Hochschule
Augsburg University of
Applied Sciences

Fakultät für
Informatik

Bachelor-Thesis

Studiengang
Technische Informatik

Optimized Implementation of a Feature Detector for Embedded Systems Based on the 'Accelerated Segment Test'

Verfasser:
Peter Fink
Am Kappengrund 24
86946 Issing
+49 8194 713
PeterFink@t-online.de
Matrikelnr.: 924547

Prüfer: Prof. Dr. Gundolf Kiefer

Hochschule für angewandte
Wissenschaften Augsburg
An der Hochschule 1
86161 Augsburg
Telefon: +49 (0)821-5586-0
Fax: +49 (0)821-5586-3222
info@hs-augsburg.de

Hochschule
Augsburg University of
Applied Sciences

Fakultät für
Informatik

Bachelor-Thesis

Studiengang
Technische Informatik

Optimized Implementation of a Feature Detector for Embedded Systems Based on the 'Accelerated Segment Test'

Verfasser:
Peter Fink
Am Kappengrund 24
86946 Issing
+49 8194 713
PeterFink@t-online.de
Matrikelnr.: 924547

Prüfer: Prof. Dr. Gundolf Kiefer
Betreuer 1: Philipp Lutz, M.Eng.
Betreuer 2: Dr. Elmar Mair
Datum: 17.04.2014

Hochschule für angewandte
Wissenschaften Augsburg
An der Hochschule 1
86161 Augsburg
Telefon: +49 (0)821-5586-0
Fax: +49 (0)821-5586-3222
info@hs-augsburg.de

Abstract

Recent developments made portable embedded systems cheaper, even smaller, and also enormously increased their computing power. But performance in a system itself cannot only be gained by using faster and better hardware or simply utilizing multiple general purpose processor cores, it can also be enhanced by adaption for special auxiliary hardware and optimization of time-critical software parts. Especially when large amounts of data have to be processed, as in image processing algorithms, the benefits of parallel data processing can exceed the additional optimization effort. This thesis will show how to take advantage of the additional processing power of the *TI DM3730* processor in the use case of a feature detector based on the *Accelerated Segment Test*. Two ways of unburdening the *CPU* by using either *SIMD* extensions on the *CPU* itself or by transferring the task to the on-chip *DSP* were implemented and validated. The evaluation of several ideas and possibilities for optimizations led to the final implementations that reduced processing times by more than 25% on both units.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

1.1 Motivation

Computer vision is already used throughout many different applications in industry, medicine, automotive, retail, surveillance and authentication [25]. For example in object detection [12, 15], tracking [33], or localization and mapping [13] it is often important to "[d]etect [...] and match [...] specific features across different images" [16]. From these three steps, which are *detection*, *description* and *matching* [16], *detection*, as the first step, is very important, because the following steps are based on the information gathered in this step. One kind of these feature detectors are so called corner detectors, which respond to corners in images.

Recent research has helped to increase efficiency, robustness and repeatability of corner detectors in general, e.g. in [22]. But not every system can provide that much computing power, especially when it comes to lightweight embedded systems like small drones, multicopters or small planes where weight and available energy is limited. They are often designed to be autonomous and so necessary tasks, e.g. to prevent damage to the system itself or persons in reach, have to be done on-the-fly and sometimes even with real-time requirements. Therefore, it is crucial to design these systems as a whole in a way, that every smallest part is efficient and uses available resources with care.

Furthermore, hardware performance improvements on low energy platforms (mostly *ARM* based) (*Advanced RISC Machines*) (*Reduced Instruction Set Computing*) help to take a step towards these aims and offer the possibility to let more complex tasks be handled on such devices. Typically, highly integrated *SoCs* (systems on chips) with numerous and specialized on-chip hardware exceeding a single core are predestined for weight-critical applications. But software needs to be adapted to special hardware at the expense of development time and costs when it is necessary to increase overall performance at the same energy consumption level, or simply to increase efficiency.

At the *German Aerospace Center* (*DLR*) the target platform on a multicopter for a feature detector is a *SoC* including a processor with vector extensions and also an on-chip *DSP* (digital signal processor), which was not used at the time of writing.

1.2 Goals

As there are more tasks on a flying drone than to detect corners in images captured from on-board cameras, the detection should require as little time as possible, no matter where or how the calculations are done. A popular, fast and reliable corner detector is Edward Rosten's FAST (Features from Accelerated Segment Test) [21, 20], which is capable of corner detection at video rate on desktop computers [14]. For this detector a reference implementation and a vectorized version using Intel's Streaming *SIMD* (single instruction multiple data) Extensions 2 (*SSE2*) are available. The first goal is the transferring and further optimizing of the vectorized version for the internal multimedia vector extension, which uses the comparable *ARM NEON* instruction set in order to profit from the parallel execution of some parts of the algorithm.

The second aim is to outsource the corner detection. Therefore the optimized code will be ported to the on-chip *DSP* that implements a different instruction set, and the base frame will be adapted to work with an available *API* (application programming interface) for the *DSP*.

Finally, a comparison of the achieved results, a detailed analysis of calculation times on the different processing units, as well as a comparison of power consumptions and a conclusion with an outlook on future work will be presented.

2 Project Environment

This chapter will present a description of the target platform with all important components, as well as a summary of the *FAST* algorithm and a description of a sample application at the *DLR*.

On small multicopters every gram of weight is crucial, but on highly automated systems computational power is important to fulfill necessary tasks on the fly. Some multicopters at the *DLR* are equipped with *Gumstix Overo® FireSTORM* modules weighing only 5.6 g, which contain a *DaVinci DM3730 SoC* from *Texas Instruments* (*TI*) as the centerpiece to accomplish the arising tasks [9]. These boards are tiny and have to be mounted on extension platforms to allow easy development and so an even cheaper solution was found in the *Beagleboard-xM*, which implements the same main chip.

2.1 Beagleboard-xM

The *Beagleboard-xM* features the *TI DaVinci DM3730*, containing a superscalar *ARM Cortex-A8* core, which is capable of running up to 1 GHz. The *ARM* core is connected to the on-board 512 MB low power mobile-*DDR RAM* (double data rate random access memory) via two cache levels of 4-way 32 KB Level-1 (L1) for data and instructions each and 256 KB 8 ways associative L2 cache [30].

Besides some specialized hardware as a *display interface* and a *camera image signal processor* the *SoC* features an on-chip *TMS320C64x+ DSP* and a *PowerVR SGX530 GPU* (graphics processiung unit) [30]. The graphics unit will not be included in further discussions of this work because it is not that powerful when executing algorithms that are highly memory dependent. In [18] is was shown that data transfers, especially from the *GPU* back to main memory, are very slow.

The *DSP* is embedded in the *IVA2.2* (TI video and audio accelerator) mega-module including local caches, a dedicated memory management unit (*MMU*), a video hardware acceleration module and some more [30]. It uses 32 KB direct mapped L1 cache for instructions and 80 KB 2-way associative L1 data cache expanded by 64 KB unified 4-way L2 cache [28]. The *C64x+* is a fixed-point *DSP* based on the *TMS320C6000 CPU* (central processing unit) using the *VelociTI™* architecture [29].

The *Beagleboard* additionally features four *USB* (universal serial bus) ports, *Ethernet*, *RS232*, audio in/out, S-Video and *DVI* (digital visual interface) interfaces. It comes with headers for easy connection of *LCDs* (liquid crystal displays) and camera modules.

On the *Beagleboard*, *Ubuntu precise 12.04.1 LTS* is running as an operating system (*OS*) with a 3.0.0 kernel including the realtime preemption patch.

2.1.1 NEON Co-Processor

ARMv7 architecture introduced an optional advanced *SIMD* extension to the *ARMv7-A* and *ARMv7-R* profiles called *NEON*. This add-on extends the already existing small set of *SIMD* instructions of the *ARMv6* architecture by defining groups of instructions operating on vectors of 64 and 128-bit vector registers. The *NEON* instructions support 8-bit, 16-bit, 32-bit, 64-bit signed and unsigned integers, as well as 32-bit single-precision floating point elements and 8-bit and 16-bit polynomials. The register bank, added for the extension, consists of 32 64-bit doubleword registers, which can also be used as 16 128-bit quadword registers. The *NEON* instructions can be utilized directly by writing assembly code or by using intrinsics in C/C++ with compilers that support them. Intrinsics look like function calls, but are replaced by (a sequence of) low-level instructions at compilation, providing efficient usage of *NEON* instructions in high-level languages. (Cf. [3].)

"The *NEON* unit is decoupled from the main *ARM* integer pipeline by the *NEON* instruction queue (*NIQ*). The *ARM* Instruction Execute Unit can issue up to two valid instructions to the *NEON* unit each clock cycle. *NEON* has 128-bit wide load and store paths to the Level-1 and Level-2 cache, and supports streaming from both. The *NEON* media engine has its own 10 stage pipeline that begins at the end *ARM* integer pipeline. [...][It] has three *SIMD* integer pipelines, a load-store/permute pipeline, two *SIMD* single-precision floating-point pipelines, and a non-pipelined Vector Floating-Point unit (VFPLite)" [1].

The implemented instructions can be grouped in logical and compare operations, general data processing and arithmetic instructions, as well as shift, multiply and load/store instructions. Generally speaking, the *NEON* instruction set is very similar to other multimedia extensions like Intel's Streaming *SIMD* Extension (*SSE*) in version two and three, which could be seen as a minimal common ground for scientific and media applications today within the large variety of available vector extensions in the desktop (x86) environment [10]. For a complete reference see [2].

2.1.2 TMS320C64x+ DSP

The *C64x+ DSP* has its own clock and runs at a maximum frequency of 800 MHz, implements an eleven stage pipeline and is separated into two data paths. Each path contains a general-purpose register file with 32 32-bit registers and four functional units. The "register files support data ranging in size from packed [...][8-bit] data [(vectors)] through 40-bit fixed-point and 64-bit [fixed-/]floating-point data. Values larger than 32 bits, such as 40-bit long and 64-bit [...][non-integers] are stored in register pairs" [26].

The total of eight functional units (.L1, .S1, .M1, .D1, .L2, .S2, .M2, .D2) are shared equally among the data paths, from which two are multipliers and six are arithmetic units with slightly different feature characteristics and supported instructions. So in a best-case scenario eight instructions can execute in parallel. If an instruction can only be performed on one specific unit at a certain cycle, it is also possible to use data from the other path's register file via two data cross paths. Additionally, each path has its own load and store path to memory, as seen in Figure 2.1.

Compared to the *NEON* extension, the *DSP* approach to parallel (multimedia) data processing is a bit different. The *DSP* features more functional units, but it keeps the 32-bit register width. With multiple units executing the same instruction in parallel it may result in the same processing speed, but dependent on the implementation it may also result in slower or faster execution. For example, logical operations on 32-bit data can be executed on two functional units on each path, which can result in four instructions being executed at the same time, which is exactly the same as one operation on a 128-bit vector (cf. [27]). Many algorithms can profit from instruction parallelism as it can be used here, especially when they cannot be easily parallelized or not at all. For implementations that are capable of vectorized data processing it is also often possible to reach the same performance as with larger data vectors, as in the *NEON* architecture.

Although the *DSP* has its own peripherals, it has no direct means of communication to the host *CPU* except the main memory. Therefore, control mechanisms are needed in order to be able to move certain tasks to the *DSP* and unburden the host processor. A popular *DSP-OS*, which manages tasks on the *DSP*, is the *DSP/BIOS operating system* from *TI*, which is loaded by the host *CPU* onto the *DSP*. Once this is accomplished, so called *nodes*, which implement the user applications, can be loaded dynamically into the *DSP/BIOS*. Although there are other possibilities to control the *DSP/BIOS* from the host side, here *DSP-Bridge*, also from *TI*, is used together with a lightweight *DSP API* from [18]. This interface includes host-side bridge logic and easy access to common tasks that are necessary when working with an co-processor. The *DSP API* and its sublayers implement a mailboxing-system between the two processors through the main memory. Through these messages *nodes* can be launched, parameters can be passed or status messages can be exchanged. (Cf. [31].)

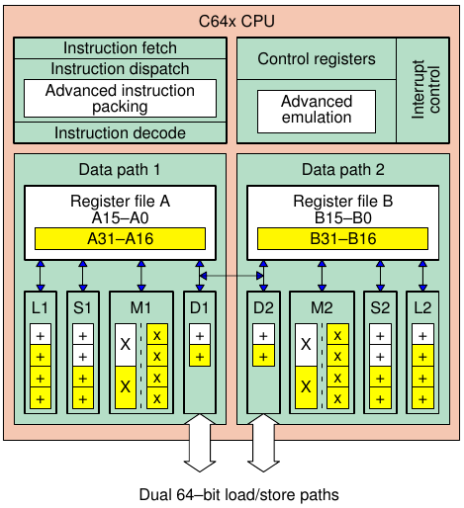

Figure 2.1: TMS320C64x DSP block diagram. The data paths 1 and 2 each contain a register file of 32 registers and four of the total eight functional units.

An important aspect concerning the usage of the *DSP* is the fact that both processors do not share their caches and both have their own *MMUs* with own virtual address spaces. This must be kept in mind when transferring data between them, because it must be ensured that the cache is invalidated before the first read access and written back to *RAM* after the last writing step, and that addresses are mapped to the other address space before exchange.

