gray scale images provided as supplemental material in [1]. Since the number of operations heavily depends on input data and implementation details, we plot size of the problem (number of pixels) divided by run time (CPU cycles) to compare performance for implementations with different amounts of work.

Branches and branch prediction. The proposed implementation of the FAST algorithm heavily relies on conditional branches to reduce the number of checks and thus the number of operations to be performed. As detailed in [4] modern processors are highly pipelined, perform speculative execution to avoid stalls and use sophisticated branch prediction mechanisms to reduce the number of required pipeline flushes. From analyzing with Intel(R) VTune™ Profiler measuring performance on a single image smaller than 256 by 256 pixels repeatedly trains the branch predictor and results in no branch missprediction penalties, while the same measurement on bigger images can result in up to 35% of pipeline slots to be wasted due to bad speculation. In order to collect realistic performance measurements that are not affected by training the branch predictor, we measure warm cache performance on small images by executing the algorithm on a set of different images preloaded into cache.

Analysis of scalar code. Our baseline implementation iterates on a single pixel at a time, performs the required checks and if all of them succeed the coordinates of the pixel are appended to the output array. From our analysis with VTune the most evident bottleneck in performance is the fact that all loads, comparisons and logical operations are performed on a single byte. Furthermore since image rows are stored contiguously in memory, iterating over a row of pixels results in good spatial locality and the linear nature of memory accesses enables the hardware prefetcher to preload data into caches. Therefore, since memory bandwidth and latency appeared to not be a bottleneck for scalar code, our first step to improve performance was to vectorize our implementation.

Vectorization approach. Instead of working with one pixel at a time (one byte) we implemented a vectorized version of the algorithm that loads sequential pixels in the image and performs checks in parallel for each pixel in the vector. This means that more pixels can be processed at the same time but with the draw-

back that as long as a pixel in the vector can be considered a corner subsequent checks need to be performed for the whole vector. This introduces a trade-off between the size of the vector and the total work to be performed (see Figure 2), we analyze this problem further in the followings sections.
We implemented vectorized versions with SSE2, AVX2 and AVX-512 that process 16, 32 and 64 pixels at the same time respectively. To keep track of which pixel can be considered a corner we use a bit mask and the checks are performed as long as the mask is not zero. If the last check passes we append to an array the coordinates of the pixels that have the correspondent bit in the mask set to 1.

AVX-512 Optimizations. The AVX-512 instruction set adds many powerful instruction, in particular we used `vpcompressq` to append the coordinate of the pixels that are valid corners. For SSE2 and AVX2 implementations we append the coordinates of the corners falling back to scalar code and checking each bit in the corner mask individually. Instead, the AVX-512 `vpcompressq` instruction, given a pointer to the corner array, a mask and a data register, can insert in the array only the data selected by the mask. Efficient computation of the coordinates can be done using precomputed offsets and using increments inside the loop instead of using the `set_epi32` intrinsic. X and Y coordinates are packed together using a `blend` instruction. To know the number of corners that are inserted in the array in the current iteration we use the `popcnt` instruction on the bit mask. This operation is necessary to increase the end of the array pointer by the number of elements inserted by `vpcompressq`.
In Figure 2 we show the performance increase using these instructions compared to a "base" AVX-512 implementation that just extends the AVX2 version with 512 bits registers.

Analysis of checks vs block size. As mentioned above, loading sequential pixels can significantly increase the total amount of work if pixels contained in the same vector require a different number of checks. This problem is visually shown in Fig.3. While a scalar version performs the strictly required work per pixel, a vectorized implementation will cause the number of checks performed for all the pixels in a vector to be the largest number of checks required by any of the pixels.

