2018 42nd IEEE International Conference on Computer Software & Applications

Measurement-based evaluation of data-parallelism for OpenCV feature-detection algorithms

Jakob Danielsson[1], Marcus Jägemar[1,2] Moris Behnam[1], Mikael Sjödin[1], Tiberiu Seceleanu[1]
[1] Mälardalen University, Västerås, Sweden
[2] Ericsson AB, Stockholm, Sweden
jakob.danielsson@mdh.se

***Abstract*—We investigate the effects on the execution time, shared cache usage and speed-up gains when using data-partitioned parallelism for the feature detection algorithms available in the OpenCV library. We use a data set of three different images which are scaled to six different sizes to exercise the different cache memories of our test architectures. Our measurements reveal that the algorithms using the default settings of OpenCV behave very differently when using data-partitioned parallelism. Our investigation shows that the executions of the algorithms SURF, Dense and MSER correlate to L3-cache usage and they are therefore not suitable for data-partitioned parallelism on multi-core CPUs. Other algorithms: BRISK, FAST, ORB, HARRIS, GFTT, SimpleBlob and SIFT, do not correlate to L3-cache in the same extent, and they are therefore more suitable for data-partitioned parallelism. Furthermore, the SIFT algorithm provides the most stable speed-up, resulting in an execution between 3 and 3.5 times faster than the original execution time for all image sizes. We also have evaluated the hardware resource usage by measuring the algorithm execution time simultaneously with the L3-cache usage. We have used our measurements to conclude which algorithms are suitable for parallelization on hardware with shared resources.**

I. Introduction

Many industrial systems often use feature detection algorithms in various applications ranging from face recognition to autonomous vehicular systems. Detecting features in a frame is a time-consuming process [5] because of the high number of traversed pixels. The number of traversed pixels depends highly on the feature detection algorithm goal, e.g., detecting objects, corners, edges, blobs or key points. The number of traversed pixels affects the application execution time, which is often a limitation for time-sensitive real-time systems.

The process of feature detection stipulates that different calculation sequences search for specific conjunctions between pixels in a frame. The length of the feature detection sequence varies significantly among the used algorithm. The number of traversed pixels per frame grows if the feature detection sequence is long leading to a further increased execution time.

One way of decrease the execution time of these calculations is to parallelize the execution and use multiple CPU-cores at the same time. The computations for a frame are often suitable for execution on parallel architectures, where each CPU can operate on a sub-frame (i.e. a partition of the original frame). Luckily, almost all processors available today are, so called, multi-core processors which have at least 2 CPU cores.

However, in a multi-core architecture, the computing units compete for access to common hardware resources, such as caches, memory banks and memory buses. This competition lead to challenges in designing parallel software to avoid bottlenecks in the data-flow and to prevent computing units from interfering with each other. Examples of performance problems related to parallel execution include cache trashing (one core evicts data from the cache that is needed by another core), cache-line ping-pong (a false-sharing problem when cores that are seemingly unrelated manipulate data-elements that are allocated close in memory), and DRAM starvation (the DRAM controller may choose to serve only memory requests from one controller for a while, since that brings up the throughput of the memory system - at the expense of long delays for some cores).

The ideal execution environment for a feature detection algorithm running on a multi-core architecture is identified by several properties. Minimizing the shared-memory congestion side effects and interprocess synchronization time are the most important ones. One possible solution to reduce the harmful effects of shared resource congestion is to monitor and understand the algorithm resource usage before-hand [15]. It is possible to obtain such knowledge by, for instance, using Performance Measurement Counters (PMC) [7].

The knowledge of how feature detection algorithms such as FAST, HARRIS or SURF affect the shared resources is an important part when incorporating them into a multi-core system, since it can give an indication on how well the algorithm scales with parallelism opportunities offered by multi-cores. Since the input data to such algorithms can be relatively large, there is a possibility that the algorithms may suffer from shared memory congestion and therefore obtain an insignificant speed-up when utilizing multiple cores. Therefore, it is possible that a feature detection algorithm has such characteristics that it is better suited for running on a single core, together with other general workloads instead of reserving the several computational units of the computer while achieving little execution time gains. However, the success of applying a parallel paradigm to a feature detection algorithm can however be an efficient tool to decrease the execution time of such heavy workloads.

In this paper we study how the feature-detection algorithms using the Open Computer Vision (OpenCV) library [4] behaves with respect to data-level parallelization in terms of L3 cache usage on multi-core processors. OpenCV is one of the most widespread libraries for image processing and hence

0730-3157/18/$31.00 ©2018 IEEE
DOI 10.1109/COMPSAC.2018.00105

IEEE computer society

these results should be valuable for a large community. The main contributions in this paper include:

- We have evaluated how the feature detection algorithms in the OpenCV features2d module [19] perform from data partitioned parallelism with respect to speed-up.
- We have measured the performance of the feature detection algorithms in the OpenCV features2d module together with each algorithm hardware resource usage. From these measurements, we deduced that the L_3-cache has the highest effect on the algorithm performance.

Outline: Section II give background information related to feature detection algorithms and their resource usage. Detailed information on our implementation is given in Section III and the experiments in Section IV. We conclude the paper by summarizing our conclusions in Section V.

II. Background

It is possible to run image processing on multi-core systems with the purpose of decreasing the execution time by using coarse-grained data parallelized algorithms [27]. Relevant work include investigating how to parallelize feature detection algorithms such as SIFT [10], [29], SURF [28], and Harris [12] for performance increase. Applying these parallelization techniques however require an in-depth investigation of the algorithm functionality and also how to adapt the functionality parameters to the hardware in use. In this paper, we have instead executed a generalized coarse-grained parallelism model which can be applicable for speed-up gains without studying the workload in detail. Since our approach does not require in-depth knowledge of neither the hardware or the software, it is also easy to migrate between different hardware setups. In this paper, we have executed a generalized coarse-grained parallelism model which can be applicable even though the work-load is not studies in detail. To the best of our knowledge, our paper is the first that investigates the effects data-level parallelism has on the shared memory using OpenCV feature-detection algorithms. The algorithms investigated in this paper are well established feature detection algorithms, available in the free and non-free branches of *features2d* in the OpenCV library. We have used the default algorithm tuning values which come with the OpenCV library in order to have a reference for the comparison.

A. Feature detection

Feature detection is a way of distinguishing anomalies in an image. Feature detection can be divided into 4 subsets, edge detection, corner detection, object detection and blob detection. In this work, we have used the common interfaces class [19] of the OpenCV library which implements 11 different feature detection algorithms listed in Table I.

