Investigating Execution-Characteristics of Feature-Detection Algorithms

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

***Abstract*—We discuss how to obtain information of execution characteristics, such as parallelizability and memory utilization, with the final aim to improve the performance and predictability of feature and corner detection algorithms for use in e.g. robotics and autonomous machines. Our aim is to obtain a better understanding of how computer vision algorithms use hardware resources and how to improve the time predictability and execution time of such algorithms when executing on multi-core CPUs. We evaluate a fork-join model applicable to feature detection algorithms and present a method for measuring how well the algorithm performance correlates with hardware resource usage. We have applied our method to the Featured from Accelerated Segment Test (FAST) algorithm. Our characterization of FAST reveals that it is an algorithm with excellent parallelism opportunities, resulting in an almost linear speed-up per core. Our measurements also reveal that the performance of FAST correlates very little with the number number of misses in the L1 data cache, L1 instruction cache, data translation lookaside buffer and L2 cache. Thus, the FAST algorithm will not have a negative effect on the execution time when the input data fits in the L2 cache.**

I. Introduction

Robots are becoming more autonomous which means they are getting more dependent upon perception algorithms for object detection. Object recognition algorithms often use preprocessing algorithms to extract changes in color or features in an image, called feature detection algorithms. Since the robot is getting a continuous video stream, it is important that the response of the feature detection algorithm is fast enough so that the robot can maintain uninterrupted perception of the environment.

Many different feature detection algorithms exist, which use many different execution patterns. Some algorithms, themselves, are depending upon other feature detection algorithms to do preprocessing before doing the actual work, while others execute directly on the raw image data. Different feature detection algorithms also work in finding different targets in an image such as lines, corners and edges. Perception often make use of many different, and well established, feature and corner detection algorithms such as FAST, SIFT, SURF, and Harris. Often they are combined to achieve better object recognition. However, the algorithms which are combined have very different execution patterns which put stress on how to use the computer hardware efficiency to meet the timing. Multi-core architectures offer great opportunities for executing multiple algorithms on different cores and also enables the workload of a single algorithm to be spread out across multiple cores. However, since feature detection algorithms use different execution patterns, they exhibit very different memory-access patterns which impact on both the predictability and execution time of the algorithms. Characteristics which affect the predictability and execution time of an algorithm can be, but are not limited to, resource utilization, execution time, possibility to distribute the workload to different cores as well as how to schedule different algorithms together to achieve maximum quality of service. To investigate how these characteristics affect the system performance, a firm understanding of the algorithms properties and how it use hardware resources is needed. In this paper, we discuss aspects which are important to consider when using a feature detection algorithm as a real-time application executing on a parallel platform. As a case study, we have evaluated the Features from Accelerated Segment Test (FAST) [12] algorithm with respect to these above mentioned categories by investigating the algorithm behavior. The rest of the paper is organized as follows. Section II presents a brief study of related work. Section III presents the requirement to build the testing tool. Section IV show our methodology and measurements on the characteristics parallelism and memory utilization, Section IV presents challenges occurring when executing FAST on multiple cores, while Finally, Section V concludes the paper and directs to future work.

978-1-5090-6505-9/17$31.00

A. The FAST algorithm

The FAST algorithm is used for detecting features and corners in images. The main mechanism of FAST is based upon using a Bresenham circle of radius 3 (depicted in figure 1) which is compared to all pixels in an image matrix. The execution of FAST is divided into two steps: determining if a selected pixel is an **Interest Point** and determining if the interest point is a **Corner**. Two threshold values are used for making this decision. The first threshold is the intensity threshold (I_t) which is a percentage value that is applied to the pixel intensity (I) of all pixels within the Bresenham circle. If a pixel in the Bresenham circle is (I_t)% darker or brighter than the currently selected pixel, it is considered as a feature. The second threshold value N is used for deciding how many pixels in the Bresenham circle should be features in order for the currently selected pixel to be considered as a corner. To determine if the selected pixel is an interest point, FAST compares the pixel intensity value of the four utmost pixels -