To further analyze this coherency problem with respect to pixels in a vector, we plot the total work that would be performed depending on the specific block size. Fig. 5 shows a histogram representation of the average number of checks that need to be performed for different block sizes. As observed, some specific block configurations result in a smaller average of checks

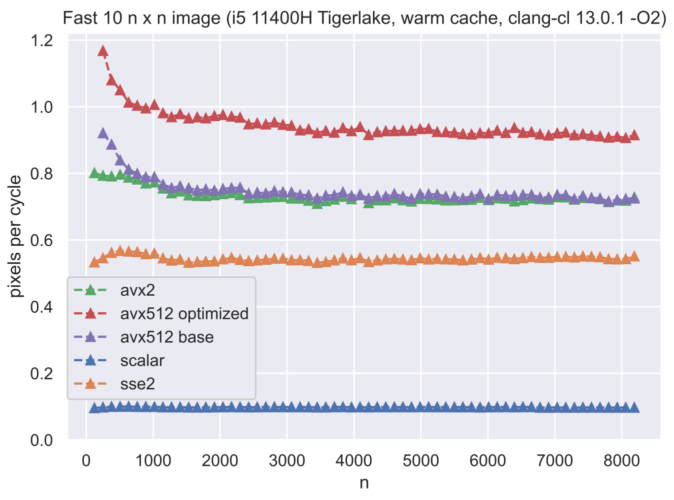


Fig. 2: Performance in pixels per cycle for the vectorized code. Vectorization greatly improves performance with respect to the scalar code. The AVX-512 base version doesn't include optimizations using AVX-512 specific instructions. The use of `vpcompressq` along with the efficient computation of the coordinates, greatly improves the performance of the AVX-512 implementation surpassing the AVX2 version.

per block to be performed, in particular, blocks with a shape closer to a square. For example, if we work with 64 pixels at a time, which is the case of AVX-512 SIMD vectors, using blocks of 8x8 pixels instead of a block of 64x1 sequential pixels will lead to a smaller number of performed checks per block. That is, the total work can be reduced by using some specific block shapes. While this fact is true for the analyzed input data, it is also a common strategy in image processing to work with blocks of pixels since there is often stronger coherency between closer pixels. Thus, we will go deeper into the idea of using pixels of different block sizes and loading them into single SIMD vectors.

In vector blocking. When working with blocks of pixels from different rows, pixels will not be sequential in memory and we have to load from multiple memory locations. Therefore, we need to consider the trade-off between using more loads for a single vector and reducing the amount of work by exploiting coherency.

However, multiple strategies exist to address the problem of loading different memory segments into a SIMD vector. First, we investigate an implementation using `gather` instructions. At first glance, this operation could seem an appropriate choice since it

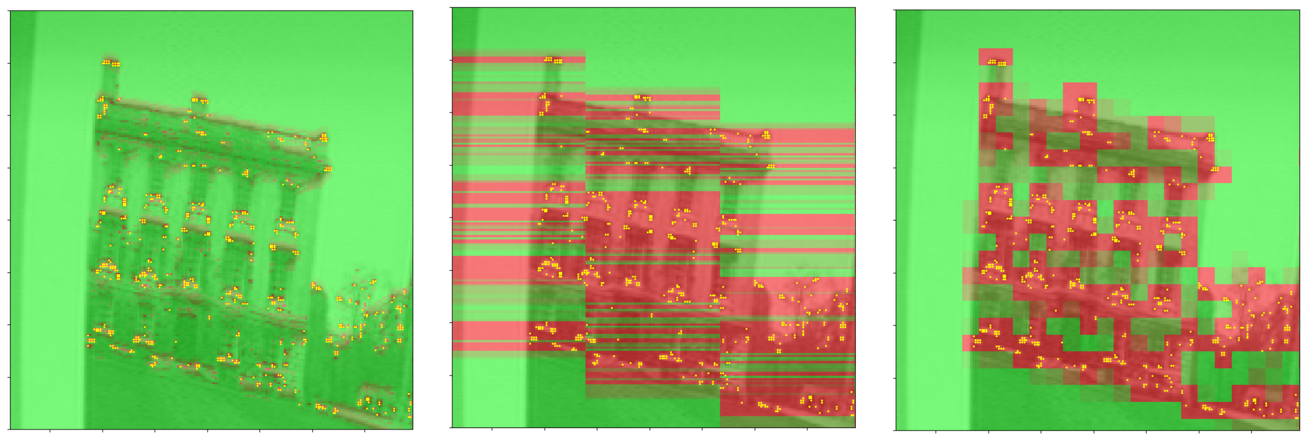

(a) Scalar 1x1 (b) AVX-512 64x1 (c) AVX-512 8x8

Fig. 3: Amount of checks performed for scalar and AVX-512 vectorized implementations using different block configurations. Pixels colored in green represent pixels where only the first check is performed before being discarded, while red tones indicate that more checks are performed. Detected corners are highlighted in yellow.

