Mälardalen University
School of Innovation Design and Engineering
Västerås, Sweden

DVA331 Thesis for the Degree of Bachelor in Computer Science

INVESTIGATING MEMORY CHARACTERISTICS OF CORNER DETECTION ALGORITHMS USING MULTI-CORE ARCHITECTURES

André Sääf
asf14002@student.mdh.se

Alvin Samuelsson
asn14010@student.mdh.se

Examiner: Dr. Moris Behnam

Supervisor: Jakob Danielsson

May 24, 2017
Abstract

In this thesis, we have evaluated the memory characteristics and parallel behaviour of the SUSAN (Smallest Unvalue Segment Assimilating Nucleus) and Harris corner detection algorithms. Our purpose is understanding how the memory affects the predictability of these algorithms and furthermore how we can use multi-core machines to improve the execution time of such algorithms. By investigating the execution pattern of the SUSAN and Harris corner detection algorithms, we were able of breaking down the algorithms into parallelizable parts and non-parallelizable parts. We implemented a fork-join model on the parallelizable parts of these two algorithms and we were able to achieve a 7.9–8 times speedup on the two corner detection algorithms using an 8-core P4080 machine. For the sake of a wider study, we also executed these parallel adaptations on 4 different Intel platforms which generated similar results. The parallelized algorithms are also subjects for further improvement. We therefore investigated the memory characteristics of L1 data and instruction cache misses, cycles waiting for L2 cache miss loads, and TLB store misses. In these measurements, we found a strong correlation between L1 data cache replacement and the execution time. To encounter this memory issue, we implemented loop tiling techniques which were adjusted according to the L1 cache size of our test systems. Our tests of the tiling techniques exhibit a less fluctuating memory behaviour, which however comes at the cost of an increase in the execution time.
Acknowledgement

We would like to express our sincere gratitude to our supervisor Jakob Danielsson for his continuous advice and encouragement throughout this thesis project. We have been extremely lucky to have a supervisor who showed so much support and interest about our work.
# Table of Contents

1 **Introduction** 5

2 **Background** 5  
2.1 Kernel Filters 5  
2.2 Corner Detection 6  
2.2.1 Harris Corner Detection 6  
2.2.2 SUSAN Corner Detection 7  
2.3 Multi-core Architecture and Caches 7  
2.3.1 Translation Lookaside Buffer 8  
2.3.2 Techniques for Improving Cache Utilization 9  
2.3.3 Prefetching 9  
2.4 Resource Monitoring 10  
2.5 Branches and Branch Predictors 10

3 **Related Work** 10

4 **Problem Formulation** 11  
4.1 Research Questions 11  
4.2 Motivation 11  
4.3 Outcomes 11

5 **Method** 11  
5.1 Implementation 11  
5.1.1 Harris Corner Detection 12  
5.1.2 SUSAN Corner Detection 12  
5.1.3 Tiling Techniques 13  
5.2 Test Setup 14

6 **Results** 14  
6.1 Parallel Capabilities 15  
6.1.1 Harris 15  
6.1.2 SUSAN 17  
6.2 Memory Characteristics 19  
6.2.1 Harris 19  
6.2.2 SUSAN 21  
6.3 Corner Results 22

7 **Discussion** 23

8 **Conclusions** 24

9 **Future Work** 25

References 27
List of Figures

2.1 Common masks ............................................. 6
2.2 High-level design of a multi-core architecture. .................... 8
5.1 Reference image used in our tests. ................................ 12
5.2 Parallelized Harris where each branch denotes a separate thread. 12
5.3 Parallelized SUSAN where each branch denotes a separate thread. 13
5.4 Row-by-row traversing. ...................................... 13
5.5 Loop tiling, traversing by block. ................................ 13
5.6 Kernel overlapping to other blocks. Dashed/Green: Loaded block, Circle/Yellow: Current pixel, Cross/Red: Kernel overlap to other blocks. ................. 14
6.1 Basic single-threaded Harris and OpenCV Harris. .................. 15
6.2 Average Harris response computation time in microseconds on a PowerPC system. 15
6.3 Average Harris response computation time in microseconds on Intel systems. ....... 15
6.4 Naive loop blocked Harris computation in microseconds on a PowerPC system. .... 16
6.5 Naive loop blocked Harris computation in microseconds on Intel systems. ........ 16
6.6 Harris Basic implementation vs Naive tiling vs “No edge” tiling on (a) PowerPC and (b) Intel systems. ........................................ 17
6.7 Average SUSAN response computation time in microseconds on a PowerPC system. 17
6.8 Average SUSAN response computation time in microseconds on Intel systems. ....... 17
6.9 Naive loop blocked SUSAN computation in microseconds on a PowerPC system. ... 18
6.10 Naive loop blocked SUSAN computation in microseconds on Intel systems. .......... 18
6.11 SUSAN Basic implementation vs Naive tiling vs “No edge” tiling on (a) PowerPC and (b) Intel systems. ........................................ 18
6.12 Harris L1 data cache replacements in number of lines replaced. ............. 19
6.13 Harris L1 data cache replacements in number of lines replaced with OpenCV. .... 19
6.14 Harris L2 cache measurements in number of cycles with pending L2 cache miss loads. 19
6.15 Harris L2 cache measurements in number of cycles with pending L2 cache miss loads with OpenCV. ........................................ 20
6.16 Harris TLB measurements in number of store misses causing a page walk. .... 20
6.17 Harris TLB measurements in number of store misses causing a page walk with OpenCV. ........................................ 21
6.18 SUSAN L1 data cache replacements in number of lines replaced. ............. 21
6.19 SUSAN L2 cache measurements in number of cycles with pending L2 cache miss loads. ........................................ 22
6.20 SUSAN TLB measurements in number of store misses causing a page walk. .... 22
6.21 Corner results of our (a) Harris and (b) SUSAN corner detector. ............. 22
1 Introduction

Finding corners in images is an essential part of computer vision and is finding usages in fields such as robotics, navigation, and avionics. Images are often represented as large scale matrices where computations put a lot of stress on the computer hardware, including the cache memory, DRAM memory and system bus. Putting stress on the memory can make it difficult to estimate the worst-case execution time (WCET) of feature detection algorithms. This issue occurs both due to eviction policies within the memory as well as the execution pattern of the algorithm itself. Since many different algorithms are scheduled together on the same system, it is also important to understand how the WCET can change with the cache characteristics on a system. Especially multi-core systems are interesting in the perspective, since they often have common memory such as the L2 cache and the last-level cache, which can cause coherence and false sharing misses. In real-time systems, where timing predictability is essential, we must be aware of the effect which may be caused due to shared memory. Thus, the characteristics which affect the WCET of these algorithms is essential to account for.