A feature detection algorithm is typically built upon a set of mathematical rules which defines a corner. These mathematical rules control not only how a corner is defined, but also how the pixels in a frame are accessed. The main mechanism of every corner detection algorithm is to traverse each pixel within a frame. Detecting a corner in an image can become a costly process in terms of hardware resources since frames become larger as a consequence of higher resolution, which lead to an increased amount of pixels which have to be traversed. Larger frames can also potentially contain more corners, which furthermore increases the processing time of an image.

TABLE I: Our investigated feature detection algorithms.

Algorithm	License	Description
Harris [11]	BSD	Corner detector
FAST [23]	BSD	Corner detector
SIFT [16]	Proprietary	Object detector
SURF [3]	Proprietary	Object detector
ORB [24]	BSD	Object detector
BRISK [14]	BSD	Corner detector
MSER [17]	BSD	Blob detector
GFTT [26]	BSD	Corner detector
STAR [1]	BSD	Corner detector
DENSE [4]	BSD	Feature extractor
Simple blob [4]	BSD	Blob detector

Fig. 1: Example of FAST and Harris.

Feature detection algorithms use different mechanisms for detecting interest points in an image. There are although some common stages for all algorithms. The first step is always to read the input image file and translate it into a matrix filled with RGB (Red, Green, Blue) data points, where each data point represents a pixel. The second common step is to convert the image in-to grayscale, which is translates the RGB values to a matrix of pixel intensities, which represent values of the brightness of the pixels. After this step, the algorithms begin to execute their respective interest point detection mechanism. The actual detection mechanisms differs a lot depending on the algorithm. To exemplify a diversity, we have depicted the mechanisms of two feature detection algorithms in Fig. 1. The figure illustrates a Sobel filter (marked 1 with purple boxes) which serves as one of the primary mechanisms for the Harris algorithm and a Bresenham Circle (marked 2 with blue boxes) which is the main mechanism of the FAST algorithm.

The second property all algorithms have in common is that the entire image matrix gets traversed at least once. Algorithms such as SURF and SIFT create new matrices that contain results from the initial image matrix. The algorithm repeatedly traverses the original image matrix until it has processed the

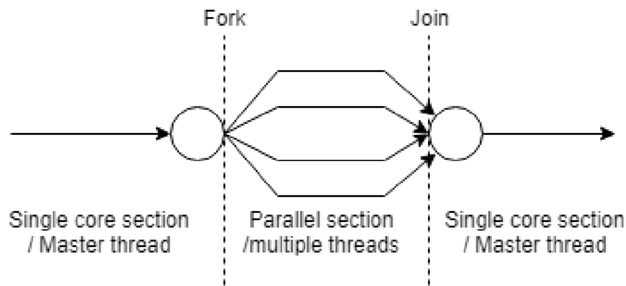

Fig. 2: The fork-join model for parallelization of algorithms.

complete image. There can also be co-dependence between the algorithms, meaning that one algorithm uses the results given by another algorithm. For example, ORB uses the result of Harris or FAST to detect objects. The last step of a feature detection algorithm is to return the pixels considered to be featured. OpenCV calls these features keypoints.

B. Parallel programming

There are various approaches reduce the execution time through parallelism [21]. Designing a feature detection program with a fork-join is one way of utilizing the core-level parallelism, which is efficient due to the mechanics of these algorithms. A fork-join model has two parts controlled by the main thread. First, the fork section where one or several tasks, feasible for parallelization, are allocated over the available CPU cores. The main thread resumes its execution when all spawned tasks have finished and entered the join section. Fig. 2 illustrates an example of the fork-join model utilizing 4 cores.

The fork-join model is a trivial way when trying to increase the performance of feature detection algorithms since there are no global variables shared. This means the algorithms can be split up to work on sub-parts of an image without interfering with another sub-part of the image.

C. Shared memory

Shared resource congestion is one of the major limiting performance factor when running applications, such that the application performance is correlated to the shared resource usage [13]. The resource usage of an application is usually measured by the Performance Monitoring Unit (PMU) [20], such as Intel [15], and deduce resource bottlenecks [7]. The application performance is typically [8, 9] measured in an application-specific metric [2]. In this paper we are mostly concerned with L_3-cache usage because it is the first system-wide shared resource, which makes it the first resource that is eligible to suffer from multi-core memory contention.

It is difficult to correlate the cache usage to execution time [6] when running applications on a HW with shared caches. Sandberg et al. [25] focus on understanding and modeling the execution behavior caused by a congested shared cache. It is also possible to quantize how cache misses affect the system performance by profiling the resource usage of a system [22].

Most non-dedicated computer systems utilize caches to be able to access data quickly. However, the cache is often a costly part of a processor, which limits the amount of available to the CPU. The limited cache size force most CPU to implement cache eviction policies to remove less-used data from the cache and replace it with new data. One of the most commonly known algorithms for replacing data inside a cache is the Least Recently Used (LRU) policy. The LRU tracks data usage, and the least recently used data is removed from the cache and replaced with the new data when the cache is congested. Multi-core systems often make use of a shared cache when communicating between threads and processes. Shared caches of a multi-core processor can, however, lead to negative behavior when using policies such as the LRU policy. When multiple threads access the same memory, the risk is that one thread requests a block of data from the DRAM that replaces the data which was about to be read by another thread. Such congestion scenarios can lead to cache thrashing, where several threads continuously replace each other's data, which in turn can lead to a significant system performance decrease. Computers which execute corner detection algorithms and use a fork-join model will, at some point, have to use the shared resources, such as caches and memory. Shared caches may not be a problem if the image fits into the local cache. Such favorable scenario happens, for example, when the feature detection algorithm can process the whole image in a single iteration, i.e., before other processes replace the cache content. However, the processed memory depends highly on the used algorithm. We have focused to investigate the effects that shared cache congestion causes on the speed-up gains when using the OpenCV feature detection algorithms utilizing a data-partitioned fork-join model.

III. Approach

Our study consists of two parts. The first part is a program that implements the OpenCV algorithms and samples the desired performance counters simultaneously as the test execution time. The second part analyzes the measurements.

A. OpenCV feature detection

OpenCV provides an overlying feature detection class that contains 11 different feature detection algorithms. We have used a data-partitioned fork-join model for evaluating the OpenCV library on multi-core systems. We have depicted the execution model in Fig. 3.