2.2 FAST Feature Detector

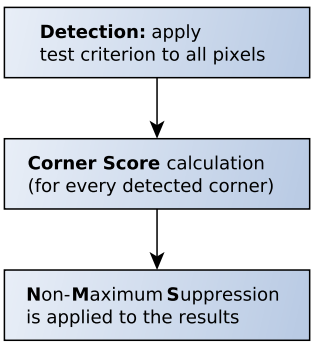

Figure 2.2: Abstract FAST algorithm flow.

The *FAST* (Features from Accelerated Segment Test) feature detector was published by Edward Rosten et al. in 2005 [20] and is a relaxed version of the *SUSAN* [23] corner detector as it uses the brightness of its surrounding pixels to decide whether a corner candidate is a corner or not. The *FAST* algorithm uses a Bresenham circle, which is a discretized circle, of radius 3 as a test mask around the center pixel p. The center pixel is also called nucleus and is marked green in Figure 2.3, whereas the circle is highlighted in red. This specifically means that the values of these 16 pixels on the test pattern have to be compared to the brightness of the nucleus p. According to the test criterion there have to be at least n contiguous pixels on the test pattern circle which are all brighter or all darker than the nucleus by more than a threshold t. The intensities of the other $16 - n$ pixels are ignored. The parameter n describes the length of the arc, which indirectly defines the angle at which corners are detected and thus it has great influence on

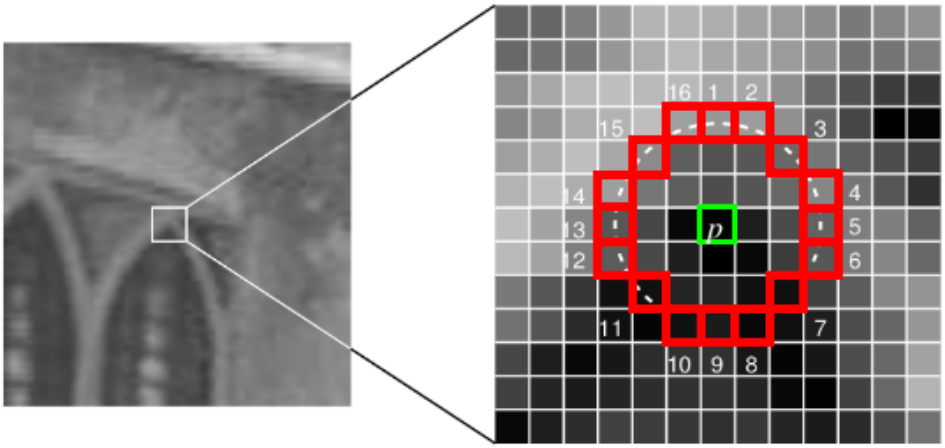

Figure 2.3: The 16 pixel segment test pattern is colored in red and is used to determine a corner. The green center pixel p is the corner candidate, whereas the dashed line indicates an arc of 12 contiguous pixels which are all brighter than p by more than a threshold t [20, 21].

repeatability and robustness of the detector. In [22], $n = 9$ was shown to have unmatched processing speed and a very high repeatability. Within this work only $n = 9$ will be used and other possible test-shapes as in [14] will not be examined either.

The described *Detection* is followed for each detected corner by a calculation of the *Corner Score* (see also Figure 2.2), which represents the strength of a corner. Looking at the nucleus the score value equals the maximum threshold at which it will still be a corner [22]. These scores are later used in the *NMS* (non-maximum suppression) to reduce the number of locally equal features. The *NMS* compares each pixel with its 8 surrounding neighbors and rejects all pixels for which a neighbor reaches a higher score [20, 21, 22, 14, 23].

2.3 Sample Application

At the *DLR* in the *XRotor*-Group for *UAVs* (Unmanned Aerial Vehicles), where this work was supported, a possible use-case of the *FAST* feature detector is in a visual compass application. To implement this, a camera equipped with a circular fish-eye lens is mounted top down under an *UAV*, through which it is able to see the horizon in almost every situation (see Figure 2.4). By detecting feature points along the horizon and tracking them during the permanent movement of the *UAV* it is possible to calculate a rotation angle. There are of course conventional magnetic field sensors available for this purpose, but they may drift after a while or may be jammed by electromagnetic fields in certain situations or environments. To solve this problem, the magnetic sensor data can be fused with data from the optical rotation sensor, where in the case of an error the information from one sensor can be discarded and a

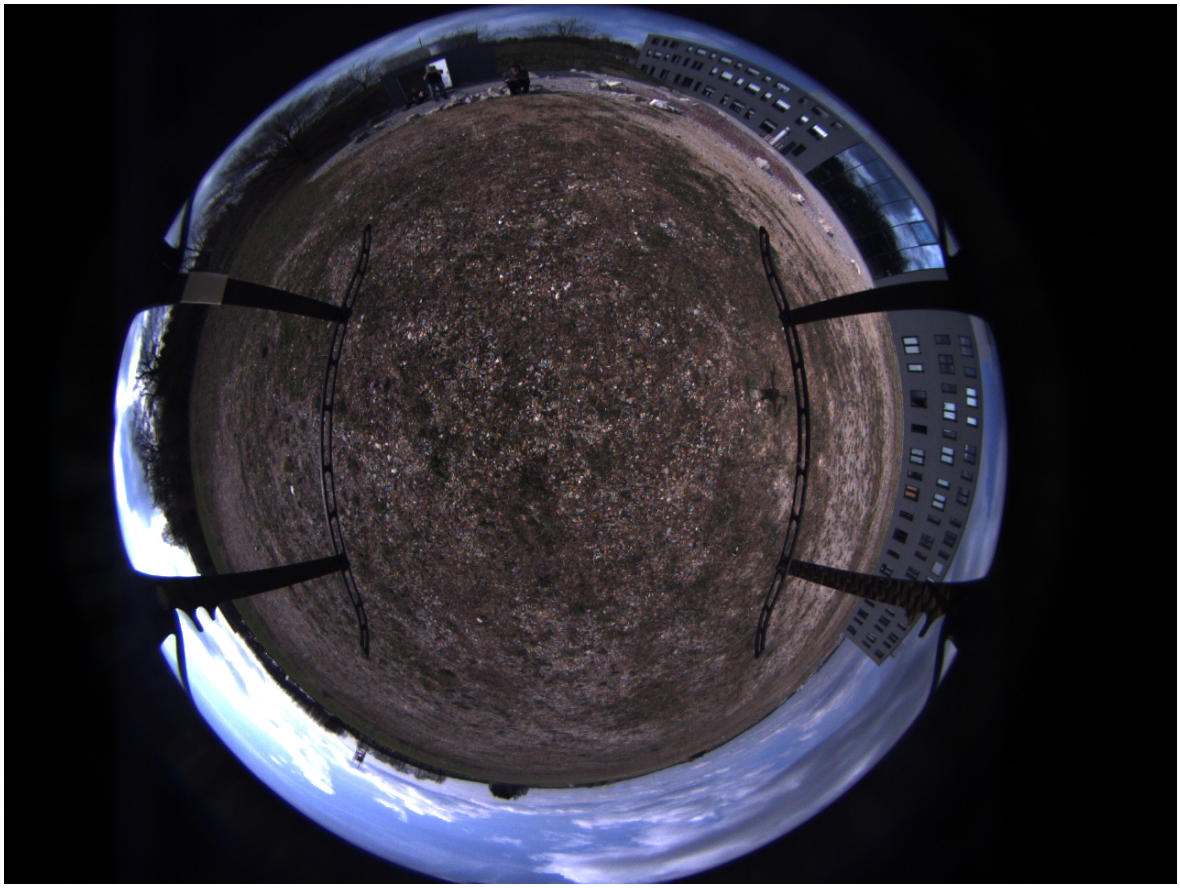

Figure 2.4: Image *Omni* taken from a quadrocopter with a fisheye lens.

reliable navigation can be assured. This is just one out of many possible use cases for optical sensors on *UAVs* for which this detector can be used.

3 Existing and Non-Optimized Implementations

Edward Rosten, one of the authors of the *FAST* detector, maintains an open source library called *libCVD - computer vision library*, which is a "high performance C++ library for computer vision, image, and video processing" [19]. It contains the reference implementation to the FAST detector [21]. A second source is another computer vision library: *OpenCV* [5]. In the source files it is said that this implementation is also from the authors of *FAST*, but the implementation is quite different, which will be clear after the following sections.

In order to examine and to evaluate the different implementations, a framework was created, which incorporates routines (from [24]) to read binary data from, and write to *Portable Grey Map (PGM)* images. A means of time measurement for comparing the efficiency and generation of results, and several debugging and testing mechanisms were also added to help with the verification process of later optimizations.

In the following, both implementations will be examined, and a third variation will be introduced.

3.1 Code Analyses of Existing Implementations

3.1.1 libCVD Implementation

The *libCVD* implementation of the detection is auto-generated and consists of about 4500 lines of code of cascaded if-structures. The nested structure represents the full segment test criterion, which tests all 16 pixels in the test pattern in a specific order. This order results from an optimized decision tree which was generated from several test sequences by the authors, as the "efficiency of the detector will depend on the ordering of the questions and [...][it] is unlikely that this choice of pixels is optimal" [22]. This implies that the detector will not likely represent the best variation of the tree for the images used in this test. Nevertheless it will be included as a reference, because no learning or training will be used in the line of this work and later on, as there is no unique adaption possible for mobile devices in general at compilation time.

The corner score calculation is similarly constructed and also mechanically generated. The same arguments as in the previous paragraph can be applied to this part.

3.1.2 OpenCV Implementation

The *OpenCV* implementation includes a generalized approach, which is not based on optimized trees and is processor independent. But it also includes a partial vectorized (*SIMD*) version for the multimedia extension *SSE2* on x86 *CPUs*.

OpenCV Single Pixel Processing

The single pixel implementation, in contrast to the vectorized one, makes use of the following fact: An arc of 9 or more pixels on the test pattern contains at least one single pair of opposing pixels that are included in the arc. See pixels 0 and 8 in Figure 3.1 on the test circle.

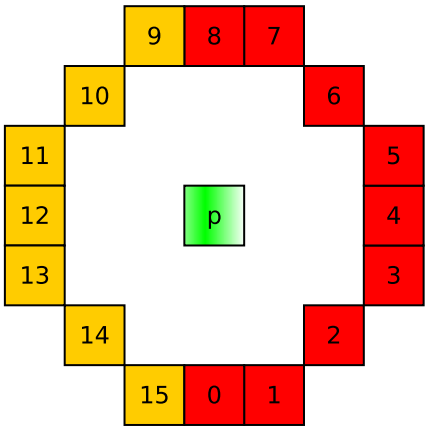

Figure 3.1: Test pattern with offsets and 9 pixels marked in red.

At first the values of the pixels $x \in \{0 \dots 15\}$ on the circle are preprocessed with the help of a lookup-table, where the intensity I_x is used as an index resulting in one of three values $V_x \in \{0, 1, 2\}$, dependent on the nucleus p and threshold t:

$$V_x = \begin{cases} 1, & \text{for } I_x < I_p - t \quad (darker) \\ 2, & \text{for } I_x > I_p + t \quad (brighter) \\ 0, & \text{for } I_p - t < I_x < I_p + t \quad (similar) \end{cases}$$

With the help of this information a three step high-speed test can be implemented. See Figure 3.2 for the whole *Detection* program flow.

- V_0 and V_8 are checked; if both are 0 (similar) then p cannot be a corner anymore and it will be discarded.
- Evaluate all remaining pixel pairs on the circle with even offset $(V_{2/10}, V_{4/12}, V_{6/14})$. If, for one of these pairs, both V_x are 0, then the maximum count will be 7 and no corner will be possible.
- Test all odd pairs of pixels $(V_{1/9}, V_{3/11}, V_{5/13}, V_{7/15})$ in the same manner.

