

Imperial College
London

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Automatic Code Generation and Pose Estimation on Cellular Processor Arrays (CPAs)

Author:
Thomas Debrunner

Supervisor:
Professor Paul Kelly

Submitted in partial fulfillment of the requirements for the MSc degree in
Computing (Artificial Intelligence) of Imperial College London

September 2017

Abstract

Modern image capturing and processing devices have relatively high power consumption and low frame rate. Cellular Processor Arrays (CPA) are new imaging devices with parallel Single Instruction Multiple Data (SIMD) computational capabilities built into every pixel. This enables massive pixel-parallel execution of image processing algorithms. Additionally, since CPAs are based on analogue technology, they consume less power compared with digital cameras. While rudimentary image processing algorithms have been shown on CPA, to the best of our knowledge, no one has so far ported higher level Computer Vision algorithms to CPA. This thesis presents novel methods for camera pose estimation and automatic filter kernel code generation on CPA.

Two main contributions are presented in this thesis. First, two 2DoF Visual Odometry algorithms based on a sum-of-absolute-difference approach are presented, along with a novel, tiling-based, 4DoF Visual Odometry system. The pose estimation algorithms have been shown to work at high frame rates up to 10,000 fps while consuming 1.23 W of power. The second and main contribution is the theoretic conception and implementation of an automatic code generation system that is capable of automatically generating programs to implement filter kernels on CPA hardware. A Viola-Jones based face detector has been implemented to show that one can port higher level Computer Vision algorithms to CPA.

Acknowledgments

I would like to offer my special thanks to the following people

- Dr Sajad Saeedi at Imperial College London for the great support in the development of this thesis
- Prof Paul Kelly at Imperial College London for the insightful discussions and supportive project supervision
- Dr Stephen Carey at the University of Manchester for providing me with a great experience at his lab
- Dr Wenbin Li at Imperial College London for providing me with quick renderings of synthetic datasets

Without their guidance and support, this thesis would not have been possible.

Contents

1	Introduction	1
2	Background	3
2.1	Cellular Processor Arrays (CPA)	3
2.1.1	Architecture	3
2.1.2	Performance	8
2.1.3	Software	9
2.1.4	Simulator	9
2.2	Visual Pose Estimation	9
2.2.1	Visual Odometry	10
2.2.2	Visual Simultaneous Localisation and Mapping (V-SLAM)	11
2.2.3	Random Sample Consensus (RANSAC)	11
2.2.4	Data Generation	11
2.3	Viola Jones Face Detection	12
3	Pose Estimation	14
3.1	Motivation	14
3.2	Applications	14
3.3	Conventions	15
3.4	Yaw & Pitch Estimation	16
3.4.1	Motivation	16
3.4.2	Approach	16
3.4.3	Implementation	19
3.4.4	Results	21
3.5	Yaw, Pitch, Roll, Z Estimation	29
3.5.1	Motivation	29
3.5.2	Approach	30
3.5.3	Implementation	32
3.5.4	Results	32
3.6	Scaling and Rotation	39
4	Automatic Kernel Code Generation	43
4.1	Motivation	43
4.2	Applications	44
4.3	Abstraction Levels	44
4.4	Contribution	45

4.4.1	Overview	45
4.4.2	Value Approximation	45
4.4.3	Filter Approximation	47
4.4.4	Set Notation	49
4.4.5	Goal Transformations	50
4.4.6	States	51
4.4.7	Graph Representation	52
4.4.8	State Transformation Plans	53
4.4.9	Reverse Splitting Algorithm	54
4.4.10	Pair generation	59
4.4.11	Non-Exhaustive Pair Generation and Heuristics	63
4.4.12	Computation Graph Relaxation	65
4.4.13	Register Allocation	71
4.5	Performance Evaluation	72
4.5.1	Heuristics vs. Exhaustive Search	74
4.5.2	Code for Well Known Filters	77
4.5.3	Comparisons with CPU and GPU implementations	82
4.5.4	Effects of Approximation	88
4.6	Face Detection	90
4.6.1	Motivation	90
4.6.2	Implementation	90
4.6.3	Results	95
4.6.4	Practicality on Current Hardware	100
5	Conclusion	101
5.1	Contributions	101
5.1.1	Pose Estimation	101
5.1.2	Automatic Code Generation	102
5.2	Future Work	103
5.2.1	Pose Estimation	103
5.2.2	Automatic Code Generation	103
5.2.3	Hardware	104

Chapter 1

Introduction

Conventional image processing systems consist of multiple distinct hardware components. The image gets captured by a digital camera before being transferred over a bus to the systems main memory. It is only after the image has been transferred that image processing algorithms can be applied. These algorithms usually run on CPUs, special purpose signal processors and on more parallel hardware architectures like GPUs or FPGAs.

While contemporary hardware bus and memory systems are easily capable of performing this task in real-time, data rates become a problem at high frame rate applications. For example, a 256×256 sensor, with 8 bits of resolution per pixel would require a bus capable of a data rate of 6.5 GB/s (Carey et al. (2012)), running at 100,000 fps. Simpler image processing algorithms like filtering can generally be done in real-time. However, there are more complex algorithms like object detection, convolutional neural networks or Simultaneous Localisation and Mapping (SLAM) algorithms that do not scale to real-time performance or require complex, energy intensive hardware.

This energy overhead especially poses difficulties to implement such systems into embedded, energy-limited devices. Cellular Vision Chip, such as the ACE400 (Dominguez-Castro et al. (1997)), ACE16K (Linan et al. (2002)), MIPA4K, (Poikonen et al. (2009)) and the various iterations of the SCAMP chip (Dudek and Hicks (2005), Dudek (2005), Dudek (2003), Carey et al. (2013)) try to address this problem by shifting image processing from a dedicated processing unit onto the image sensors focal plane.

Since image processing is done right on the chip in a pixel parallel fashion, the vision sensor can output preprocessed abstract information. This information can take on the form of position of features, subsampled flow vectors, pixels that changed etc. Since full video frames do not need to be transferred the requirements on the bus and memory system get relaxed. This can significantly reduce power consumption.

Unlike traditional computers, most of the chips mentioned above store the captured and intermediate images as analogue instead of digital values. Arithmetic operations are carried out directly on analogue values. The reasoning behind this is an effort to increase speed while keeping area and power dissipation at a minimum as argued by (Dudek and Hicks (2000))

In this thesis, we are mainly going to focus on the latest iteration of the SCAMP

chip, the SCAMP-5 (Carey et al. (2013)).

The thesis has two main focuses, the first one being pose estimation (Visual Odometry). In Visual Odometry, we try to estimate the camera's ego motion solely based on analysing a stream of video frames. To be usable in a real-world application, these systems have to work in real-time. Generally, we are not interested in storing the video frames for later use. Estimating the pose right on the image sensor and only reporting the pose vectors to the host system can greatly reduce the hardware requirements on bus, memory and host processor. This makes the problem especially well-suited to be implemented on cellular processor arrays.

The second main focus of this thesis lies in the fact that cellular processor arrays have a vastly different design than ordinary CPUs and GPUs. Having large number of processors with only very rudimentary capabilities in operations and I/O, requires a very different style of programming. It turns out that a lot of image processing algorithms can be reduced to the application of convolutional kernels. In the meantime, performing an arbitrary convolutional kernel on the cellular processor array requires significant effort in coding and optimisation. The second and most important part of this thesis is devoted to the automatic generation of fast, correct code that implements the desired convolutional kernel. A novel algorithm has been proposed that works backwards from the desired result to the initial state in order to find a good program for the kernel. Furthermore, an application of the code generation algorithm is shown by the implementation of a Viola and Jones (2001) based face detector.

The rest of the thesis is organised as follows:

1. **Chapter 1, Introduction** contains a short overview of the chapters to come.
2. **Chapter 2, Background** introduces all the relevant concepts required to understand the rest of the thesis, as well as points the reader into the right direction to find further resources.
3. **Chapter 3, Pose Estimation** Introduces two novel 2DoF Visual Odometry algorithms designed for CPA hardware. In addition to that, a novel tiling-based four degrees of freedom algorithm is presented and evaluated.
4. **Chapter 4, Automatic Kernel Code Generation** presents a concept and an algorithm to automatically generate code to perform convolutional filters on cellular processor arrays. A demo application showing a face detector, similar to the one by Viola and Jones (2001), is presented.
5. **Chapter 5, Conclusion** summarises all the previous results, draws a conclusion and presents opportunities for future.

Chapter 2

Background

2.1 Cellular Processor Arrays (CPA)

Programmable Focal-Plane processors have been around for a long time. While there was theoretical interest in the topic, real implementations of processors on image sensors did not appear until the mid 1990s. For example, Dominguez-Castro et al. (1997) describe in their paper a CMOS chip capable of storing and processing four binary images. The chip, called the ACE400 is a 20×22 processor array modelled after Cellular Neural Network (CNN) Universal Machine (Chua and Yang (1988)).

Interesting applications using the ACE400 chip were shown by Zarandy et al. (2002), with one being the detection and classification of simple objects printed on a rotating ring at 10,000 fps. The other application involved the analysis of single sparks from a car spark plug. The unconventionally high frame rate allowed them to capture around seven frames per spark to properly analyse its properties.

A different approach was taken by Dudek and Hicks (2000), first describing a single, analogue, sampled current microprocessor, leading to the SCAMP vision chip (Dudek and Hicks (2005)). Other than the CNN based chips (Chua and Yang (1988)), which use parallel, hard wired convolution templates for most operations, the SCAMP consists of a grid of analogue general-purpose processors operating in a single instruction, multiple data (SIMD) manner. Despite having speed disadvantages, Dudek (2004) argues that the sequential SIMD processor is a better practical choice, due to smaller circuit area and better achievable accuracy.

Multiple versions of the SCAMP chip have been implemented, starting from a very small 21×21 prototype (Dudek and Hicks (2001)) to the more recent 256×256 array (Carey et al. (2013)). All implementations share a very similar architecture which is briefly outlined in the sections to come. As this thesis is mainly based on SCAMP chip, the subsequent background sections focus on SCAMP.

2.1.1 Architecture

This section is about the architecture of the SCAMP-5 chip, which is the device used for this study.

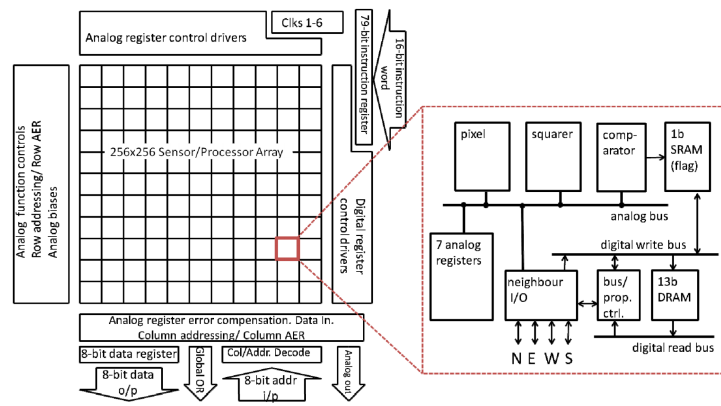


Figure 2.1: Architecture of the 256×256 SCAMP vision chip (Carey et al. (2013)). The chip consists of an array of 256×256 processing elements, each one assigned to exactly one pixel, along with support and driving circuitry. The architecture of a single processing element is depicted in figure 2.2

Carey et al. (2013) describes a chip consisting of a square array of 256×256 analogue processing units (PE) along with control and readout circuitry to drive the chip. On this chip, a single instruction stream is distributed to all processing elements simultaneously. This means that, all processing elements execute the same instruction at the same time, on their local data. The algorithm that is performed on the system is thus defined by a sequence of (79-bit) instruction words, issued by the system controller (Carey et al. (2013)).

Analogue Processing Elements (PE)

Figure 2.2 Shows the detailed architecture of a single processing element. All PEs incorporate seven S²I analogue sampled current registers, 13 single bit digital DRAM registers and one SRAM single bit FLAG register (Carey et al. (2011)).

Carey et al. (2011) states that the analogue subsystem is closely related to the previous SCAMP3 chip, described in Dudek (2005). In this design, all analogue registers are connected to a common analogue bus, where data is represented as currents. An advantage of this analogue design is that arithmetic operations can be performed directly on the bus by current summation and division (Dudek (2005)). Using this technique, negation, addition, subtraction and division by a small integer number can be performed without the need for additional, dedicated, ALU hardware. Division by a small integer factor can be achieved by simultaneously loading the current from one register into multiple target registers, splitting the total current into multiple equal parts (Dudek (2003)).

The local analogue bus is connected to the output of the special purpose NEWS register of the four adjacent processors (north, east, south, west), enabling it to pass analogue values between cells (Dudek (2005)). Access to an analogue value of a neighbouring cell is thus a two step process, which involves copying the data from the source register into the NEWS register first, before copying the NEWS register of the neighbouring cell to the target register. According to Carey et al. (2011),

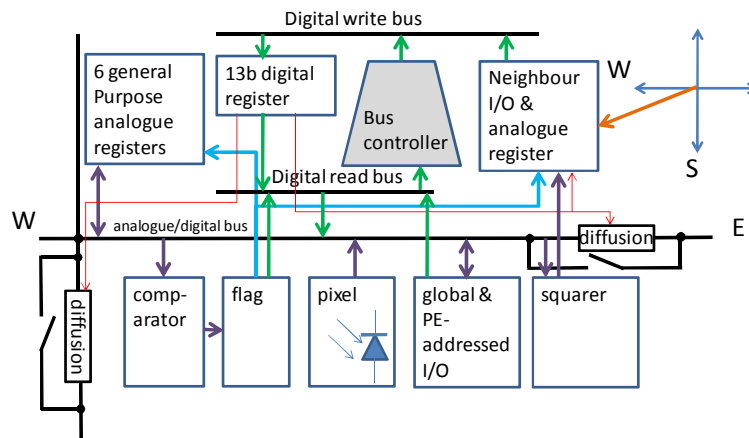


Figure 2.2: Architecture of the processing element (Carey et al. (2012)). Each processing element contains 6 general purpose analogue register plus one special purpose analogue register for neighbour communication. There is an analogue photodiode system to capture new frames, as well as connections to global I/O. Note that, apart from the squarer and comparator, the cell does not contain any analogue arithmetic unit, as operations are carried out by current summation/division on the bus.

a double register transfer (transfer a value into another register and back) should not involve an error larger than 0.25% of the maximum signal value. Furthermore, the PEs have capabilities for the fast execution of low pass filters (diffusion) as well as for asynchronous trigger wave propagation on the digital registers (Carey et al. (2011), Carey et al. (2012)).

Every PE contains a photo detector circuit with near linear characteristics, which allows the system to directly store the current from the photo detector to a register (Dudek (2005)). Doing this on all processing elements simultaneously captures a photo with every processing element exactly holding one pixel of the acquired image.

The FLAG register provides local autonomy by inhibiting any data to be written when the register is not set (Dudek (2005)). The comparator is used to set the FLAG register, by deciding if a current on the analogue bus is positive or negative (Dudek (2005)). The 13 general purpose digital registers and the FLAG register can be read out onto the Local Read Bus (LRB), where they perform a logical nor operation of all the source bits selected at the same time (Carey et al. (2011)). This value then gets written back, either direct or inverted to the registers via the Local Write Bus to the register bank. Carey et al. (2011) also states that, unlike it was the case with the analogue registers, the LWB is directly multiplexed to the digital output of the neighbouring cells, allowing single cycle digital communication with the neighbours. The digital system thus provides intrinsic capabilities for not, nor and or logical functions in a single cycle (Carey et al. (2011)).

Control System

A sequence of Instruction Code Words (ICW) dictates the algorithm that is performed on the chip, whereas every PE executes the same instruction on its own data stream

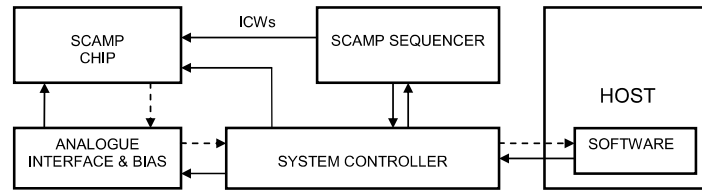


Figure 2.3: SCAMP system components (Barr et al. (2006)). The SCAMP chip does not contain any control logic by itself, it executes the instruction stream received by the SCAMP sequencer. The actual control of the system is performed by the system controller which sends the appropriate instructions to the SCAMP sequencer as it encounters them in the program. The algorithm shown in this table shifts the contents of the analogue register *A* 10 pixels to the right.

Line #	Instruction	Target Processor	Comment
23	...		
24	<code>A = B</code>	SCAMP	Copy contents of B into A
25	<code>_load(s0, 0)</code>	Controller	Initialize sequencer register <i>s0</i> to 0
26	<code>NEWS = A</code>	SCAMP	Copy contents of A into NEWS
27	<code>A = WEST</code>	SCAMP	Copy western NEWS register to A
28	<code>_add(s0, 1)</code>	Controller	Add 1 to <i>s0</i>
29	<code>_compare(s0, 10)</code>	Controller	Compare <i>s0</i> with 10
30	<code>_jump(c, 26)</code>	Controller	if <i>s0</i> < 10, jump to instruction 26
31	...		