Fig. 3 shows how the workload is distributed to the different cores of our system. At the fork stage, each thread has its affinity set to a core which is not in use by the algorithm, which means thread 0 gets affinity 0 and therefore executes on core 0 and so on. The thread affinity is furthermore used for partitioning the Image. For partitioning the image, we have chosen to divide each image on a height basis. The threads work horizontally on the indexes calculated according to equation (1) where $Work_x$ is the work indexes and $ImageSize_x$ is the horizontal size of the image. The vertical workload is calculated according to equation 3 and 4, where UpperBound is upper vertical index bound, LowerBound is the lower vertical index bound, $ImageSize_y$ is the size of the

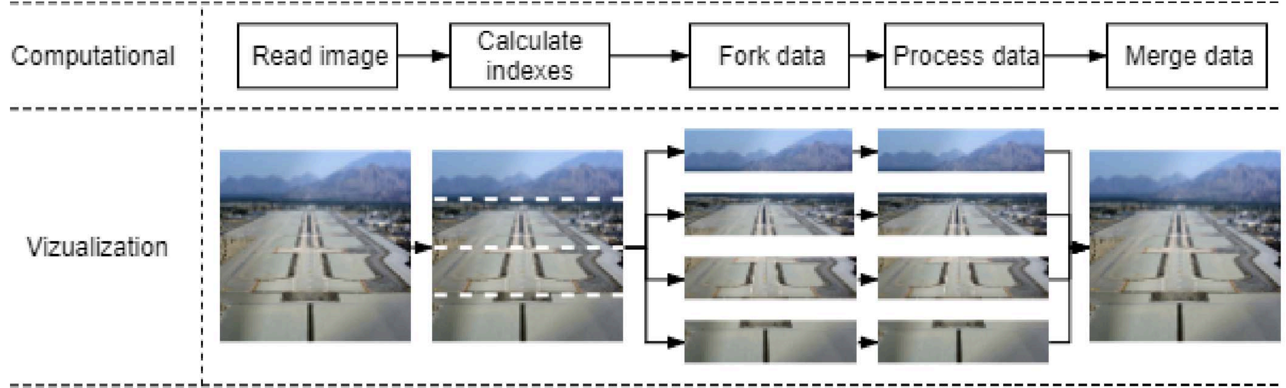

Fig. 3: The image data is partitioned to support the fork-join model where parallelization is supported.

entire image and *aff* is the core affinity of the current thread, which is indexed between 0 and n-1, where 0 is the first core and n-1 is the last core.

$$Work_x = ImageSize_x \tag{1}$$

$$if(aff) = 0, \quad UpperBound \quad = 0 \tag{2}$$

$$if(aff) > 0, \quad UpperBound \quad = \frac{ImageSize_y}{aff + 1} \tag{3}$$

$$LowerBound = \frac{ImageSize_y}{aff + 2} \tag{4}$$

B. Performance Monitoring

We have implemented a system function that simultaneously monitor the resource usage and performance of an application.

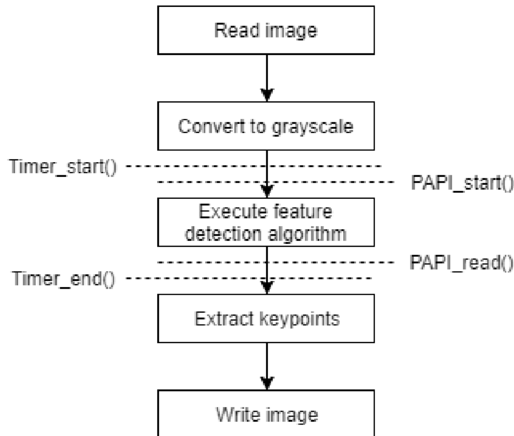

Fig. 4: The algorithm performance measurement sequence.

1) Application Performance: We measure the execution time of each algorithm using the high resolution clock chronox (c++11 library) for measuring the algorithm execution time. The placement of the timestamps are depicted in Fig. 4.

2) Resource Usage: We monitor the number of shared cache misses by using the Performance API library (PAPI) [18] which provide an interface towards the PMU [20]. We insert PAPI start before the algorithm start and PAPI read when the algorithm is finished, as depicted in Fig. 4.

IV. Experiment

We have run our experiments on a quad-core Intel® Core™ i5-3570 processor running at 3.40GHz using g++ version 5.4 with -pthread, -std=c++11 and -O3 as compiler arguments. The HW specifications are listed in Table II. Streaming SIMD Extensions (SSE) instructions are enabled by OpenCV as default configuration.

We measure two different parameters in our test suite, utilizing different amount of cores. The first parameter is Application performance which measures the total execution time of the feature detection algorithms utilizing 1, 2, 3 and 4 cores. We then use the execution time to calculate the speed-up gained from using multiple cores compared to single core. The second parameter is the execution-time measured per image partition, which means we execute the same image partitions but on single core and compare them to our per-core multi-core respective values. At the same time, we also measure the L_3-cache misses which describes the shared resource usage

TABLE II: Hardware specifications Intel® Core™ i5-3570.

Feature	Hardware Component
Core	4xIntel® Core™ i5-3570 CPU (Ivy Bridge) 3.4GHz
L_1-cache	32 KB 8-way set assoc. instruction caches/core + 32 KB 8-way set assoc. data cache/core
L_2-cache	256 KB 8-way set assoc. cache/core
L_3-cache	6 MB 12-way set assoc. shared platform cache
MMU	64 Byte line size, 64 Byte Prefetching, D-TLB: 32 entries 2 MB/4 MB 4-way set assoc. + 64 entries 4 KB 4-way set assoc., I-TLB: 128 entries 4 KB 4-way set assoc., L_2U-TLB: 1 MB 4-way set assoc., L_2U-TLB: 512 entries 4 KB/2 MB 4-way assoc.

Fig. 5: Test images.

during the test execution. We have also measured the amount of keypoints detected using single-core on a full image, to be able to see what effects the amount of detected keypoints has on the speed-ups gained.

In our tests we have used images designed to fit different parts of the Cache memory of our test system Intel® Core™ i5-3570k. The specification for our test images are listed in table. We present the test image size variations in Table III.

We have also executed tests against different images, within a similar environment. The images are presented in Fig. 5 and follow the specifications presented listed in Table III.

The purpose of each test is to reveal the feasibility of our data-partitioned program model when using the standard OpenCV feature detection algorithms. The default parameters of the STAR algorithm in the OpenCV feature detection suite uses a specific set of image scales when executing the Laplacian operator. Some of these image scales are so large that they are not feasible for our smaller image variations, which results in a non-proportional speed-up when partitioning small images to even smaller image partitions. We have therefore exempted these inaccurate STAR detector results.