If the nucleus is still not rejected after the high-speed test, then $I_{0\dots15}$ will have to be examined again. Not all information is lost due to the fact that searches for darker or brighter values can be omitted dependent on $1 \in \{V_0 \dots V_{15}\}$ and $2 \in \{V_0 \dots V_{15}\}$. The test pattern is examined counterclockwise and all contiguous pixels with intensities brighter or darker than the nucleus,

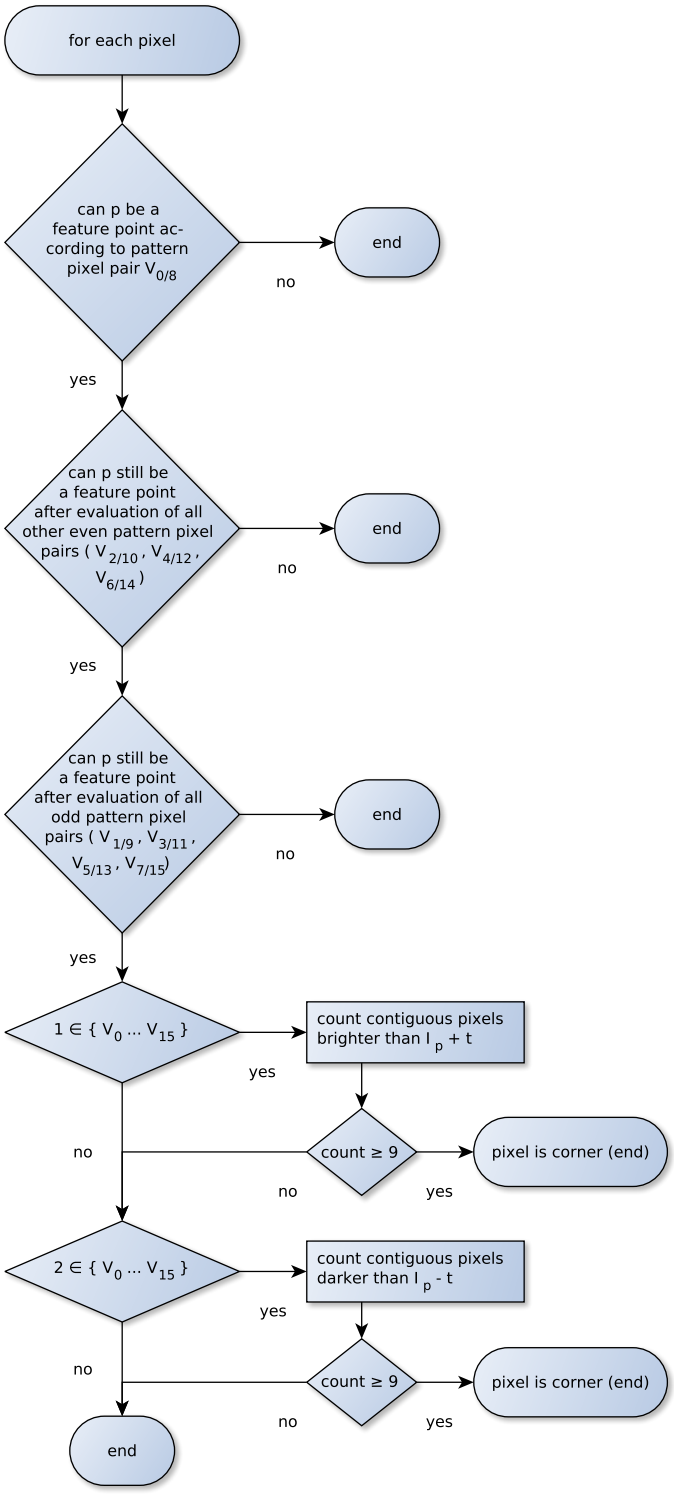

Figure 3.2: Program flow of single pixel *Detection* (*OpenCV*).

by more than the threshold, are counted. When the arc is interrupted by pixels, which do not belong to the same class of brighter or darker, the counter is reset to 0. If the counter exceeds 8, then a valid corner is found.

After the detection of a corner, the *Corner Score* is calculated immediately. The score value is the maximum threshold at which the corner candidate is still a feature point. The *Corner Score* routine is designed in a way that it does not use any information whether a *positive* (where the pixels on the circle are brighter than the center pixel) or a *negative* corner was detected. Both the positive and the negative score is calculated. Therefore all pixels on the test pattern are stored in an array as shown in Figure 3.3. For each of the 16 possible positions of the 9-pixel-arc on the pattern, which could determine the corner, the minimum is selected (for the positive score). Out of these minima the maximum is selected and compared to the negative *Corner Score* value, which is calculated in the same way, except that minimum and maximum functions are inverted. The score with the greater distance to the nucleus' intensity represents the (*Corner Score* + 1), because the pixels on the test pattern must exceed the threshold by one at least. To obtain the *Corner Score*, 1 is subtracted and the result is returned.

Corner Scores from three image lines are buffered in an array, where after every completion of a single row the NMS is applied. Coordinates from corners which do not have a neighbor with a higher score are appended to the feature-point list. A neighbor which was not declared a feature point beforehand is treated as a corner with score 0.

OpenCV SIMD Processing

When compiling the *OpenCV* implementation for a *CPU* that supports the *SSE2* instruction set, there is an alternative implementation included that makes use of the multimedia extension. Of course the code is not suitable for the *ARM CPU*, but as *NEON* is very similar, an

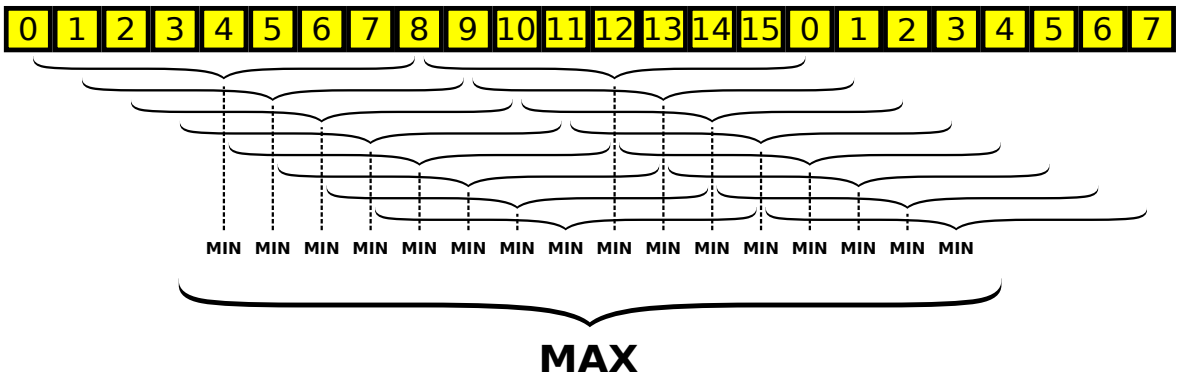

Figure 3.3: *Corner Score* calculation: Test pattern pixels with offsets in an array; the minima of all possible 9-pixels-arcs are calculated and the maximum from these results is $t_{max} + 1$ for the upper threshold. For the lower threshold the search for minimum and maximum are exchanged.

adapted *NEON* version will be created in the next section. In the following paragraphs the *SIMD* version will be analyzed and explained.

The program flow itself remains the same as without vectorization, but the *Detection* part is a bit different, as Figure 3.4 shows: The algorithm uses *SIMD* instructions for each line in the given image until a complete vector cannot be filled with values anymore. The remaining pixels of the row are processed one by one by the single-pixel approach. The *Detection* itself can be divided into two subparts: The *Abort Criterion* and the *Vector-Detection*. Both parts use vectors to process 16 pixels in parallel in 128-bit vectors. (See Figure 3.5 for a vectorized version of the test pattern.) The *Corner Score* calculation uses *SIMD* instructions for a single score, as they do not have to be calculated for all pixels, and the ones which have to be calculated are not necessarily immediately consecutive pixels in memory.

The *Abort Criterion* replaces the high-speed test and is adapted for a whole vector. It would not make sense to keep the three-stages test, as the calculation for the whole vector has to be done anyway if a single element is a corner. Instead of calculating anything concrete, which can later be used in the *Vector-Detection*, it tries to get a general overview of the vector's content. Therefore, it uses a variation of the original high-speed test for the $n = 12$ *FAST* detector [20], which checks the 90°-pixels on the test pattern (offsets 0, 4, 8, 12 in Figure 3.5). To obtain an arc of 9 or more contiguous pixels, at least two of these four pixels are included. So the *Abort Criterion* examines all possibilities of two neighboring 90°-pixels. If for none of these pairs both exceed the nucleus p by at least the threshold t, then this element cannot be a corner. In the next step the decision is made whether the *Vector-Detection* is skipped for this vector or not. If none of the vector's elements can be a corner, then the detection is skipped and the next vector is loaded from memory. If only the second half of the vector can contain one or more corners, then the *Abort Criterion* is repeated with a new 16 elements vector loaded with 8 pixels offset to the previous. This results in a second run of the *Abort Criterion*, but the *Vector-Detection* runs on a vector, which can contain more corners with the same effort.

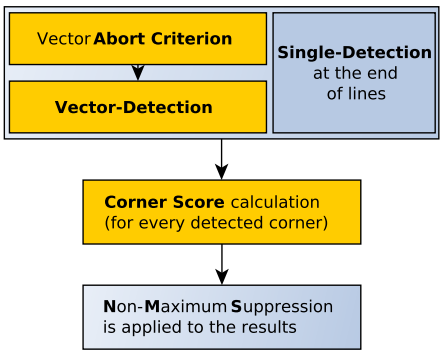

Figure 3.4: Vectorized *FAST* algorithm flow. The yellow parts are vectorized.

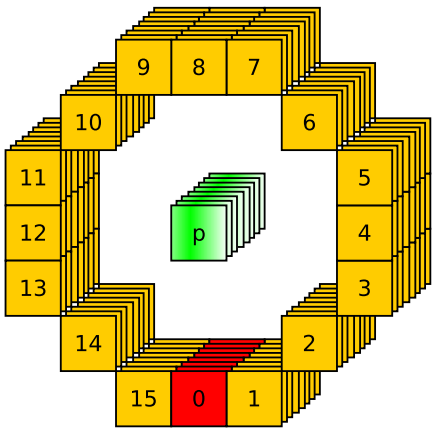

Figure 3.5: Vectorized test pattern with offset-vector 0 marked in red.

Figure 3.6: Vectorized counter example. `c0` is the counter and `m0` contains the comparison results. The first and the fourth element are incremented, whereas the second is reset and the third stays at 0.

The *Vector-Detection* counts the maximum length of a contiguous arc of pixels on the test pattern, which are all greater or lower than the nucleus ± the current threshold, for 16 elements in the vector at the same time by comparing the test pattern pixels with the absolute threshold values $p \pm t$. The counting is done by subtracting the result of the comparison from the current counter value. As a comparison of two elements results in `0x00` or `0xFF`, and `0xFF` has a decimal value of -1, when interpreted as two's complement, the signed subtraction results in -0 or $+1$. If the arc is interrupted, the counter must be reset to 0. To implement this feature, the counter is *AND*-combined with the comparison result as the last step of one counting cycle. For a graphical illustration of an example counting sequence see Figure 3.7 and for the instructions used in one counting step, see Figure 3.6 as an example.

The *Corner Score* is also available as a vectorized implementation. Here it is not realized as parallel processing of multiple scores, but as a calculation of a single *Corner Score* with the means of vector instructions. Therefore the array of test pattern pixels is generated as in the non-vector approach. The algorithm itself stays also the same, except that multiple minimum/maximum extractions are executed parallel on a vector. At the end a serialization has to be performed on the vector to obtain a single result from the results in the vector elements. In Figure 3.8 the *NEON* implementation is illustrated.

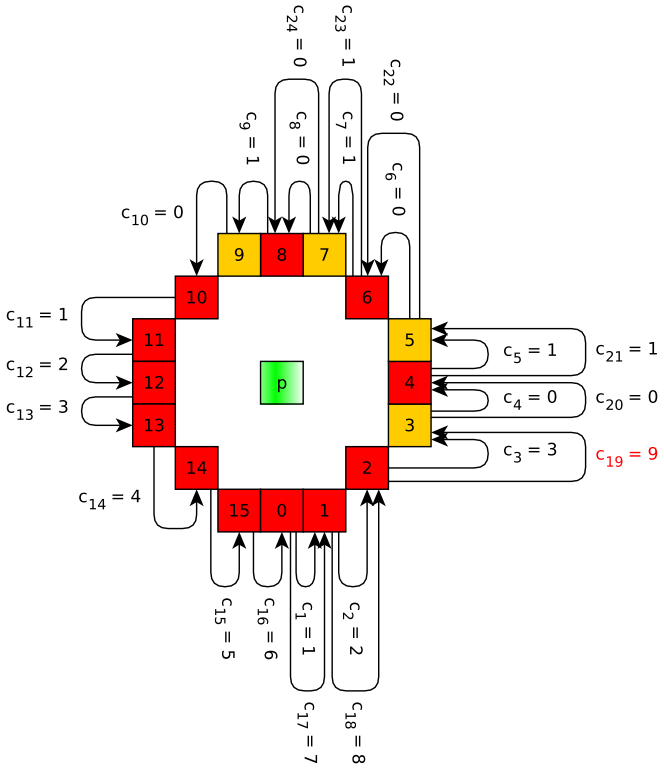

Figure 3.7: Counting pixels on the contiguous arc on the test pattern. The red pixel exceed the threshold and c is the counter with the initial value $c_0 = 0$. 24 counting steps are performed, because it is the worst case to count a 9 pixel arc. The illustration shows a slice of the vectorized implementation. At c_{19} the maximum count is reached and saved as maximum count.

3.2 Porting of the OpenCV SIMD Implementation

The algorithm was already implemented with the help of *SIMD*-extensions to increase the performance. Because *SSE2* and *NEON* share large parts of their instruction sets and their possible vector lengths, it is likely that a *NEON* version would also be faster on the *ARM CPU*. Therefore it would be nice to have a working and comparable *NEON* implementation before starting to analyze the different approaches in terms of performance, because the porting itself is not expensive in terms of effort. The *SSE*-implementation uses intrinsics that look like standard *C* function calls, but they are usually replaced by the compiler with a single corresponding *SIMD* assembler instruction. So in the first step the given code was ported without any optimizations to the *NEON* architecture.