Table 2.1: Example instruction sequence (Barr et al. (2006)). SIMD SCAMP instructions that operate on the analogue processor array arrive in conjunction with standard instructions sequential instruction for the controller processor. The controller sends the SCAMP instructions to array via sequencer. Whenever possible, SIMD instructions get glued to controller instructions to be issued in parallel.

(SIMD) (Barr et al. (2006)), if their FLAG bit is set.

The purpose of the control system is to control the, potentially non-linear, stream of instructions to the array processor, as well as to provide immediate arguments to and read out results from the array processor (Barr et al. (2006)).

According to Barr et al. (2006), ICWs for the processor array get issued by the SCAMP controller, which is implemented as a small, dedicated standard microprocessor. In this system, every SCAMP op-code consists of two parts glued together, with the first part of the instruction controlling the SCAMP array via the sequencer, while the latter part controls the controller itself. By following just its own instruction stream while issuing ICWs to the SCAMP array at the same time, the controller implicitly controls the algorithm performed on the array (Barr et al. (2006)).

Table 2.1 shows an example algorithm from Barr et al. (2006). The purpose of the algorithm is to shift the image in the analogue register *A* ten times to the right.

A SCAMP and a controller instruction always gets issued at the same time, if there is an instruction available at compile time. If not, the empty slots get filled with `nop`

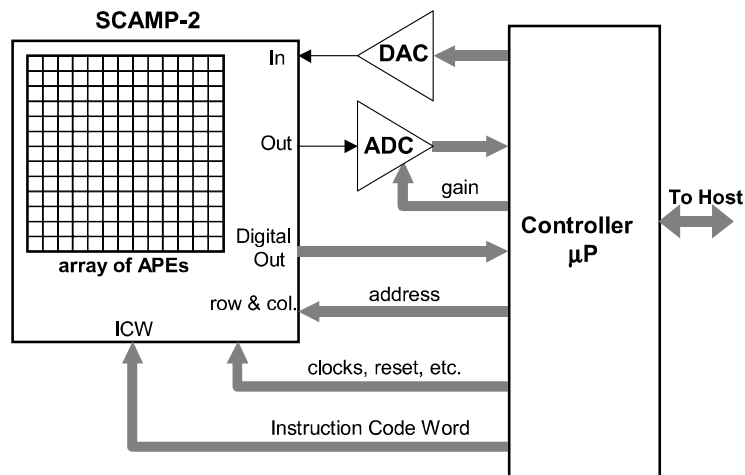


Figure 2.4: SCAMP-2 readout architecture (Dudek (2003)). The readout architecture of the SCAMP-5 mostly equivalent (Carey (2017)). The controller in this graphic includes the sequencer presented in Figure 2.3. The only means for transferring data to/from the controller is by means of sequential DA/AD converters. Pixels can get read out sequentially (by selecting one at a time) or as the sum of a group of pixels by selecting multiple. In this case, current summation takes place on the readout bus. The adjustable gain ADC has to be tuned accordingly.

statements (no operation) (Barr et al. (2006)). This ensures that the the controller and the array processor always operate in parallel, improving system performance.

Readout Architecture

The SCAMP controller can load data to and from the processor array by means of A/D and D/A converters for analogue values, and directly for digital values (Dudek (2003)).

Figure 2.4 shows the SCAMP readout architecture as shown in Dudek (2003). To read out a single cell, the controller selects the required cell and makes the cell output the current from the required register onto the global readout bus. This current then gets digitized in the global variable gain ADC (Dudek (2003)). The paper emphasizes that we can select multiple cells at the same time, whose currents then add up on the global readout bus. Together with the correct gain settings for the variable gain amplifier, we can perform local and global summation tasks constant time (Dudek (2003)).

The readout circuit has a flexible addressing system that facilitates selecting multiple cells of interest at the same time (Dudek (2003)). Row and column addresses can not only be set to specific values, but can also contain "don't care" bits, resulting in a bit pattern that can adapt to various addressing needs. Cells with row/column addresses that match the given pattern get selected (Dudek (2003)). There are four addressing modes of special interest: (Dudek (2003))

1. **Single PE** Full addresses, no don't cares.

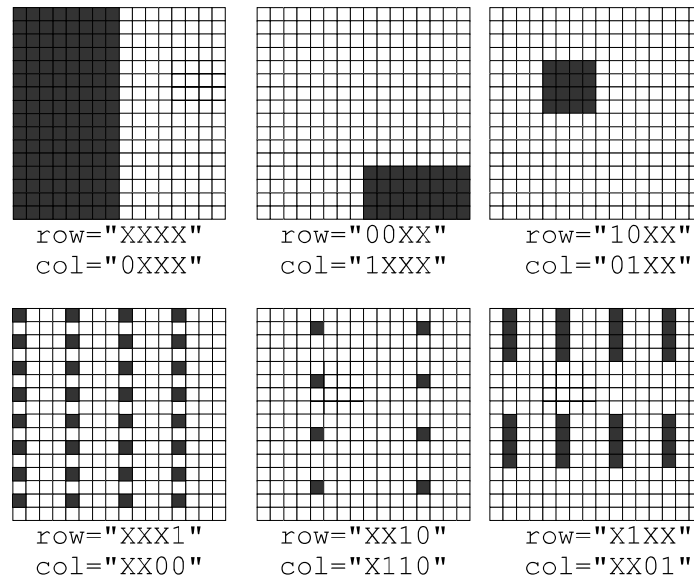


Figure 2.5: SCAMP addressing modes (Dudek (2003)). Cells shaded dark are selected (activated) cells. The image shows patterns that can be selected by setting the selection and don't care bits the right way. Half-frame and periodic selections can be performed trivially with this method.

2. **Full array** All bits don't care. Matches entire array.
3. **Block selection** Don't cares in the less significant bits.
4. **Periodic selection** Don't cares in the more significant bits.

Some addressing examples can be seen in **Figure 2.5**.

The SCAMP system has a way to load data into analogue registers by means of a DAC (Barr et al. (2006)). This input current from a single global DAC is distributed over the array to every PE (Dudek and Hicks (2005)). Every selected PE can then copy the value from the input bus to a local register.

2.1.2 Performance

The SCAMP-5 device, introduced in (Carey et al. (2013)), is rated at a clock frequency of 10 MHz, achieving a computational performance of up to 655GOPS at 1.9 pJ/OP. Carey et al. (2013) report a power consumption of 1.23 W (peak) and 0.2 mW (idle).

An application has been shown in Carey et al. (2012) in which the SCAMP-5 hardware manages to track an object on a rotating disc at 100,000 fps. According to the paper, tracking five objects on the disc at the same time reduces the frame rate to 25,000 fps. The solution in Carey et al. (2012) tracks the position of closed circles, which is very well-suited to the chip's asynchronous processing capabilities.

2.1.3 Software

Upon switching on the SCAMP device, a small program called the OS is loaded from the host computer onto the chip's memory (Barr et al. (2006)). This software initializes and calibrates the chip. The OS runs on the system controller, which controls communication to the host, the analogue hardware and ADC/DAC converters (Barr et al. (2006)). The OS can easily be extended or modified to change the behavior of the chip (Barr et al. (2006)). Drivers and C++ libraries are available for both Linux and Windows platforms (Barr et al. (2006))

2.1.4 Simulator

The paper (Barr and Dudek (2008)) introduces APRON, a cellular processor array simulation tool. Most work in this thesis was developed and tested on the APRON platform.

APRON provides a simulation and prototyping environment for general cellular processor arrays. The system can not only simulate the software on standard desktop computer hardware, but can also be used as a front-end to run the code directly on the target hardware (Barr and Dudek (2008)). APRON comes with a human readable programming language called APRON script. Programs written in this language get compiled by the APRON compiler to run on the simulator, or by a custom compiler to run on dedicated hardware (Barr and Dudek (2008)). The simulator also features a plugin interface to allow the developer to implement custom simulation behaviour to simulate specific hardware (Barr and Dudek (2008)). Plugins to simulate the behaviour of SCAMP-5 are published by (Chen (2016)). For this thesis, further plugins were developed, mainly for efficiently storing measurement values to files.

Error Model

Carey et al. (2013) describes an error model for a double register transfer (copy values from register A to B requires two transfers due to the inverting nature of the SI registers) as:

$$A_{i,j} = B_{i,j} + k_1 B_{i,j} + k_2 + \epsilon_{i,j}(t) + \delta_{i,j} \quad (2.1)$$

According to Carey et al. (2013), the fixed error k_2 can be easily compensated for in software. The paper assigns a value of $k_1 = 0.07$ for a register range of 0 to 100, an RMS value of 0.09 for the random error ϵ and an RMS value of 0.05 to the fixed pattern noise δ .

2.2 Visual Pose Estimation

This section summarises related work that has been developed for Visual Odometry (VO) (**Section 2.2.1**) and Simlutaneous Localisation And Mapping (SLAM) (**Section 2.2.2**). In later chapters of this thesis, algorithms are presented to solve the Visual Odometry task.

2.2.1 Visual Odometry

Visual Odometry (VO) is a method of estimating a camera's location and pose solely from analysing the video feed and a known initial location and pose. In the most general case, a VO system can estimate the camera's movement in the full physical six degrees of freedom space (Naroditsky et al. (2012)); however more restricted setups that assume a the cameras motion to be restricted to fewer dimensions often perform better in accuracy and still manage to capture all of the agents expected movements (Campbell et al. (2005)). VO systems can generally be divided into two groups, that is monocular and binocular systems. While binocular systems rely on the input of at least two cameras with known physical configuration, monocular systems have to rely on the input of a single camera. Stereo VO systems are capable to measure absolute distances to 3d objects in space by triangulation (Scaramuzza and Fraundorfer (2011)). Monocular systems on the other hand have to perform the entire task of estimation on 2D image data. This generally does not inform about the absolute scale. The absolute scale can only be recovered by knowing the dimensions of an object in the scene. Scaramuzza et al. (2009) show that there exist certain situations in which one can in fact estimate the absolute scale from a single camera. Most notably is the case where the camera is mounted onto a wheeled vehicle at an offset of its center of motion. Scaramuzza et al. (2009) show that as soon as the vehicle performs a turn, the absolute scale can be recovered via nonholonomic constraints.

One of the first implementations of a system similar to Visual Odometry was done by Moravec (1980) in 1980. The system used a single camera on a horizontal sliding rail to capture images of the environment from two different angles. The robot would operate in a stop and go fashion, capturing 9 images from different angles, extracting features and correlating them in every stop. Matthies and Shafer (1987) extended the same approach with a better error estimation model. Pioneering results were shown in Nistér et al. (2004), utilising a Random Sample Consensus (RANSAC, Fischler and Bolles (1981)) approach to filter out outliers. Nistér et al. (2004) also came up with the term "Visual Odometry" for the first time, as an analogy to wheel odometry. More recent approaches in feature tracking based algorithms and RANSAC are presented in the works by Cheng et al. (2005), implementing a VO system for the Mars Exploration Rover and Tardif et al. (2008), implementing a purely incremental monocular VO system. A different approach is followed by Milford and Wyeth (2008). Instead of tracking and comparing individual features, they used an appearance based method of comparing frames by a dense sum-of-absolute-difference method. A similar approach as Milford and Wyeth (2008) is followed in this thesis. Another appearance based approach was presented by Goecke et al. (2007) utilising the Fourier Mellin transform for a global full frame comparison. Yet another approach was followed in the paper by Corke et al. (2004), in which an approach based on sparse optical flow is presented. More recently, Steinbrücker et al. (2011) showed a real-time VO solution that also incorporates depth information from RGB-D sensors.

2.2.2 Visual Simultaneous Localisation and Mapping (V-SLAM)

Although most of the algorithms presented in this thesis belong in the domain of Visual Odometry (VO), a short summary of the works performed in a similar line of research, Visual Simultaneous Localisation and Mapping (V-SLAM) is presented here. While VO is incremental by nature with the goal to ensure local consistency over the last few frames, SLAM is concerned in achieving a global consensus over the whole trajectory (Scaramuzza and Fraundorfer (2011)). A SLAM algorithm constantly keeps track of a global map of the environment in order to detect if the agent returns to a previously visited location (loop closure). This is crucial to ensure global consistency (Scaramuzza and Fraundorfer (2011)). They also note that a VO system can be used together with loop closure detection and global optimisation to form a complete SLAM system.

One of the earliest contributions to the fields were presented in Smith et al. (1990) and Leonard and Durrant-Whyte (1991). These early contributions were generally incapable of running in real-time, vastly limiting their applicability to real world problems. They also sensors such as laser measurements. Real-time and vision-based, contributions came from Davison (2003) (MonoSLAM), Chiuso et al. (2000), Deans and Hebert (2005) as well as the very accurate Parallel Tracking and Mapping from Klein and Murray (2007). These systems were first limited to smaller indoor locations. Approaches that are capable of mapping larger scenes were presented by Clemente et al. (2007) as well as Mei et al. (2011). More recently, Newcombe and Davison (2010) showed a method to incorporate depth information measured from commodity depth sensors, such as the Microsoft Kinect into the SLAM problem (KinectFusion). This approach has been pushed forward by various researchers in recent years. Most notably are the works by Kerl et al. (2013) (DVO), Forster et al. (2014) (SVO), Engel et al. (2014) (LSD-SLAM), Mur-Artal et al. (2015) (ORB-SLAM) and Whelan et al. (2015) (ElasticFusion).

2.2.3 Random Sample Consensus (RANSAC)

At the heart of most contemporary VO and V-SLAM system is the Random Sample Consensus (RANSAC) algorithm introduced by Fischler and Bolles (1981). RANSAC is an iterative algorithm, that is capable of fitting a mathematical model under the existence of outliers (Saeedi et al. (2014)). It does this by random sampling of the measured data and assessing the quality of every sample subset it takes.

2.2.4 Data Generation

There are many datasets in existence used to verify the performance of Visual Odometry (VO) and SLAM systems, one of the most notable being the dataset by Sturm et al. (2012). The dataset contains real-world images, with ground truth recorded by higher frame rate motion capturing cameras. Another notable dataset is the ICL-NUIM dataset by Handa et al. (2014) from Imperial College. Unlike the set by Sturm et al. (2012), the ICL-NUIM dataset consists of renderings of synthetic environments.



Figure 2.6: Examples of two Haar-Like features applied to a face image. Evaluated feature is the pink sum of pixel values in pink area minus the sum of the pixel values in the blue area. Both features are weak classifiers for a face, with the first one representing the fact that the eye poerion is usually darker than the cheek portion, whereas the other represents a bright nose part with shadows on both sides.

ICL-NUIM

The ICL-NUIM datadatasetset (Handa et al. (2014)) is a collection of synthetic images, rendered along recorded, real-world trajectories. Two environments are modelled, one showing a quite densly populated office scene, while the other shows a more sparsly populated living room scene. The synthetic nature of the dataset ensures a perfect ground truth (Handa et al. (2014)). It also gives the ability to quickly render new trajectories in the same enviromnent. The *POVRay* (<http://povray.org/>) raytracing software is used to render the individual frames in photorealistic quality.

In this thesis, the original code by Handa et al. (2014) was used to render new trajectories in the same environments at higher CPA frame rates.

2.3 Viola Jones Face Detection

Face detection is classical problem in computer vision. One of the early contributions came from Kohonen (1989) using a neural network inspired approach to extract characteristic eigenvectors that describe a face. This approach was later extended by Kirby and Sirovich (1990) as well as Turk and Pentland (1991). Turk and Pentland (1991) coined the term "Eigenfaces" which is now commonly used for this method. An alternative, more computationally efficient method was presented by Viola and Jones (2001). They showed that a strong object detection algorithm could be combined from a series of weak classifiers, boosted using a method called *Adaptive Boosing* (Freund et al. (1999)) to form a strong classifier. Viola and Jones (2001) used so called *Haar-Like features* which are inspired by the Haar Functions introduced by hungarian mathematician Alfred Haar (Haar (1910)). In the context of object detection, the Haar-Like features reduce to partial sums of the pixels in the image.

Figure 2.6 shows two examples of such Haar-Like features. The feature is evaluated by taking the difference of pixel sums in parts of the image. Every Haar-Like feature forms a weak classifier for a face. Taken together as a boosted cascade, they form a strong classifier for faces. According to Viola and Jones (2001), one does not

have to apply all features to every position in the image, as one can reject locations early on, if the first weak classifiers return a negative result. Summations of pixel values is generally an expensive operation, with performance scaling proportionally to the size of the summed area. A contribution of the Viola and Jones (2001) is to use a alternative *integral image* representation of the image, allowing to compute arbitrary partial sums of the image in constant time. A more in-depth analysis of the algorithm is provided by Wang (2014).

Chapter 3

Pose Estimation

This chapter introduces three algorithms for camera pose estimation. Three different algorithms have been implemented in order to solve various pose estimation tasks. As it is considered an easier problem, the first contributions are two algorithms for 2DoF pitch and yaw estimation (**Section 3.4**). These algorithms then get extended to a solution for 4DoF estimation, adding roll and z -translation to the systems capabilities. (**Section 3.5**).