A. Data partitioned measurements

An important behavior to observe when using data partitioned parallelism is the speed-up given by executing the algorithm on multiple cores. This measurement gives us an absolute value on how well the algorithm responded to our proposed parallel data partitioned model. In this section, we present and discuss the speed-up gained by utilizing 2, 3 and 4 cores compared to 1 core. Each test on each core was repeated 500 times to provide a median of the execution times. The median execution time is then used to calculate the speed-up according to Equation (5), where S is the speed-up gained, t_0 is the single-core execution time, t_i is the execution time of core i and n is the number of cores used.

$$S = \frac{t_0}{\{max(t_i) : 0 \leq i < n\}} \quad (5)$$

Fig. 6 shows the speed-up of each feature detection algorithm. The y-axis denotes the gained speed-up, and the x-axis represents 3 test images, each one with 6 image size variations. The first cluster of 6 image sizes belongs to the image shown in Fig. 5 a, the second set to Fig. 5 b, and the third set to Fig. 5 c. We categorize speed-ups into three categories: The first is *linear speed-up*, where the resulting execution time is equal to the single core execution divided by the number of cores used. The second is *sub-optimal speed-up*, which provides a smaller speed-up than the linear one. The third and final is *super-linear speed-up* which provides a more significant speed-up than a linear one.

To increase readability, we will refer to specific test cases as $Img_\#_{figure_size}$ where # is the figure number.

The numbers for the BRISK detector show a sub-optimal speed-up using 4 cores. The achieved speed-up is small when using the smallest image but increases with the image size. However, the speed-up is at its peak at Img_1_{262KB}, Img_2_{1MB} and Img_3_{262KB}. When further increasing the image sizes, the speed-up decreases again. We call this behavior a pyramid-like behavior.

The Dense Feature detector shows a small speed-up using any of our multi-core tests, the peak speed-up is at roughly 70% faster than the original 1 core version. Furthermore, there is no gain at all from using multi-core until increasing the image size to 1 MB. The Smaller sizes of 32 KB, 128 KB, and 256 KB actually decrease the execution time compared to the single core version. The Dense detector also shows a pyramid-like behavior and has peak performance at $Img_2_{6.1MB}$ and peak speed-up at $Img_1_{24.6MB}$ and $Img_3_{24.6MB}$, however,

TABLE III: Image size variations and their cache boundess.

Figure nr.	Image Size	Mem. Req.	Cache boundness
1	103x103	32 KB	L_1-cache
2	209x209	131 KB	4 × L_1-cache
3	295 x295	262 KB	L_2-cache
4	591x591	1 MB	4 × L_2-cache
5	1431x1431	6.1 MB	L_3-cache
6	2862x2862	24.6 MB	4 × L_3-cache

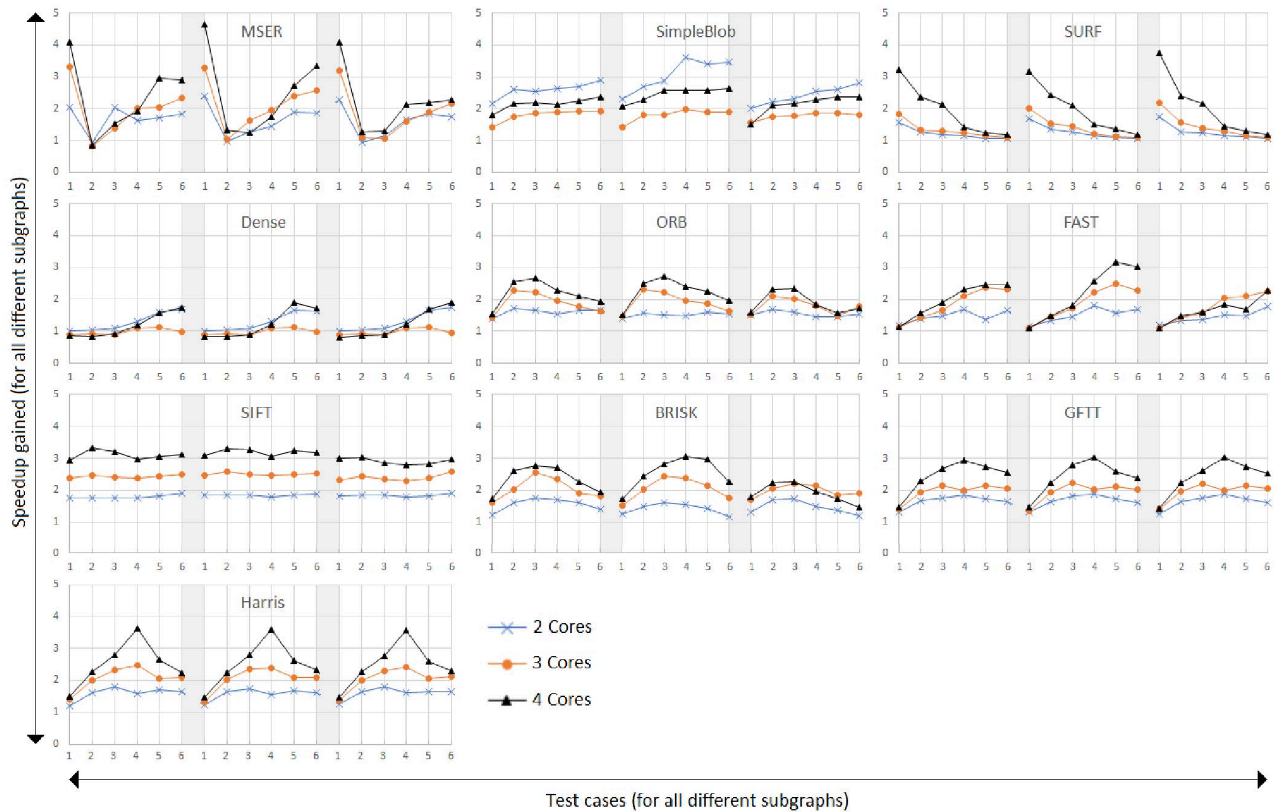

Fig. 6: Feature detection algorithm speed-up factors for various test-cases when running a multi-core test system.

the differences between the speed-ups are roughly 15%, meaning it is small and could just be a coincidence.