Two well-known techniques for detecting corners in an image are SUSAN [1] and Harris [2] corner detector. By investigating the memory characteristics of these algorithms and applying caching techniques such as blocking and loop unrolling, it may be possible to decrease the WCET and furthermore make a good estimation of the average case execution time. We will investigate cache behaviour of these algorithms by measuring the low-level metrics of cache and TLB misses using an Intel architecture with low level metrics obtained from Charmon [3], which is an extension to the Perf tool, and how it affects the execution time, which may help in finding better caching techniques for SUSAN and Harris. We have also made parallel implementations of SUSAN and Harris to study how well these algorithms scale over multiple cores. Improving the cache usage with common techniques has a goal of re-using data, which may prove to increase both the cost-efficiency and timing predictability of a feature detecting algorithm.

2 Background

Corner detection is an integral part of robotic computer vision systems today. It allows for detecting corners in images and to extract certain features. Corner detection has good results on images with heavy contour and can be defined as the end of an edge or the intersection of two edges. Three different methods are used to categorize corner detection [4]: Template based corner detection uses a template to compare the image with. This method can receive good results with a well-fitting template, but may suffer otherwise. Contour based corner detection relies on edge detection to define contours of the image. The resulting edges are then used to detect corners. Direct corner detection uses mathematical computations, usually based on the derivative of an image. SUSAN and Harris corner detector are considered direct corner detectors due to their computational based methods.

In the following subsections we describe the process of implementing a corner detection algorithm. Furthermore, we describe the algorithms focused in this study, namely the SUSAN and Harris corner detection algorithms.

2.1 Kernel Filters

One functionality that many corner detection algorithms have in common is the use of masks (sometimes referred to as a kernel or a filter), such as Sobel or Prewitt. Masks are represented as small matrices and are used to estimate the derivatives of an image [5, pp. 425–427]. The mask $H$ is used on grids $A_i$ of the same size with each pixel $i$ in the grayscale image as the centre of the grid as $D_i = H * A_i$, whereas $*$ is the convolution operation. The result of the operations will be a matrix of the pixels’ estimated derivative. An image can be seen as a function $f(x, y)$, where the image gradient $G$ essentially has two components: an $x$-derivative and an $y$-derivative. This means we must calculate both derivatives to find the change in intensity. These derivatives can then be combined to get the gradient magnitude $M = \sqrt{x^2 + y^2}$. The general Prewitt masks $H_x$ and $H_y$ are defined in figure 2.1 (a) and (b).
The Sobel operator (also referred to as Sobel derivative) is a common alternative mask. It uses two 3 × 3 matrices, as shown in figure 2.1 (c) and (d). These matrices are convoluted and used to calculate the gradients magnitude. They are almost identical to the Prewitt operator matrices, but gives a higher weight to the centre pixel to suppress noise [6]. Noise is unwanted colours in the images, usually as a result of different levels of brightness when the image was taken.

$$\begin{bmatrix}
-1 & 0 & 1 \\
-1 & 0 & 1 \\
-1 & 0 & 1 \\
\end{bmatrix} \quad \begin{bmatrix}
1 & 1 & 1 \\
0 & 0 & 0 \\
-1 & -1 & -1 \\
\end{bmatrix} \quad \begin{bmatrix}
-1 & 0 & 1 \\
-2 & 0 & 2 \\
-1 & 0 & 1 \\
\end{bmatrix} \quad \begin{bmatrix}
1 & 2 & 1 \\
0 & 0 & 0 \\
-1 & -2 & -1 \\
\end{bmatrix}$$

(a) Prewitt horizontal (b) Prewitt vertical (c) Sobel horizontal (d) Sobel vertical

Figure 2.1: Common masks

2.2 Corner Detection

Corner detection algorithms often use the result of a kernel filter to find corners, and may require images to be pre-processed before being processed by the corner detection algorithm. Image pre-processing is used to prepare the data for the algorithm and may include transforming an image to a grayscale image or normalizing the image data.

In this section, we describe the Harris corner detection algorithm which uses the Sobel kernel to get an image gradient, and the SUSAN corner detection algorithm which uses a mask to compare pixel intensities to find corners.

2.2.1 Harris Corner Detection

The basic principle of the Harris corner detection algorithm is defined in 7 steps, as shown in algorithm 1. The first step is to generate the image derivatives with the Sobel operators. The derivatives are then multiplied together to define $I_x^2$, $I_y^2$, and $I_{xy}$, whereas $I$ is defined as pixel intensity. These values are then averaged over a window, often using a Gaussian or a mean filtering method (described as $G$ in the algorithm), to achieve an isotropic response. The filtered derivatives are defined as a matrix $B(x,y)$ for each pixel $(x,y)$ in step 4. This matrix is used in step 5 to calculate the initial response $R(x,y)$ for each pixel $(x,y)$ in step 4. The last two steps apply a threshold on $R$ to eliminate non-important results and use non-maximum suppression to find the largest response value in each local window (the window the mask is applied to).

**Algorithm 1**

Harris corner detection

```plaintext
for all Pixels (x, y) in Image do
  $I_x \leftarrow H_x * A_{xy}$    ▶ Step 1: Derivatives
  $I_y \leftarrow H_y * A_{xy}$
  $I_x^2 \leftarrow I_x * I_x$    ▶ Step 2: Products
  $I_y^2 \leftarrow I_y * I_y$
  $I_{xy} \leftarrow I_x * I_y$   ▶ Step 3: Apply filter
  $S_{x2} \leftarrow G * I_x^2$
  $S_{y2} \leftarrow G * I_y^2$
  $S_{xy} \leftarrow G * I_{xy}$
  $B(x, y) \leftarrow \begin{bmatrix} S_{x2} & S_{xy} \\ S_{xy} & S_{y2} \end{bmatrix}$    ▶ Step 4: Define as matrix
  $R \leftarrow Det(B) - k(Trace(B))^2$    ▶ Step 5: Compute response
  Apply threshold for $R$
  Apply non-maximum suppression over local area
end for
```
Harris and Stephens introduced the algorithm mathematically as finding large eigenvalues within the local window [2]. Moving the window in the direction of an edge will give small changes in intensity, while shifting the window at a corner will give significant change in all directions. Large eigenvalues mean that stretching the window in any direction will result in high distortion, which means it can be considered as a corner.