In the *Abort Criterion* and the *Vector-Detection* only the intrinsics in Table 3.1 were used and most of them are available in both instruction sets. The *SSE* intrinsic `_mm_movemask`, which generates an 16-bit mask of the vector elements' *MSBs*, could not be represented with a single instruction and was substituted for the first implementation with code that was proposed in [6] (see Listing 3.1).

The *Corner Score* calculation also uses intrinsics from Table 3.1 and works with 16-bit signed integers, because 8 bit cannot cover the possible range $-255 \leq (I_p - I_x) \leq 255$. The serialization phase, where the maximum vector element is extracted (for the positive threshold), contained the *SSE* intrinsics `_mm_unpackhi_epi64` and `_mm_srli_si128`. The first instruction unpacks and interleaves the high halves of two given vectors and the second shifts the vector right by a number of bytes, but *NEON* does not feature direct complements. So the code was modified to serialize the vector slightly differently, but with the same logical result and the same amount of intrinsics. The two code fragments are shown in Listing 3.2 and 3.3. `vextq_s16`

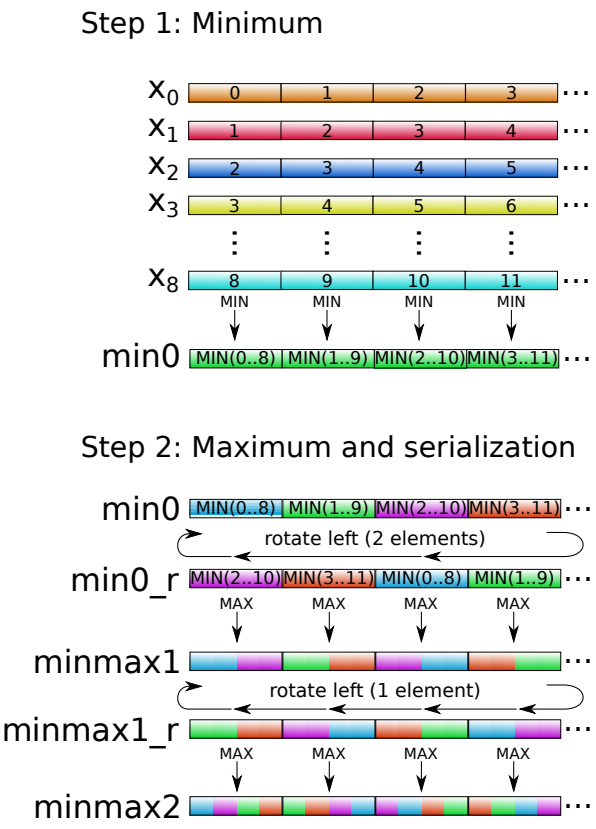

Figure 3.8: Vectorized *Corner Score* calculation for positive thresholds. In step one, the parallel search for the minimum is outlined. The numbers in vectors x_0 through x_8 represent the pixels' offsets on the pattern. The rotation in the second step is implemented with an extract-instruction. The example is simplified, as it only shows a four-elements-vector. The double vector size, as used in the implementation, requires a third application of rotation and maximum selection in step two, according to Listing 3.3. In the end `minmax2` contains the maximum of all elements from `min0`.

extracts a specific number of 16-bit elements from the first vector and takes the remaining elements from the second, which makes it a logical rotation of elements for the whole vector when used with the same vector. In Figure 3.8 the vectorized *Corner Score* implementation is shown.

Listing 3.1: *Movemask* equivalent [6]

```
uint16_t movemask(uint8x16_t input){

    const uint8_t __attribute__ ((aligned (16))) _Powers[16] =
    { 1, 2, 4, 8, 16, 32, 64, 128, 1, 2, 4, 8, 16, 32, 64, 128 };

    //~ Set the powers of 2 (do it once for all, if applicable)
    static uint8x16_t powers = vld1q_u8(_Powers);

    //~ Compute mask from input
    uint64x2_t _Mask = vpaddlq_u32(vpaddlq_u16(vpaddlq_u8(vandq_u8(input,
        powers))));

    //~ Get resulting bytes
    uint16_t mask;
    vst1q_lane_u8((uint8_t *)&mask + 0, (uint8x16_t)_Mask, 0);
    vst1q_lane_u8((uint8_t *)&mask + 1, (uint8x16_t)_Mask, 8);

    return mask;
}
```

Listing 3.2: *Corner Score* serialization: *SSE* implementation

```
q0 = _mm_max_epi16(q0, _mm_unpackhi_epi64(q0, q0));
q0 = _mm_max_epi16(q0, _mm_srli_si128(q0, 4));
q0 = _mm_max_epi16(q0, _mm_srli_si128(q0, 2));
threshold = (short)_mm_cvtsi128_si32(q0) - 1;
```

Listing 3.3: *Corner Score* serialization: *NEON* implementation

```
q0 = vmaxq_s16(q0, vextq_s16(q0, q0, 4));
q0 = vmaxq_s16(q0, vextq_s16(q0, q0, 2));
q0 = vmaxq_s16(q0, vextq_s16(q0, q0, 1));
vst1q_lane_s16((int16_t *) &threshold, q0, 0);
threshold--;
```

The resulting code was validated by detecting the same corners in the same test images on both a *CPU* capable of executing *SSE* instructions and the *CPU* on the *Beagleboard*, on which the *NEON* implementation was executed. Also, a test was successfully completed in which a random vector including its environment was generated and then analyzed by the

Table 3.1: *SIMD* implementation intrinsics overview

Description	**SSE Intrinsic**	**NEON Intrinsic**
load a 16x8-bit vector from memory	_mm_loadu_si128	vld1q_s8
load a 8x16-bit vector from memory	_mm_loadu_si128	vld1q_s16
bitwise AND	_mm_and_si128	vandq_u8
bitwise XOR	_mm_xor_si128	veorq_s8
bitwise OR	_mm_or_si128	vorrq_s8
subtract unsigned 8-bit integers with saturation	_mm_subs_epu8	vqsubq_u8
add unsigned 8-bit integers with saturation	_mm_adds_epu8	vqaddq_u8
subtract signed 8-bit integers	_mm_sub_epi8	vsubq_s8
subtract signed 16-bit integers	_mm_sub_epi16	vsubq_s16
compare two signed 8-bit integers	_mm_cmpgt_epi8	vcgtq_s8
select the maximum of unsigned 8-bit integers	_mm_max_epu8	vmax_u8
select the maximum of signed 16-bit integers	_mm_max_epi16	vmaxq_s16
select the minimum of signed 16-bit integers	_mm_min_epi16	vminq_s16
generate 16-bit mask of the elements' *MSBs*	_mm_movemask_epi8	not available
set all elements to the same value	_mm_set1_epi16	vdupq_n_s16
set all elements to 0	_mm_setzero_si128	not available use vdupq_n_s16(0)
unpack and interleave 64-bit integers from the high halves	_mm_unpackhi_epi64	not available
shift a number of bytes right and insert zeros	_mm_srli_si128	not available

vectorized version itself and compared to a simple implementation which examined one pixel after another on the test pattern for each element on the vector. Finally, the implementation was successfully validated.

3.3 Profiling of the Available Implementations

In order to get an overview of the efficiency of the different implementations and to find the parts and sections of the code that are worth optimizing, a profiling was done. Details on how these measurements were performed are described in Chapter 5. Figure 3.9 and 3.10 show the results of all three implementations for two different images at realistic thresholds. Both charts show similar results at different scales. The *SIMD* implementation is the fastest version, whereas the *libCVD* implementation is even faster than the *OpenCV* version without *SIMD* support. Inside the *NEON* port over 85% of the whole calculation time is spent in the *Detection*, whereas the *Corner Score* calculation already profited from its vectorization and does not carry much of a weight any more.

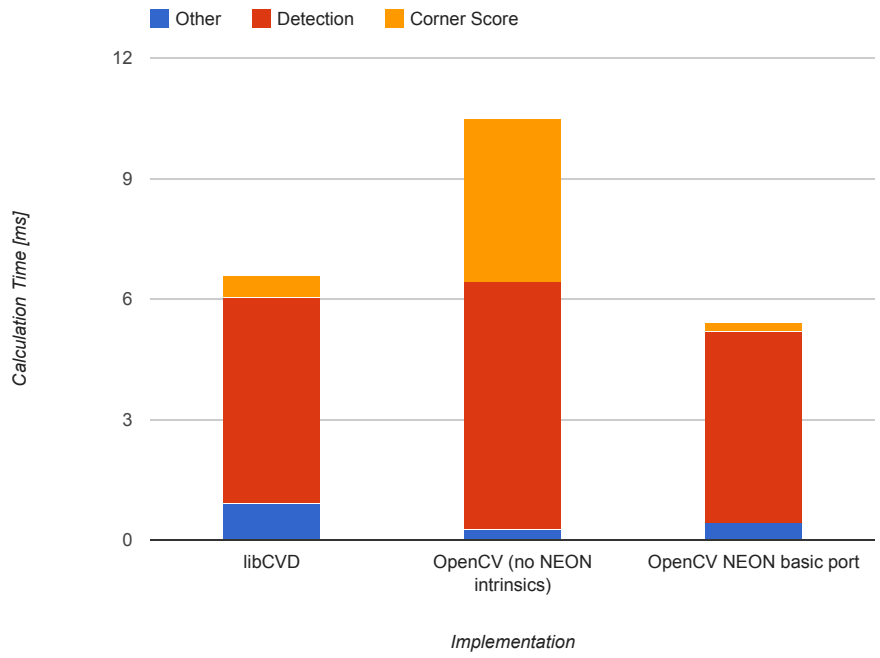

Figure 3.9: Profiling results from test image *Lena*. *Other* includes the *NMS*, initialization and the basic loops inside the algorithm.

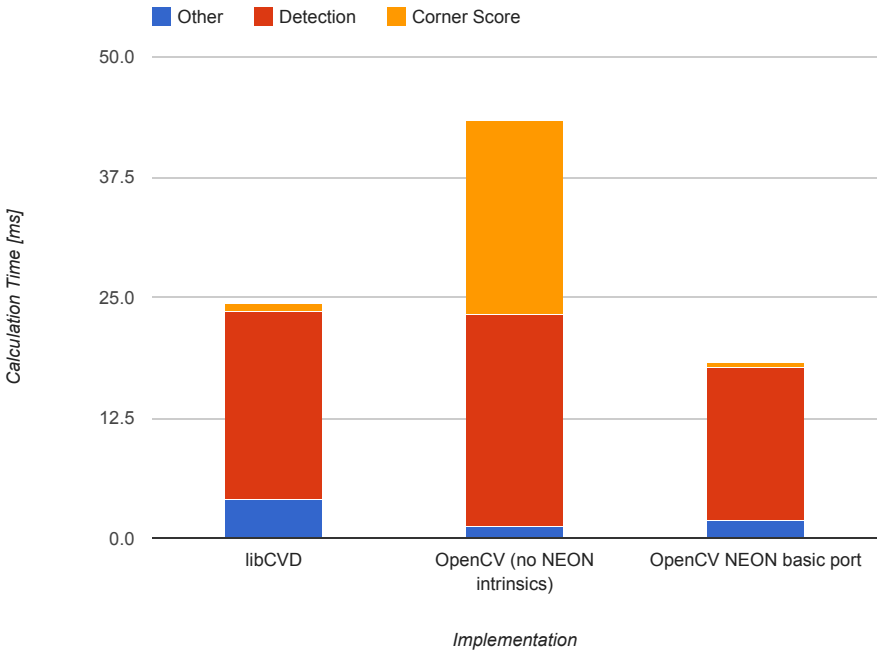

Figure 3.10: Profiling results from test image *Omni*. *Other* includes the *NMS*, initialization and the basic loops inside the algorithm.

4 Optimization

4.1 NEON Optimizations

In Section 3.3 it was shown that the *OpenCV* basic *NEON* port was the fastest of the reviewed implementations and it will be further optimized in this section. There is no way of vectorization or pipeline optimizations in the *libCVD* implementation because of the high amount of conditional branches, which interrupt the program flow enormously. Also, the *OpenCV* implementation without *NEON* intrinsics will not be optimized because the *NEON* port is already a kind of optimization from this version. So in this section, it will be tried to optimize the *NEON* port in respect of code, algorithm and cache with focus on the *Detection*, as it consumes the most time and therefore offers the highest potential of optimization.

4.1.1 Code Optimizations

Although *gprof* [8], a profiling tool, was not able to generate exact timing information due to its sampling resolution, it was able to show in the first tests that the *movemask* equivalent from Listing 3.1 was called quite often. It occurs once in the *Abort Criterion* and once at the end of the *Vector-Detection* for every vector. It was clear from the beginning that it was not necessarily the fastest replacement and would consume quite a bit of the whole calculation time, but it was a simple and fast solution for a quick porting of the code. Finally, both cases were exchanged with some modified code in order to avoid the mask exactly as generated by *movemask*.

For the *Abort Criterion* it is only important to know whether there are possible corners in the first and in the second half of the vector. Thus, a saturated narrowing of each half is performed, which results in a 64-bit vector. This vector is stored in an array of two 32-bit integers, because there are no instructions for a narrowing down to 32-bit or less. The detection whether the whole vector or the half is empty can now be checked by only comparing the two integers with zero. Listing 4.1 shows the code before, and the bottom section of 4.2 shows it after the optimization. The code does not seem to be more efficient on the first sight, because three lines replace a single line in the listing, but the old implementation contains a function call to the even longer *movemask* routine. All in all the execution time could be reduced by this modification.