marked in Figure 1 as pixel 1, 5, 9 and 13 with the intensity of the currently selected pixel. If at least 3 of the utmost pixels are considered features, the current pixel is marked as an interest point and the algorithm continues to execute, else break. In the second step, the FAST algorithm compare the currently selected pixel to the rest of the pixels in the Bresenham circle. If at least N contiguous pixels are considered as features, the current pixel is considered as a corner [12]. Since different thresholds will affect execution time, we have executed the tests using 10%, 20%, 30% and 40% as threshold values for I_t and a threshold value of 12 for N.

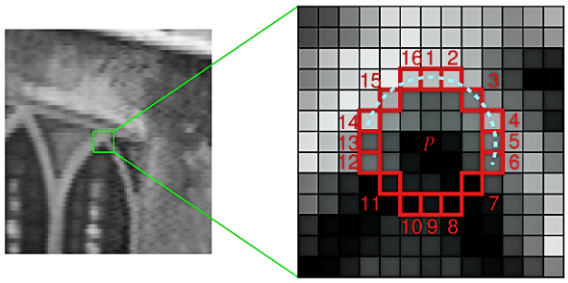

Fig. 1. FAST algorithm corner example where p is the currently selected pixel for analysis

B. Hardware Resource Monitoring

The computational performance of the CPU is not the only limiting factor when evaluating the performance of an algorithm. Often, an algorithm process data that read from main memory putting a severe strain on memory buses and various cache levels. An algorithm may also suffer from other congestion-related side-effects; For example, if the branch prediction unit fails to predict the execution flow of the program. One of our goals is to monitor and evaluate the hardware resource usage of several edge detection algorithms. Having a thorough understanding of the system-level performance together with the low-level shared resource utilization makes it possible to understand better how to implement and deploy different algorithms [2] efficiently. Understanding the hardware usage of the algorithms will also make it easier to select the optimal hardware without expensive resources overprovisioning providing a greater throughput [3]. We estimate that it is possible to substantially improve the overall system performance of co-located algorithms by optimizing the core allocation of algorithms [4]. We will use a performance monitoring application that utilize the Performance Monitor Unit (PMU) to continuously monitor the performance of selected processes. Our application utilizes the Perf API for convenient PMU configuration.

II. Related work

Different works for comparing Edge detection techniques have been done, whereas the comparisons often include correctness of the algorithms [7] [15] [14] [6] and FPS comparison [11]. Furthermore, many works try to optimize feature detection algorithms using specialized hardware environments such as FPGA [8] [13] [10]. Other studies also compare image processing algorithms using different hardware units such as CPU, GPU and FPGA [1]. Furthermore, Paul et al. [9] presented a method which investigates the resource usage in the Harris Corner detection algorithm.

In this study, we instead try to focus on a broad scope by identifying characteristics which are important to investigate when using feature detection algorithms on limited hardware. Furthermore the ultimate goal of this work is to be able to determine how the quality of service can be affected of feature algorithm using a system with specific characteristics. In this paper we investigate the mechanics of the FAST algorithm by evaluating the code as well as analyzing the algorithms effects on the memory of the system in which it is running.

III. Method

In this work, we evaluate three characteristics including memory resources, opportunities for parallelism and resource usage with respect to the FAST algorithm. For a richer proof of concept, we used the 8-core Freescale P4080 with a 2017-03 NXP fsl-core linux distribution for evaluating the parallel issues due to the high multi-core capabilities while we used an Intel Core i-3570k using Ubuntu 16.04, kernel version 4.4 for investigating PMU related topics. We used a 512x512 bitmap version of the Lena image as test data, due to its frequent usage in other corner detection research papers. We executed all FAST tests on a 512x512 pixel bitmap, depicted in figure 2[1].