2.2.2 SUSAN Corner Detection

SUSAN (Smallest Unvalue Segment Assimilating Nucleus) is an edge and corner detection algorithm that has relatively good performance since it does not require image derivatives or noise reduction [1]. A circular mask (typically of 37 pixels) is applied to each pixel as its nucleus (the centre pixel in the mask), where the neighbour pixels’ intensities are compared to the intensity of the nucleus. Equation 1 describes this operation as \( c(⃗{r}, ⃗{r}_0) \), where \( ⃗{r}_0 \) is the nucleus coordinates, \( ⃗{r} \) is a pixel in the mask, and \( I(⃗{r}) \) is the pixel intensity. Constant \( t \) is used as a threshold for how sensitive the feature detection will be and how much noise is ignored. This equation is not used in practice, but is given as a general idea of what is done, while Smith suggests equation 2 as a more efficient solution [1]. In this thesis, we have used the latter version, suggested by Smith.

\[
c(⃗{r}, ⃗{r}_0) = \begin{cases} 
1, & \text{if } |I(⃗{r}) - I(⃗{r}_0)| \leq t \\
0, & \text{otherwise}
\end{cases} 
\]  
(1)

\[
c(⃗{r}, ⃗{r}_0) = e^{-\frac{(I(⃗{r}) - I(⃗{r}_0))^2}{t}} 
\]  
(2)

Pixels which have an intensity within a certain threshold of the nucleus will be defined as a certain area, a “unvalue segment assimilating nucleus” or USAN. The USAN’s area for each nucleus is defined in equation 3 by summing the number of pixels within the threshold in the mask.

\[
n(⃗{r}_0) = \sum_{⃗{r}} c(⃗{r}, ⃗{r}_0) 
\]  
(3)

An initial edge response \( R(⃗{r}_0) \) is given by equation 4. The USAN area is compared to a threshold \( g \), set to \( 3n_{\text{max}}/4 \) where \( n_{\text{max}} \) is the maximum value \( n \) can take.

\[
R(⃗{r}_0) = \begin{cases} 
g - n(⃗{r}_0), & \text{if } n(⃗{r}_0) < g \\
0, & \text{otherwise}
\end{cases} 
\]  
(4)

Finally, non-maximum suppression is used to find corners. This algorithm differs from derivative-based solutions, as we only use the intensity values with respect to the local circular mask.

2.3 Multi-core Architecture and Caches

Modern processors often have multiple processing units (cores) for executing instructions in parallel. The workload of a program can be distributed among different cores to achieve speedup. Cores often have their own private cache memory as well as a shared higher-level cache memory. A shared memory introduces problems of memory contention and coherence, where a lower-level cache require uniformity with the shared memory in a higher-level cache. Since images can be represented as large-scale matrices, they can often quite easily be adapted to multi-core systems.

Cache memory is the memory which is located closest to the CPU. A CPU typically has many cache memories, which are located within different distances from the CPU. The cache memories often follow a hierarchy where the cache located closest to the CPU is called L1 cache, which is also the fastest cache. The L1 cache is then followed by an L2 cache which in many cases is shared between two cores. Finally, there can be a cache which is shared by all cores, called L3 cache or last level cache (LLC). Higher-level caches get larger in size, but get progressively slower. This hierarchy is used to speed up the data access and avoid using the slow DRAM as much as possible. Requesting data that is found in cache memory is called a cache hit. A cache hit saves significant time compared to fetching from the DRAM memory, which also helps easing the memory bus usage in multiprocessor systems with shared caches. Data is typically loaded into the cache in chunks.
blocks) of data due to spatial locality — data stored close to each other are often related and may be requested soon. Requesting data that is not found in cache is called a cache miss. A cache miss will result in data getting fetched from either a higher-level cache or main memory, which can cost hundreds of CPU cycles. Due to spatial locality, following blocks are often pre-fetched to avoid additional cache misses when entering a new block. Optimizing algorithms for spatial locality is an important step in receiving good results, since it will reduce the need to access slow main memory.

Figure 2.2: High-level design of a multi-core architecture.

There is always a need to replace blocks in the cache, which is done according to a replacement policy, often by replacing the least recently used (LRU) block. How the replacement policy maps memory addresses to a specific cache entry varies. If a block can be assigned anywhere in the cache it is called a fully associative cache, while a cache that assigns one specific block for each main memory block is called direct mapped. CPUs often have a combination of the two as an $N$-way set associative cache, where blocks can be assigned anywhere within a set of $N$ entries. The number of slots each block may be assigned to can give significant changes in performance due to blocks not necessarily being replaced by a contending block and the hardware having to check the whole set if it contains a requested cache line. Increasing the number of ways per set has been shown to increase the hit rate [7], up to an 8-way or 12-way set where going any further usually does not improve the hit rate by a considerable amount.

False sharing is a side effect of cores having a separate cache level. If two cores need to work on the same block of data (not necessarily the same data in the block), the cores could load the blocks into their own private cache. Algorithm 2 shows two threads which may be run on separate cores, reading or writing to different data on the same block. The data structure $M$ gets copied to each core's private cache. When thread 2 writes to $M.y$, the cache entry of thread 1 will be invalidated, causing re-read from main memory. This is an unnecessary access to main memory as thread 1 never use $M.y$. The problem remains even if both threads only write to the block as a logical write is a physical read-write, leading to problems if a block in an image matrix spans over two different working cores.