Listing 4.1: Movemask (*Abort Criterion*) before optimization

```
uint16_t mask = movemask((uint8x16_t) m0);

if (mask == 0)
    continue;
if ((mask & 255) == 0)
{
    j -= 8;
    ptr -= 8;
    continue;
}
```

Listing 4.2: *Abort Criterion* (and *movemask* replacement) after optimization

```
uint8x16_t m0, m1;
uint8x16_t v0 = vld1q_u8((const uint8_t *) ptr);
uint8x16_t v1 = vqsubq_u8(v0, t);
v0 = vqaddq_u8(v0, t);

uint8x16_t x0 = vld1q_u8((const uint8_t *)(ptr + pixel[0]));
uint8x16_t x1 = vld1q_u8((const uint8_t *)(ptr + pixel[4]));
uint8x16_t x2 = vld1q_u8((const uint8_t *)(ptr + pixel[8]));
uint8x16_t x3 = vld1q_u8((const uint8_t *)(ptr + pixel[12]));
m0 =  vandq_u8(vcgtq_u8(x0, v0), vcgtq_u8(x1, v0));
m1 =  vandq_u8(vcgtq_u8(v1, x0), vcgtq_u8(v1, x1));
m0 =  vorrq_u8(m0, vandq_u8(vcgtq_u8(x1, v0), vcgtq_u8(x2, v0)));
m1 =  vorrq_u8(m1, vandq_u8(vcgtq_u8(v1, x1), vcgtq_u8(v1, x2)));
m0 =  vorrq_u8(m0, vandq_u8(vcgtq_u8(x2, v0), vcgtq_u8(x3, v0)));
m1 =  vorrq_u8(m1, vandq_u8(vcgtq_u8(v1, x2), vcgtq_u8(v1, x3)));
m0 =  vorrq_u8(m0, vandq_u8(vcgtq_u8(x3, v0), vcgtq_u8(x0, v0)));
m1 =  vorrq_u8(m1, vandq_u8(vcgtq_u8(v1, x3), vcgtq_u8(v1, x0)));
m0 =  vorrq_u8(m1, m0);

uint32x2_t m_half = vqmovn_u64((uint64x2_t) m0);
uint32_t mask_array[2];
vst1_u32(mask_array, m_half);

if ((mask_array[0] ==  0) && (mask_array[1] == 0))
    continue;
if (mask_array[0] == 0)
{
    j -= 8;
    ptr -= 8;
    continue;
}
```

In the code at the end of the *Vector-Detection* the *movemask* routine was replaced by a storage of the whole vector, because here the result from every element is needed. Of course the storage itself is slower, but the subsequent comparisons are shorter, no shifting has to be done and the *movemask* call is omitted. Listing 4.3 and 4.4 show the differences.

In further investigations it was discovered that the *SSE* version included a conversion of the 8-bit unsigned values to a signed representation. The conversion was implemented by a subtraction of the half range (128) in order to keep the whole range after converting to signed values. Originally the values were compared with the intrinsic `_mm_cmpgt_epi8` for signed 8-bit elements, because no unsigned counterpart exists in any *SSE* version. *NEON* features such an unsigned comparison and can profit from the removal of the conversion, which is needed for every vector. Listing 4.2 shows the code with already removed conversions.

Listing 4.3: Movemask (*Vector-Detection*) before optimization

```
1 int m = movemask(vcgtq_u8(max0, K16));
2
3 for( k = 0; m > 0 && k < 16; k++, m >>= 1 )
4     if(m & 1)
5     {
6         cornerpos[ncorners++] = j+k;
7         if(nonmax_suppression)
8             curr[j+k] = (uint8_t)pastCornerScore(ptr+k, pixel, thr);
9     }
```

Listing 4.4: Movemask (*Vector-Detection*) after optimization

```
1  uint8_t m_array[16];
2  vst1q_u8(m_array, (vcgtq_u8(max0, K16)));
3
4  for( k = 0; k < 16; k++)
5     if(m_array[k] != 0)
6     {
7         cornerpos[ncorners++] = j+k;
8         if(nonmax_suppression)
9             curr[j+k] = (uint8_t)pastCornerScore(ptr+k, pixel, thr);
10    }
```

Another possible optimization was the usage of *inline assembler* in the three parts, where parallelization is used. The *Abort Criterion* was implemented in assembler and it showed that there are no pitfalls for the compiler to generate inefficient assembler code from the intrinsics, and that there is no way to gain even more time. The assembler output of the other two parts were examined and no improvement possibilities were found.

4.1.2 Algorithm Optimizations

It was also tested whether the processing of single pixels at the end of each line could be replaced by an additional vector processing loop with only a part of its content being valid data. In some cases a rather small reduction of execution time could be reached, but no universal improvement could be measured, as it depends on the actual line length and the number of features found in each row. A benefit in some cases is definitely possible, but this should be evaluated separately for each case.

The algorithm was also modified in a test to integrate the *Corner Score* calculation into the *Vector-Detection* loop and to keep it really parallel. If the vector contained only feature points, for which the score had to be calculated anyway, then the algorithm could profit from calculating the score in parallel. In standard use cases, in which vectors do not contain that many corners to justify the massive prolonging of the *Vector-Detection*, it is faster to only compute the score for the really detected feature points. Another reason why this implementation is slower than the original is the fact that the *CPU* features only 16 *NEON* registers, but far more values have to be kept accessible to calculate the *Corner Score* within the detection loop. Even an more optimized version with the whole *Vector-Detection* unrolled manually was slower than the basic *NEON* port.

4.1.3 Cache Optimizations

A reason for different algorithms revealing a poor performance is an inefficient cache usage, which causes pipeline stalls and waiting cycles. As the *FAST* algorithm only needs seven image lines to be quickly accessible, images of moderate resolutions should fit into the *CPU*'s *L1-cache*, which features 32 KB. To prove this statement, a cache simulation was performed with *Dinero IV* [7]. Therefore all read accesses to image data were logged including the addresses, which were then used for the simulation. The simulation results for the tested images showed only cold misses, which are obligatory and cannot be avoided.

4.2 DSP Implementation and Optimization

The *CPU* implementation was optimized up to a stage at which the optimization effort slowly exceeds the resulting gain in performance, because the large improvements with big impacts were already covered and only small non-optimal parts may be hidden somewhere. To reduce the resource footprint a bit more on the *CPU* itself, an implementation for the on-chip *DSP* will be created and adapted for the special needs of the co-processor. As many of the optimizations done in the previous sections do not only apply for the *NEON* instruction set, but are also applicable for the *DSP*, it was tried to keep as many as possible.

On the host side implementation, a possibility to choose between execution on *CPU* and *DSP* was added. Furthermore, the image data must be copied to an aligned buffer suitable for the *DSP*. An output buffer must be allocated, and the buffers have to be mapped, too.

On the *DSP* side, the *FAST* source code was striped down to plain C and e.g. the use of `std::vector` was removed. The *C++* vector contained the results after completion and was replaced by an array of configurable size, because the *API* only allows buffers with defined sizes as exchange between the two units. Apart from that, the lookup-table used for the non-vector detection at the end of the lines could not be stored inside the stack for some reason, as it contained 511 32-bit integer values. Placing it in the stack frequently caused the *DSP* to crash, and only a complete reboot could help. Storing this table in the heap solved the problem. Of course, the parallel parts differ from their *NEON* counterparts, because of their plain variation in vector length, but the algorithm stays the same. In Listing 4.5 the *Abort Criterion* with *DSP* intrinsics is shown, whereas Listing 4.2 states the *NEON* version.

Listing 4.5: Abort Criterion (*DSP*)

```
1  uint32_t m0, m1;
2  uint32_t v0 = _mem4((void *) ptr);
3  double v1_d = _mpyu4(v0, 0x01010101);
4  uint32_t v1 = _spacku4(_ssub(_hi(v1_d), t2), _ssub(_lo(v1_d), t2));
5  v0 = _saddu4(v0, t);
6
7  uint32_t x0 = _mem4((void *)(ptr + pixel[0]));
8  uint32_t x1 = _mem4((void *)(ptr + pixel[4]));
9  uint32_t x2 = _mem4((void *)(ptr + pixel[8]));
10 uint32_t x3 = _mem4((void *)(ptr + pixel[12]));
11 m0 =  _cmpgtu4(x0, v0) & _cmpgtu4(x1, v0);
12 m1 =  _cmpgtu4(v1, x0) & _cmpgtu4(v1, x1);
13 m0 =  m0 | (_cmpgtu4(x1, v0) & _cmpgtu4(x2, v0));
14 m1 =  m1 | (_cmpgtu4(v1, x1) & _cmpgtu4(v1, x2));
15 m0 =  m0 | (_cmpgtu4(x2, v0) & _cmpgtu4(x3, v0));
16 m1 =  m1 | (_cmpgtu4(v1, x2) & _cmpgtu4(v1, x3));
17 m0 =  m0 | (_cmpgtu4(x3, v0) & _cmpgtu4(x0, v0));
18 m1 =  m1 | (_cmpgtu4(v1, x3) & _cmpgtu4(v1, x0));
19 m0 =  m1 | m0;
20
21 if (m0 == 0)
22     continue;
```

In the first part the threshold (`t` and `t2`, respectively a 8 and a 16-bit vector) is added to and subtracted from the nucleus' intensities with saturation to get the upper and lower boundaries value without any overflow or underflow of the 8-bit range. The *DSP* features a saturated addition on bytes but only a saturated subtraction on 16-bit integers. Here a trick is used

to widen the data type via a multiplication with 1, which results in a double register. The upper and the lower half of the double is taken for the subtraction, followed by a narrowing with saturation of the results down to 8 bit per element.

In contrast to the other already mentioned multimedia instruction sets, the *DSP* implements a compare instruction that implies the generation of a bit-mask. Additionally, an expand command exists, which can generate the results matching the other instruction sets (`0xFF` or `0x00` for *true* or *false*). For the *Abort Criterion* in Listing 4.5 it does not matter which comparison is used, as only logical operations are performed on these values. But the expanding instructions can be omitted because the mask was generated anyway in the original *SSE*-implementation at the end. However, in the *Vector-Detection* the comparison result must be expanded to implement the algorithm. (Cf. 4.6.)

The *SIMD* versions for logical operators like *OR* and *AND* can be replaced here by standard bitwise operators as | and &, because they are not bound to any data type and work on 32-bit integer as well as 32-bit vectors of any element length.

Remaining 128-bit *NEON* intrinsics were simply replaced with available 4-element *DSP* intrinsics shown in Table 4.1.

The option to step forward only half a vector was omitted on the *DSP*, because there was no noteworthy difference in execution time in a short test. This results from the short vector, for which two pixels are traded for another loop of the *Abort Criterion* and unaligned memory accesses, as two pixels equal 16 bits.

Listing 4.6: Detection (*DSP*)

```
1  for ( k = 0; k < 24; k++ )
2  {
3      uint32_t x = _mem4((void *) (ptr + pixel[k]));
4
5      m0 =  _xpnd4(_cmpgtu4(x, v0));
6      m1 =  _xpnd4(_cmpgtu4(v1, x));
7
8      c0 = _sub4(c0,  m0) & m0;
9      c1 = _sub4(c1,  m1) & m1;
10
11     max0 = _maxu4(max0, c0);
12     max1 = _maxu4(max1, c1);
13 }
```

Table 4.1: SIMD DSP implementation intrinsics overview

Description	NEON Intrinsic	DSP Intrinsic
load vector with signed 8-bit elements from memory	vld1q_s8	_mem4
load vector with signed 16-bit elements from memory	vld1q_s16	_mem4
bitwise AND	vandq_u8	&
bitwise OR	vorrq_u8	|
subtract unsigned 8-bit integers with saturation	vqsubq_u8	not available
subtract signed 8-bit integers	vsubq_s8	_sub4
subtract signed 16-bit integers	vsubq_s16	_sub2
add unsigned 8-bit integers	vqaddq_u8	_saddu4
compare two unsigned 8-bit integers	vcgtq_u8	_xpnd4(_cmpgtu4())
select the maximum of unsigned 8-bit integers	vmaxq_u8	_maxu4
select the maximum of signed 16-bit integers	vmaxq_s16	_max2
select the minimum of signed 16-bit integers	vminq_s16	_min2

The *Corner Score* vectorization was translated with the intrinsics in Table 4.1, with the serialization realized as a left rotation (`_rotl`) by 16 bit and a masking of the lower 16 bits as shown in Listing 4.7. (Cf. *SSE* and *NEON* implementations in Listing 3.2 and 3.3.)

Listing 4.7: *Corner Score* serialization: *DSP* implementation

```
1 q0 = _max2(q0, _sub2(0, q1));
2 q0 = _max2(q0, (int32_t) _rotl((uint32_t) q0, 16));
3 threshold = q0 & 0x000000FF;
4 threshold--;
```

The version which was tested for *NEON* with the *Corner Score* calculation inside the *Vector-Detection* was tested again on the *DSP*, because it features more registers and could have profited from the difference in architecture. But it turned out that the additional costs inside the *Vector-Detection* are much larger than the benefits from not having to calculate every *Corner Score* separately. The rather small 4-element vectors exacerbate the situation on the *DSP* even more and renders the proposal redundant.