allows to load into a vector from different memory locations using a single instruction, with a vector of offsets passed as input. Nevertheless, performance of `gather` operations depends on the operands and is in general expensive, with a latency up to 20 cycles on modern Intel architectures. As shown in Fig. 4, an AVX2 implementation using this approach for a block configuration formed by 8 segments of 4 bytes leads to a worse performance than the AVX2 sequential implementation.

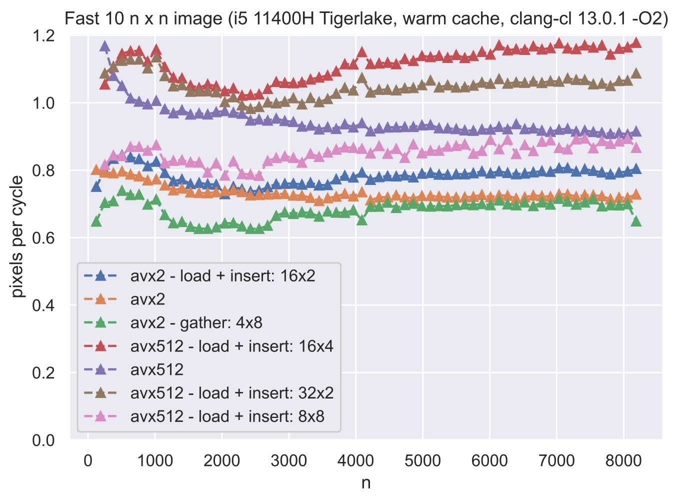


Fig. 4: Pixels per cycle plot for different in-vector blocking strategies. The AVX2 with 4x8 block size uses gather instructions to load from different memory addresses, whereas all the other implementations use load and insert operations to form the corresponding SIMD vector.

In our case, memory offsets are constant per image and we do not need to recompute the stride at every iteration. Therefore, the ability of `gather` intrinsics to use arbitrary memory offsets is not required. Instead, we can use multiple loading instructions to load memory segments individually from different addresses and combine them into a vector with insert operations. Fig. 4 shows the performance results obtained with this method for different block configurations. As observed in the plot, some in-vector blocking implementations outperform the original sequential implementation of the respective version. In the case of AVX2 version, we obtain the best results using a 16x2 block configuration, while for AVX-512 the 16x4 block size is the one that achieves the highest performance.

Looking at Fig. 4, performance using an 8x8 block size for AVX-512 could seem not as good as expected. However, we should consider that working with 8x8 blocks instead of 16x4 blocks doubles the number of load and insert instructions and leads to more dependencies that significantly affect performance. Fig. 6 shows the trees of load and insert dependencies for the described AVX-512 block configurations.

Roofline analysis. In order to compare the performance of our implementations with respect to the upper bounds given by the roofline model we estimate work and data movement as follows. Every pixel is accessed up to 17 times (once when considered as corner candidate and up to 16 times when checked as a neighbour), however those accesses only happen with a stride of at most 7 rows, here we make the assumption that the required number of rows always fits at least in the L3 cache, for a 12MB L3 cache this holds for