Loading blocks could potentially cause a replacement to occur, leading to cache misses if the replaced block is to be accessed again. With access to certain information about the cache, such as its size, it is possible to create algorithms based on this information. This is called cache-aware algorithms and suffers from the fact that computers often have different cache and block sizes, as well as a different number of cores in multi-core systems. Multicore-oblivious algorithms purpose is to function on a variety of different systems, where the number of cores, cache size, cache levels and block size is unknown. Similarly, a cache-oblivious algorithm is designed to make use of the cache without size or other parameters explicitly given.

2.3.1 Translation Lookaside Buffer

Virtual memory is used as a security measure and to help applications by abstracting the physical memory. Translating a virtual address to a physical address requires reading from a page table that is stored in main memory, which makes the operation slow. To speed up translation between
Algorithm 2
Example of two functions running concurrently, causing false sharing

\[
M \leftarrow \text{Struct \{ int } x; \text{ int } y; \text{ \}} \\
\text{function } \text{Thread1} \\
\quad \text{while } true \text{ do} \\
\quad \quad \text{Read } M.x \\
\quad \text{end while} \\
\text{end function} \\
\text{function } \text{Thread2} \\
\quad \text{while } true \text{ do} \\
\quad \quad M.y += 1 \\
\quad \text{end while} \\
\text{end function}
\]

virtual and physical memory addresses, the memory management unit include on-board caches that store recently translated addresses — translation look-aside buffers, or TLB. Due to spatial locality, storing recent translations will reduce TLB misses as nearby data are often related and subsequently referenced in the near future. A TLB miss will cost additional CPU cycles, which means reducing TLB misses gives a higher performance, especially when prefetching. This will be further discussed in section 2.3.2.

The general design choice for TLBs vary: small caches being fully associative or larger caches with small associativity [8]. A TLB often store a part of the virtual page number (tag) together with the physical frame number and status bits, such as a valid bit if the page is present, a reference bit to track page usage, a valid bit to track valid translations, and a dirty bit which is set when the page is modified. The design choice also affects the replacing scheme and must take into consideration that TLB misses are more common than a page fault (a page currently not mapped in the TLB that may have to be read from disk). TLB misses can be handled by both software and hardware. There are however no large differences in the basic operations being made [8].

2.3.2 Techniques for Improving Cache Utilization

False sharing can be hard to detect in code, but there exist numerous ways to avoid it. There are generally two different categories: writing code that avoids false sharing or by using runtime schedule parameters specified for an algorithm and the CPU running it [9]. In parallel systems, it is common to let different cores loop through specified ranges and not interfere with each other, and will therefore speedup the process of looping through a block. This may however cause severe false sharing, as every core modifies the same block of data. Resolving this can be done by using a local copy for the range the thread has been assigned, and when all threads are done add them together by letting one core write the changes to the block. We can also pad each core’s range so that they all are on unique blocks (by padding with as much extra data as needed), thus disabling false sharing. Again, we need to write the modifications to the block.

Avoiding TLB misses is a tricky subject, which essentially can only be done by changing the memory pattern in an algorithm. Block optimization can help avoid unnecessary cache and TLB misses. Dividing the loop into smaller chunks or blocks that fit in the cache can lead to a substantial improvement.

2.3.3 Prefetching

Prefetching is a technique that fetches a block of data from main memory to the cache memory before it is referenced [8, p. 482]. Predicting that a block of data may be referenced in the near future can improve performance by avoiding stalls due to memory loads. Modern processors often have hardware support for prefetching, where Intel supports hardware prefetching in both the L1 cache and the L2 cache [10, Section 2.3.5.4]. Intel’s hardware prefetcher is triggered by a few conditions, including ascending access to very recently loaded data. Another type of prefetching is software prefetching, where a compiler (as an optimization step or with compiler directives in
the code) inserts instructions that calls for loading cache lines into either the L1 data cache or L2 cache.

2.4 Resource Monitoring

Monitoring resource usage of an algorithm can provide detailed information that can be used in performance analysis as part of tuning the algorithm for improving its performance. Most processors have special on-chip hardware performance counters that store low-level metrics such as cache misses, TLB misses, RAM accesses, and branch misses. These performance counters can be extracted using tools such as Charmon [3].

2.5 Branches and Branch Predictors

Branches and conditional branches are computer instructions that can cause a change in the instruction execution sequence given a condition (e.g. loops and if-then-else structures). Modern processors prefetch instructions for execution before previous instructions are finished to achieve higher performance [8, p. 281]. Instruction prefetching can cause a control hazard due to a taken branch (the branch condition is satisfied), which means that the prefetched instruction is not the one that is needed and may cause a stall before starting over with the branch target.

Branch predictors are circuits that tries to predict whether a branch will be taken to improve the instruction flow. The branch predictor keeps history of whether branch instructions are taken or not taken to base the predictions on.

3 Related Work

Caching techniques have been heavily researched due to their importance in many areas, especially in multi-core systems. Issues like false sharing and coherence have proposals like hierarchical multi-level caching models [11], and balanced parallel models [12]. These studies focus on improving cache utilization for common mathematical applications like matrix multiplication, which is a subproblem of corner detection algorithms that use masks.

Creating more cache friendly code using various coding techniques has been researched. Kowarschik et al. [13] shows how different cache techniques such as blocking and padding can improve execution time in computationally intensive code. Michael et al. [14] investigated blocking and padding techniques by improving the locality of nested loops and making it possible by transforming the code using common cache techniques such as tiling and unrolling. Furthermore, He et al. [15] researched nested for loops when using them for joining data together and the algorithm being cache oblivious by using recursive partitioning, recursive clustering and buffering. He et al. then performs an experimental setup on computers with a single core. This means that parallelizing can be explored further. The study reveals that it is possible to achieve good cache performance by only manipulating the code with various cache techniques. While their focus relies on nested for loops, which is common when manipulating matrices, our purpose is to investigate the memory characteristics of feature detection algorithms in order to achieve a better understanding in how they can be optimized and scheduled together.

Adaptive Harris corner detection algorithms have been investigated where the threshold parameter $k$ is based upon the maximum response of the image [16, 17]. Paul et al. [18] proposed an adaptive resource-aware algorithm based upon parameters like CPU usage. Their results show an effective improvement of false corners, but also execution time in the resource-aware adaptation of Harris corner detector.