5 Experimental Results

In this chapter the execution times of the different implementations as well as several subparts will be examined and compared. The results will show the reduction of calculation time or the performance boost through various optimizations. Additionally, a comparison of the power consumptions on both units will be presented.

5.1 Test Settings

The time measurements in this chapter were performed by an internal means of time measurement, as external tools as *gprof* [8] or *callgrind* [32] do not offer the necessary resolution. *gprof*, for example, provides a resolution of 0.01 s, which is too inaccurate for these fast implementations to generate plausible results. In order to get preciser results, the relevant code parts were measured with the internal system clock and the system call `clock_gettime()`.

For all measurements only stream relevant code parts were measured, which must be executed for every single picture. Initialization, reading the picture from a file and printing results were omitted, as well as allocation of page-size aligned buffers in case of the *DSP*. These tasks will have to be performed only once in an use case, where thousands of pictures are processed in a stream. To get runtime information from subroutines or parts of the algorithm, sections were removed from the implementations and the specific times were calculated by subtraction of consecutive measurements. The measurements concerning the consumed time on the *CPU* in case of the algorithm running on the *DSP* were generated with the same technique. Here, the blocking time after the *node* was launched from the host side until the results are available to the *CPU*, was measured to get the pure *DSP*-time. These values might not be 100% accurate, as there are system calls deep inside the kernel involved, but they mark a dimension which can be calculated with. To generate representative results, all values given in this chapter are the average of 100 iterations.

All experiments were made with two different test images: *Omni* (Figure 2.4 converted to 8-bit gray-scale) as a real-world scenario measuring 1280 x 960 pixels and the smaller picture *Lena* (see Figure 5.2), featuring a resolution of 512 by 512.

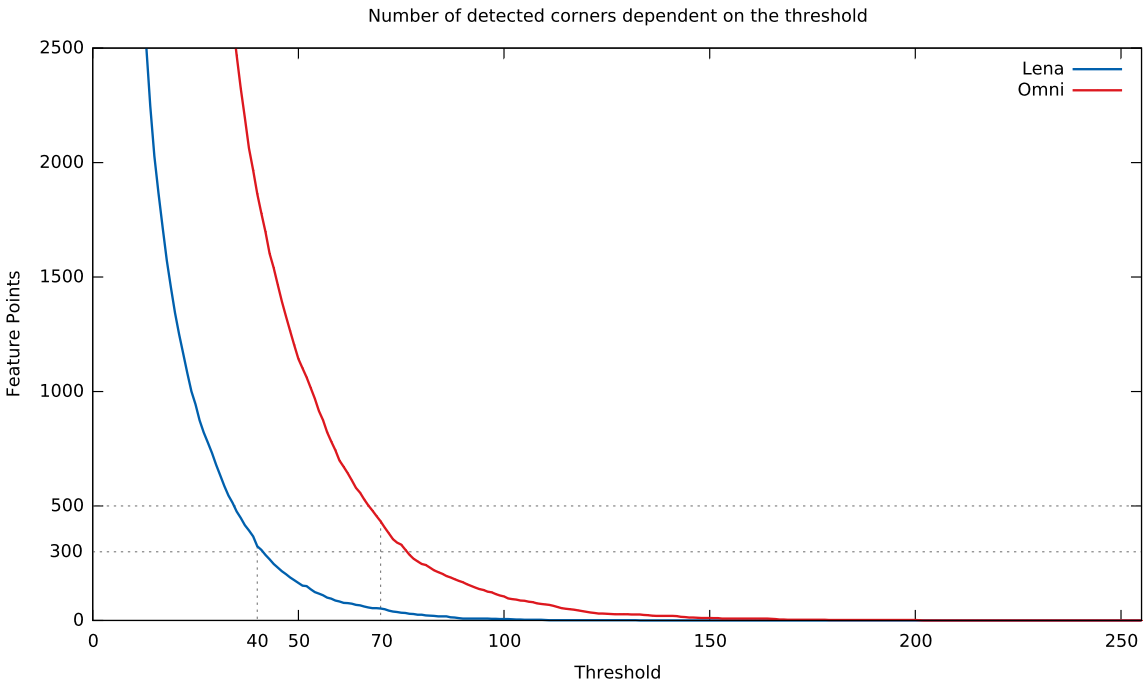

Figure 5.1: Dependence between the number of detected corners and the threshold.

As it is shown in Figure 5.1, the number of detected corners can vary a lot when the threshold is changed. The Figure only shows the corners up to a number of 2500. In fact, the feature point count rises for *Lena* up to 18574 and 46887 for *Omni* at a threshold of 1.

To generate reasonable measurements and comparisons, a target area of a required amount of corners had to be specified. Different applications and environments need sometimes more and sometimes less feature points. For example in [11] the range of about $100 - 1500$ was used in experiments with different detectors. The requirements from the *DLR* demanded $300 - 500$ corners per image, so this was selected as the target area. The lower scaled image *Lena* was chosen to be measured nearer to the lower boundary at a threshold of 40, which generates 324 detected corners. (Cf. Figure 5.2.) *Omni*, which features a higher resolution, was selected to be examined at a threshold of 70 resulting in 433 feature points.

Figure 5.2: Test image *Lena* (512 x 512) with 324 detected corners at a threshold of 40.

5.2 Optimization Results

Figures 5.3 and 5.4 show the results in comparison to the profiling information gathered before the optimization phase and Table 5.1 shows the calculation times of the fastest implementation on both units. Six different implementations were examined:

- *libCVD*: Original *libCVD* implementation.
- *OpenCV* (no *NEON* intrinsics): Original implementation compiled without *SSE2* support.
- *OpenCV NEON* basic port: *SSE SIMD* intrinsics ported to *ARM NEON* without further optimization effort, as described in Chapter 3.2.
- *NEON* optimized: Ported and optimized *OpenCV* implementation.
- *DSP / OpenCV* (no intrinsics): Original *OpenCV* implementation without *DSP* intrinsics.
- *DSP* optimized: *OpenCV* implementation after optimization for the *DSP*.

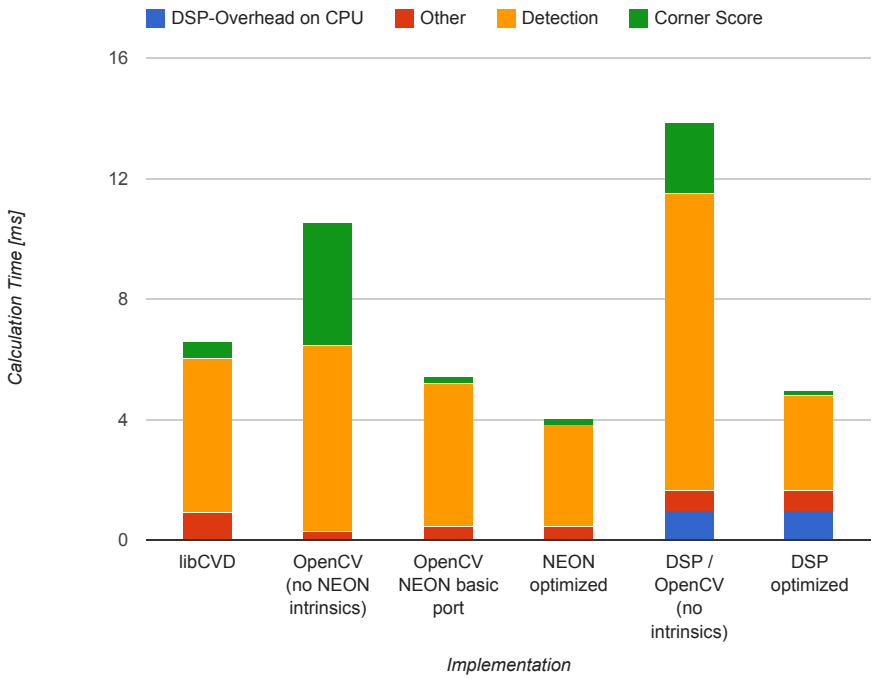

Figure 5.3: Comparison of execution times before and after the optimization based on image *Lena*.

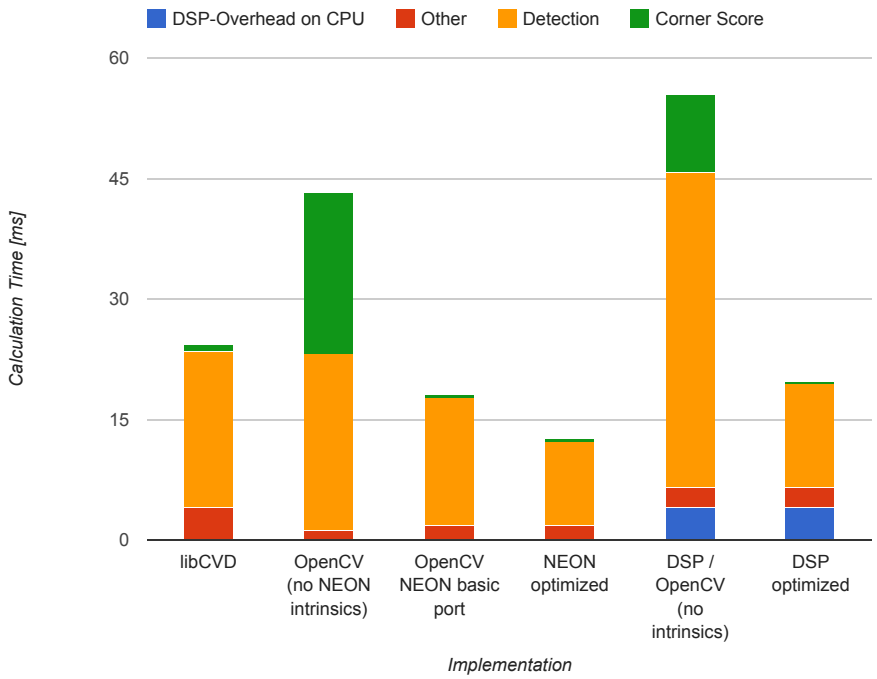

Figure 5.4: Comparison of execution times before and after the optimization based on image *Omni*.

The results for both images show the same trend and point out that optimization helped to reduce the calculation time. For the image *Lena* the overall execution time was reduced by 38.9% when compared to the *libCVD* implementation or 25.8% when compared to the basic *OpenCV NEON* port. For *Omni* these values are even higher : 47.9% and 30.3% respectively.

When outsourcing the algorithm to the *DSP*, there are two possible comparison modes. First, the whole calculation time can be used as a criterion. Or second, the pure *CPU* time can be taken into account, because the *CPU* is not occupied by the detection after the launch of the *node* on the *DSP* and is available to other processes. When comparing the complete execution time, the optimized *DSP* implementation is 23.8% (55.4%) slower than the optimized *NEON* version, but 8.2% faster (8.3% slower) than the basic *NEON* port for image *Lena* (*Omni*). If only the *CPU* time is taken into account, there is a gain of 82.4% (78.0%) compared to the *NEON* basic port and still 76.3% (68.4%) compared to the optimized *NEON* implementation for image *Lena* (*Omni*).

The differences between both test images can be explained by the difference in homogeneity in the image content. Whereas in *Lena* the inhomogeneous areas are rather small and smooth, *Omni* features a large center area, where sharp edges and many corners are located. Of course

Table 5.1: Minimum calculation time overview

Image	Threshold	Calculation Unit	Processing Time
Lena	40	CPU	4.03 ms
Lena	40	DSP	4.98 ms
Lena	40	CPU-time @ DSP	0.95 ms
Omni	70	CPU	12.68.03 ms
Omni	70	DSP	19.70 ms
Omni	70	CPU-time @ DSP	4.00 ms

the amount of data also influences the speed, because the internal parts do not all scale well in the same extend with the image resolution and the numbers of pixels.

5.3 Comparison between CPU and DSP

As already mentioned in the previous section there are efficiency differences on both platforms. These can be seen quite clearly in Figure 5.5, which shows the absolute total time consumptions in comparison.

The interesting point in this comparison is the fact that the *DSP* outperforms the *CPU* at higher numbers of detected corners or, indirectly speaking, at lower thresholds. Of course this is not of any interest for the outcome of the optimization, as this change in performance is outside the required target region of corners, but it is possible to learn something about the performance of different parts from some additional measurements, which are represented in Figures 5.6 and 5.7.

At first, one can see the reason why the *DSP* is faster than the *CPU* at high numbers of detected features: It is especially the slow *Corner Score* calculation, which consumes less than half of the time on *DSP*, although the *CPU* runs at a higher frequency than its opponent. At very low thresholds, at which the scores have to be calculated for many corners, this function clearly carries weight. The fact that this routine scales linearly with the amount of detected feature points, combined with the results in Figures 5.6 and 5.7, leads to the conclusion that the *Corner Score* is always faster on the *DSP*.