3.1 Motivation

In this section, we present pose estimation algorithms based on Visual Odometry (VO). In Visual Odometry, we estimate the pose of a camera solely based on its input video feed and a known initial pose. The estimation is then performed via integrating small incremental movements. Conventional VO systems generally require performant and energy intensive hardware. For example, Whelan et al. (2013) presents a robust Visual Odometry algorithm that achieves around 43 fps on a desktop GPU. It is expected that the massive parallelism of the cellular processor array could achieve much higher frame rates at a lower energy consumption. Furthermore, running at a much higher frame rate could naturally increase the system's robustness to drift as it integrates smaller incremental movements. This could also open up the possibility to use simpler algorithms using larger approximations, as the effects of approximations have less effect.

3.2 Applications

Visual Odometry (VO) systems found application mainly in robotics with the most notable example being planetary rovers (Lacroix et al. (1999), Olson et al. (2000)). Especially in GPS denied environments, such as indoor or underground locations, exact localisation of an agent becomes a harder problem. Additional to VO, there are other methods available, such as wheel odometry or inertial measurement units (IMUs). While wheel odometry can suffer from slippage in uneven terrain and IMUs

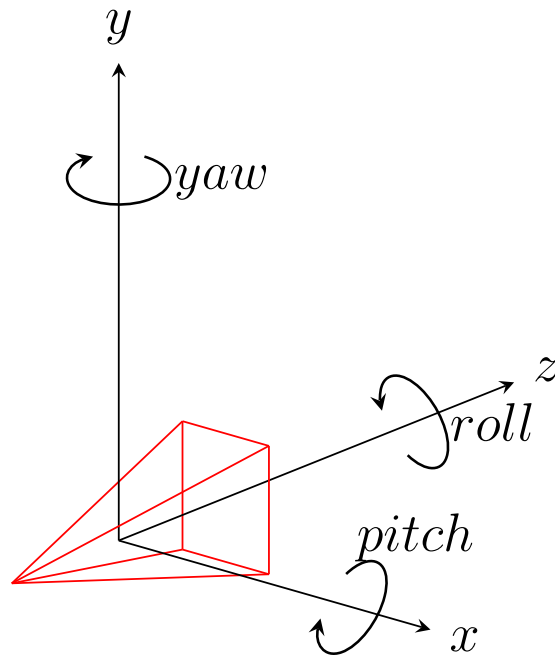


Figure 3.1: Conventions for the camera coordinate system chosen to be used in this thesis

can suffer from time-dependent driftage, VO presents itself as a valuable additional method to existing methods.

Compared to traditional VO approaches, it is expected that the algorithms implemented on the cellular processor array only have a very low power consumption. This makes the CPA implementation a good candidate to be used in very energy limited systems such as small robots or aerial vehicles.

3.3 Conventions

There are multiple conventions one can use in order to define Euler angles and coordinate systems, but in this thesis we are going to stick to the following convention. We assume a **left-handed** camera coordinate system with the camera pointing in the direction of the **positive z-axis**. Pitch, yaw and roll are therefore defined as rotations around the x , y and z axis respectively. **Figure 3.1** illustrates the chosen coordinates and the directions of the Euler rotations. The order of Euler rotations is chosen as **x-y-z** (pitch-yaw-roll).

3.4 Yaw & Pitch Estimation

3.4.1 Motivation

According to the definition above, a yaw and a pitch in the cameras coordinate system should result in a horizontal and vertical image shift. As shifting an image horizontally and vertically are intrinsic operations on the cellular processor array, it is thought to be an easier problem to estimate yaw and pitch compared to other degrees of freedom. However, a more complicated approach that also incorporates roll and z translation are discussed later on in the thesis.

3.4.2 Approach

Angular Velocity Estimation

The goal of this contribution is to provide a system that can estimate the angular velocities of the yaw and pitch rotations between two video frames, according to the cameras coordinate system. Knowing the initial pose of the camera, we can then estimate the camera's pose after any number of frames, in global coordinates.

Since we ruled out translation, a pitch or yaw rotation should result in the same apparent movement for all visible objects on the image plane. The idea of the algorithm is to store the last frame and to capture a new one. We assume that if the camera did rotate by a small angle between the two frames, the the new frame should essentially be equivalent to the old frame but shifted some pixels into a certain direction. Computing the sum of absolute difference between a shifted version of the new frame and the old frame gives us a measure on how good the matching of the two frames is. Finding motion between the frames is therefore equivalent to finding a shift offset that gives an alignment with the lowest sum of absolute difference value. We assume that any motion of the camera between two frames is purely yaw and pitch motion according to the cameras coordinate system.

Let $I^{(t)}(x, y)$ be the pixel value at image position x, y at time t , f be the focal length of the camera and let the camera be free of distortions. Assume a camera rotation of β radians (yaw) and α radians (pitch) between two consecutive frames.

Figure 3.2 shows the situation in which we rotate the camera around the y -axis (yaw). Doing so, we observe a point seen at a distance of x (along the x -axis). After the transformation, we find the same point at x' in the new image.

The relation between the two points can be established easily from Figure 3.2 as

$$x' = f \cdot \tan\left(\beta - \arctan\frac{x}{f}\right) \quad (3.1)$$

which can be reformulated to

$$x' = f \cdot \frac{\tan(\beta) - \frac{x}{f}}{1 + \tan(\beta) \cdot \frac{x}{f}} \quad (3.2)$$

If we assume $\tan(\beta) \cdot \frac{x}{f} \approx 0$ and only consider small angles β , we can use the approximation $\tan(\beta) \approx \beta$, we can simplify this to $x' \approx \beta \cdot f - x$. Considering

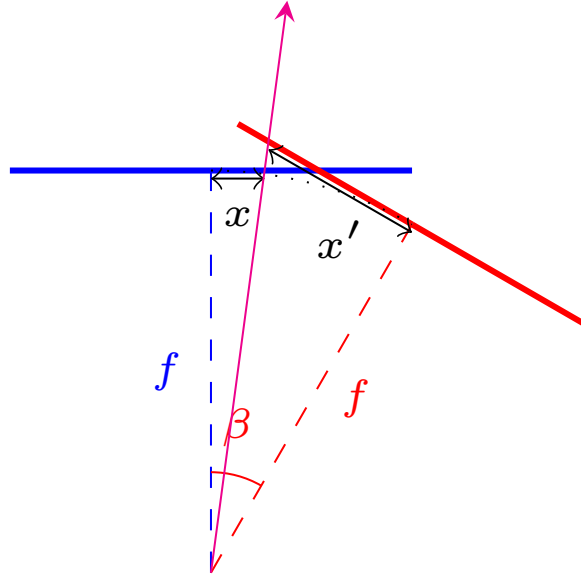


Figure 3.2: Apparent position of a pixel after rotation. The blue and red thick lines represent the image plane before and after a rotation by α . The magenta line is the vector pointing to a real world location. The values x and x' are the distances from the image center (in x -direction) where we expect to find the real world location in both images.

coordinates starting at the bottom left (making x positive), we can state the following approximation: An object we previously found at pixel location (x, y) should now have moved to location $(x + \beta f, y + \alpha f)$

If we further assume similar lighting conditions for both frames at time t and $t + 1$, we can say that for the image intensity I at time t

$$I^{(t)}(x, y) \approx I^{(t+1)}(x + \beta f, y + \alpha f) \quad (3.3)$$

and

$$|I^{(t)}(x, y) - I^{(t+1)}(x + \beta f, y + \alpha f)| \approx 0 \quad (3.4)$$

We define the sum of absolute absolute differences as the following, with H being the image height, and W being the image width.

$$\text{SAD}(u, v) = \sum_{x=\max(1, \beta f)}^{\min(W, W - \beta f)} \sum_{y=\max(1, \alpha f)}^{\min(H, H - \alpha f)} |I^{(t)}(x, y) - I^{(t+1)}(x + u, y + v)| \quad (3.5)$$

From (3.4) it follows that

$$\text{SAD}(u, v) \begin{cases} = 0 & \text{if } u = \alpha f \text{ and } v = \beta f \\ \geq 0 & \text{otherwise} \end{cases}, \quad (3.6)$$

where as equality in the second case can only occur if the image has self similar sub images, such as it is the case with a uniform colour. However, if we have enough variance in the image, we can assume that $SAD(u, v) > 0$ for $u \neq \alpha f, v \neq \beta f$

Finding the camera rotation between two consecutive frames thus becomes a search problem with the goal to minimise the cost function $SAD(u, v)$.

$$(\alpha, \beta) = \frac{1}{f} \cdot \underset{u, v}{\operatorname{argmin}}(SAD(u, v)) \quad (3.7)$$

In practice, we can not expect the cost function to equal zero at the correct offset, however, the minimising approach of (3.7) still holds.

With a given frequency (frame rate) of r (units $[\frac{1}{s}]$), we can estimate the angular velocities as $\omega_x = r \cdot \alpha$ and $\omega_y = r \cdot \beta$.

Visual Odometry

So far, we only measured the angular velocities of the camera between individual frames in relation to the cameras coordinate system. However, for a real-world application we may want to know the cameras pose at a certain frame relative to the world coordinate system. We approach this by Visual Odometry, a process of accumulating small individual rotations in order to estimate the cameras global pose at a certain frame.

To perform this task, we need the angular velocities along the x and y axes estimated in the section before, as well as an initial camera pose. Let $[\alpha^{(0)}, \beta^{(0)}, \gamma^{(0)}]$ be the Euler rotations (according to our definition) of the cameras initial pose. Given the well-known definitions of the 3D rotation matrices along the axes

$$R_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \alpha & -\sin \alpha \\ 0 & \sin \alpha & \cos \alpha \end{bmatrix} \quad R_y = \begin{bmatrix} \cos \beta & 0 & \sin \beta \\ 0 & 1 & 0 \\ -\sin \beta & 0 & \cos \beta \end{bmatrix} \quad R_z = \begin{bmatrix} \cos \gamma & -\sin \gamma & 0 \\ \sin \gamma & \cos \gamma & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (3.8)$$

We define the initial rotation matrix that maps coordinates in the initial camera coordinate system to the world coordinate system as

$$R^{(0)} = R_x(\alpha^{(0)}) \cdot R_y(\beta^{(0)}) \cdot R_z(\gamma^{(0)}) \quad (3.9)$$

Let $[\omega_x^{(i)}, \omega_y^{(i)}]$ be the angular velocities measured around the respective axes in the cameras coordinate system at frame i . We perform linear interpolation assuming that the angular velocity is constant between two consecutive frames. Therefore, given a frame rate of r , the angles travelled between two consecutive frames $i - 1$ and i are given by $\alpha^{(i)} = \frac{1}{r} \cdot \omega_x^{(i)}$ and $\beta^{(i)} = \frac{1}{r} \cdot \omega_y^{(i)}$

The transformation matrix in camera coordinates that implements these rotations can be written as

$$R^{(i)} = R_x(\alpha^{(i)}) \cdot R_y(\beta^{(i)}) \quad (3.10)$$

Therefore, the transformation matrix T_i that maps coordinates in the camera coordinate system of frame i to the world coordinates is given by

$$T_i = R^{(0)} \cdot R^{(1)} \cdots R^{(i)} = \overset{\curvearrowright}{\prod}_{k=0}^i R^{(k)} \quad (3.11)$$

Note that the order of matrix multiplications is important. The $\overset{\curvearrowright}{\prod}$ means right multiplication of values.

In order to compare the results more easily, the camera pose transformation matrix then gets converted into the three smallest Euler angles according to our definition above. Note that despite our system can not detect any rotations around the z -axis, the Euler decomposition of T generally has a non-zero z component. This comes from the fact that our motion estimations are always in respect to the camera's coordinate system. With the right sequence of rotations around the x and y axes of the camera coordinate system, one can induce motions that look like rotations around the z -axis viewed from the world coordinate system.

3.4.3 Implementation

Two different algorithms have been implemented to solve the search problem:

1. **Independent "cross" search, (Algorithm 1)** We assume that the actual image movement is relatively small. In every iteration of the algorithm, we capture a new frame. At first, we compute the sum of absolute difference from the current frame and the previous frame. The value obtained this way, is the score of the hypothesis that the camera did move between the two frames. After that, for both dimensions, we shift the new image in a local neighbourhood and compute the sum of absolute value for each, always keeping the minimum value. Whenever we find a lower value than before, the best hypothesis gets updated to the current shift.

The algorithm explores a "cross" of hypotheses, with the center point being the no movement hypothesis.

2. **Gradient descend inspired search, (Algorithm 2)** The problem of finding u and v is the problem of finding an alignment of the current frame onto the previous frame, such that the sum of absolute difference between the two frames is minimal.

This algorithm performs the search for the best alignment by using a gradient descend inspired approach. The algorithm starts out with $u = 0$ and $v = 0$, assuming there was no movement between the frames. The sum of absolute difference for no movement gets stored as the initial best value. From there, we shift the current frame one pixel in all four directions, computing the sum of absolute differences in each direction. We keep the minimum value obtained as well as the direction we took to get there. This identifies the direction to shift the image in the next iteration, assuming we always go into the direction which leads to the lowest immediate value. After identifying the direction, we add the direction to the results variables u or v and shift the current frame in

Algorithm 1 Independent search in both dimensions. Note that the algorithm essentially does four times the same, shifting the new image into a certain direction compute the sum-of-absolute differences at each shift.

```

1: procedure INDEPENDENTSEARCH(last_frame, current_frame)
2:    $u \leftarrow 0$ 
3:    $v \leftarrow 0$ 
4:   shift_frame  $\leftarrow$  current_frame
5:   min_sum  $\leftarrow$  MAX_VALUE ▷ Reset minimal correlation
6:   for  $i \in \{0, MAX\_SHIFT\}$  do ▷ Explore east direction
7:     total_sum  $\leftarrow$   $\sum_{image} |shift\_frame - last\_frame|$ 
8:     if total_sum < min_sum then
9:       min_sum  $\leftarrow$  total_sum
10:       $u \leftarrow i$ 
11:      shift_frame  $\leftarrow$  shift_east(shift_frame)
12:   shift_frame  $\leftarrow$  current_frame
13:   for  $i \in \{1, MAX\_SHIFT\}$  do ▷ Explore west direction
14:     total_sum  $\leftarrow$   $\sum_{image} |shift\_frame - last\_frame|$ 
15:     if total_sum < min_sum then
16:       min_sum  $\leftarrow$  total_sum
17:        $u \leftarrow -i$ 
18:       shift_frame  $\leftarrow$  shift_west(shift_frame)
19:   shift_frame  $\leftarrow$  current_frame
20:   min_sum  $\leftarrow$  MAX_VALUE ▷ Reset minimal correlation
21:   for  $i \in \{0, MAX\_SHIFT\}$  do ▷ Explore south direction
22:     total_sum  $\leftarrow$   $\sum_{image} |shift\_frame - last\_frame|$ 
23:     if total_sum < min_sum then
24:       min_sum  $\leftarrow$  total_sum
25:        $v \leftarrow i$ 
26:       shift_frame  $\leftarrow$  shift_south(shift_frame)
27:   shift_frame  $\leftarrow$  current_frame
28:   for  $i \in \{1, MAX\_SHIFT\}$  do ▷ Explore north direction
29:     total_sum  $\leftarrow$   $\sum_{image} |shift\_frame - last\_frame|$ 
30:     if total_sum < min_sum then
31:       min_sum  $\leftarrow$  total_sum
32:        $v \leftarrow -i$ 
33:   shift_frame  $\leftarrow$  shift_north(shift_frame)
return  $u, v$ 

```

the obtained direction. We start the next iteration of the process. Again, we assume no further movement between the current frame and the last frame. If we detect a direction in which we can get an even lower minimal sum, we got into this direction and start the next iteration. The process ends when we identified a local minima, which manifests itself that we get higher sum of absolute difference values in all four directions. This algorithm assumes the problem to be locally convex.

The algorithms running on the cellular processor array report the (scaled) value of the angular velocities back to the host CPU. Scaling and Visual Odometry is then performed on the host CPU. Since these tasks are just multiplications of 3×3 matrices, even very low power devices should be capable of doing that.

3.4.4 Results

Testing Methods

Both implementations were tested on the APRON simulator (Barr and Dudek (2008)). It has been shown that the implementations work on real hardware as well, however due to lack of available hardware, all the quantitative results in this sections were obtained using the simulator software.

A real-world trajectory, recorded with the gyroscope of a smartphone, was used in the following two environments to create synthetic datasets:

1. *Office scene* an office room with pillars, checkerboard floor and ceiling panels. The large amount of individual objects are expected to pose an easier problem for the pose estimation algorithms, as there are more features for the algorithm to track.
2. *Living room scene* a living room with little furniture and plain white walls. It is expected that this environment poses a bigger challenge to the algorithm, as the frames have less variance. Especially the ceiling is very uniform.

Figure 3.3 shows some frames taken out from both datasets. The frames captured with the camera pointing to the ceiling may pose the biggest challenge to the algorithm. Both environments were rendered with the same trajectory, to generate comparable results.

Angular Velocity Estimation

Figure 3.4 shows the measured angular velocities of the algorithms on both datasets. The shaded areas at the bottom give an indication about the variance of the dataset at the given frames. It is expected that the less variance there is in the images, the harder it gets for the algorithm to successfully track the motion. **Figure 3.5** shows the squared error to the ground truth. The error is the sum of the squared error of the pitch and yaw rotations. **Table 3.1** shows the means and the standard deviations of the squared errors for the datasets.