In this work, we explore how to find better cache behaviour for corner detection algorithms to achieve a more reliable execution time. Previous work focused on general cache behaviour, how to improve the accuracy of the algorithms, or how to improve the algorithms using cache-aware techniques. Cache-oblivious algorithms have the advantage of no run-time overhead of monitoring system resources like CPU usage, and no need for additional parameters of the target system. Using a cache-oblivious model is a good first step in optimizing an algorithm, where further fine-tuning using a cache-aware model can be used to reach a nearly optimal algorithm on specific target systems.
4 Problem Formulation

The objective of this thesis is to investigate the cache behaviour of the SUSAN and Harris corner detection algorithms. The intention is to explore different caching techniques to, if possible, achieve a more predictable execution time for these algorithms. Furthermore, this thesis aims to investigate how multi-threaded implementations affects the execution time of the corner detection algorithms using the different caching techniques.

4.1 Research Questions

The following questions will be researched:

- How does the cache and the TLB misses affect the predictability of Harris and SUSAN?
- How can the predictability of Harris and SUSAN corner detection be improved using caching techniques and multi-threaded implementations?
- Since there are very well established Harris and SUSAN corner detection algorithms available, will they suffer from these behaviours and how well will they perform against a cache-aware implemented version of SUSAN and Harris?

4.2 Motivation

Achieving a more predictable execution time could potentially be beneficial for use in systems with hard execution time constraints, such as those used in robotics navigation and avionics. Since multi-core architectures allows for algorithms to be executed in parallel, understanding the parallel capabilities of the algorithms can be used to expand these algorithms for use in multi-core systems.

4.3 Outcomes

The main goals of this thesis follow:

- Implement SUSAN and Harris using caching techniques
- Investigate parallel capabilities of the implementations
- Investigate memory characteristics of the implementations
- Compare our Harris implementations with the OpenCV Harris implementation

5 Method

Our work follows an experimental investigation of the two corner detection algorithms, Harris and SUSAN. Section 5.1 describes our implementations of Harris and SUSAN made in C++, whereas the test setup can be seen in section 5.2. We will also describe how we evaluate our implementations and compare it to other implementations that are non-parallel or use libraries such as OpenCV. Based on the results of this investigation, we aim to improve the cache and TLB behaviour of the algorithms using caching techniques such as loop tiling.

5.1 Implementation

All our implementations follow the same initial steps of loading an image from file and converting it to grayscale. The image data is loaded from a bitmap (BMP) image file to a one-dimensional array, where figure 5.1 shows our 1280 x 1024 pixel test image. Converting to a grayscale image follows a luminance-preserving approach of matching the luminance of the original image in the resulting grayscale image. This is achieved using a weighted linear combination of the RGB values: \( Y = 0.2126R + 0.7152G + 0.0722B \). Parallelization is achieved using pthreads (POSIX threads) by dividing the image into \( N \) blocks for parallelization over \( N \) cores. Section 5.1.1 and 5.1.2 will further show the specific implementation of the algorithms. Improvement suggestions are presented in section 5.1.3.
5.1.1 Harris Corner Detection

Since the Harris algorithm is divided into 7 steps (where Gaussian filtering essentially is optional, as it is used to smooth out noise) we have considered which steps that could benefit from parallelization, whereas our measurements will focus on the Sobel operator and response calculations. Most steps only require access to nearby pixels as we apply a mask, which means dividing the image into blocks to avoid contention is relatively simple. We are not using any padding, which means we will lose one row of pixels at the image border to avoid out-of-bounds accesses due to the $3 \times 3$ mask, which is not too large of a loss of potential corners. Step 1–6 are distinct from the rest of the blocks assigned for parallelization if we let the row of pixels between blocks have a derivative of 0, while the last step of non-maximum suppression might require a synchronization step before proceeding, depending on the radius of suppression. Figure 5.2 describe our fork-join model of the Harris algorithm.

The basic implementation is straightforward with no explicit regard for the cache. Traversing the image with a $3 \times 3$ mask require loading data from three different rows into the cache (above, below, and same row as the centre pixel being processed). Improved algorithms will try to optimize the cache usage by re-using the loaded data more efficiently.

5.1.2 SUSAN Corner Detection

Our implementation of SUSAN similarly to Smith [19] uses a lookup table (LUT) to speed up brightness comparisons around each pixel. Parallelization requires only reading of nearby pixels,
which can be done without concern when reading over other blocks, but with the requirement of a synchronization barrier before starting non-maximum suppression. The circular mask of 37 pixels will result in 3-pixel loss around the image border. Figure 5.3 describe our fork-join model of the SUSAN algorithm.

5.1.3 Tiling Techniques

The first optimization technique is a naive loop tiling method. Instead of traversing the array row-by-row as in figure 5.4, the image is divided into smaller blocks that fit into the cache, as seen in figure 5.5. A problem with the naive loop tiling is that our kernel relies on reading an area around each pixel. If we load a block into the cache and proceed to process a pixel at the edge of the block, we need to load a line from the block where the kernel overlaps. Figure 5.6 depicts this scenario, where the yellow pixel is the current pixel, green pixels belong to a loaded block and red pixels are located in other blocks that the kernel overlaps.

Solutions for this problem can be explored in several ways. One way that we will be investigating is to skip calculations for edge pixels, which in turn results in the loss of eventual corner pixels. Depending on the block size the number of pixels lost may be too large. Prefetching will also play a role in contending for the cache, where prefetching could replace lines that are not yet processed. Intel’s Sandy Bridge architecture prefetches one line at a time for the L1 cache following a few conditions, where an ascending access to very recently loaded data is one of them [10, Section 2.3.5.4]. An optimal algorithm would have to take this into consideration by for example
not creating a tight-fitting block or in other ways avoid early replacements of not fully processed lines. Our tiling method will not accommodate for prefetching, but will explore options as skipping calculations for edge pixels. Optimizing the blocking algorithm to accommodate for prefetching will be left as future work as we focus our investigation on the naive block algorithm and the block algorithm which ignores block edges.