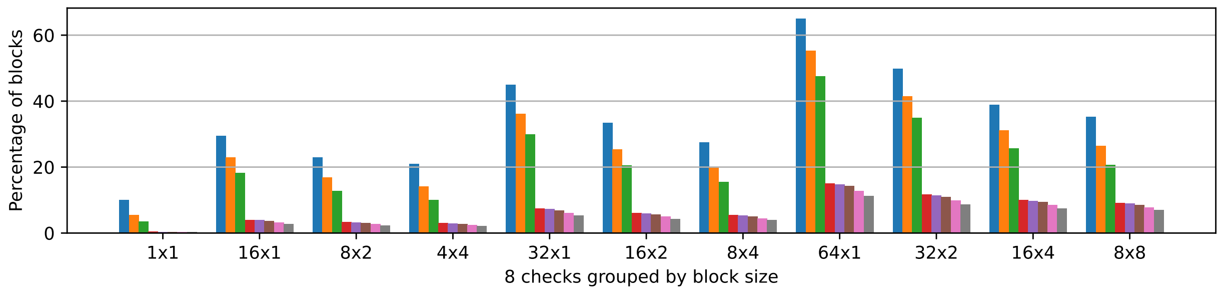
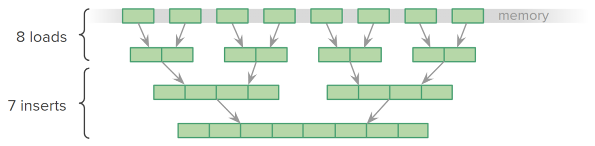


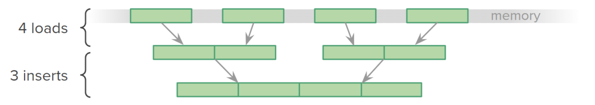
Fig. 5: The 8 bars in each group represent the percentage of pixel blocks in the image that pass the respective check, bigger blocks, especially of non-square dimensions, need to perform substantially more checks in total since they perform as many checks as required by the worst case pixel in the block. The data is averaged across all images of the "box" data set used for testing.

	Ports	Peak byte ops/cy	Peak pixels/cy	Mesaured pixels/cy	Percentage of peak
scalar	0, 1, 5, 6	4	0.15	0.097	62.7%
SSE2	0, 1	32	1.43	0.54	37.6%
AVX2	0, 1	64	2.42	0.70	29.0%
AVX-512	0, 5	128	3.47	0.89	25.8%

Table 1: Peak and measured performance for a 8192×8192 image.



(a) Tree AVX-512 8x8



(b) Tree AVX-512 16x4

Fig. 6: AVX-512 dependency trees for loading from multiple rows.

images of width up to about 1.8×10^6 pixels. Therefore we assume that the image is only loaded from memory once and thus the data movement equals the size in bytes of the image. For each implementation we measure total work by instrumenting the code to count the number of checks performed. We then estimate the cost of each check in terms of byte operations per cycle by inspecting the generated machine instructions and by counting micro-operations (uops) from the output of the open source tool uiCA [5] which gives throughput and port usage information for basic blocks; an example of the tool's output for the first check of the AVX2 implementation is reported in Appendix A. When estimating total work for a specific implementation we only consider execution ports with integer ALUs which are being used in the generated code, as these ports bottleneck the loop's throughput. In particular for our scalar implementation we consider all integer operations executed on port 0, 1, 5 and 6; for SSE2 and AVX2 we consider all vector operations on port 0 and 1; for AVX-512 we consider all vector operations on port 0 and 5 which have 512 bit vector integer ALUs on the Tiger Lake architecture. Combining this information with the throughput upper bounds for the relevant ports we can also compute a tight performance upper bound in terms of pixels per cycle for a given input as follows

$$W \geq \sum_{i=1}^{8} W_i N_i$$

$$T_{peak} \geq \frac{W}{\pi_{peak}} \quad \text{cycles}$$

$$P_{peak} \leq \frac{\#\text{pixels}}{T_{peak}} \quad \text{pixels / cycle}$$

where W_i is the number of byte operations required for the i^{th} check, N_i is the number of times the i^{th} check is performed and π_{peak} is the peak performance of the machine in byte operations per cycle for the relevant ports and instruction set. We think that the work estimated for SSE2 and AVX2 is accurate since these implementations bottleneck the considered ports with exclusively vector instructions. On the other hand for the AVX-512 port 0 and 5 are also shared with scalar logical operations thus our counting method overestimates the number of required byte operations. The upper bounds computed from our model and the measured performance are summarized in table 1.