Additionally, the *Abort Criterion* on *DSP* underbids its counterpart for both pictures by more than 50% regardless of the threshold, because the *Abort Criterion* seems to fit very well into the *DSP*'s architecture and it is not dependent on the threshold. The *Abort Criterion*'s implementation uses many logical instructions, which can be processed by four subunits on the *DSP* resulting in a very fast execution, although the implementation includes a multiplication and the vector contains four times less elements than a *NEON* vector. Furthermore, it can

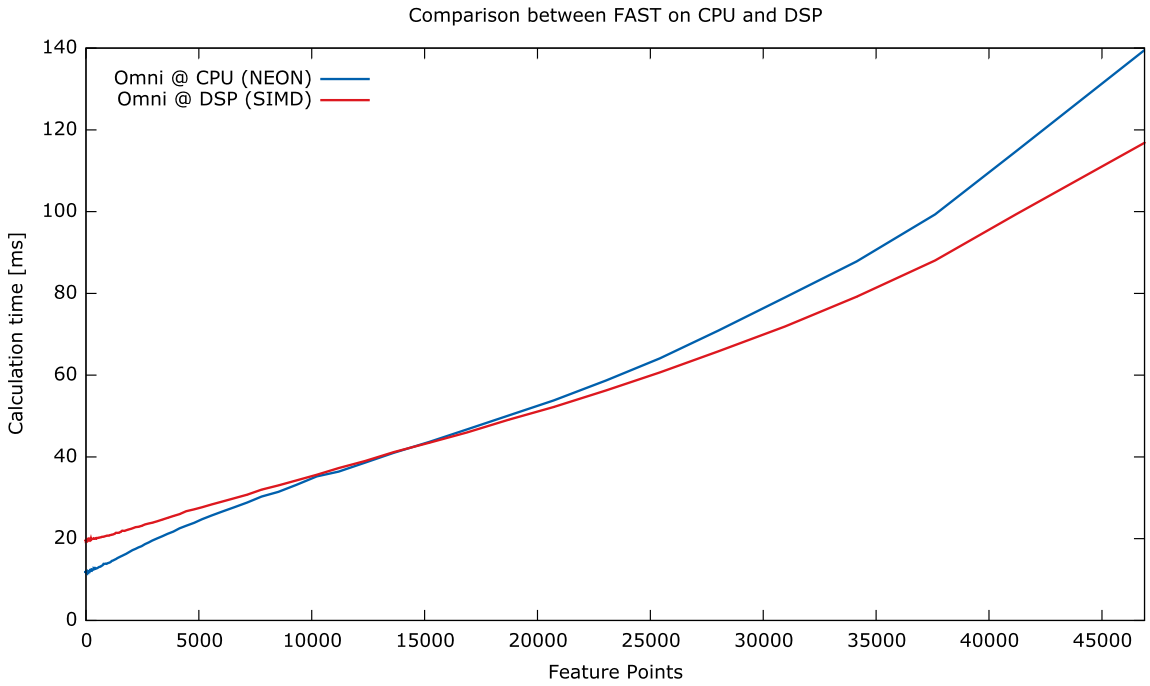

Figure 5.5: Comparison of execution times on *CPU* and *DSP* based on test image *Omni*.

profit from the implicit generation of the bit-mask compared to *NEON*. Refer to Chapter 4.2 for more information about the bit-mask on the *DSP*.

For a decreasing number of detected corners the decision maker for the *CPU* is the efficiency of the *Vector-Detection*, which compensates the low performing *Abort Criterion* and the declining fraction of the *Corner Score* calculation. The *Vector-Detection* can be so fast on the *CPU* because of its simple vector length and the resulting data throughput.

Considering realistic thresholds, the limiting factor is the *Vector-Detection* on the *DSP* and the *Abort Criterion* on the *CPU*, which should be addressed first in case of further optimizations.

Power Consumption Measurement

Although the *CPU* and *DSP* are on the same chip, the different usage and hardware implementation might influence the overall power consumption, which is a crucial factor on battery powered devices. To compare both implementations, a possibility to measure the power consumption as close to the *SoC* was needed and found. The *Beagleboard-xM* features a two-pin header for current measurements of the *SoC* [4], at which an oscilloscope was connected via a differential amplifier. On these pins the voltage drop over a 0.1 Ω resistor in the power supply can be measured. In theory the consumed energy would be in proportion to the integral of

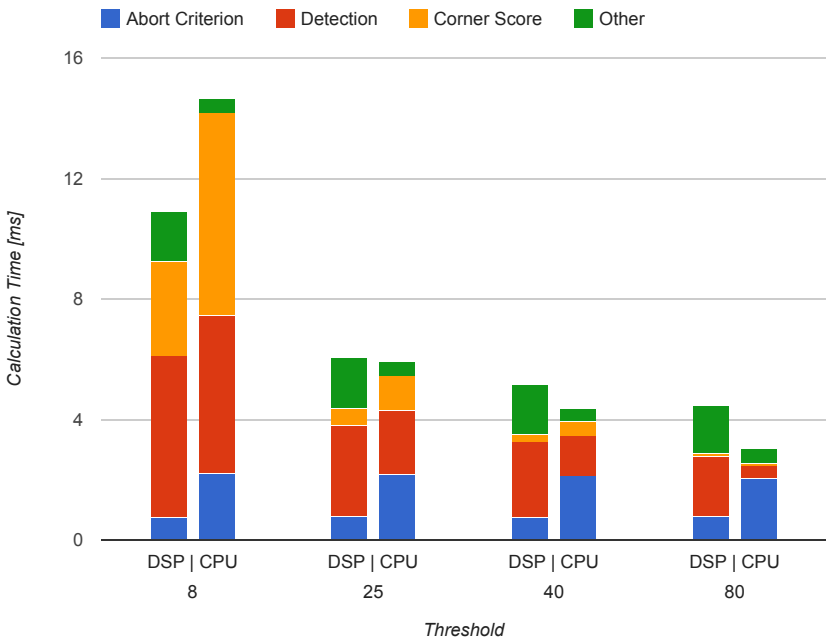

Figure 5.6: Absolute time consumption of test image *Lena* at different thresholds.

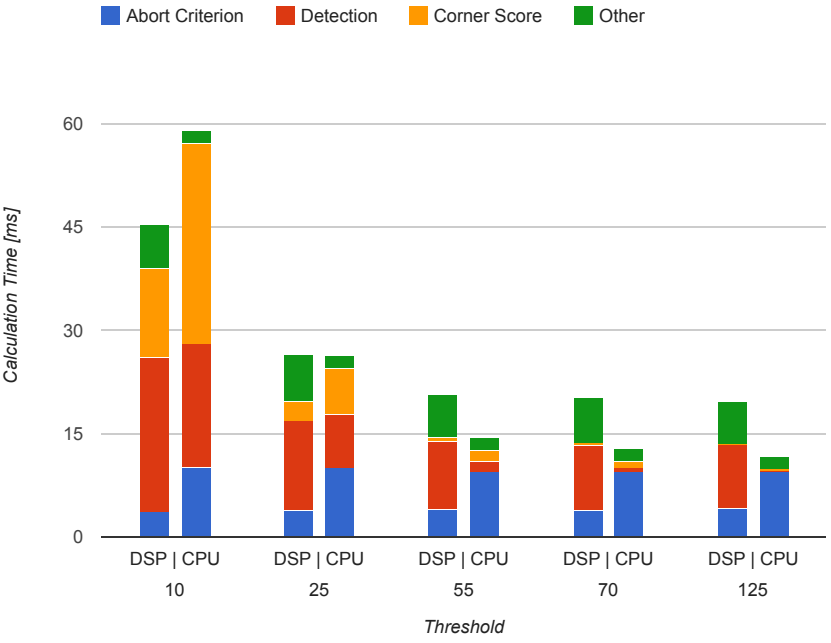

Figure 5.7: Absolute time consumption of test image *Omni* at different thresholds.

the measured voltage drop. To indicate the start and the end of the algorithm, the code was modified to toggle a *GPIO*-pin, which was used to trigger the measurement. Of course no exact values were supposed to be the results of this measurement, also because of missing equipment with a higher resolution, but a visual comparison is possible. In Figures 5.8 and 5.9 the image *Omni* was processed at a threshold of 20, which results in a longer execution time and a better measurement. The yellow rising edge displays the start of the stream-relevant part of the implementations and the falling edge represents the end. The blue graph is the measured voltage drop after the difference amplification with factor 20.

The results from this measurements show that less energy is consumed when executing the corner detection on the *CPU*. This can be explained by a shorter execution time and by the fact that the *DSP* must be waken up from a power-saving state, which cannot be compensated by the *CPU* through saving the same amount of energy at the same time.

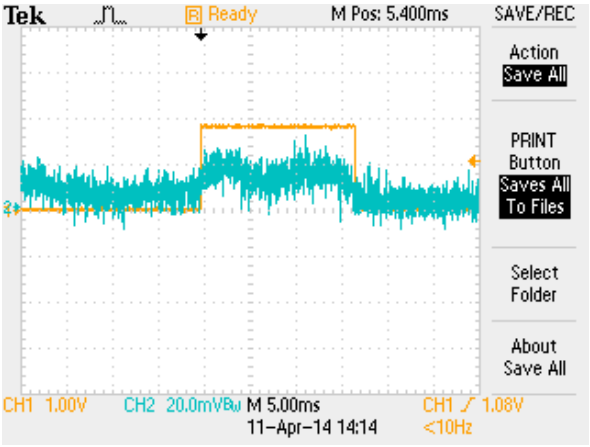

Figure 5.8: Power consumption measurement of image *Omni* on *CPU*.

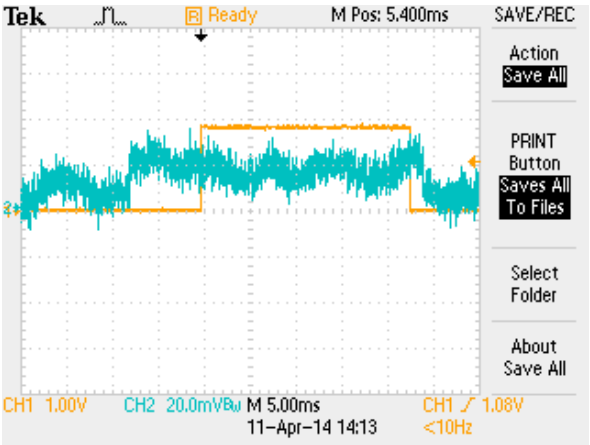

Figure 5.9: Power consumption measurement of image *Omni* on *DSP*.

6 Conclusion and Outlook

6.1 Conclusion

The *FAST* feature detector was successfully ported to the *TI DaVinci DM3730 SoC* platform consisting of an *ARM Cortex-A8* core and a *TMS320C64x+ DSP*. The algorithm was further examined and optimized to fulfill the needs of mobile devices and battery powered robots, trying to archive a low performance footprint to be able to run other important tasks on the same system and to reduce the overall power consumption.

Before optimization a framerate of 54 images per second (fps), at a resolution of 1280 x 960, would have been possible on the *CPU* with the *NEON* basic port, but now 78 fps, 30% more, are reachable. When outsourcing the algorithm to the *DSP* 50 fps are possible with the optimized implementation. When looking at lower resolutions, the maximum framerate would have been 184 fps at a resolution of 512 x 512, which was increased by 25% up to 248 fps on the *CPU* and up to 200 fps on the *DSP*.

Because the *CPU* is only occupied for about a fifth of the whole processing time when running the detection on the *DSP*, it is advisable to transfer the calculation to the *DSP* if the co-processor is available and the marginally reduced processing speed can be accepted. In this case, the *CPU* is free and available for other important tasks in the system. This of course results in an increased power consumption of the system, because a second calculation unit is running. If the system is not running on its limits and power consumption is preferred over the additional processing resources, then the optimized *NEON* implementation should be the choice.

This, once again, shows that optimization is important and necessary on efficient embedded devices, and often the result outweighs the invested development time and effort. Of course not every algorithm can gain from *SIMD* and parallel executed instructions and there are a lot of difficulties along the way. Time-critical parts have to be spotted, sections of the algorithm have to be parallelized, tests and analyses have to be made to evaluate the optimizations. But not every time the whole optimization process down to hand written assembler has to be undergone, as for example in the *FAST* algorithm usage of intrinsics generated about as good output as the hand written inline assembler.

In the context of this work, additionally a census descriptor was implemented and optimized for the *NEON* architecture by the author for the *DLR* to make up a fully applicable module, which is able to cover the first two steps of the three steps of *detection - description - matching*. A short summary and description is included in Appendix A.1.

6.2 Outlook

At the time of writing a new generation of multicopters is designed at the *DLR*, for which vision based sensors will be even more important than in preceding models. Time will tell whether and where exactly they will use the *FAST* detector in one of the sensors, or for one of their tasks in the subject of navigation, tracking, object recognition or other purposes relying on optical information.

The fully tested and optimized implementation can now be used on *UAVs* or other (mobile) devices featuring an *ARM* processor providing the *NEON* instruction set or a *SoC* with the on-chip *DSP*. Even applications in the field of augmented reality are possible on embedded devices as phones or tablets. To make it publicly available and to give something back to the open source community, an attempt will be made to supply the optimized *NEON* implementation back to the *OpenCV* library.

For future works, of course the third step of *matching* could be a point of interest, or the integration of the implementations into the robot operating system (*ROS*) [17], which is widely used among the multicopters.