5.2 Test Setup

The algorithms are tested on four different Intel processors with private L1/L2 caches and a shared L3 cache. We have also executed tests using an 8-core PowerPC QorIQ P4080ds to measure the execution speedup on more cores. The P4080 also runs on a minimalistic kernel (NXP fsl-core 4.1.30-rt34+g4004071), which results in less interference from the kernel. The i5-2500k and i5-3570 processors were run on Ubuntu 16.04 with a Linux 4.4 kernel, while the i5-4210U and i5-3317U processors were run on Debian 9.0 with a Linux 4.9 kernel.

<table>
<thead>
<tr>
<th>Model</th>
<th>Freq.</th>
<th>Cores/Threads</th>
<th>L1</th>
<th>L2</th>
<th>Shared L3</th>
</tr>
</thead>
<tbody>
<tr>
<td>i5-2500k</td>
<td>3.3 GHz</td>
<td>4/4</td>
<td>32 KB 8-way</td>
<td>256 KB 8-way</td>
<td>6 MB 12-way</td>
</tr>
<tr>
<td>i5-3570</td>
<td>3.8 GHz</td>
<td>4/4</td>
<td>32 KB 8-way</td>
<td>256 KB 8-way</td>
<td>6 MB 12-way</td>
</tr>
<tr>
<td>i5-4210U</td>
<td>2.7 GHz</td>
<td>2/4</td>
<td>32 KB 8-way</td>
<td>256 KB 8-way</td>
<td>3 MB 12-way</td>
</tr>
<tr>
<td>i5-3317U</td>
<td>2.6 GHz</td>
<td>2/4</td>
<td>32 KB 8-way</td>
<td>256 KB 8-way</td>
<td>3 MB 12-way</td>
</tr>
<tr>
<td>e500mc</td>
<td>1.5 GHz</td>
<td>8/8</td>
<td>32 KB 8-way</td>
<td>128 KB 8-way*</td>
<td>2 × 1 MB 32-way</td>
</tr>
</tbody>
</table>

Table 5.1: Processor specifications used in the tests. L1 and L2 are core-private caches and L3 is a shared cache, with specified set associativity.(*) The e500mc processor has a L2 cache shared between two cores.

The GNU Compiler Collection (GCC) was used to compile our implementations. GCC has compiler flags which can be used to optimize the code, which may result in reduced code size and execution time. All tests were compiled using the -O3 flag, which turns on various optimization flags [20].

The three different methods (the basic linear traversing, the naive tiling, and the “no edge” tiling) were run 100–1000 times for each core on each test system, for both SUSAN and Harris. The block size for the tiling techniques was set to $256 \times 128$, which is the L1 cache size in our test systems. The OpenCV Harris corner detection algorithm lacks multi-thread support and was therefore run on a single thread.

6 Results

This section presents the results of our implementations of Harris and SUSAN. The execution times and speedup ratio will be compared for the basic algorithm, the naive tiling, and the alternative “no edge” tiling method. The cache behaviour is presented for the three different methods, and OpenCV in the case of Harris.
Furthermore, figure 6.1 shows a comparison between our basic linear Harris to the OpenCV Harris function.

![Figure 6.1: Basic single-threaded Harris and OpenCV Harris.](image)

OpenCV shows a 300 % increase in execution time compared to the basic linear Harris. Note that OpenCV also allocates memory for the destination matrix and include pre-processing, hence the large difference.

6.1 Parallel Capabilities

Each measurement of parallel capabilities will be presented as an average from 1000 tests. A highly desirable execution time for a parallel implementation is to execute at a linear speedup when using additional cores. A linear speedup is defined as a speedup of $P$ when running an algorithm on $P$ cores, compared to a single-core implementation.

In section 6.1.1, the results from our parallel Harris implementation is described, whereas section 6.1.2 describes the results from our parallel SUSAN implementation.

6.1.1 Harris

The measurements of the Harris corner detector using the P4080ds machine is depicted in figure 6.2 and using the Intel machines in figure 6.3.

![Figure 6.2: Average Harris response computation time in microseconds on a PowerPC system.](image)

Harris executed at close to linear speedup using an Intel i5-2500k processor, achieving a 3.7 times speedup by dividing the work over 4 cores, 2.8 times on 3 cores, and 1.9 times on 2 cores. Executing on processors with two physical cores and two virtual cores (Intel i5-4210U and i5-3317U), the speedup landed at an average of 1.85 times for 2 and 4 threads. This shows that our implementation of Harris does not scale with virtual cores. A fully linear speedup may have been hindered by kernel interrupts, whereas the P4080ds machine was used to validate the speedup on
a system with minimal kernel interference, and revealed that an execution time very close to linear speedup was possible. Figure 6.4 and 6.5 shows how the naive Harris loop tiling algorithm, where the mask may cause reads outside of the block, has similar scaling as the basic linear implementation. The naive tiling shows a 25 % increase in execution time, a slowdown compared to the basic implementation. This increase could be due to the mask causing reads outside of the block, causing cache contention.

Our last test compares the execution time result between the two tiling techniques and the basic implementation. The measurements taken when using an e500mc processor are shown in figure 6.6(a) and the measurements using an Intel i5-3570k are shown in figure 6.6(b). The “no edge” tiling technique shows a 4–5 % improvement in execution time compared to the naive tiling technique using both an i5-2500k and e500mc processor. Compared to the basic
implementation, the tiling techniques introduce a 15–20% increase in execution time on both processors.

6.1.2 SUSAN

Figure 6.8 depicts the basic SUSAN implementation’s parallel capabilities when running on Intel processors and figure 6.7 shows its parallel capabilities running a P4080 machine.

The basic linear SUSAN implementation achieved almost linear speedup when executing on a P4080 machine with a minimalistic kernel. Close to linear speedups were achieved on an Intel i5-2500k processor at 3.8, 2.9, and 1.9 speedup executing on 4, 3, and 2 cores, respectively. Similarly to the Harris results, virtual cores did not achieve any additional speedup, which can be seen on our i5-4210U and i5-3317U processor results.
The naive tiling algorithm’s average execution times are depicted in figure 6.10 running on our Intel processors and figure 6.9 when running on an e500mc processor. SUSAN’s naive tiling show similar parallel capabilities as the basic tiling method, reaching close to linear speedup on all processors.