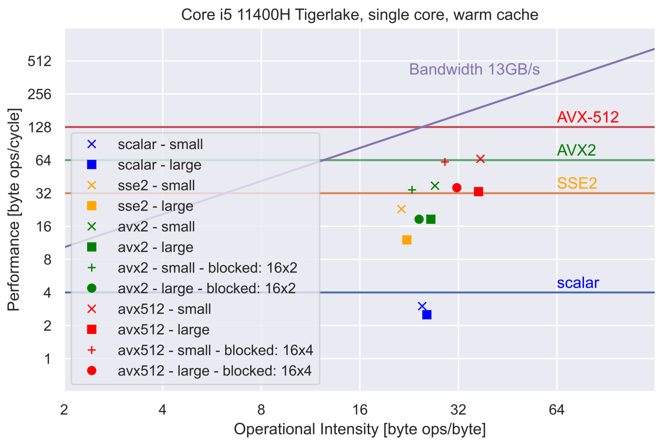
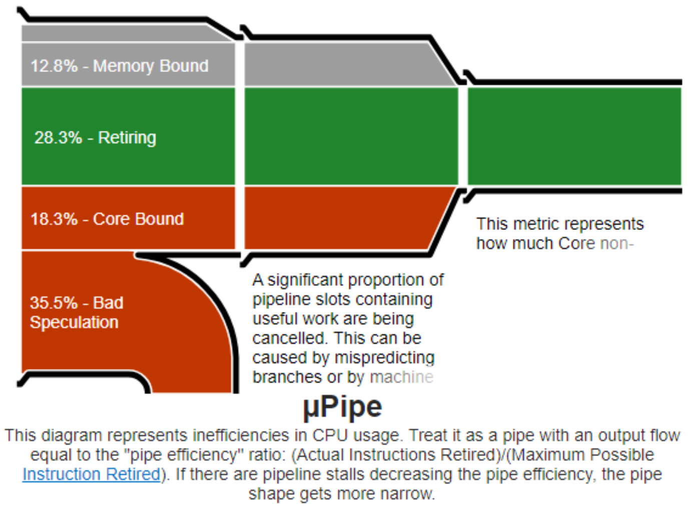


Fig. 7: Roofline plot for scalar and vectorized implementations on small (256x256) and large (8192x8192) images. Performance is measured on a single image with repeated invocations thus small images are not subject to bad speculation penalties and the input data fits in cache.

The resulting roofline plot in Fig. 7 shows how increasing the vector size also increases the total amount of work performed and thus the operational intensity, furthermore non-blocked implementations perform more work than blocked ones due to lower coherency between pixels in a vector as discussed above. As far as the scalar implementation is concerned we measured a higher operational intensity than with small vector sizes because we count all integer operations, including logical operations and address computations which are executed per byte instead of per vector, therefore they are up to 16, 32 or 64 times what is required for the respective vector size. As expected, the performance is substantially higher with repeated measurements on the same small image since fewer cache misses or branch misprediction penalties occur, these results thus achieve unrealistic performance for most cases but they provide a reasonable upper bound for the respective implementations. Finally we analyzed with VTune our AVX-512 implementation to understand what are the bottlenecks preventing us to achieve the theoretical peak performance. In the following figure we include the summary of the VTune analysis on the 8192x8192 image.



This metric represents how much Core non-

A significant proportion of pipeline slots containing useful work are being cancelled. This can be caused by mispredicting branches or by machine

This diagram represents inefficiencies in CPU usage. Treat it as a pipe with an output flow equal to the "pipe efficiency" ratio: (Actual Instructions Retired)/(Maximum Possible Instruction Retired). If there are pipeline stalls decreasing the pipe efficiency, the pipe shape gets more narrow.

We note that the main performance bottlenecks identified by VTune are bad speculation penalties and suboptimal usage of execution ports.

Handling borders. The algorithm loads neighbouring pixels in a ring of radius three, therefore a border of three pixel all around the image needs to be discarded to avoid out of bound accesses. This means that we cannot start the vectorized loop at the start of the row but we need to align with the vector size. For example the original SSE2 version of the FAST-10 algorithm in libCVD, for pixel 3 to 16 in a given row, calls the scalar version of the algorithm and then starts the vectorized loop on the remainder of the row. The same problem is also present at the end of the row if the row isn't a multiple of the register length.
Our implementation, instead of falling back to the scalar version, at the start and end of each row, loads an unaligned vector and then discard the results that would be computed again in the aligned loop. This means that for each row, 6 pixels are checked twice and discarded once, however, little work is added and this approach scales better than falling back to scalar for the peeling.
The performance of our final implementations is summarized in Fig. 8 and compared against the libCVD implementations. In the vector blocking version of the algorithm the last rows unaligned with the column height also need to be special cased to avoid out of bounds accesses. This can be done by falling back to a vectorized version that uses blocks of fewer rows.