Algorithm 2 Gradient descent inspired search

```

1: procedure GRADDESCSEARCH(last_frame, current_frame)
2:    $u \leftarrow 0$ 
3:    $v \leftarrow 0$ 
4:    $min\_sum \leftarrow \sum_{image} |shift\_frame - last\_frame|$  ▷ Get baseline value
5:   do
6:      $next\_action \leftarrow STAY$  ▷ Assume we found minimum
7:      $shift\_frame \leftarrow \mathbf{shift\_east}(current\_frame)$ 
8:      $t\_sum \leftarrow \sum_{image} |shift\_frame - last\_frame|$ 
9:     if  $t\_sum < min\_sum$  then
10:       $min\_sum \leftarrow t\_sum$ 
11:       $next\_action \leftarrow EAST$  ▷ Lower sum value, if shifted east
12:       $shift\_frame \leftarrow \mathbf{shift\_west}(current\_frame)$ 
13:       $t\_sum \leftarrow \sum_{image} |shift\_frame - last\_frame|$ 
14:      if  $t\_sum < min\_sum$  then
15:         $min\_sum \leftarrow t\_sum$ 
16:         $next\_action \leftarrow WEST$  ▷ Lower sum value, if shifted west
17:         $shift\_frame \leftarrow \mathbf{shift\_north}(current\_frame)$ 
18:         $t\_sum \leftarrow \sum_{image} |shift\_frame - last\_frame|$ 
19:        if  $t\_sum < min\_sum$  then
20:           $min\_sum \leftarrow t\_sum$ 
21:           $next\_action \leftarrow NORTH$  ▷ Lower sum value, if shifted north
22:           $shift\_frame \leftarrow \mathbf{shift\_south}(current\_frame)$ 
23:           $t\_sum \leftarrow \sum_{image} |shift\_frame - last\_frame|$ 
24:          if  $t\_sum < min\_sum$  then
25:             $min\_sum \leftarrow t\_sum$ 
26:             $next\_action \leftarrow SOUTH$  ▷ Lower sum value, if shifted south
27:          switch  $next\_action$  do ▷ Shift current frame for next iteration
28:            case  $EAST$ 
29:               $current\_frame \leftarrow \mathbf{shift\_east}(current\_frame)$ 
30:               $u \leftarrow u + 1$ 
31:            case  $WEST$ 
32:               $current\_frame \leftarrow \mathbf{shift\_west}(current\_frame)$ 
33:               $u \leftarrow u - 1$ 
34:            case  $NORTH$ 
35:               $current\_frame \leftarrow \mathbf{shift\_north}(current\_frame)$ 
36:               $v \leftarrow v + 1$ 
37:            case  $SOUTH$ 
38:               $current\_frame \leftarrow \mathbf{shift\_south}(current\_frame)$ 
39:               $v \leftarrow v - 1$ 
40:          while  $next\_action \neq STAY$ 
41:          return  $u, v$ 

```

(a) Example frame from *living room* set(b) Example frame from *office* set(c) Low variance frame, little features to hold on to. *Living room* set.(d) Low variance frame in the *office* set. The ceiling has more texture.

Figure 3.3: Example frames from both tested datasets. It is expected that frames with low variance are harder to track.

<i>Units:</i> $\left[\frac{\text{rad}^2}{\text{s}^2}\right]$	Mean Error	Std. Error
Living room, Cross	12.45 e-6	32.20 e-6
Living room, Gradient	5.14 e-6	6.80 e-6
Office, Cross	6.04 e-6	8.88 e-6
Office, Gradient	5.24 e-6	7.11 e-6

Table 3.1: Mean and standard deviations of the squared errors in the measurement of angular velocities for both algorithms on both datasets.

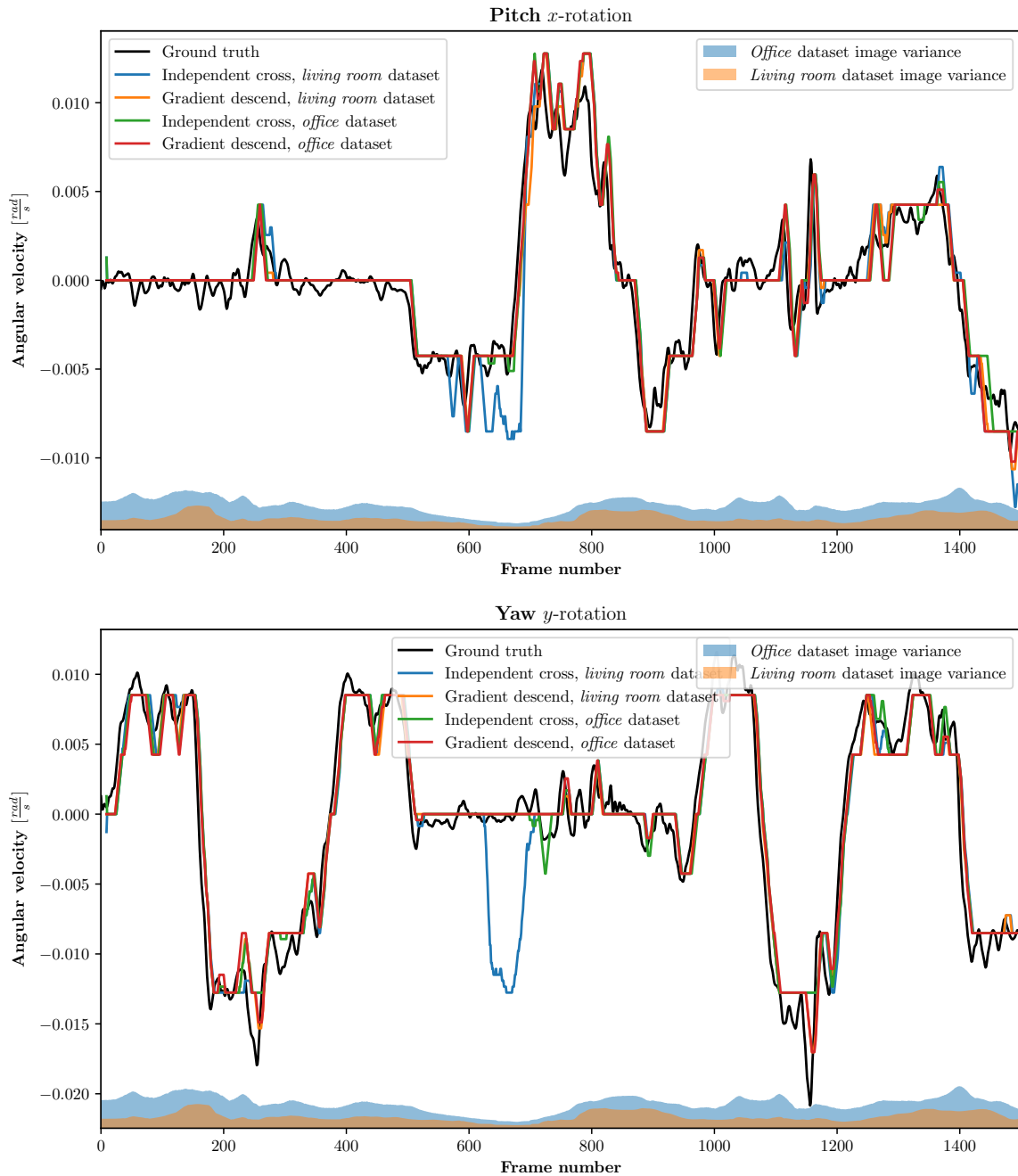


Figure 3.4: Angular velocities as estimated by the pose estimation algorithms. Both the *independent cross* and the *gradient descend* algorithms were applied to the *office* and *living room* datasets. The region at the bottom shows the (scaled) image variances of the original video frames.

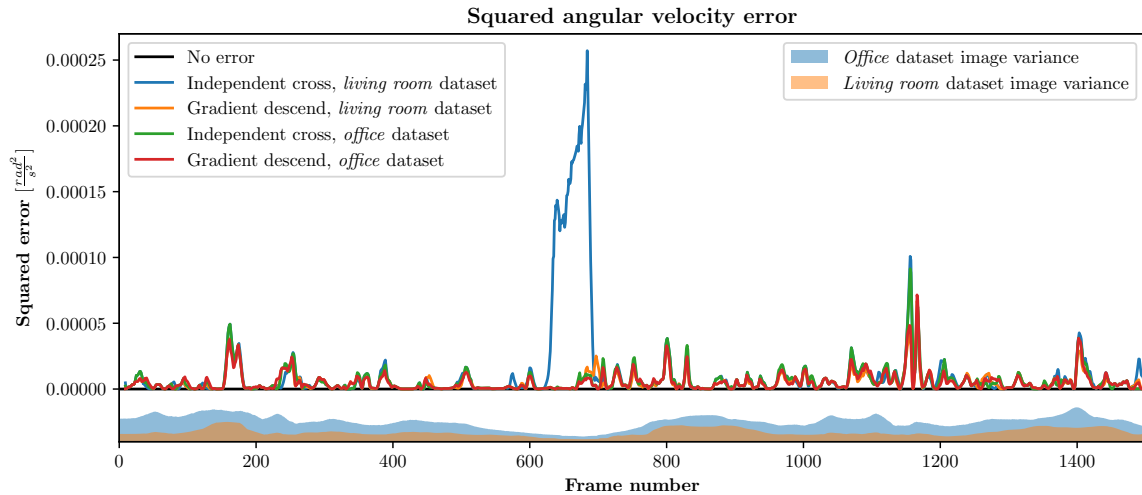


Figure 3.5: Squared errors of the algorithms on the datasets for the angular velocity estimation.

All combinations except the *cross* algorithm on the *living room* dataset exhibit good tracking performance with little error. The *cross* algorithm fails to estimate the angular velocity properly in a region from frames 650 to 700. This is thought to be caused by the very little variance the frames in this region exhibit in the *living room* dataset. (See Figure 3.3). The *office* dataset also has little image variance in this region, however still enough to not be vulnerable to this problem. The *gradient descend* based algorithm performs significantly better in this difficult area. The reasoning behind this is that the *cross* algorithm always finds the global minima in both directions, potentially skipping local minima. As the sum of absolute differences is, due to the lack of enough image variance, very small for all shifts of the new frame, it is very well possible that there is a spurious global minima further away from the correct solution, which may be purely induced by noise or lighting changes. The *gradient descend* based approach has a more conservative approach. As it assumes that the problem is convex, it stops as soon as it finds itself in a local minima. An interesting thing to note is, that except one outlier, all the other combinations performed surprisingly well in this part compared to the rest of the sequence.

In general, we can see that the *gradient descend* based approach performs better in all cases. Contrary to the expectation, it actually performed better on the *living room* dataset than on the *office* dataset.

Visual Odometry

The measured angular velocities were accumulated according to equation 3.11. The resulting pose matrices were transformed into Euler angles according to our definition (left handed coordinate system, xyz order). The poses obtained for the different algorithm and dataset combinations can be seen in Figure 3.6. The squared error of the posed in relation to ground truth is depicted in Figure 3.7. The error means and standard deviations are listed in Table 3.2. **Table 3.3** shows the maximum values

<i>Units:</i> $[rad^2]$	Mean Error	Std. Error	Max. Error	End Error
Living room, Cross	0.236	0.283	1.703	0.467
Living room, Gradient	0.026	0.016	0.083	0.034
Office, Cross	0.045	0.034	0.138	0.133
Office, Gradient	0.039	0.027	0.118	0.049

Table 3.2: Squared mean and standard pose errors for the algorithm and dataset combinations. All values in $[rad^2]$

<i>Units:</i> $[rad]$	Max. Pitch	Max. Yaw	Max. Roll	End Pitch	End Yaw	End Roll
Liv. r., Cross	0.721	-0.533	1.083	0.185	-0.319	0.576
Liv. r., Grad.	0.188	0.272	-0.095	0.122	0.136	0.022
Off., Cross	0.289	0.302	-0.096	0.254	0.256	0.062
Off., Grad.	0.254	0.308	-0.099	0.108	0.194	-0.002

Table 3.3: Individual dimension maximum and end errors of the estimated pose angle for the different algorithm and dataset combinations

and end values of the errors in radians, for each dimension.

As expected, the algorithm/dataset combinations that performed best in the angular velocity estimation perform best in the VO task as well. This is due to the fact, that the pose estimation takes the measured angular velocities as input data and integrates them over time.

All algorithms experience drift, however, the gradient descend algorithm is significantly better with the worst run exhibiting only half the squared error of the best run with the cross algorithm.

Performance

Because of the lack of real hardware, the following performance calculations are estimated based on published performance figures of the SCAMP chip and static analysis of the source code to the algorithms presented. The different operations on the SCAMP chip take various amounts of clock cycles. The visual processor runs at a clock frequency of $10MHz$, while the controller runs at $200MHz$ (Carey (2017)). Since the controller is much faster than the vision chip, and can run operations in parallel, we ignore the effect of the controller for our runtime estimates. **Table 3.4** shows the assumed number of SCAMP operations required to perform the operation on the vision chip. These values were obtained by reverse engineering the source code provided by Chen (2016).

Carey et al. (2013) state a power consumption of the vision chip at the maximum instruction rate of $10MHz$ as $1.23W$ and an idle power consumption of $0.2mW$. Power consumption for a fixed frame rate below the maximum frame rate is computed under the assumption that the processor finishes a frame at maximum clock speed and then switches to idle state until the beginning of the new frame.

Table 3.5 shows estimated performance calculations for the independent cross and the gradient descend based algorithms. The independent cross algorithm is not

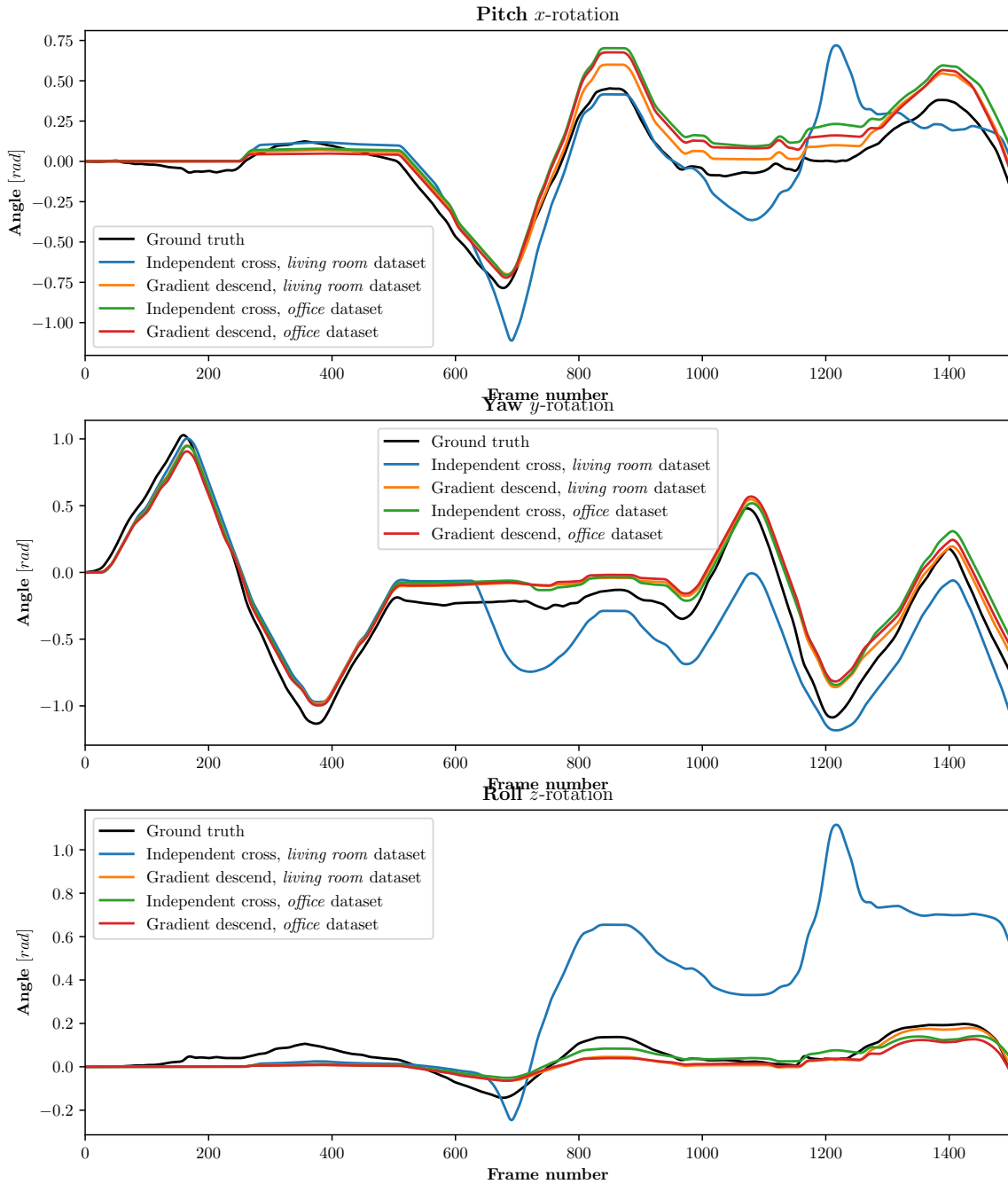


Figure 3.6: Euler angles of the camera poses as estimated by the different algorithm-dataset combinations.