Fig. 2. Input data for FAST

A. Opportunities for Parallelism

A traditional way of executing algorithms in parallel is using the fork-join model, which partitions a workload into smaller workloads and executes these smaller workloads on different cores. When working with image processing, this is a simple and intuitive way of increasing the performance, since a pixel matrix often can be divided into sub-matrices by the amount of cores used. There is no need for synchronization within the algorithm steps because the FAST algorithm does not use global shared variables. The FAST algorithm with a fork-join using an 8 core machine is depicted in Figure 3. The optimal performance of an algorithm running in parallel is defined by Execution time using one core divided by amount of cores being used. The execution model in Figure 3 depicts a straightforward fork-join execution model which is possible for the

[1]The figure was selected because it is one of the standard test images in the image processing community

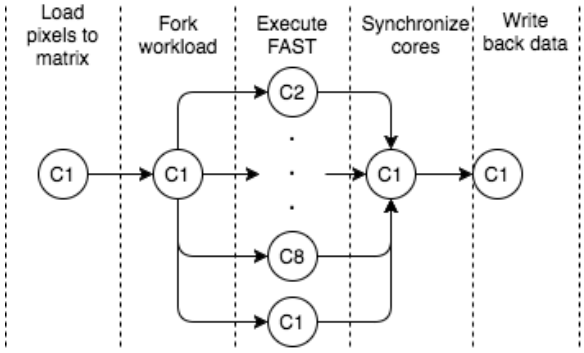

Fig. 3. FAST 8 core execution model

fast algorithm. This leads to a near-optimal execution speed-up when using multiple cores. For proof of concept, we took measurements of FAST running on 1 to 7 cores using an 8 core Freescale P4080 machine and executing a test 100 times on the same picture. Table I shows the percentage deviation from the optimal performance when executing on multiple cores using I_t thresholds of 10%, 20%, 30% and 40%.

Cores	$I_t = 10\%$	$I_t = 20\%$	$I_t = 30\%$	$I_t = 40\%$
2 cores	4,41%	3,05%	1,98%	1,26%
3 cores	4,69%	3,21%	1,79%	0,98%
4 cores	5,10%	3,61%	2,21%	1,06%
5 cores	5,09%	2,93%	1,59%	0,92%
6 cores	5,86%	4,34%	2,80%	1,69%
7 cores	4,51%	2,79%	2,58%	1,95%

TABLE I
FORK-JOIN MEASUREMENTS OF FAST

As shown in Table I, we see only a small percentage deviation to the optimal execution time even when the threshold is set to 10% which can be considered a very sensitive threshold. Thus we can conclude that FAST is an algorithm very well suited for parallelism. Executing tests using 5 cores managed to decrease the deviation percentage compared, even though the trend was an increasing percentage. This can be an indication of uneven workload between the different cores.

B. Resource Utilization

In our investigation, we have focused on two memory-related issues. The first issue relates to the code size of the algorithm itself. A long and complex execution flow causes several performance-related side-effects, such as instruction cache misses and branch mispredictions. The second memory-related issue relates to the data processed by the algorithm. A memory-bound algorithm has a large working set and triggers a high degree of data cache misses, Data-TLB reloads and memory bus contention which can cause a decrease in performance. If two algorithms are memory-bound, it may not be sufficient to distribute them on different CPU cores because they often share HW resources. We have used a system-level metric (SLM) as performance indicator that describes the number of pixels traversed per time unit. Simultaneously, we monitor Low-Level Metrics (LLM) describing the memory subsystem usage. We use the Pearson coefficient [5] to denote how well SLM correlates with each LLM whereas 0 mean no correlation at all and 1 full correlation. We tested the correlation by executing a test-suite that fetches SLM and LLM at 100Hz. We ran this test using a fork-join model with four cores, where core 0 was set up as a synchronization core and core 1-3 as workload cores.