The FAST feature detector has a low speed-up using the smaller image sizes and the speed-up increases with the image size. However, FAST reaches a sub-optimal performance at each speed-up peak which is between 2 to 3 times speed-up when using multi-core. The insignificant speed-up gained on the smaller images can be explained as an effect of the overhead gained by the data-partitioned parallelism. If the overhead of an algorithm is dominant, initializing the algorithm multiple times will make the algorithms parallelism less efficient, or even worse (as seen in the Dense algorithm) when using images so small that the work-load execution time does not match the overhead execution time.

The GFTT feature detector has a similar speed-up result for all three test suites. The smallest image has a speed-up of roughly 50%, which is similar to the speed-up of the largest image. Furthermore, the GFTT feature detector achieves a close to optimal speed-up using the 1 MB image. Due to the major speed-up differences, GFTT presents an even stronger pyramid behavior than the Dense and BRISK feature detector.

The speed-up obtained by using 2,3 and 4 cores on Harris are similar to the speed-ups of the GFTT feature detector which is reasonable since it is based upon the same fundamentals as GFTT. The 1 MB image provides the best speed-up, however, in the Harris case a speed-up of almost 4 instead of 3. Furthermore Test suite 1 and 2 of the Harris test are similar in the matter of speed-up behavior, but the 3rd test suite has a lesser peak speed-up at the 1 MB image.

The speed-up obtained utilizing four cores using the MSER feature detector show a different behavior from the other feature detectors. The speed-ups illustrate a reverse pyramid behavior, whereas the 32 KB image obtains a small super linear speed-up and the other images show a lesser speed-up. The trend is a speed-up to the 6 MB version of the images, and then a stall of the speed-up.

The speed-up of ORB illustrate a small pyramidic behavior with a peak at the 3rd size variation of each image. The speed-up the progressively decreases as the image size increases.

The Simpleblob speed-up illustrates a small speed-up as the image sizes increases. This is an on-going process as the speed-up is lowest at the smallest image variation and highest at the largest image variation. The exception is the test results from Img_2_{1MB}, which provides a slightly higher speed-up than the other 1 MB sizes.

The SIFT speed-up is the only algorithm which presents a close to consistent speed-up on all of the frames. Although the speed-up obtained from all frames is sub-optimal, the speed-up gained from SIFT is close to the same on the 32 KB version as the speed-up gained on the 24.6 MB version. This result suggest that SIFT is a scalable solution for every image size.

TABLE IV: Detected key points.

Image	Size	HARRIS	SimpleBlob	SIFT	SURF	ORB	MSER	GFTT	FAST	Dense	BRISK
1	32KB	90	0	55	61	50	24	276	330	324	13
1	128KB	110	0	281	285	358	29	737	1184	1225	72
1	256KB	217	0	450	644	453	59	1000	2211	2500	173
1	1MB	613	3	1502	2341	500	187	1000	7264	9801	565
1	6MB	1000	9	5632	10945	500	743	1000	23828	57121	2214
1	24MB	1000	38	16652	33346	500	1989	1000	51253	227529	5898
2	32KB	81	0	56	65	56	33	200	280	324	12
2	128KB	137	0	185	321	339	66	489	868	1225	51
2	256KB	203	0	433	545	428	97	720	1355	2500	108
2	1MB	593	7	1459	1769	500	198	1000	4443	9801	363
2	6MB	1000	9	3645	7856	500	614	1000	22833	57121	853
2	24MB	389	15	4824	28929	500	1021	1000	71934	227529	1154
3	32KB	100	0	80	93	59	35	199	318	324	18
3	128KB	347	0	245	385	370	63	763	1151	1225	86
3	256KB	497	0	455	710	461	97	1000	1824	2500	154
3	1MB	1000	12	1581	2399	500	276	1000	6212	9801	537
3	6MB	1000	70	6015	8288	500	1043	1000	10831	57121	1447
3	24MB	1000	133	27472	41037	500	3084	1000	142727	227529	6782

The SURF detector illustrates a behavior which originally expected for all algorithms, since the smaller images fit entirely in the L1 cache and potentially could be processed directly. SURF executes the 32 KB images at a super-linear which gradually decreases when the image size is increased.

B. Keypoints detected

OpenCV denotes features detected as keypoints. Due to the varying sizes of the images, there will be a variance in detected keypoints even though the algorithm in scale-invariant, simply because there are less pixels available. Table IV presents the keypoints detected in each image variation for each algorithm. Since we are using the default settings of OpenCV, some algorithms use a threshold value of how many keypoints can be detected at max, this occurrence can be seen in the HARRIS, GFTT and ORB detectors. As the number of detected keypoints increases with the image size, except for the algorithms which have a threshold value, we can conclude that the keypoint detection does not have a negative impact on the speed-up gained by an algorithm. This occurrence is especially clear in the FAST detector, which has a larger speed-up at the largest frame with 21253 (image 1), 71934 (image 2) and 142727 (image 3) keypoints detected than the smallest frame which only finds 330 (image 1), 280 (image 2) and 318 (image 3).

C. Execution time differences

We have measured the execution time of the program when it is run in parallel and compared it to a Sequential execution of the program to monitor any eventual losses in the execution time of the parallel program due to shared memory contention and overhead execution times. We executed this test using 4 different cores, introducing synchronization points between each core execution. The sequential version of our program is depicted in Fig. 9. Our sequential version of the program thus executes one image partition, running on one core before executing the next image partition on another core. The maximum execution time of the executing cores represent the execution time of the entire program, since a program is never faster than the slowest core. Each test was conducted 500 times to provide a median value.

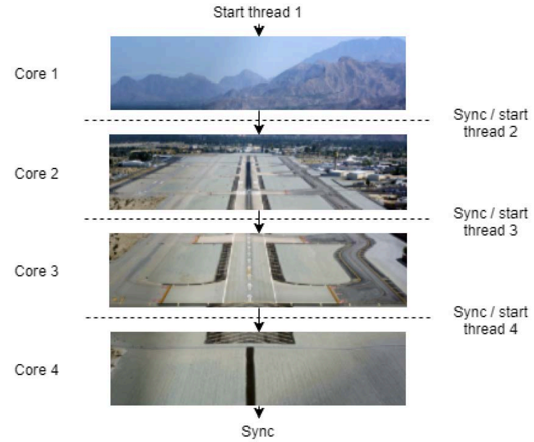

Fig. 9: Sequential version of the test program.