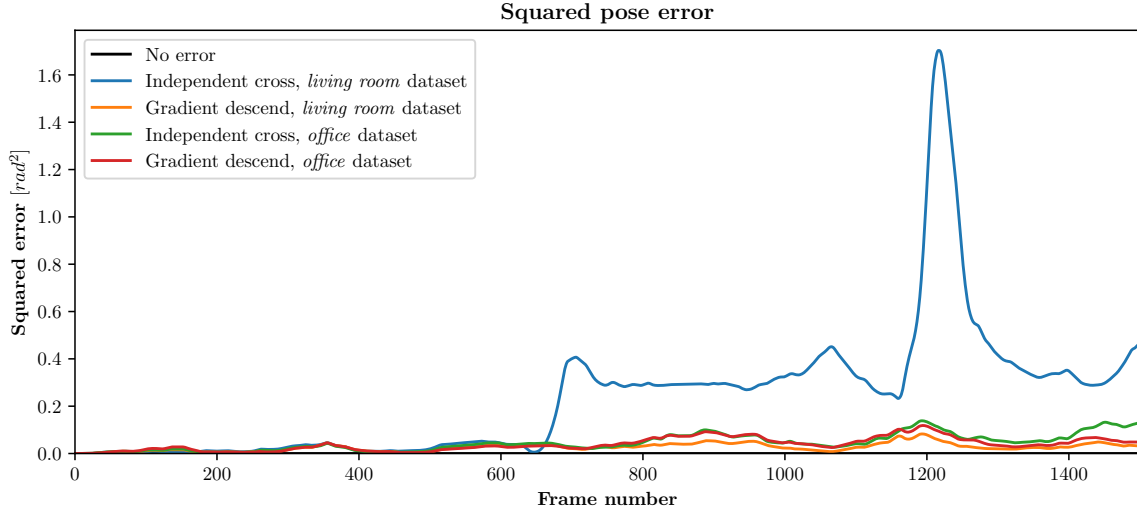


Figure 3.7: Squared error of the estimated pose for all the algorithm dataset combinations

Operation	Clock cycles
<i>north, south, east, west, copy</i>	2
<i>add, sub</i>	2
<i>neg</i>	1
<i>where, all</i>	1
<i>abs</i>	5
<i>div2</i>	5
<i>or, not</i>	1
<i>nor</i>	2
<i>nand</i>	4
<i>and</i>	5
<i>dshift</i>	1
<i>global sum</i>	64

Table 3.4: Assumed clock cycle counts for the individual operations. These values were obtained by static observation in reverse engineering of the source code (Chen (2016)).

	Independent cross	Gradient descend
Avg. cycles per frame	1176	846.72
Std. cycles per frame	0	314.58
Theoretical max. FPS.	8503	11810
Power @ max. FPS	1.23 W	1.23 W
Estimated power @ 60FPS	8.9 mW	6.4 mW

Table 3.5: Estimated performance values for the algorithms on the SCAMP chip.

dependent on the data, every frame takes the same amount of instructions. Thus, the standard deviation of the number of instructions is 0. The gradient descend based algorithm however, performs more steps on larger shifts. The mean and standard deviation values were obtained based on the number of steps the algorithm performed on the *living room* and *office* datasets. The gradient descend algorithm in its current implementation has no upper bound on the number of steps it performs. However, in order to guarantee a certain frame rate, it would be trivial to limit the algorithm to a maximum number of iterations. **Table 3.5** demonstrates that the gradient descend based algorithm not only has a better tracking performance, but also requires less instruction cycles in the mean case.

The values presented here are to be taken with caution, as they are based on an assumption about the number of clock cycles without actual measurements. In addition to that, the calculations do not incorporate limitations such as bus speeds or sleep/wake times into account. Nevertheless, they show the order of magnitude of performance figures we can expect from a CPA system. It clearly shows that using a CPA can give extremely high frame rates at a very low energy consumption for problems like these.

3.5 Yaw, Pitch, Roll, Z Estimation

This section introduces a novel method created to extend the two degrees of freedom approach presented in Section 3.4 with two additional degrees of freedom, roll and z -translation. The algorithm is based on splitting the image up into individual tiles, estimating a basic vector in each tile. These estimates are then used by a statistical classifier in order to estimate the global movement.

3.5.1 Motivation

Section 3.4 introduced two algorithms based on the sum of absolute differences in order to estimate the camera ego rotations around the x and y axes relative to the camera coordinate system. While the results show promising results that the system works as intended, the use of a system that only estimates rotations around two axes is limited. In general, we experience the following two limitations:

1. As soon as the camera experiences a roll motion, the visual odometry can no longer accurately estimate the cameras position, as roll rotations are not

tracked.

2. We don't have any possibility to capture a translation of the camera, we have to assume that the camera remains at the same place.

Adding roll estimation to the system allows us to capture all 3 degrees of freedom of rotation which allows us to capture every possible rotational movement. Adding z-translation allows us to capture camera movement in the direction we are looking at. In general, this is the most interesting direction of movement, as a wheeled robot or car is mechanically restricted to move into other directions. Furthermore, it is expected to be an easier problem to separate roll and z motion from yaw and pitch, as translations into x and y directions would look similar to yaw and pitch to the camera. The system could be extended to a 6DoF approach using 3 cameras pointing in different directions.

3.5.2 Approach

The approach chosen for this problem splits the image into 16 square tiles, estimating the displacement vector for each tile according to the gradient descend based algorithm introduced in Section 3.4. For each of the four degrees of freedom we have an expectation how the individual vectors should be aligned if a motion in this dimension occurs. **Figure 3.8** shows the expected vectors for each tile, given a motion in one of the four degrees of freedom. The basic concept of the algorithm is to use a statistical method, in order to match the 16 measured displacement vectors with a linear combination of the expected bases. In order to capture roll and z-translation at the same time, we can not expect the same apparent movements of objects in the whole image plane. Instead, this assumption is relaxed to only hold (approximately) for small sub images. The chosen approach divides the image plane into 16 tiles, where on each tile, we compute the $[u, v]$ vector according to equation (3.7).

An expected vector direction and relative magnitude was calculated for each tile and for each expected mode of motion (pitch, yaw, roll, z). The resulting vectors are presented in Figure 3.8.

The idea of the algorithm is that every motion between two frames is considered to be a linear combination of four degrees of freedom, while every degree of freedom has an expected vector direction in each tile. So measuring the apparent vector in each tile allows us to recover the coefficients of the linear combination, which gives us the hypothesis in all four dimensions.

Let

$$\mathbf{m} = [u_1 \ v_1 \ u_2 \ v_2 \ \dots \ u_{16} \ v_{16}]^T \quad (3.12)$$

be the measured apparent motion vector components found in each tile.

Let $\{\mathbf{b}_{yaw}, \mathbf{b}_{pitch}, \mathbf{b}_{roll}, \mathbf{b}_z\}$ be the vectors in the same form as \mathbf{m} , but normalised and with the components of their corresponding base vector field. The resulting, normalised \mathbf{b} vectors are

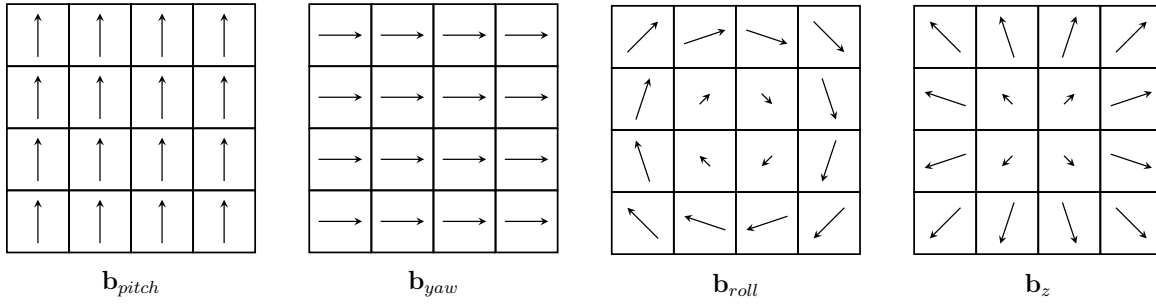


Figure 3.8: Expected vector direction for each tile given a certain mode of motion between two frames. Any legal motion the camera experiences between two frames is assumed to appear as a linear combination of these vectors, according to Equation 3.13.

$$\begin{aligned}
 \mathbf{b}_{pitch} &= \frac{1}{4} [0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1]^\top \\
 \mathbf{b}_{yaw} &= \frac{1}{4} [1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1]^\top \\
 \mathbf{b}_{roll} &= \frac{1}{4\sqrt{10}} [3 \ -3 \ 3 \ -1 \ 3 \ 1 \ 3 \ 3 \ 1 \ -3 \ 1 \ -1 \ 1 \ 1 \ 3 \ -1 \ -3 \ -1 \ -1 \ -1 \ 1 \ -3 \ -3 \ -3 \ -3 \ -1 \ -3 \ 1 \ -3 \ 3]^\top \\
 \mathbf{b}_z &= \frac{1}{4\sqrt{10}} [-3 \ -3 \ -1 \ -3 \ 1 \ -3 \ 3 \ -3 \ -1 \ -1 \ -1 \ 1 \ -3 \ -1 \ -3 \ 1 \ -1 \ 1 \ 1 \ 3 \ 1 \ -3 \ 3 \ -1 \ 3 \ 1 \ 3 \ 3 \ 3]^\top
 \end{aligned}$$

Based on the four base vector fields, we then describe our model as

$$\mathbf{m} - (\alpha \cdot \mathbf{b}_{yaw} + \beta \cdot \mathbf{b}_{pitch} + \gamma \cdot \mathbf{b}_{roll}) + \delta \cdot \mathbf{b}_z = \epsilon \quad (3.13)$$

With ϵ being the error we encounter in our estimate. The intuition behind this model is, that it describes a way to linearly decompose the measured \mathbf{m} vector into a linear combination of the known \mathbf{b} vectors. The parameters $\alpha, \beta, \gamma, \delta$ from the linear combination are then the scaled, angular velocities in their respective direction. The presented form is equivalent to a standard generalised linear regression problem, for which there are multiple strategies to solve it.

OLS solution

A straight forward solution to the problem is the Ordinary Least Squares (OLS) approach. This approach requires us to assume, that the error on the estimates follows a Gaussian distribution. With this assumption, we can write the Ordinary Least Squares problem. Given $B = [\mathbf{b}_{yaw} \ \mathbf{b}_{pitch} \ \mathbf{b}_{roll} \ \mathbf{b}_z]$ we can state that

$$[\alpha \ \beta \ \gamma \ \delta]^\top = (B^\top B)^{-1} B^\top \mathbf{m} = B^\top \mathbf{m} \quad (3.14)$$

Since the matrix B is an orthonormal matrix, that is $B^\top B = B^{-1} B = I$. In every frame, the apparent $x - y$ vectors are measured in all 16 tiles. The CPU then

computes the matrix vector product $B^T \mathbf{m}$ to get the motion estimations in each of the four dimensions.

With B constant and known at compile time. Motion estimation is therefore just a linear combination of the vector components in \mathbf{m} . This operation requires at most $4 \cdot 32 = 128$ multiply and add operation on a standard CPU and should thus easily fit into the computational budget of even very limited processing devices. One can expect to reduce this even further by explicitly multiplying the expression out and exploit the sparse nature of B .

RANSAC solution

Especially for computer vision related problems, we can, in general, not assume that the error on the measurements follows a gaussian distribution, but instead consists of several very accurate measurements with a lot of outliers. A more robust estimation technique that assumes outliers to be present (RANSAC) was introduced by Fischler and Bolles (1981). In addition to OLS, RANSAC can be used to eliminate outliers from the regression problem in Equation 3.13.

3.5.3 Implementation

The tile algorithm has been implemented to run on the APRON (Barr and Dudek (2008)) simulator because of the lack of available physical hardware. Only the measurement of the 16 individual tile vectors takes place on the CPA, the matrix multiplication for the actual estimate happens outside on a conventional processor. Newer versions of the SCAMP chip which incorporate a more sophisticated system controller (Carey (2017)) would be powerful enough to perform the estimation on the same device as well. Yaw and Pitch get estimated using the ordinary least squares method, Roll and Z-translation get estimated using a 5-sample RANSAC with 90% inlier confidence threshold.

3.5.4 Results

Testing Methods

The algorithm has been tested on the same trajectory rendered in synthetic environments as in Section 3.4. However, this time, the ground truth trajectory and the rendered images also include roll movements. The camera remains at the same place throughout the entire scene, only three degrees of freedom are analysed.

In addition to that, the algorithm is tested on a trajectory of a forward facing camera mounted on a person moving forward. The same trajectory was rendered in two different environments, one showing a kitchen scene while the other shows a living room scene.

Units: $[\frac{rad^2}{s^2}]$	Mean Error	Std. Error
Living room	7.038 e-06	8.963 e-06
Office	5.831 e-06	7.902 e-06

Table 3.6: Mean and standard deviations of the squared angular velocity errors. (Tiled algorithm, both datasets)

Units: $[rad^2]$	Mean Error	Std. Error	Max. Error	End Error
Living room	0.108	0.079	0.322	0.214
Office	0.042	0.046	0.194	0.063

Table 3.7: Means and standard deviations of the squared errors of the estimated poses using the tiled algorithm on both datasets.

Angular Velocities

Figure 3.9 shows the estimated angular velocities for the tiled algorithm on the *living room* and the *office* datasets. **Table 3.6** shows the mean and standard deviations of the squared error of the tiled algorithm in the three degrees of freedom pose estimation task.

One can see that the tracking of yaw and pitch angles is still consistent, albeit less accurate than in the approach presented in Section 3.4. The roll rotation estimate is considerably less accurate than the yaw and pitch estimate.

Pose Odometry

Figure 3.10 shows the estimated poses of the algorithm together with ground truth on both datasets. The poses are in $x - y - z$ Euler angles, according to our definition (section 3.3). **Table 3.7** and **Table 4.1** show the mean squared and the maximum individual errors reached on the sequence.

It is evident, that the pose estimation suffers from large drift, especially in the pitch dimension towards the end of the sequence. A maximum pitch error of $0.498rad$ as experienced in the living room dataset, corresponds to an error of approximately 28° . The yaw estimation is considerably better, however, still encounters larger errors than observed in the two degrees of freedom approach.

Units: $[rad]$	Max Pitch	Max Yaw	Max Roll	End Pitch	End Yaw	End Roll
Living room	0.498	0.290	-0.271	0.362	0.266	-0.109
Office	0.376	0.193	-0.214	0.136	0.193	-0.084

Table 3.8: Maximum and end errors for the individual dimensions of the estimated poses for the tiled algorithm.

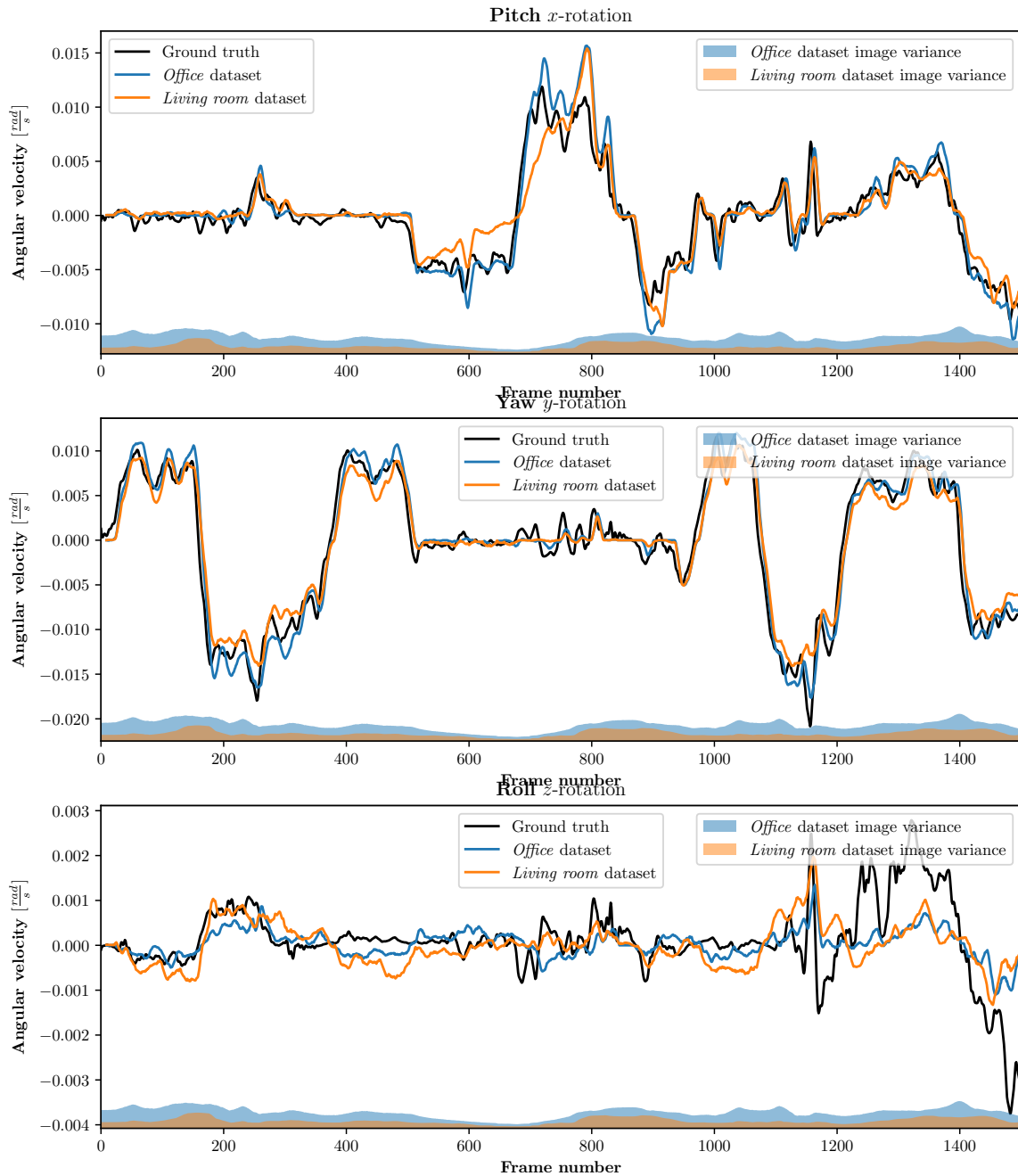


Figure 3.9: Estimated angular velocities of the tiled four degree of freedom algorithm. Applied on both datasets. The shaded regions at the bottom symbolise the image variance of the datasets.