Core : I threshold	L1D miss	L1I miss	L2 miss	DTLB miss
Core 1 : I_t=10%	0.195	0.114	0.191	0.233
Core 2 : I_t=10%	0.103	0.067	0.084	0.21
Core 3 : I_t=10%	0.056	0.395	0.029	0.133
Core 1 : I_t=20%	0.004	0.365	0.145	0.024
Core 2 : I_t=20%	0.206	0.172	0.197	0.24
Core 3 : I_t=20%	0.076	0.395	0.427	0.4
Core 1 : I_t=30%	0.073	0.013	0.07	0.234
Core 2 : I_t=30%	0.198	0.187	0.078	0.169
Core 3 : I_t=30%	0.208	0.083	0.204	0.131
Core 1 : I_t=40%	0.012	0.035	0.119	0.083
Core 2 : I_t=40%	0.21	0.187	0.246	0.172
Core 3 : I_t=40%	0.431	0.395	0.427	0.4

TABLE II
FORK-JOIN MEASUREMENTS OF FAST

Table II show Pearson coefficient obtained from the workload cores during the tests of correlation between SLM and the LLM. Our measurements indicate that the performance correlates very little with the memories measured in this test. This occurrence may be due to the fact that FAST has few memory operations and instead uses many branch operations. The current pattern however suggest that core 3 correlates best with the LLM we chose. The correlation may be an effect of how the picture is divided. With the current division of the picture, core 3 would detect the least corners, and would use the least branches and would therefore correlate better with the memory.

IV. RESOURCE USAGE CHALLENGES

Many resources affect the performance of an algorithm apart from the earlier mentioned ones. Different parallel paradigms are useful depending on the algorithm, for example, the fork-join model. Due to the many if statements of FAST, it is very unlikely that a forked algorithm will execute with at the same speed on different cores. If one core is overwhelmed with corner detections it can lead to one core executing the algorithm slower than the other cores, leaving the other cores underutilized, Figure 4 illustrates such behavior. Altering

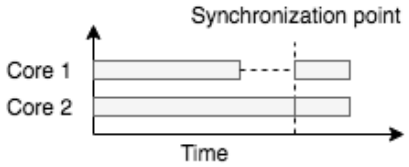

Fig. 4. Core idle issue due to synchronization

the threshold values of FAST can dramatically change the result of found corners in an image. To test the resource utilization of FAST, we executed 1000 tests on a single picture, measuring the execution time of each individual core,

whereas core 1-7 were used as work-set cores and core 0 as housekeeping/synchronization thread. Figure 5 depicts the mean execution time of the 1000 tests for each individual work-set core using I_t values of 10%, 20%, 30% and 40%.

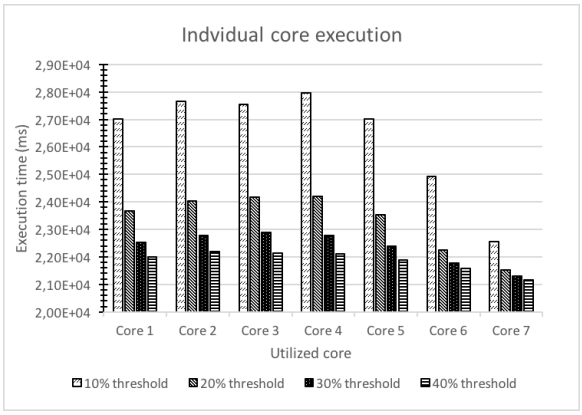

Fig. 5. Execution time per core with different I_t values using FAST

Each core also had different amount of corner detection, Table III shows the amount of corner points detected in each individual core.

Core	10%	20 %	30 %	40 %
1	2018	761	373	206
2	2460	873	432	232
3	2301	842	377	203
4	2391	796	390	178
5	1938	588	432	102
6	1284	253	99	39
7	546	99	42	11

TABLE III
PRESENTS THE AMOUNT OF CORNERS DETECTED PER CORE

From the measurements conducted in this section, we can conclude that the inter-core synchronization time correlates with the amount of corners detected. This mean executing FAST on images which does not have corners evenly distributed, may lead to a utilization loss when executing FAST on multiple cores.