Failed optimizations. One common optimization strategy consists in exploiting spatial locality by blocking. However, as mentioned in the roofline analysis section, the algorithm only reuses data from the previous 7 rows. In fact, the blocking implementations that have been tried perform worse than the implementation without blocking. Note that the purpose of the blocking strategy here described is different to the pre-

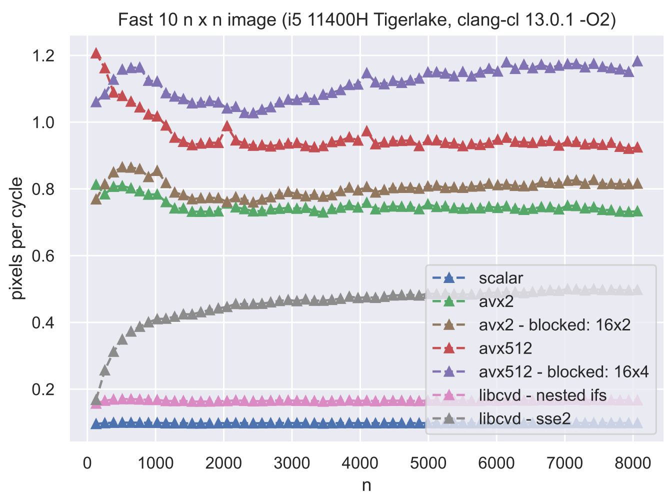


Fig. 8: Performance in pixels per cycle of the implementations with peeling. The libcvd SSE2 implementation falls back to scalar for the peeling which greatly degrades performance for small images.

vious in-vector blocking. In that case, we blocked and went deeper into different block sizes to exploit vectorization and reduce the additional work introduced by incoherent pixels in a vector.

Another possible optimization is to increase instruction level parallelism (ILP) by unrolling the loop. This way, the processor can make better use of execution ports and avoid stalls caused by instruction dependencies. Nevertheless, there exist two reasons why this may not be a good idea in this case. First, port usage analysis made with uiCA shows that relevant ports are fully saturated for the most frequent parts of the loop (an example for the first check can be seen in Appendix A). Second, unrolling may increase the total amount of work, since we are processing more pixels at the same time and the work performed for all of them is the same of that of the pixels that needs to perform more checks. Despite that, we have tried unrolling implementations, but as expected they did not improve performance.

4. CONCLUSIONS

In our work we have analyzed the performance of the FAST-10 algorithm for corner detection for modern processor architectures. We have made use of tools such as VTune and uiCA to back up our analysis and to find promising optimization paths.
We have extended the algorithm with different vectorization instructions and analyzed the tradeoff between work and vector size, which led to a blocking version of the algorithm to reduce the total work. We have looked into AVX-512 specific instructions to improve performance and implemented an efficient peeling mechanism that achieves higher performance for small images.
Finally we made a roofline analysis for different versions of the algorithm and image sizes, analyzing the operational intensity and showing the effect that the branch predictor has on performance for small and big images.

5. CONTRIBUTIONS OF TEAM MEMBERS

Dario Mylonopoulos. AVX2 implementation. AVX-512 specific optimizations, in particular efficient coordinate computations. Analysis of checks vs block size. Branch prediction and roofline analysis.
Lorenzo Liso. Scalar and AVX-512 implementation. AVX-512 specific optimizations, in particular efficient appending with `vpcompress` and `popcnt` instructions. Border peeling using masks.
Saidanur Sideif-zada. SSE2 implementation. FAST-10 algorithm analysis.
Sergio Sancho Portolés. Analysis of checks vs block size. In-vector blocking implementations with gather and insert instructions. Memory blocking and ILP optimizations.