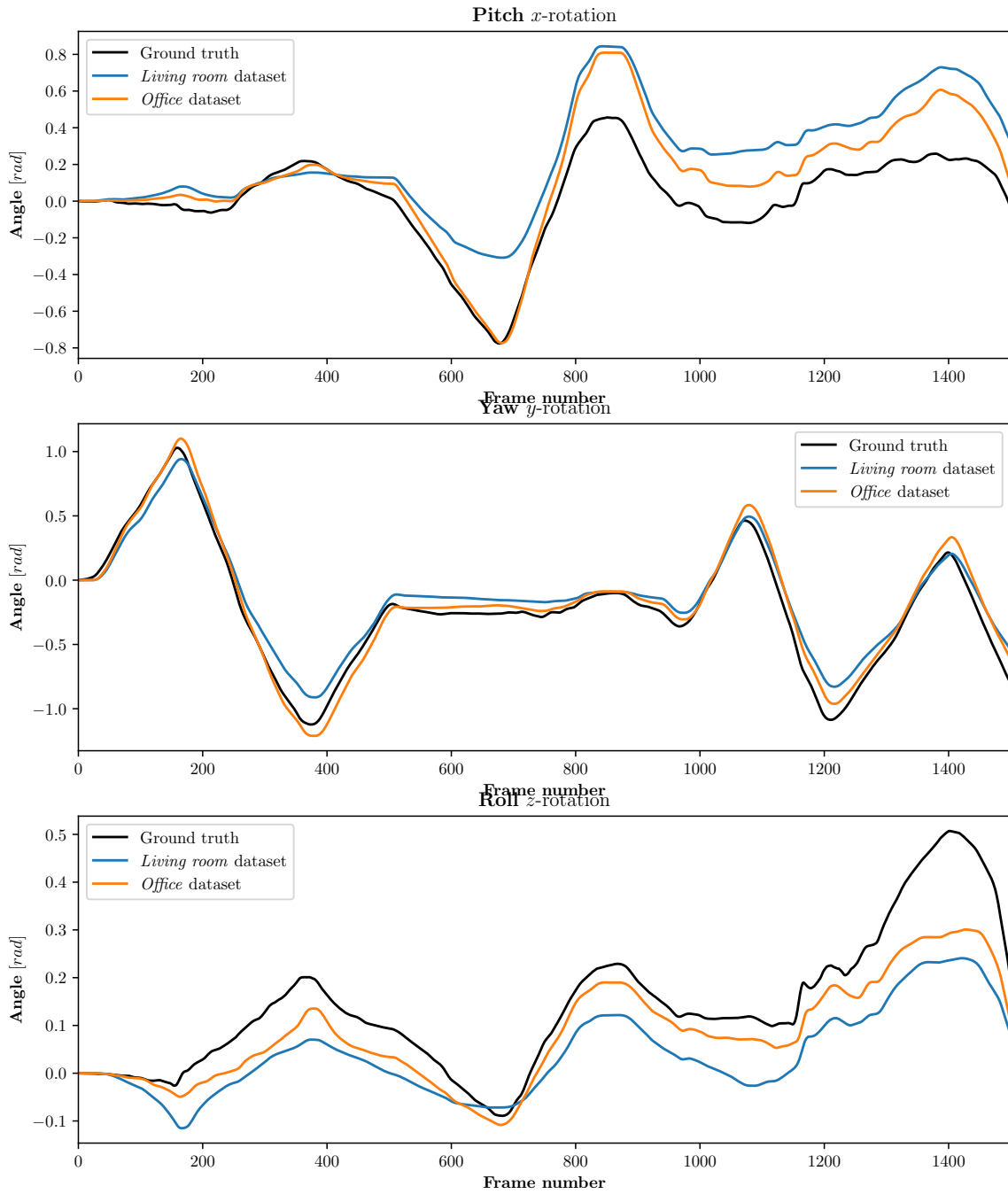


Figure 3.10: Estimated poses for the tiled algorithm on both datasets.

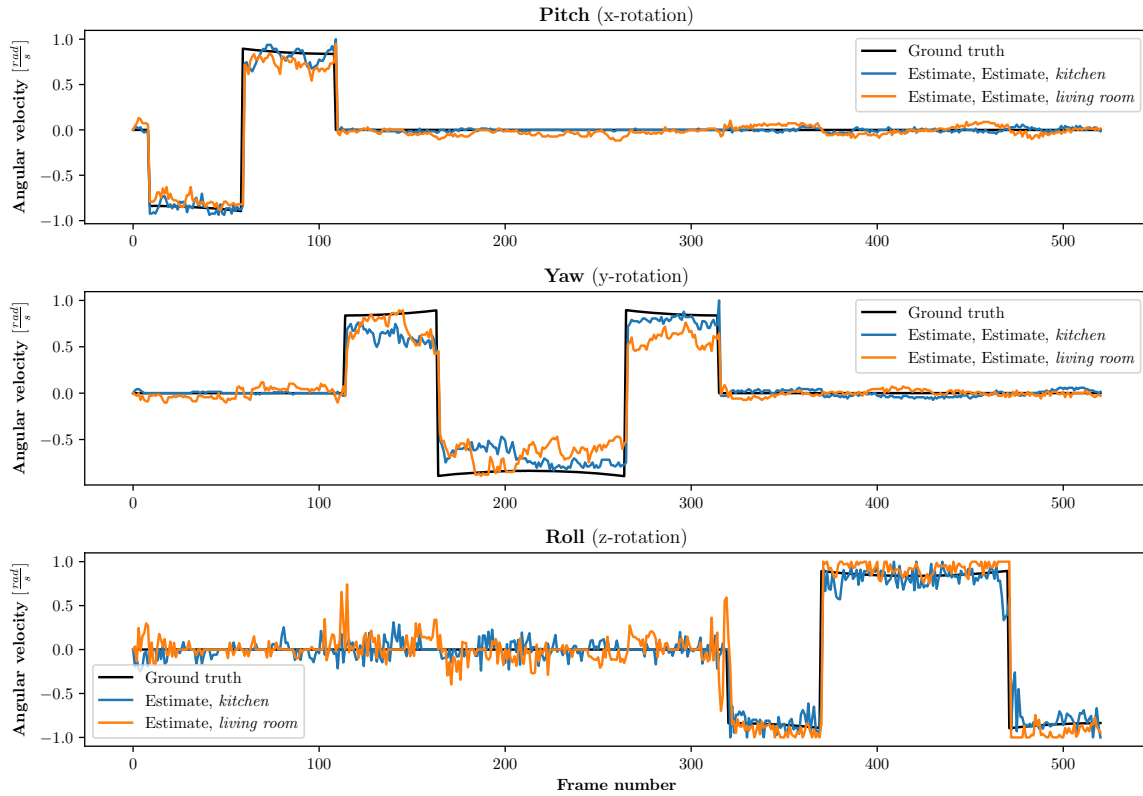


Figure 3.11: Tiled angular velocities in full 4DoF estimation

Units: $[rad^2]$	Pitch	Yaw	Roll
Avg. Squared Error	0.0056	0.0026	0.0009
Max. Squared Error	0.0085	0.0398	0.0078

Table 3.9: Orientation errors for the tiled algorithm, *kitchen* dataset.

Translation

The algorithm was tested on a synthetic dataset showing the point of view from a person walking slowly towards a kitchen counter (*kitchen* dataset) and towards a sofa in a living room (*living room* dataset). **Figures 3.11, 3.12 and 3.15** show the estimated angular velocities, camera orientation and camera position in global coordinates. One can see that the four degrees of tracking algorithm works well at tracking the camera's movement both in rotation as well as in z -translation. The *living room* dataset shows a better performance at translation estimation, whereas the *kitchen* dataset exhibits a better performance in pose estimation. Pose errors are relatively small with largest squared error being around 0.04 rad . Contrary to the expectation, both the largest error was encountered in the yaw direction, rather than roll and z directions. Since we are dealing with a monocular system, the scale of the translational movement can not be automatically established. The scale shown in Figure 3.15 was chosen using prior information about the dataset.

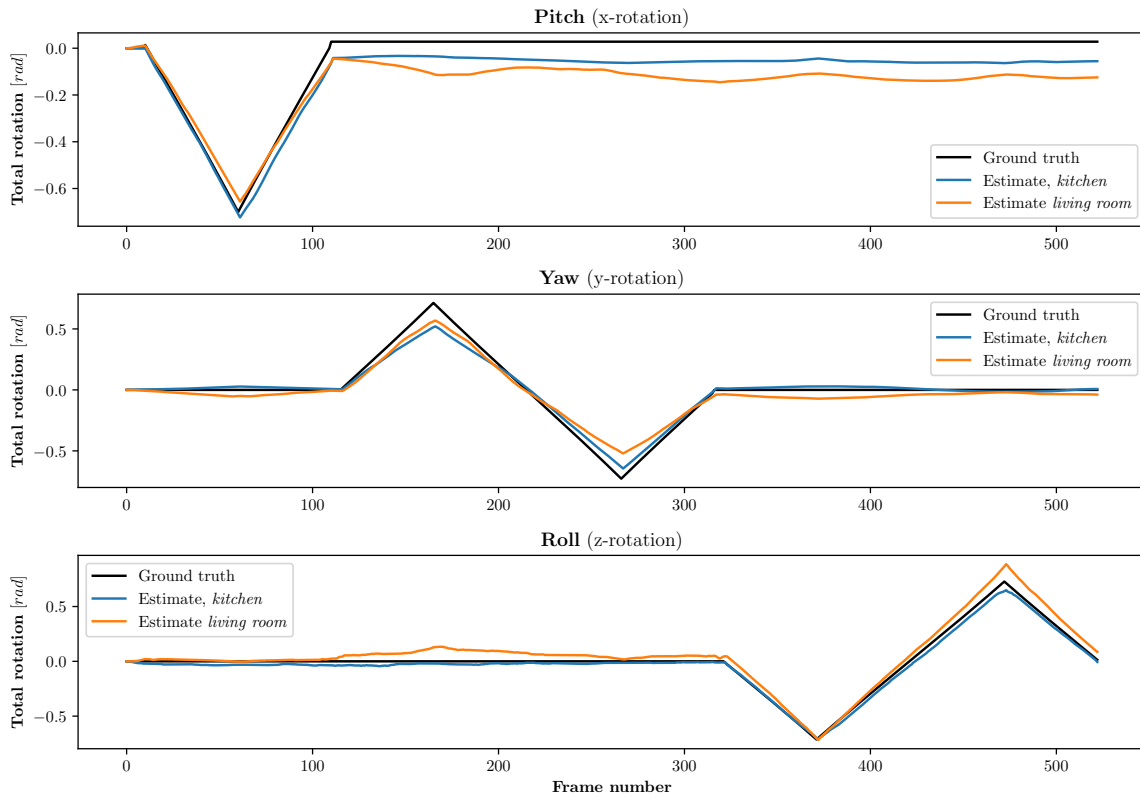


Figure 3.12: Tiled algorithm estimated poses in full 4DoF estimation

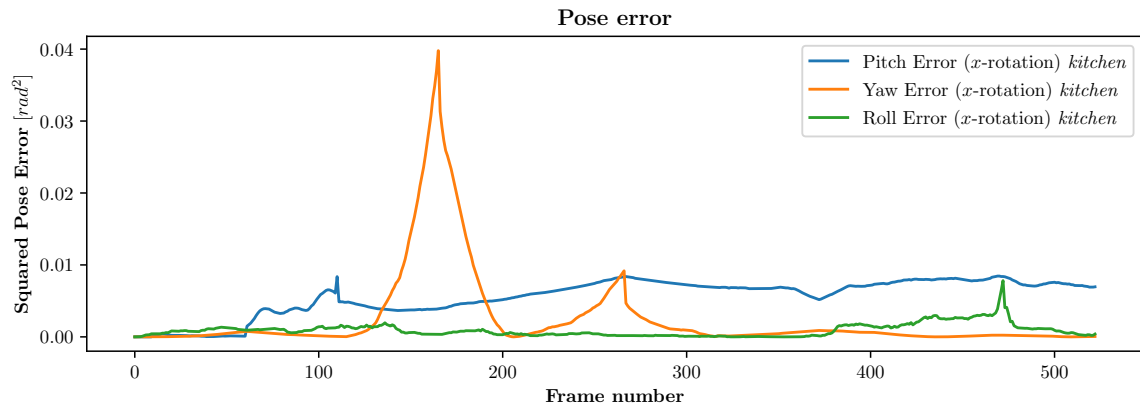


Figure 3.13: Tiled algorithm estimated pose error on *kitchen* dataset in full 4DoF estimation

Units: [rad ²]	Pitch	Yaw	Roll
Avg. Squared Error	0.0161	0.0042	0.0043
Max. Squared Error	0.0301	0.0464	0.0300

Table 3.10: Orientation errors for the tiled algorithm, *kitchen* dataset.

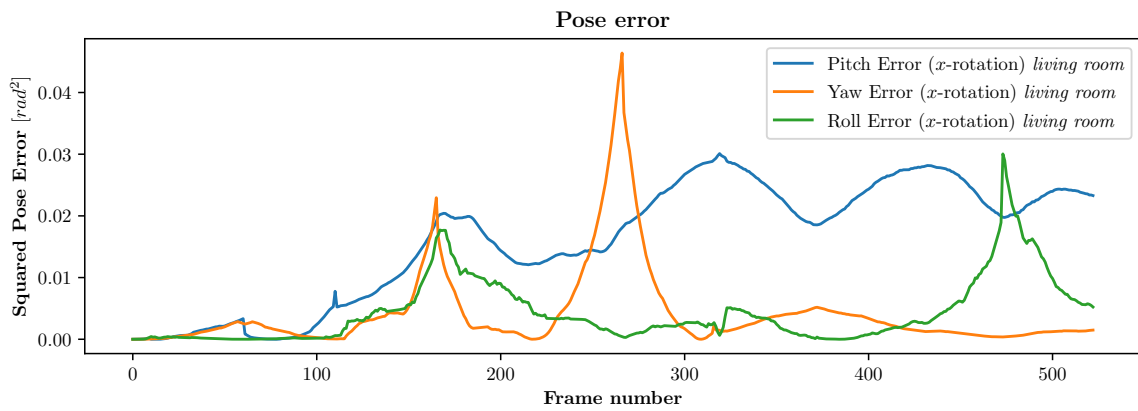


Figure 3.14: Tiled algorithm estimated pose error on *living room* dataset in full 4DoF estimation

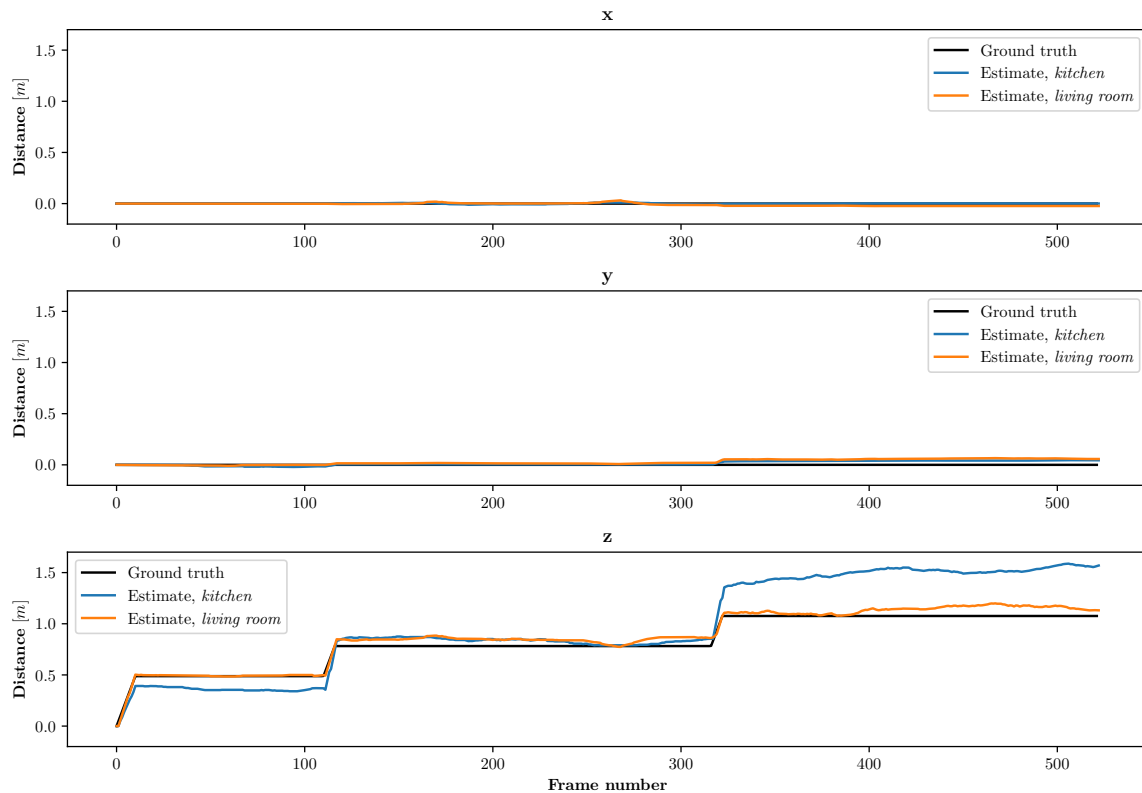


Figure 3.15: Tiled algorithm estimated camera location in full 4DoF estimation

3.6 Scaling and Rotation

While not directly used in the current Pose estimation algorithms presented in this thesis, it is thought to be of importance to be able to perform rotation and scaling operations on CPA for future algorithms. This section introduces an algorithm that allows one to rotate and scale an image on the CPA. While shifting images, or part of images is a very natural operation on the cellular processor array, scaling and rotating is not. The chip behaves according to the single instruction, multiple data paradigm (SIMD) which ensures that every processing element always executes the same instruction. In rotating and scaling however, no two pixels share the same motion vector between the original and the transformed image.