We call the difference between our sequential execution and our parallel execution ΔT, which is calculated according to equation (6) where i is the core used, which are indexed starting from 0 and n is the number of cores used. tp is the median execution time using a parallel approach and ts is the median execution time using a sequential approach.

$$\Delta T = \{max(tp_i) : 0 \leq i < n\} - \{max(ts_i) : 0 \leq i < n\} \quad (6)$$

ΔT allows us to quantify how much of the program execution time is affected by utilizing a multi-core architecture. Fig. 7 illustrates the ΔT per core per image.

Fig. 7 depicts the ΔT on the y-axis using a logarithmic scale w the x-axis represents 3 test images, each one with 6 different image variations, separated with a gray field. The SURF algorithm performed worst in this test, with a ΔT of roughly 900000 microseconds compared to the sequential version using the largest image size.

FAST and Dense are the best overall algorithms according to the ΔT calculations, where the majority of the values are placed within the 80 microseconds range. There few outliers

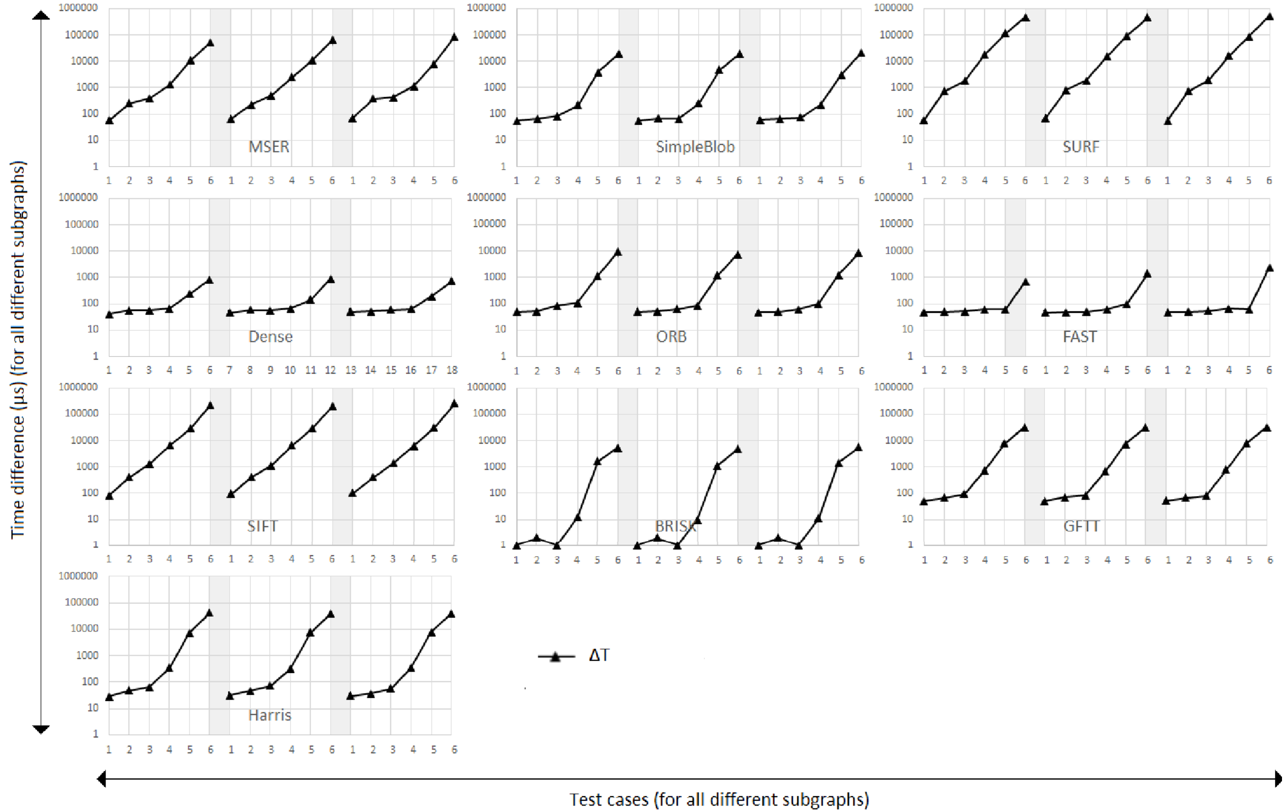

Fig. 7: Differences in execution time using parallel and sequential approach

ranging 2300 microseconds using our largest image sizes which are small compared to the other algorithms.

D. Execution Characteristics

Given the different speed-up behaviors, there are certain events occurring within the hardware, which limits the size of the possible speed-up. We measured 16 different low-level metrics to investigate possible bottlenecks. However, the most important metric to measure is the first system-wide shared resource, which in this case is the L_3-cache, since it is the first shared resource with least amount of memory which makes it most likely to suffer from thrashing by other threads. We have chosen to visualize only the L_3-cache misses metric due to space limitations. Fig. 8 depicts the total amount of L_3-cache misses for both the sequential and parallel versions plotted on the left Y-axis, and the percentage deviation, denoted as ΔC plotted on the right Y-axis. The L_3-cache misses are the measured median values from 500 executions, while ΔC is calculated according to the total cache misses of all used cores when run in parallel divided by the total cache misses of all cores when run sequentially, denoted as ParallelMisses and SequentialMisses in equation (7).

$$\Delta C = \frac{ParallelMisses(C_{1..4})}{SequentialMisses(C_{1..4})} \tag{7}$$

The ideal value of ΔC is 0% L_3-cache difference which indicates that no thrashing has occurred. If thrashing occurs in the cache, the ΔC will increase. If the difference is negative, it means the memory is efficiently re-used by other threads and produces less L_3-cache misses than the sequential version.

Compared to the other algorithms, FAST has a low L_3-cache usage, see Fig. 8, which is proportional to the amount of corners detected. We can also observe that FAST suffers a comparatively low amount of additional cache misses due to memory contention. The largest ΔC are in the smaller frames, but the difference in total is almost negligible. Since the speed-up of FAST is independent on how many cache misses are produced in L_3-cache, we can conclude that FAST is non-cache bound and therefore suitable for parallel executions.

Similarly to FAST, SIFT has a relatively low ΔC at the 6 MB image, which implies that SIFT re-use a lot of the data of the 6 MB variation of the image. The speed-up of SIFT remains unaffected by the ΔC indicating that SIFT is computationally heavy but is not memory bound.