![Figure 6.9: Naive loop blocked SUSAN computation in microseconds on a PowerPC system.](image)

![Figure 6.10: Naive loop blocked SUSAN computation in microseconds on Intel systems.](image)

The last test compares the execution time results of all SUSAN implementations (the two tiling techniques and the basic implementation). Figure 6.11(b) shows the test results from when running on an i5-3570k, and figure 6.11(a) depicts the execution times on a P4080ds machine.

![Figure 6.11: SUSAN Basic implementation vs Naive tiling vs “No edge” tiling on (a) PowerPC and (b) Intel systems.](image)

The “no edge” tiling shows a 2 % improvement in execution time compared to the basic SUSAN implementation running on an e500mc processor, while our Intel processor shows an equal execution time between the “no edge” tiling method and the basic implementation. The naive tiling show a 5–10 % increase in execution time on all systems.
6.2 Memory Characteristics

This section presents low-level metric measurements for our single-threaded implementations of Harris and SUSAN corner detection. The metrics are sampled at a 100 Hz frequency over 1000 test runs, and contain the following event measurements: L1D Replacements (Level 1 data cache lines replaced), L1 Instr. Misses (Level 1 instruction cache misses), L2 Pending (cycles with pending L2 cache miss loads), and DTLB Walks (TLB store misses causing a page walk).

6.2.1 Harris

Table 6.1 presents the averaged low-level metric results of our Harris corner detection implementations.

<table>
<thead>
<tr>
<th>Method</th>
<th>L1D Replacements</th>
<th>L1 Instr. Misses</th>
<th>L2 Pending</th>
<th>DTLB Walks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic</td>
<td>530919</td>
<td>5911</td>
<td>1170129</td>
<td>1046</td>
</tr>
<tr>
<td>Naive</td>
<td>563747</td>
<td>4818</td>
<td>1006505</td>
<td>784</td>
</tr>
<tr>
<td>Improved</td>
<td>570470</td>
<td>5770</td>
<td>1531753</td>
<td>810</td>
</tr>
<tr>
<td>OpenCV</td>
<td>4682621</td>
<td>5421</td>
<td>19899319</td>
<td>2555</td>
</tr>
</tbody>
</table>

Table 6.1: Average low-level metric results for Harris corner detector.

The L1 data cache lines replaced seem to affect the execution time the most, as the basic implementation has both more L1 instruction cache misses and more TLB misses than the tiling techniques, but has a faster average execution time. Figure 6.12 shows a stable L1 cache behaviour throughout the measurements. OpenCV is very fluctuating and has 8–9 times more L1 data cache lines replaced than the other implementations, as seen in figure 6.13.

![Figure 6.12: Harris L1 data cache replacements in number of lines replaced.](image)

OpenCV’s large number of cache misses may be because of the difference in amount of pre-processing and memory allocation that is captured by our measurements, which makes it difficult to compare the results. Nevertheless, figure 6.14 and 6.15 depicts our L2 cache measurements.
The naive tiling algorithm had slightly better L2 cache results than the rest, also showing a more stable activity. OpenCV yet again shows fluctuation and a larger number of L2 misses compared to the other algorithms. Figure 6.17 depicts the same patterns with a larger amount of TLB page walks required. Furthermore, figure 6.16 depicts the TLB behaviour of our Intel system.

The naive blocking is very steady in the TLB, whereas the basic Harris algorithm fluctuates with a larger average than both the tiling algorithms. Comparing TLB misses to the execution time results, TLB misses does not seem to be a leading factor in causing a decreased performance for our algorithms.
Figure 6.17: Harris TLB measurements in number of store misses causing a page walk with OpenCV.

### 6.2.2 SUSAN

Table 6.2 presents the averaged low-level metrics results for SUSAN.

<table>
<thead>
<tr>
<th></th>
<th>L1D Replacements</th>
<th>L1 Instr. Misses</th>
<th>L2 Pending</th>
<th>DTLB Walks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic</td>
<td>95399</td>
<td>2382</td>
<td>462274</td>
<td>200</td>
</tr>
<tr>
<td>Naive</td>
<td>117298</td>
<td>1740</td>
<td>932792</td>
<td>732</td>
</tr>
<tr>
<td>Improved</td>
<td>115967</td>
<td>1859</td>
<td>645516</td>
<td>176</td>
</tr>
</tbody>
</table>

Table 6.2: Average low-level metric results for SUSAN corner detector.

Execution times of the basic implementation indicate that L1 instruction cache misses have low impact on execution time, whereas L1 data cache replacements and the L2 cache may affect the execution times more. Figure 6.18 depicts how the basic implementation had less L1 data replacements than the loop tiled versions. Furthermore, figure 6.19 shows a larger interval where the basic algorithm had occasional spikes of pending L2 cache miss loads compared to the tiling techniques, but still achieved a lower average.

Figure 6.18: SUSAN L1 data cache replacements in number of lines replaced.

All our SUSAN implementations show fluctuation in how many TLB misses that cause page walks, as seen in figure 6.20. The naive tiled algorithm caused a relatively large number of misses compared to the other two algorithms, reaching 200–300 % more TLB misses on average.
Figure 6.19: SUSAN L2 cache measurements in number of cycles with pending L2 cache miss loads.

Figure 6.20: SUSAN TLB measurements in number of store misses causing a page walk.

6.3 Corner Results

Figure 6.21 present the SUSAN and Harris corner results. SUSAN was run with a geometric threshold $g$ of 1850, and Harris used a response threshold of 20000.

Comparing the results on our test image that was used for the previous tests is challenging because of the contrast. At a glance, the corners detected by Harris show a more consistent appearance along the few noticeable edges in the image, while SUSAN has a larger amount of corners detected within a certain area of the gravel road in the image.
7 Discussion

Our Harris and SUSAN implementations reached a 3.7 times speedup on 4 cores using an Intel i5-2500k processor — close to a linear speedup. One reason for the sublinear result may be due to kernel interrupts. The P4080ds machine has less interference due to a minimalistic kernel, achieving almost linear speedup at the cost of a slower execution time overall due to a slower processor. The fact is that three methods (the basic linear, the naive tiling, and the tiling without edge calculations) for both SUSAN and Harris scale well with threading, but the tiled results falling behind the basic linear traversing in execution time was a bit unexpected. Two possible reasons for these results may be increased control flow complexity due to nested loops causing branch misses, and linear traversing having straightforward prefetching conditions. Another possible reason for this behaviour may be the compile flags used for the tests. In the tests, we used the -O3 optimization flag which includes multiple optimization techniques. Whether this had an effect on the relative performance between the algorithms is unclear. Re-running the tests without -O3 may change this behaviour.