The SCAMP chip allows us to temporarily deselect some processing elements, only executing the next instruction on the selected CPAs. Using this technique, the operation could be implemented nevertheless.

Approach

We use a technique called *rotation by shearing* introduced in Paeth (1986). The presented approach decomposes a rotation by matrix M into three shearing operations, two along the x - axis and one along the y - axis. With a definition of shearing along x and y axis according to

$$S_x(\alpha) = \begin{bmatrix} 1 & \alpha \\ 0 & 1 \end{bmatrix} \quad S_y(\alpha) = \begin{bmatrix} 1 & 0 \\ \alpha & 1 \end{bmatrix} \quad (3.15)$$

Paeth (1986) shows that a rotation matrix can be decomposed according to

$$\begin{bmatrix} \cos \alpha & -\sin \alpha \\ \sin \alpha & \cos \alpha \end{bmatrix} = \begin{bmatrix} 1 & -\tan \frac{\alpha}{2} \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 \\ \sin \alpha & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & -\tan \frac{\alpha}{2} \\ 0 & 1 \end{bmatrix} \quad (3.16)$$

Which is a series of shears along the x and y axis. What remains to be done is to implement a function that performs a shear on the cellular processor array. It turns out that the same algorithm can also be used to implement scaling with minimal changes.

Consider a scaling matrix with scaling coefficient γ . Again, we can decompose the matrix into a scaling to x -direction and a scaling into y -direction, giving us

$$\begin{bmatrix} \gamma & 0 \\ 0 & \gamma \end{bmatrix} = \begin{bmatrix} \gamma & 0 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 \\ 0 & \gamma \end{bmatrix} \quad (3.17)$$

which is a multiplication of a stretch of the image in x direction, followed by a stretch of the image in y direction.

Implementation

Let row r_i , $i \in \{-128, 127\}$ be the i - th row in y direction from the centre of the array. In the SCAMP case with 256 rows, r_1 would be row number 128 counted from the bottom of the chip, starting at 0.

We observe, that a shear by $\alpha \in [-1, 1]$, along the x -axis requires us to shift row r_i by $\alpha \cdot i$ pixels east. Furthermore, we observe that for any row r_j with $|j| > |i|, i \cdot j > 0$, r_j has to be shifted at least $\alpha \cdot i$ pixels. So when shifting row r_i , it makes sense to also shift all the other rows which incur at least the same shift as r_i to save instructions.

The algorithm for a shear into x -direction is outlined in algorithm 3. The *select_rows* function first gets called on half of the entire array, and then on one row less in every iteration of the loop. On the SCAMP chip, this is very easily implemented by selecting the entire half-array first and then shifting the selection up or down by one pixel in every loop iteration. Selecting the entire half array is trivial, thanks to the flexible addressing system (See section 2.1.1). Shears into the y direction are implemented according to the exact same pattern with switched coordinates.

Furthermore, we observe that a scale along the y axis by γ requires us to shift row r_i $\gamma \cdot i$ rows north. This can be done by the exact same algorithm, however instead of *shift_west* we perform *shift_south* and instead of *shift_east* we perform *shift_north*

Results

The algorithm has been implemented and tested on both real hardware as well on the simulator. **Figure 3.16** shows the result of a rotation and scaling by the algorithm recorded on real SCAMP hardware.

Rotation requires us to perform three shears one after the other. It is evident from the images, that rotation does degenerate the image quality quite considerably as it involves quite a few shifts. However, for small rotation angles, the decrease in image quality is less severe.

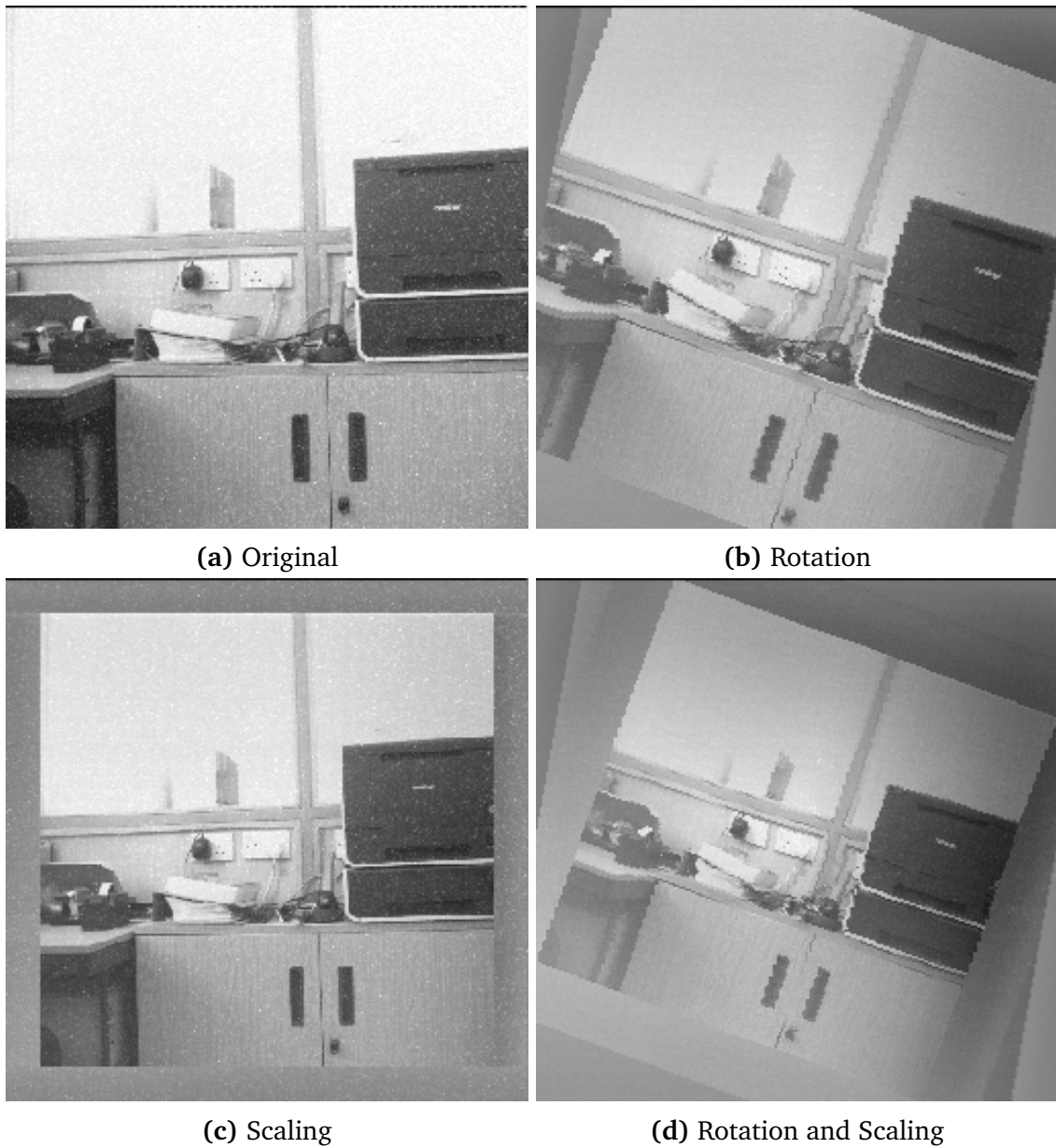


Figure 3.16: Results of applying the rotation and scaling algorithms on real SCAMP hardware

Algorithm 3 Algorithm that performs a shear by α into x-direction on the SCAMP chip. Function *select_rows* makes sure that the next operation only gets executed by the rows given as argument. *shift_east* and *shift_west* perform a shift by one pixel on the selected rows

```

1: procedure SHEARX( $\alpha$ )
2:    $acc \leftarrow 0$ 
3:   for  $i \in 0 \dots 127$  do
4:      $acc \leftarrow acc + |\alpha|$ 
5:     if  $acc \geq 1$  then
6:        $acc \leftarrow acc - 1$ 
7:       select_rows( $r_i \dots r_{127}$ )
8:       if  $\alpha > 0$  then
9:         shift_east
10:      else
11:        shift_west
12:    $acc \leftarrow 0$ 
13:   for  $i \in -1 \dots -128$  do
14:      $acc \leftarrow acc + |\alpha|$ 
15:     if  $acc \geq 1$  then
16:        $acc \leftarrow acc - 1$ 
17:       select_rows( $r_i \dots r_{-128}$ )
18:       if  $\alpha > 0$  then
19:         shift_west
20:      else
21:        shift_east

```

Chapter 4

Automatic Kernel Code Generation

In this chapter, an algorithm is presented to automatically generate program code to execute an arbitrary convolutional kernel on a Cellular Processor Array. The problem consists of a very large search tree, which is in most cases intractable to explore exhaustively. Therefore, we introduce heuristics to point the algorithm to good solutions quickly. The algorithm manages to find good solutions for arbitrary filters quickly. For filters for which a manual implementation exists, the algorithm has proven to generate equivalent or better code.

4.1 Motivation

Cellular Processor Arrays, defined by the nature of their design, require the programmer to think of algorithmic problems in a different way. There are problems, such as the "Sum-Of-Absolute-Differences" class of algorithms described in Chapter 3, that fit to the platform in a very natural way. However, there are algorithms that fit on CPA less intuitively. For simple convolutional filters such as the 3×3 Gaussian filter, or the 3×3 Sobel filter, developers have come up with hand-crafted implementations for CPA. As an example, one could implement the 3×3 Gaussian filter in the following way:

$$K = \frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} = \frac{1}{4} \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \cdot \frac{1}{4} [1 \ 2 \ 1] \quad (4.1)$$

Translated to a SCAMP program, we get a solution with 12 SCAMP instructions, (**Listing 4.1**).

Listing 4.1: Algorithm to perform the 3×3 separable Gaussian filter. Assume original Image in A, and result in D.

```
1 D = div2(A)
2 C = div2(D)
3 E = east(C)
4 D = add(D, E)
5 E = west(C)
6 D = add(D, E)
7 D = div2(D)
8 C = div2(D)
9 E = north(C)
10 D = add(D, E)
11 E = south(C)
12 D = add(D, E)
```

For more complicated filters or large quantities of filters however, it becomes harder or even impractical to manually generate good quality code implementations. An automatic generation of the filter code delegates a time consuming part of the development process to the software, saving developer time. At the same time, it improves abstraction by making the filter representation independent from the underlying hardware, so a future CPA, or similar product may be able to use the same codebase. As many Computer Vision problems can be expressed in terms of convolutional filters, being able to automatically generate code for them forms a basic building block in achieving higher level Computer Vision algorithms on CPA.

4.2 Applications

The first stages of convolutional neural networks for image classification, such as AlexNet (Krizhevsky et al. (2012)), usually consist of pure convolutional layers. The coefficients of the filter kernels are all learned in the training phase of the network. Since we can not make any assumptions on the values prior to learning, the convolutional filters that result appear to be random. An automatic code generation can produce the code implementation of these learned convolution filters automatically, making it easier to deploy these kinds of networks to CPAs.

The algorithm presented in Viola and Jones (2001) uses a large set of very simple Haar features to construct powerful face detection classifier. The algorithm can not only be trained to detect faces, but any class of objects with distinct shape and features. Since we can see the Haar-Like features as very simple convolution kernels, the automatic code generation can produce code to implement a Haar-Like classifier with a large number of features on a CPA. An implementation of a face detector is shown in a later Section 4.6.

4.3 Abstraction Levels

One can identify different levels of abstraction when talking about code for the SCAMP CPA chip. **Table 4.1** shows different abstraction levels of code for the CPA system. Code for the hardware abstraction to levels 1 and 2 is already available (Chen (2016)).

This thesis takes the abstraction to a higher level, by allowing to generate code from a high level description of a filter Kernel. Higher level abstractions like com-

Level	Description	Example
4	Higher level algorithmic description	for (...)
3	Filter Kernel description	$K = [...]$
2	Mapping of arithmetic instructions to load instructions	A = add(C, D)
1	LOAD instructions to move currents between registers	A, B = load(C, D)
0	79-bit instruction word to set switched in PEs	1011001..

Table 4.1: Code abstraction levels

plers for general purpose programming languages have not yet been published and are not in the scope of this thesis.

The approaches presented in subsequent chapters produce code in abstraction level 2, that can be further compiled down to machine instruction words (level 0) by existing tools.

4.4 Contribution

This section describes the theoretic approach as well as experimental results obtained for the code generation problem. The code generation algorithm presented in this thesis takes an arbitrary convolutional filter as an input and outputs a CPA program executable on the SCAMP hardware. Section 4.4.1 provides a short overview of the steps involved in generating a program for CPA.

4.4.1 Overview

The automatic filter code generation can be seen as five individual steps. In a first step, the coefficients of the input filter get approximated to fit to the hardware capabilities of the device. This step is explained in **Section 4.4.2**. Next, the filter gets transformed into a set notation to be decomposed by the reverse split algorithm. This step is described in **Section 4.4.3**.

The reverse split algorithm, described in **Section 4.4.9**, then tries to decompose the filters set into additions of subsets, while keeping track of the operations it performed to get there. This way, by trying to reach the initial state from the final state, we get a plan on how to compose the final state from the initial state.

In a fourth step, we try to reduce the amount of necessary instructions by exploiting equivalence transforms on the computational graph. This operation is described in **Section 4.4.12**. In a last step, a graph colouring algorithm computes a valid register allocation for the computation graph. This is described in **Section 4.4.13**.

The physical register allocation yields a program that can be executed on the cellular processor array without any further modifications.

4.4.2 Value Approximation

While exposing massive parallelism, the individual processing elements of the SCAMP chips are very limited in functionality. Multiplications with arbitrary constants is unsupported on the hardware. However, since multiplications by constants are the underlying principle of digital image filtering, this operation had to be implemented in an alternative fashion.

The SCAMP chip does not support multiplications by arbitrary constants, however, divisions by integer numbers can be performed by sending current into multiple registers simultaneously. So, a division by n requires at least $n + 1$ available registers, including the source registers. Since the chip only has 6 available analogue registers, of which are not always all available and for reasons of simplicity, we restrict any

division to be a division by two. Multiplications by integer numbers can also be emulated in software, by adding the value to itself. Instead of multiplying by a constant, we approximate the effect of multiplication by summing up scalings by powers of 2 of the pixel value.

With these restrictions in place, we approximate target values by divisions and multiplications of two of the original values. Let $\alpha \in \mathbb{R}$ be an arbitrary value in a convolution kernel. Let I be a pixel value. We approximate the scaled value as

$$\alpha \cdot I \approx \sum_{d=-\infty}^D a_d \cdot \frac{1}{2^d} \cdot I, \quad (4.2)$$

with $a_d \in \{-1, 0, 1\}$. These are the coefficients defining if we subtract, ignore or add a certain scaling of the value in order to approximate α . D is the depth of the approximation. Intuitively, approximating to a higher depth yields a better approximation.

In a practical implementation, we can set the $-\infty$ value of the sum to a value that is sufficiently low to capture α . A start value of $s = -\lceil \log_2(|\alpha|) \rceil$ is small enough.