The SURF algorithm has a relatively high ΔC, especially with larger image sizes. L_3-cache misses reveal an increase of 800000 misses in total using the parallel version compared to the sequential one. Concluding that SURF is cache bound is further strengthened by Fig. 5, which depicts an insignificant speed-up when executing on the largest image. It is debatable

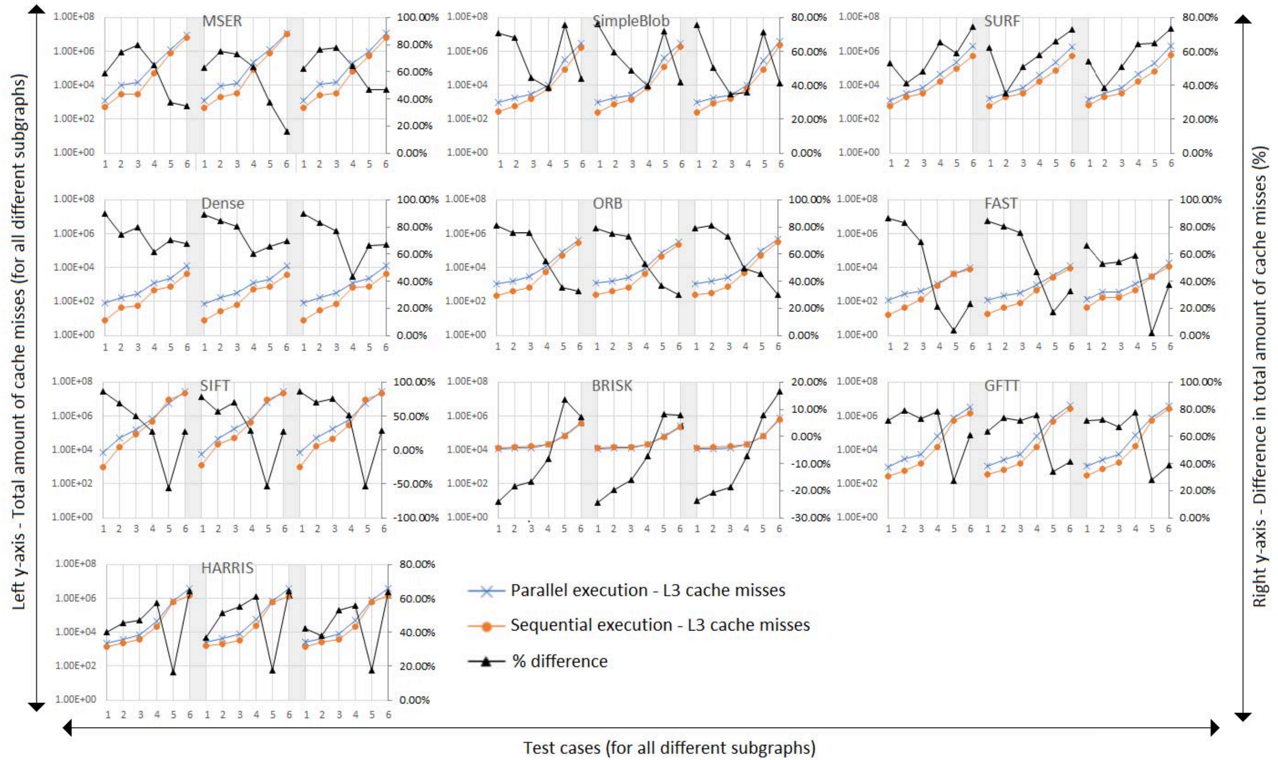

Fig. 8: L3 misses using parallel and sequential version.

how much the increased amount of corners affect the speed-up; however, Fig. 7 reveals a ΔT of almost 1 Second for the largest images, suggesting that the amount of corners detected have small to possibly no effect on the speed-up.

The ORB algorithm has a fairly low ΔC for the larger images and also shows a low ΔT version compared to the other Object detectors. However, the ORB speed-up does not correlate at all with these facts, wherefore we can conclude that ORB is not L_3-cache bound.

The Harris and GFTT algorithms are similar in regards of Speed-up behavior, ΔC and ΔT. However, neither Harris nor GFTT receive a speed-up boost despite the fact that the L_3-cache misses difference is considerably lower for the larger image sizes which indicates that neither Harris nor GFTT are L_3-cache bound.

Dense has a high ΔC for all image variations. Although the total number of cache misses are low, we must also consider the execution time of Dense, which is also low. Since the Dense algorithm presents a ΔT of roughly 3000, it loses 2/3 of its potential execution time when using parallel version. Combining this with the fact that Dense has a high ΔC it is an indication that the Dense algorithm is L_3-cache bound.

BRISK shows a low ΔC as well as a low ΔT even though BRISK has a fairly bad speed-up at the larger images. Due to this fact, we can conclude that BRISK is not L_3-cache bound.

The MSER algorithm can be considered L_3-cache bound due to the correlation between speed-ups gained in the larger images and the ΔC. In Fig. 6, we see a stall in speed-ups from $Img_1_{6.1MB}$ to $Img_1_{24.6MB}$ and $Img_3_{6.1MB}$ to $Img_3_{24.6MB}$. However, the figure shows a speed-up from $Img_2_{6.1MB}$ to $Img_2_{24.6MB}$.

A similar pattern can be detected in Fig. 8 whereas the ΔC differs by 40% in $Img_1_{6.1MB}$, $Img_1_{24.6MB}$, $Img_3_{6.1MB}$ and $Img_3_{24.6MB}$, but only differs 20% for $Img_2_{24.6MB}$..

SimpleBlob has an irregular behavior according to ΔC. The differences for each test-case are common, but it is hard to find any correlation between the ΔC and the speed-ups gained. Simpleblob, however, has a high total amount of L_3-cache misses, and when adding the fact that SimpleBlob has relatively small ΔT compared to its extensive execution time (830000 microseconds), it indicates that SimpleBlob is not observably bound to the L_3-cache.

V. Conclusions

We have evaluated how default configured OpenCV feature-detection algorithms perform when using a data-partitioned parallel programming model for 2,3 and 4 cores. The algorithms performed differently using our data-set. The Harris algorithm obtained the highest speed-up at almost 4 times faster than the original single-core performance. However, this result depends heavily on the image size. SIFT was by far the most stable algorithm showing a speed-up of roughly 3 times the single core performance for all image sizes. SURF, on the other hand, received the worst speed-up, basically insignificant

for larger images, which are the most computationally heavy. We have concluded that the parallelizing speed-ups of SURF, Dense, and MSER, are correlated to L_3-cache usage. Our measurements suggest that a system designer should not co-locate these algorithms with other L_3-cache bound tasks. We have also concluded that FAST, ORB, BRISK, HARRIS, GFTT, SIFT and SimpleBlob are not L_3-cache bound indicating that they can be efficiently utilized on multi-core systems, even though other tasks heavily load the L_3-cache. We further conclude that FAST, Dense, Harris, ORB, GFTT and BRISK all suffer from various degrees of overhead penalties when processing smaller frames.