Excluding OpenCV, the L1 data replacements were very stable throughout the tests. The better execution time and less L1 replacements for the basic implementations were most likely because the L1 cache could fit all lines for the mask in the L1 cache, even as we traversed the whole row. Traversing the array-based matrix by width in an increasing order can make use of prefetching well and two of the loaded rows may be re-used for the next row of pixels. This resulted in the basic algorithm being able to execute faster than both of our implemented tiling methods in both Harris and SUSAN.

The L2 cache results revealed that the naive tiling performed best for Harris but worst for SUSAN. One reason for this behaviour may be the larger mask causing irregularities in the memory. This can also be seen in the basic implementation for SUSAN, whereas the L2 measurements occasionally had periods of large fluctuation. However, the basic implementation generated a better execution even though the L2 behaviour.

The tiling methods again showed promising results in the metric “TLB store misses caused page walks”. Basic Harris executed with very fluctuating TLB results compared to the tiling methods. However, because of the difference in execution times, the results may indicate that the TLB misses were not as crucial as the L1 replacement behaviour. SUSAN showed similar results where the difference in TLB behaviour did not show differences in the execution times.

The OpenCV comparison should not be considered as a proof that our implementation is better than the OpenCV implementation. There is a difference in operations, where OpenCV for example allocates memory for the destination matrix and include pre-processing. The large cache fluctuations were most likely due to image border generation and memory allocation, which makes estimating the worst-case execution time for OpenCV hard.
8 Conclusions

In this study, we have conducted a comparison between the SUSAN and Harris corner detection algorithms. We implemented three different versions of both SUSAN and Harris: a basic linear traversing algorithm, a naive loop tiling algorithm, and a loop tiling algorithm that skips calculations for the block edges. By extracting hardware performance counters from the system in use, we investigated memory characteristics for these algorithms using multi-core architectures. We list three research questions that we aimed to answer throughout our thesis project.

The first research question was how the cache and the TLB misses affect the predictability of SUSAN and Harris. The Harris results showed a larger number of L1 instruction cache misses and TLB misses of the basic linear implementation than the corresponding misses of the loop tiled implementations. With 10% less L1 data cache lines replaced and a 15–20% faster execution time for the basic version, we can conclude that the L1 data replacements affect the execution time the most. SUSAN showed similar results together with a decrease in L2 cache misses, insisting that the larger mask also relies heavily on the L1 cache and the L2 cache.

Our second research question was on how the predictability of Harris and SUSAN corner detection can be improved using caching techniques and multi-threaded implementations. The results revealed that our attempts of improving Harris and SUSAN using caching techniques were outperformed by the basic linear algorithm. Traversing by blocks the size of the L1 cache showed less fluctuation in both the L1 cache, the L2 cache, and the TLB. However, the cost of this was an increase in execution time. From these results, we can conclude that using techniques which tries to optimize a Harris and a SUSAN corner detector according to the cache may be unfeasible. One reason may be that the required rows of data for each row of calculations were small enough to fit in the cache on our $1280 \times 1024$ pixel test image. Further traversing to a new row could simply start replacing blocks from the previous top row of pixels, which had already been used as part of their required calculations. Loop tiling require an increased number of comparisons due to the nested loops and bounds checks, which increase the control flow complexity. The increased complexity may have played a role in the increased execution time for the loop tiled algorithms. Linear iteration in an ascending order also allows efficient usage of hardware prefetching, leading to more efficient cache usage.

Parallelization of the algorithms showed that linear speedup is possible. Our Intel systems showed a sub-linear speedup, whereas our P4080ds machine with a minimalistic kernel to reduce kernel interference showed 7.9–8 times speedup on 8 cores. Comparing the execution times of Harris show a 4–5 times improvement over SUSAN. One reason for this disparity is the difference in kernel size. The $3 \times 3$ Sobel kernel require a total of 12 memory reads (6 for each derivative) whereas the circular mask of SUSAN require 37 memory reads for each pixel — 3 times as many memory reads. The memory bottleneck is not as extensive on the P4080ds machine, where Harris is 30% faster than SUSAN for 1–8 cores.

Our last question was on how the computer vision library OpenCV’s Harris corner detection function perform against a cache-aware corner detection algorithm. OpenCV shows a very large number of cache misses and a large fluctuation. Comparing the OpenCV Harris with our implementations is hard due to the difference in implementation, where OpenCV for example includes more pre-processing. The cache and TLB behaviour was however interesting due to the large difference in average misses, but is likely due to the increased amount of work done by the OpenCV Harris corner detector.
9 Future Work

Future work include vectorization of Harris corner detection using SIMD (single instruction multiple data) instructions. Processing an array of bytes, as in our implementation, allows for loading and operating on 14 pixels at a time using 128-bit SSE or AltiVec registers (128 / 8 bits per pixel, minus two for the edges of the $3 \times 3$ kernel). This implementation would possibly further improve the execution time of Harris due to the extra computational units. However, this may be unfeasible when using SUSAN due to the large mask.

Further improving the loop tiling algorithms may be investigated in terms of accommodating for prefetching. Optimizing memory accesses for hardware prefetching may be difficult, but software prefetching could be investigated using Streaming SIMD Extension (SSE) prefetch instructions. The results of our work show that linear iteration provides a better cache behaviour in average, but that the blocking algorithms show a more stable cache behaviour. An improvement using such techniques could possibly provide more predictable worst-case execution times.

The focus in this report relies on investigating the cache behaviour of SUSAN and Harris and improving the algorithms with respect to this behaviour. Techniques which may be used for further improvement may be static and dynamic page colouring of caches. These techniques partition the caches so that one core may not evict the other core’s cache lines, which may be adapted to the shared memory usage when executing SUSAN and Harris on multiple cores.
References