In case of additional investigations on optimizations for the *FAST* feature detector the following points should be addressed first: The part with the highest time consumption for the case of $300 - 500$ detected corners per image is the *Vector-Detection* on the *DSP* and the *Abort Criterion* on the *CPU*. An idea would be to skip the second run of the *Abort Criterion* in case of a half-vector step, because the vector was not completely rejected in the first pass and therefore there must already be a possible corner in the first half of the vector. This would, of course, require some additional code, but the whole second run could be omitted.

Bibliography

[1] ARM Ltd.: *Architecture and Implementation of the ARM Cortex-A8 Microprocessor*. https://pixhawk.ethz.ch/_media/software/optimization/neon_whitepaper.pdf 2.1.1

[2] ARM Ltd.: *ARM NEON Instruction Summary*. http://infocenter.arm.com/help/topic/com.arm.doc.dui0489c/CJAJIIGG.html 2.1.1

[3] ARM Ltd.: *Introducing NEON™ Development Article*. http://infocenter.arm.com/help/topic/com.arm.doc.dht0002a/DHT0002A_introducing_neon.pdf 2.1.1

[4] BeagleBorad.org, Coley G.: *BeagleBoard-xM System Reference Manual Rev C.1.0*. http://www.beagleboard.org/static/BBxMSRM_latest.pdf 5.3

[5] Bradski, G.: OpenCV. In: *Dr. Dobb's Journal of Software Tools* (2000) 3

[6] Daoust, Yves: *SSE _mm_movemask_epi8 equivalent method for ARM NEON*. http://stackoverflow.com/a/12383618 3.2, 3.1

[7] Edler, Jan ; Hill, Mark D.: *Dinero IV Trace-Driven Uniprocessor Cache Simulator*. http://www.cs.wisc.edu/~markhill/DineroIV/ 4.1.3

[8] Graham, Susan L. ; Kessler, Peter B. ; McKusick, Marshall K.: *gprof: a Call Graph Execution Profiler*. 1982 4.1.1, 5.1

[9] Gumstix Inc.: *Overo® FireSTORM COM Overview*. https://store.gumstix.com/index.php/products/267/ 2

[10] Hassaballah, M. ; Omran, Saleh ; Mahdy, Youssef B.: A Review of SIMD Multimedia Extensions and Their Usage in Scientific and Engineering Applications. In: *Comput. J.* 51 (2008), November, Nr. 6, 630–649. http://dx.doi.org/10.1093/comjnl/bxm099. – DOI 10.1093/comjnl/bxm099. – ISSN 0010–4620 2.1.1

[11] Heinrichs, Matthias ; Hellwich, Olaf ; Rodehorst, Volker: *Robust Spatio-Temporal Feature Tracking* 5.1, A.1.1, A.1.2

[12] Jeong, Kanghun ; Moon, Hyeonjoon: Object Detection Using FAST Corner Detector Based on Smartphone Platforms. In: *Proceedings of the 2011 First ACIS/JNU International Conference on Computers, Networks, Systems and Industrial Engineering*.

Washington, DC, USA : IEEE Computer Society, 2011 (CNSI '11). – ISBN 978–0–7695–4417–5, 111–115 1.1

[**13**] Klippenstein, J. ; Zhang, Hong: Performance evaluation of visual SLAM using several feature extractors. In: *Intelligent Robots and Systems, 2009. IROS 2009. IEEE/RSJ International Conference on*, 2009, S. 1574–1581 1.1

[**14**] Mair, Elmar ; Hager, Gregory D. ; Burschka, Darius ; Suppa, Michael ; Hirzinger, Gerhard: Adaptive and Generic Corner Detection Based on the Accelerated Segment Test. In: *Proceedings of the European Conference on Computer Vision (ECCV'10)*, 2010 1.2, 2.2

[**15**] Majumder, Goutam ; Bhowmik, Mrinal K. ; Bhatacharjee, Debotosh: Automatic Eye Detection Using Fast Corner Detector of North East Indian (NEI) Face Images. In: *Procedia Technology* 10 (2013), Nr. 0, 646 - 653. http://dx.doi.org/http://dx.doi.org/10.1016/j.protcy.2013.12.406. – DOI http://dx.doi.org/10.1016/j.protcy.2013.12.406. – ISSN 2212–0173. – First International Conference on Computational Intelligence: Modeling Techniques and Applications (CIMTA) 2013 1.1

[**16**] Moreels, Pierre ; Perona, Pietro: Evaluation of Features Detectors and Descriptors based on 3D Objects. In: *International Journal of Computer Vision* 73 (2007), Nr. 3, 263-284. http://dx.doi.org/10.1007/s11263-006-9967-1. – DOI 10.1007/s11263–006–9967–1. – ISSN 0920–5691 1.1

[**17**] Quigley, Morgan ; Conley, Ken ; Gerkey, Brian ; Faust, Josh ; Foote, Tully B. ; Leibs, Jeremy ; Wheeler, Rob ; Ng, Andrew Y.: ROS: an open-source Robot Operating System. In: *ICRA Workshop on Open Source Software*, 2009 6.2

[**18**] Reithmeier, Michael D.: *Evaluation of the Ti DaVinci DM3730 Coprocessors focusing on Image Processing Algorithms in order to disburden the CPU.* September 2012 2.1, 2.1.2

[**19**] Rosten, E.: libCVD - computer vision library. (2010), May 3

[**20**] Rosten, Edward ; Drummond, Tom: Fusing points and lines for high performance tracking. In: *IEEE International Conference on Computer Vision* Bd. 2, 2005, 1508–1511 1.2, 2.2, 2.3, 3.1.2

[**21**] Rosten, Edward ; Drummond, Tom: Machine learning for high-speed corner detection. In: *European Conference on Computer Vision* Bd. 1, 2006, 430–443 1.2, 2.3, 2.2, 3

[**22**] Rosten, Edward ; Porter, Reid ; Drummond, Tom: FASTER and better: A machine learning approach to corner detection. In: *IEEE Trans. Pattern Analysis and Machine*

Intelligence 32 (2010), 105–119. http://dx.doi.org/10.1109/TPAMI.2008.275. – DOI 10.1109/TPAMI.2008.275 1.1, 2.2, 3.1.1

[23] Smith, S. M. ; Brady, J. M.: SUSAN - A New Approach to Low Level Image Processing. In: *International Journal of Computer Vision* 23 (1995), S. 45–78 2.2

[24] Stewart, Matt: *Stewart Vision; CMU*. http://www.contrib.andrew.cmu.edu/~mstewar1/Source%20Code/stewart_vision.c 3

[25] Szeliski, Richard: *Computer Vision: Algorithms and Applications*. 1st. New York, NY, USA : Springer-Verlag New York, Inc., 2010. – ISBN 1848829345, 9781848829343 1.1

[26] TI, Texas Instruments Inc.: *TMS320C64x Technical Overview*. http://www.ti.com/lit/ug/spru395b/spru395b.pdf. Version: January 2001 2.1.2

[27] TI, Texas Instruments Inc.: *TMS320C64x/C64x+ DSP CPU and Instruction Set Reference Guide*. http://www.ti.com/lit/ug/spru732j/spru732j.pdf. Version: July 2010 2.1.2

[28] TI, Texas Instruments Inc.: *DM3730, DM3725 - Digital Media Processors*. http://www.ti.com/lit/ds/sprs685d/sprs685d.pdf. Version: July 2011 2.1

[29] TI, Texas Instruments Inc.: *TMS320C6000 Programmer's Guide*. http://www.ti.com/lit/ug/spru198k/spru198k.pdf. Version: July 2011 2.1

[30] TI et al.: *TI Wiki: AM/DM37x Overview*. http://processors.wiki.ti.com/index.php/AM/DM37x_Overview 2.1

[31] Uhrig, Jonas B.: *Integration and Development of DSP Algorithms for the ROS Environment*. October 2013 2.1.2

[32] Weidendorfer, Josef ; Kowarschik, Markus ; Trinitis, Carsten: A Tool Suite for Simulation Based Analysis of Memory Access Behavior. In: *In Proceedings of International Conference on Computational Science*, Springer, 2004, S. 440–447 5.1

[33] Xiao, Ming ; Li, Wanli ; Hu, Tianjiang ; Pan, Liang ; Shen, Lincheng ; Bu, Yanlong: SAR aided navigation based on FAST feature. In: *Software Engineering and Service Science (ICSESS), 2013 4th IEEE International Conference on*, 2013. – ISSN 2327–0586, S. 861–864 1.1

[34] Zabih, Ramin ; Woodfill, John: Non-parametric local transforms for computing visual correspondence. Version: 1994. http://dx.doi.org/10.1007/BFb0028345. In: Eklundh, Jan-Olof (Hrsg.): *Computer Vision — ECCV '94* Bd. 801. Springer Berlin Heidelberg, 1994. – DOI 10.1007/BFb0028345. – ISBN 978–3–540–57957–1, 151-158 A.1.1

Ich, Peter Fink, Matrikel-Nr. 924547, versichere hiermit, dass ich die vorliegende Arbeit mit dem Thema

> *»Optimized Implementation of a Feature Detector for Embedded Systems Based on the 'Accelerated Segment Test'«*

selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, wobei ich alle wörtlichen und sinngemäßen Zitate als solche gekennzeichnet habe. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Issing, den 17. April, 2014

Peter Fink

A Appendix

A.1 Census Descriptor

A.1.1 Binary Census Descriptor

"A very robust patch representation, the Census Operator, was introduced by Zabih and Woodfill [34]. [...] The Census Transform [...] is a non-linear transformation which maps a local neighborhood surrounding a pixel [...][p] to a binary string representing the set of neighboring pixels[' intensities]" [11]. Originally it mapped eight surrounding pixels darker than the nucleus to the digit 1 and brighter ones to 0, but this was changed in the implementation to improve understandability. This change does not matter in case only this implementation is used and no compatibility to other implementations must be ensured, but it should be kept in mind for such cases. The implementation now maps pixels to signature S as follows:

$$S_x = \begin{cases} 0, & \text{for } I_x \leq I_p - t \qquad (\textit{darker or equal}) \\ 1, & \text{for } I_x > I_p + t \qquad (\textit{brighter}) \end{cases}$$

Figure A.1 shows an example and also illustrates the ordering of the pattern pixels in the signature vector, which also differs from the original. It was changed due to the vectorization. The signature vector is a 8-bit integer that can now be used to describe a feature point, which can then be matched against other feature points e.g. in the following images or the like.

The implementation was optimized to use *NEON* vectors again. There were two possibilities of vectorization: Firstly, process one descriptor with means of *SIMD* instructions, or second,

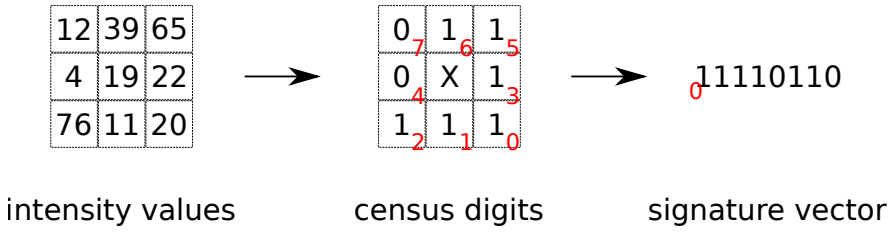

Figure A.1: Census descriptor (binary).

collect data from multiple feature points, because they are not stored consecutively in memory, and process multiple descriptors at once. Both implementations were created and tested, and the second version performed better, although the data has to be collected in the first place.

A.1.2 Ternary Census Descriptor

Because it was said in [11], that the descriptor would profit from a ternary representation, a second version was created in the same way. Now pixels are mapped to signature S, consisting of two bits, as follows:

$$S_x = \begin{cases} 0, & \text{for } I_x < I_p - t & (darker) \\ 1, & \text{for } I_p - t \leq I_x \leq I_p + t & (similar) \\ 2, & \text{for } I_x > I_p + t & (brighter) \end{cases}$$

t represents a given threshold which pixels must exceed to be *brighter* or *darker*. Pixels in between are marked as *similar*. Figure A.2 shows the mapping to a 16-bit signature, which must be doubled to store the increased information.

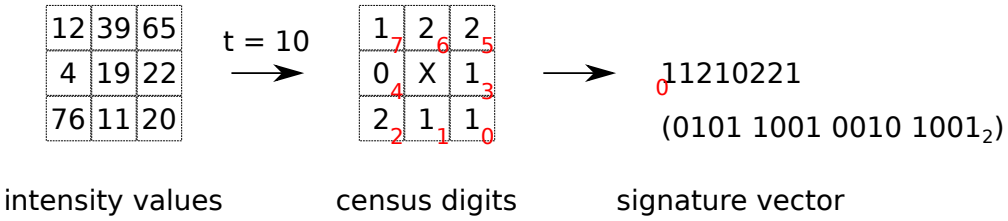

Figure A.2: Census descriptor (ternary).

A.2 Content of the CD

The attached CD contains additional content.
Besides the work in PDF format, all source files and compiled executables created in the context of this work are stored on the medium.

`./sources/`
: Source code directory

`./README`
: Information about the content on the CD

`./Thesis_Peter_Fink_924547.pdf`
: Thesis in PDF format