V. CONCLUSION

In this study, we have evaluated an implementation of the FAST algorithm regarding aspects of resource utilization, opportunities for parallelism and memory consumption using different thresholds. Our results show that FAST is a relatively simple algorithm with strong opportunities for parallelism. We see a challenge with choosing threshold values which has to be investigated further. If choosing a high threshold, there is a risk of not detecting important corners. If however choosing a low threshold, there is a possibility of loading the system inefficiently. It could however be possible to schedule FAST more efficiently by programming already finished cores to help the non finished cores finish. This approach could reduce the synchronization performance loss. By monitoring the PMU counters, we also conclude that FAST does not suffer much from misses in the memory.

Future work includes conducting a deeper study of the execution characteristics, investigating both execution behavior as well as memory behavior of more well-known feature and corner detection algorithms such as Harris, SURF and SIFT. By carrying out such a study, it is possible to understand how different feature detection algorithms should be partitioned and scheduled together. Ultimately, this knowledge can lead to a more time-predictable and dependable corner detection suite.

REFERENCES

[1] Shuichi Asano, Tsutomu Maruyama, and Yoshiki Yamaguchi. Performance comparison of fpga, gpu and cpu in image processing. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pages 126–131. IEEE, 2009.

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

[3] Stijn Eyerman and Pierre Michaud. Defining metrics for multicore throughput on multiprogrammed workloads. Technical report, Ghent University - Team ALF, 2013.

[4] Marcus Jägemar, Andreas Ermedahl, and Sigrid Eldh. Decision support for OS process scheduling based on HW-, OS- and system-level performance counters, 2016.

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

[6] Luo Juan and Oubong Gwun. A comparison of sift, pca-sift and surf. *International Journal of Image Processing (IJIP)*, 3 (4):143–152, 2009.

[7] Raman Maini and Himanshu Aggarwal. Study and comparison of various image edge detection techniques. *International journal of image processing (IJIP)*, 3(1):1–11, 2009.

[8] Rajesh Mehra and Rupinder Verma. Area efficient fpga implementation of sobel edge detector for image processing applications. *International Journal of Computer Applications*, 56(16), 2012.

[9] Johny Paul, Walter Stechele, Manfred Kröhnert, Tamim Asfour, Benjamin Oechslein, Christoph Erhardt, Jens Schedel, Daniel Lohmann, and Wolfgang Schröder-Preikschat. Resource-aware harris corner detection based on adaptive pruning. In *International Conference on Architecture of Computing Systems*, pages 1–12. Springer, 2014.

[10] Paulo Ricardo Possa, Sidi Ahmed Mahmoudi, Naim Harb, Carlos Valderrama, and Pierre Manneback. A multi-resolution fpga-based architecture for real-time edge and corner detection. *IEEE Transactions on Computers*, 63(10):2376–2388, 2014.

[11] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 779–788, 2016.

[12] Edward Rosten and Tom Drummond. Fusing points and lines for high performance tracking. In *Computer Vision, 2005. ICCV 2005. Tenth IEEE International Conference on*, volume 2, pages 1508–1515. IEEE, 2005.

[13] Varun Sanduja and Rajeev Patial. Sobel edge detection using parallel architecture based on fpga. *International Journal of Applied Information Systems*, 3(4):20–24, 2012.

[14] GT Shrivakshan, C Chandrasekar, et al. A comparison of various edge detection techniques used in image processing. *IJCSI International Journal of Computer Science Issues*, 9(5): 272–276, 2012.

[15] Kaiman Zeng, Nansong Wu, Lu Wang, and Kang K Yen. Local visual feature detection and description for non-rigid 3d objects. *Advances in Image and Video Processing*, 4(2):01, 2016.