6. REFERENCES

[1] Edward Rosten, Reid Porter, and Tom Drummond, "Faster and better: A machine learning approach to corner detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 32, no. 1, pp. 105–119, 2010.

[2] E. Rosten and T. Drummond, "Fusing points and lines for high performance tracking," in *Tenth IEEE International Conference on Computer Vision (ICCV'05) Volume 1*, 2005, vol. 2, pp. 1508–1515 Vol. 2.

[3] "libcvd - computer vision library," https://www.edwardrosten.com/cvd/.

[4] Agner Fog, "The microarchitecture of intel, amd and via cpus: An optimization guide for assembly programmers and compiler makers," *Copenhagen University College of Engineering*, vol. 2, 2012.

[5] Andreas Abel and Jan Reineke, "uops.info: Characterizing latency, throughput, and port usage of instructions on intel microarchitectures," in *ASPLOS*, New York, NY, USA, 2019, ASPLOS '19, pp. 673–686, ACM.

A. OUTPUT OF UICA

Output of the tool uiCA [5] on the instructions generated by clang-cl 13.0.1 for the first check of the AVX2 implementation.

```
Throughput (in cycles per iteration): 7.00
Bottlenecks: Dependencies, Front End, Port 0, Port 1

MITE  MS  DSB  LSD  Issued  Exec.  Port 0  Port 1  Port 2  Port 3  Port 4  Port 5  Port 6  Port 7  Port 8  Port 9

                 1     1      1                     0.48    0.52                                                    mov rax, qword ptr [rsp+0x98]
                 1     1      1                                                    1                                lea rcx, ptr [rax+r13*1]
                 1     1      1                     0.52    0.48                                                    mov rdx, qword ptr [rsp+0x90]
                 1     1      1                                                  0.33    0.67                       lea rax, ptr [rdx+rcx*1]
                 1     1      1                     0.48    0.52                                                    vmovdqu ymm0, ymmword ptr [rdx+rcx*1]
                 1     1      1               1                                                                     vpsubusb ymm1, ymm0, ymm5
                 1     1      1       1                                                                             vpaddusb ymm0, ymm5, ymm0
                 1     1      1                     0.52    0.48                                                    mov rcx, qword ptr [rsp+0x68]
                 1     1      1                     0.48    0.52                                                    vmovdqu ymm2, ymmword ptr [rcx+rax*1]
                 1     1      1                     0.52    0.48                                                    mov rcx, qword ptr [rsp+0x60]
                 1     1      1                     0.48    0.52                                                    vmovdqu ymm3, ymmword ptr [rcx+rax*1]
                 1     1      1               1                                                                     vpminub ymm4, ymm1, ymm2
                 1     1      1       1                                                                             vpcmpeqb ymm4, ymm1, ymm4
                 1     1      1       1                                                                             vpmovmskb ecx, ymm4
                 1     1      1               1                                                                     vpminub ymm4, ymm2, ymm0
                 1     1      1               1                                                                     vpcmpeqb ymm2, ymm2, ymm4
                 1     1      1       1                                                                             vpmovmskb edx, ymm2
                 1     1      1                                                            1                        shl rdx, 0x20
                 1     1      1                                                            1                        or rdx, rcx
                 1     1      1                                                    1                                not rdx
                 1     1      1               1                                                                     vpminub ymm2, ymm1, ymm3
                 1     1      1       1                                                                             vpcmpeqb ymm2, ymm1, ymm2
                 1     1      1       1                                                                             vpmovmskb ecx, ymm2
                 1     1      1               1                                                                     vpminub ymm2, ymm3, ymm0
                 1     1      1               1                                                                     vpcmpeqb ymm2, ymm3, ymm2
                 1     1      1       1                                                                             vpmovmskb ebp, ymm2
                 1     1      1                                                            1                        shl rbp, 0x20
                 1     1      1                                                    1                                or rbp, rcx
                 1     1      1                                                            1                        not rbp
                 1     1      1                                                    1                                mov rcx, rbp
                 1     1      1                                                    1                                or rcx, rdx
                 1     1      1                                                            1                        jz 0xffffffffffffff7d

                32    32     32       7       7     3.48    3.52                     5.33    5.67                       Total
```