A. Future work

We have used the default OpenCV parameters in this study, which mean that results from the feature-detection may differ due to different tuning. Therefore, further studies should try to find an optimal tuning for each frame and execute the the parallel feasibility tests described in our study. It is also possible to investigate the feasibility of co-executing feature detection algorithms on different cores. Running SURF which we concluded to be L_3-cache bound on one core and running FAST which is not L_3-cache bound on the three remaining cores could potentially be an efficient approach when the objective of a system is to detect both blobs and corners.

References

[1] M. Agrawal, K. Konolige, and M. R. Blas. Censure: Center surround extremas for realtime feature detection and matching. In *European Conference on Computer Vision*, pages 102–115. Springer, 2008.

[2] A. R. Alameldeen and D. A. Wood. IPC considered harmful for multiprocessor workloads. *IEEE Micro*, pages 8–17, 2006.

[3] H. Bay, T. Tuytelaars, and L. Van Gool. Surf: Speeded up robust features. *Computer vision–ECCV 2006*, pages 404–417, 2006.

[4] G. Bradski. The opencv library. *Dr. Dobb's Journal: Software Tools for the Professional Programmer*, 25(11):120–123, 2000.

[5] T. P. Chen, D. Budnikov, C. J. Hughes, and Y. Chen. Computer vision on multi-core processors: Articulated body tracking. In *Multimedia and Expo, 2007 IEEE International Conference on*, pages 1862–1865. IEEE, 2007.

[6] C. Ding, X. Xiang, B. Bao, H. Luo, Y. Luo, and X. Wang. Performance metrics and models for shared cache. *Journal of Computer Science and Technology*, 29(4):692–712, 2014.

[7] Stéphane Eranian. What can performance counters do for memory subsystem analysis? In *Proceedings of the 2008 ACM SIGPLAN workshop on Memory systems performance and correctness: held in conjunction with the Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'08)*, pages 26–30. ACM, 2008.

[8] S. Eyerman and L. Eeckhout. System-level performance metrics for multiprogram workloads. *IEEE Micro*, 28(3):42–53, 2008.

[9] S. Eyerman, K. Hoste, and L. Eeckhout. Mechanistic-empirical processor performance modeling for constructing CPI stacks on real hardware. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 216–226, 2011. doi: 10.1109/ISPASS.2011.5762738.

[10] H. Feng, E. Li, Y. Chen, and Y. Zhang. Parallelization and characterization of sift on multi-core systems. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 14–23. IEEE, 2008.

[11] C. Harris and M. Stephens. A combined corner and edge detector. In *Alvey vision conference*, volume 15, pages 10–5244. Manchester, UK, 1988.

[12] F. Hosseini, A. Fijany, and J. Fontaine. Highly parallel implementation of harris corner detector on csx simd architecture. In *European Conference on Parallel Processing*, pages 137–144. Springer, 2010.

[13] M. Jägemar, A. Ermedahl, S. Eldh, and M. Behnam. A Scheduling Architecture for Enforcing Quality of Service in Multi-Process Systems. In Proceedings of Emerging Technologies and Factory Automation. *Analysis*, ETFA 2017.

[14] S. Leutenegger, M. Chli, and R. Y. Siegwart. Brisk: Binary robust invariant scalable keypoints. In *Computer Vision (ICCV), 2011 IEEE International Conference on*, pages 2548–2555. IEEE, 2011.

[15] D. Levinthal. Performance Analysis Guide for Intel ® Core i7 Processor and Intel ® Xeon 5500 processors. *Intel Cooperation*, pages 1–72, 2009.

[16] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91–110, 2004.

[17] J. Matas, O. Chum, M. Urban, and T. Pajdla. Robust wide-baseline stereo from maximally stable extremal regions. *Image and vision computing*, 22(10):761–767, 2004.

[18] P. J. Mucci, S. Browne, C. Deane, and G. Ho. Papi: A portable interface to hardware performance counters. In *Proceedings of the department of defense HPCMP users group conference*, volume 710, 1999.

[19] Open Computer Vision. Common interfaces of Feature detectors. URL https://docs.opencv.org/2.4/modules/features2d/doc/common_interfaces_of_feature_detectors.html.

[20] D. Patil, P. Kharat, and A. K. f Gupta. Study of Performance Counters and Profiling Tools. In *Proceedings of 21st IRF International Conference.*, number March, pages 45–49, Pune, India, 2015. ISBN 9789382702757.

[21] M. J. Quinn. Parallel programming. *TMH CSE*, 526, 2003.

[22] N. Rameshan, R. Birke, L. Navarro, V. Vlassov, B. Urgaonkar, G. Kesidis, M. Schmatz, and L. Y. Chen. Profiling memory vulnerability of big-data applications. In *Dependable Systems and Networks Workshop, 2016 46th Annual IEEE/IFIP International Conference on*, pages 258–261. IEEE, 2016.

[23] E. Rosten and T. Drummond. Machine learning for high-speed corner detection. *Computer Vision–ECCV 2006*, pages 430–443, 2006.

[24] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski. Orb: An efficient alternative to sift or surf. In *Computer Vision (ICCV), 2011 IEEE international conference on*, pages 2564–2571. IEEE, 2011.

[25] A. Sandberg, A. Sembrant, E. Hagersten, and D. Black-Schaffer. Modeling performance variation due to cache sharing. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pages 155–166. IEEE, 2013.

[26] J. Shi et al. Good features to track. In *Computer Vision and Pattern Recognition, 1994. Proceedings CVPR'94., 1994 IEEE Computer Society Conference on*, pages 593–600. IEEE, 1994.

[27] Hiroki Sugano and Ryusuke Miyamoto. Parallel implementation of good feature extraction for tracking on the cell processor with opencv interface. In *Intelligent Information Hiding and Multimedia Signal Processing, 2009. IIH-MSP'09. Fifth International Conference on*, pages 1326–1329. IEEE, 2009.

[28] Nan Zhang. Computing optimised parallel speeded-up robust features (p-surf) on multi-core processors. *International journal of parallel programming*, 38(2):138–158, 2010.

[29] Q. Zhang, Y. Chen, Y. Zhang, and Y. Xu. Sift implementation and optimization for multi-core systems. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8. IEEE, 2008.