Proof. Assume $\alpha \geq 0$ then it follows that $s = -\lceil \log_2(\alpha) \rceil$. Let $\hat{\alpha} = \sum_{d=-\lceil \log_2(\alpha) \rceil}^D a_d \cdot \frac{1}{2^d}$ is the approximation of α and $D > 0$. With this assumption, the maximum value achievable is $\hat{\alpha} = \sum_{d=-\lceil \log_2(\alpha) \rceil}^D \frac{1}{2^d} \geq \sum_{d=-\lceil \log_2(\alpha) \rceil}^0 \frac{1}{2^d} = 2^0 + 2^1 + \dots + 2^{\lceil \log_2(\alpha) \rceil} \geq 2^{\lceil \log_2(\alpha) \rceil} \geq 2^{\log_2(\alpha)} = \alpha$. For $\alpha < 0$. We have same case as for $\alpha \geq 0$, but with $a'_d = -a_d \forall d$. \square

The approximation problem then reduces to

$$\alpha \cdot I \approx \sum_{d=-\lceil \log_2(|\alpha|) \rceil}^D a_d \cdot \frac{1}{2^d} \cdot I \quad (4.3)$$

Based on equation (4.3), an algorithm to find an optimal set of a_d coefficients given a desired α and depth is outlined in **Algorithm 4**. The algorithm iterates over the sequence $2^{\lceil \log_2(|\alpha|) \rceil}, \dots, 2, 1, \frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \dots, \frac{1}{2^D}$. At every iteration, it decides the a_i coefficient current item according to one of the following rules:

1. If the current absolute error is smaller than half of the contribution of the current item, adding or removing it would bring us further away from the target. We ignore the current item.
2. If the current absolute error is between half the current item and three quarters of the current item, adding it would bring us closer to the target, but adding the next item brings us even closer to the target. We ignore the current item.
3. If the current error is bigger than three quarters of the current item, and adding it brings us closer to the target, we add the current item
4. If the current error is bigger than three quarters of the current item, and subtracting it brings us closer to the target, we subtract it.

Error of Approximation

The algorithm outlined in **Algorithm 4** finds the optimal approximation of a given value and requested depth. However, it introduces a systematic approximation error depending on the input value. The error decreases for higher approximation depths at the expense of additional approximation terms.

Figure 4.1 shows the best approximation errors obtained by algorithm 4. The D value signifies the maximum depth the algorithm is restricted to. One can see that, as expected, going to greater depths does reduce the error for most values. However, the returns on going to deeper approximations get smaller with every new level.

4.4.3 Filter Approximation

In this Section, we extend the approximation equation (4.3), to approximate a complete convolutional filter.

Given a convolutional kernel $K \in \mathbb{R}^{m \times n}$, and maximum approximation depth D , let I be a large enough image, and $I_{i,j}$ the pixel at image coordinates i, j . Let R be the set of coordinates u, v in the neighbourhood of the center point of the convolution filter. For n odd: $u \in \{-\frac{n-1}{2} \dots \frac{n-1}{2}\}$, for m odd: $v \in \{-\frac{m-1}{2} \dots \frac{m-1}{2}\}$. For n even: $u \in \{-\frac{n-2}{2} \dots \frac{n}{2}\}$, for m even: $v \in \{-\frac{m-2}{2} \dots \frac{m}{2}\}$.

We can write the effect of convolutional kernel K onto the image I at position i, j as follows

$$I'_{i,j} = \sum_{u,v \in R} K_{u,v} \cdot I_{i+u,j+v}. \quad (4.4)$$

To implement Equation (4.4) on the CPA, we approximate the values for $K_{u,v} \cdot I_{i+u,j+v}$ using the method described in Equation (4.3). The approximated kernel then looks as follows

Algorithm 4 Find coefficients to approximate value

```

1: procedure APPROXIMATE(target, depth)
2:   result  $\leftarrow$  0
3:   for  $d \leftarrow -\lceil \log_2(|\alpha|) \rceil$ , depth do
4:     if result = target then
5:       return
6:     current  $\leftarrow$   $\frac{1}{2^d}$ 
7:     if  $|result - target| > 3/4 \cdot current$  then
8:       if  $|result + current - target| < |result - current - target|$  then
9:         Set  $b_d = 1$ 
10:      else
11:        Set  $b_d = -1$ 
12:      else
13:        Set  $b_d = 0$ 
14:      result  $\leftarrow result + b_d \cdot current$ 

```

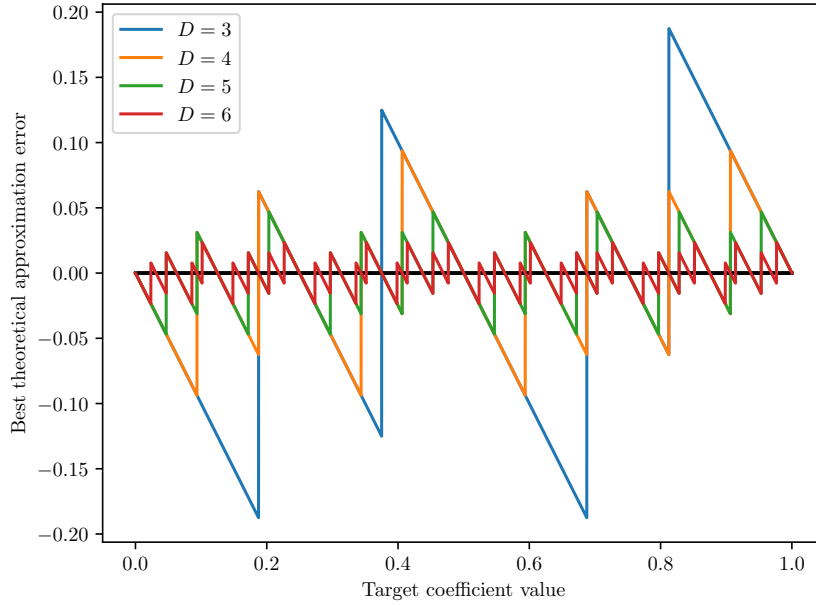


Figure 4.1: Theoretical best approximation errors for certain depths for coefficients in $[0, 1]$.

$$K_{u,v} \cdot I_{i+u,j+v} \approx \sum_{d=-\lceil \log_2(|\alpha|) \rceil}^D a_{d,u,v} \cdot \frac{1}{2^d} \cdot I_{i+u,j+v} \quad (4.5)$$

This results in the following equation for our updated image value

$$I'_{i,j} = \sum_{u,v \in R} \left(\sum_{d=-\lceil \log_2(|\alpha|) \rceil}^D a_{d,u,v} \cdot \frac{1}{2^d} \cdot I_{i+u,j+v} \right) \quad (4.6)$$

With $\frac{1}{2^d} = 2^{-d} = 2^{-d+D} \cdot 2^{-D}$ we can write this as

$$I'_{i,j} = \sum_{u,v \in R} \left(\underbrace{\sum_{d=-\lceil \log_2(|\alpha|) \rceil}^D a_{d,u,v} \cdot 2^{-d+D}}_{:=N(u,v)} \right) \cdot 2^{-D} \cdot I_{i+u,j+v} \quad (4.7)$$

Since D is the maximum approximation depth, $2^{-D} \cdot I$ is the smallest scaling of the input value occurring in the sum. **Equation 4.7** shows a rule on how to assemble the desired filter kernel by a sum of the **elementary scalings** (maximum, 2^{-D}) of the input values.

$N(u, v) \in \mathbb{Z}$ then gives the number of elementary scalings of the image value at $i + u, j + v$ required, to sum up to the desired filter. An approximated filter can

therefore be fully described by the function $N(u, v)$ together with the approximation depth D .

4.4.4 Set Notation

This section introduces a different notation of the approximate filter, as well as other preliminary definitions that are used throughout the chapter.

Let the function $\text{repeat}(a, N)$ signify the set of N copies of a .

Definition 1. Let an **atom** be a tuple $([id], u, v)$ where id is an arbitrary integer unique to all other defined tuples, u, v denote the coordinates relative to the center of the filter. An atom symbolises the contribution of an elementary scaling (lowest possible scale) at coordinates (u, v) to the final filter sum.

Definition 2. Let a **goal** be a set of atoms. The atoms in a goal are to be viewed as an addition of the atoms values.

Definition 3. Let a **final goal** FG be the goal (set of atoms), such that $FG = \{\text{repeat}([*id], u, v), N(u, v) : \forall (u, v) \in R\}$ where $*id$ is an arbitrary integer, unique in the set. For every position in $u, v \in R$ we add exactly $N(u, v)$ atoms to the set, each with a unique id .

Definition 4. The **initial goal** represents the set of atoms that is present in a register before doing any operations. Let D be the approximation depth, then $IG = \{\text{repeat}([*id], 0, 0), 2^D\}$ where $*id$ are the ids of the atoms at $u = 0, v = 0$ in the final set for as long as there are such atoms, arbitrary integers, unique in the set otherwise. We add exactly 2^D atoms with coordinates $(0, 0)$ to the set. Each atom has to have a unique id .

Example 1. A 3×3 gaussian filter can be written as

$$K = \begin{bmatrix} \frac{1}{16} & \frac{2}{16} & \frac{1}{16} \\ \frac{2}{16} & \frac{4}{16} & \frac{2}{16} \\ \frac{1}{16} & \frac{2}{16} & \frac{1}{16} \end{bmatrix} \quad (4.8)$$

With an approximation depth of $D = 4$ we manage to approximate all values without an error, according to Algorithm 4. In set notation, the final goal of this filter becomes

$$FG = \left\{ \begin{array}{cccc} ([0], -1, -1), & ([1], 0, -1), & ([2], 0, -1), & ([3], 1, -1), \\ ([4], -1, 0), & ([5], -1, 0), & ([6], 0, 0), & ([7], 0, 0), \\ ([8], 0, 0), & ([9], 0, 0), & ([10], 1, 0), & ([11], 1, 0), \\ ([12], -1, 1), & ([13], 0, 1), & ([14], 0, 1), & ([15], 1, 1) \end{array} \right\} \quad (4.9)$$

With the initial goal being

$$IG = \left\{ \begin{array}{cccc} ([6], 0, 0), & ([7], 0, 0), & ([8], 0, 0), & ([9], 0, 0), \\ ([100], 0, 0), & ([101], 0, 0), & ([102], 0, 0), & ([103], 0, 0), \\ ([104], 0, 0), & ([105], 0, 0), & ([106], 0, 0), & ([107], 0, 0), \\ ([108], 0, 0), & ([109], 0, 0), & ([110], 0, 0), & ([111], 0, 0) \end{array} \right\} \quad (4.10)$$

With $D = 4, 2^{-D} = \frac{1}{16}$

Since the function $N(u, v)$ is given by the approximation algorithm and the approximation depth D , we can easily construct the final goal of any arbitrary filter automatically.

Looking at the correspondence of this set notation to the actual implementation on the chip one can see that the notion of a goal is essentially a sum of different scalings of pixel values from positions around the center pixel. A goal can be stored in a register. Every atom corresponds to 2^D scale of a pixel value at a certain offset. Having a goal stored in a register therefore means having the sum of the scalings of pixel values in a register as described by the atoms of the goal.

4.4.5 Goal Transformations

The nature of the CPA allows us to transform goals into other goals. Since every processing unit performed the same operations up until this point, the adjacent processing units of a particular pixels hold the same values, but computed on their source data. A value stored in a CPA register can be seen as a goal, an accumulated set of different scalings of pixel values. Shifting the goal to one side essentially converts the goal to a goal with same scalings, but with different coordinates. We differentiate between shift transformations that operate by shifting goals to neighbouring processing units, scale and negate transformations that operate on the processing unit by dividing, negating and adding to itself. Any combination of these transformations is still a valid transformation.

Negation

Let F and G be two goals. Goal F is the negation of goal G iff $\forall a \in G \exists -a' \in F$ and $\exists a \in G \forall -a' \in F$ s.t. $a_u = a'_u, a_v = a'_v, a_{id} \neq a'_{id}$.

Example 2. A negation of $G = \{([0], 1, -2), ([1], 0, 1)\}$ is $F = \{-([2], 1, -2), -([3], 0, 1)\}$

Shifts

Assume a processing element holds goal $\{([0], 0, 0)\}$ in a register, from its point of view, the processing unit on the left holds goal $\{([\epsilon], -1, 0)\}$, that could be obtained by shifting the value in. However, we can look at the problem solely from the local processing unit, by saying, that we can convert a goal into another by changing the coordinates and the ids of all atoms in the goal.

Let G and F be goals. If $G \cap F \neq \emptyset$, we can not transform one goal into the other, as their corresponding elements would no longer be the same after shifting. This would violate the equality constraint.

Otherwise, if $G \cap F = \emptyset$ we can transform G into F iff $\forall a \in G \exists a' \in F$ and $\exists a \in F \forall a' \in G$ and $\exists m, n$ s.t. $a_u + n = a'_u, a_v + m = a'_v, a_{id} \neq a'_{id}$. (There is an offset mapping from every element in F to an element in G). The values a_u, a_v, a_{id} represent the atoms x, y and id properties respectively.

Example 3. The goal $G = \{([0], 0, 0)\}$ can be transformed into $F = \{([1], 5, -2)\}$ with $m = 5, n = -2$.

The goal $G = \{([0], 0, 0), ([1], 1, 0)\}$ can be transformed into $F = \{([2], 0, -1), ([3], 1, -1)\}$ with $m = 0, n = -1$.

The goals $G = \{([0], 0, 0), ([1], 1, 0)\}$ and $F = \{([2], 0, -1), ([3], 1, -2)\}$ can not be transformed into each other, as there exists no m, n to shift G to F

Scales

Let $|\cdot|$ denote the cardinality of a set. In addition to shifting goals, we can scale some goals by a factor of two. Let F and G be goals. We can scale F down to transform into G iff $G \subset F$, $|F| = 2 \cdot |G|$ and $\forall a \in G \exists a' \in F$ s.t. $a_u = a'_u, a_v = a'_v, a_{id} \neq a'_{id}$ (for every atom in G there exists an atom in F with same coordinates but different id). The same condition holds true for scaling up, with $F' = G$ and $G' = F$.

Example 4. The goal $G = \{([0], 0, 0), ([1], 0, 0), -([2], 0, 1), -([3], 0, 1)\}$ can be transformed into $F = \{([0], 0, 0), -([2], 0, 1)\}$.

The goal $G = \{([0], 0, 0), ([1], 0, 0), ([2], 0, 1)\}$ can be transformed to $F = \{([0], 0, 0), ([1], 0, 0), ([4], 0, 0), ([5], 0, 0), ([2], 0, 1), ([3], 0, 1)\}$

Additions

While not considered a goal transformation, additions are still listed here to complete the set of possible operations. Let F and G be two goals. We define the addition of F and G only to be valid, if $F \cap G = \emptyset$ (the sets are disjoint). Then, the addition of goals F and G is the union of both sets: $F + G = F \cup G$

4.4.6 States

Definition 5. A *state* is an assignment of goals to integer numbers (slots). One can see this as an assignment to virtual registers.

Example 5.

$$S_e = \begin{cases} 0 : \{([0], -1, -1), ([1], 0, -1)\} \\ 1 : \{([9], 0, 0)\} \\ 2 : \{([13], 0, 1), ([14], 0, 1)\} \end{cases} \quad (4.11)$$

The state S_e contains 3 slots, slots 0 and 2 have goals with 2 atoms each, while slot 1 has a goal with only one atom.

Definition 6. The *initial state* is the state with the initial goal assigned to slot 0.

$$S_0 = \{0 : IG\} \quad (4.12)$$

Definition 7. A *final state* is a state which contains the final goal.

$$S_F = \begin{cases} \vdots \\ k : FG \\ \vdots \end{cases} \quad (4.13)$$

With $k \in \mathbb{N}$ an arbitrary slot.

Definition 8. A **transformation** between state slots is a combination of shift, scale and negate transformation of a goal in a source state slot with subsequent assignment to a target state slot. A transformation of state slot a to state slot b with h horizontal shift, v vertical shift s scaling and n negation is represented as $b \leftarrow \text{transform}(a : \rightarrow (v) \uparrow (h) + (s) \neg(n))$

Definition 9. An **addition** between state slots a and b with subsequent assignment to a target state slot c can be written as $c \leftarrow \text{add}(+a, +b)$, where $-$ signs can be used to invert an operand.

4.4.7 Graph Representation

A similar notation is introduced for representing the state transformations and additions as a directed graph. The symbols in **Table 4.2** can be used to express a series of transformations and additions on the state as a graph.

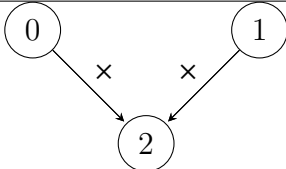
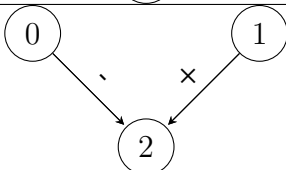
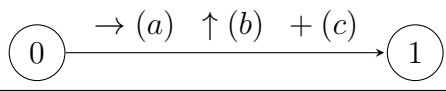
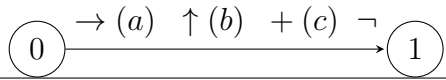
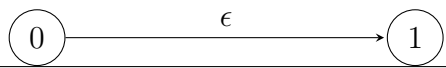
Operation	Representation	CPA instructions
Addition of state slot 0 and 1 into state slot 2		add
Addition of negative slot 0 and positive slot 1 into 2 (Subtraction)		sub
Shift of slot 0 right by a , up by b , scale down c times into slot 1		north, east, south, west, div2
Shift and scale slot 0, negate and store into slot 1		north, east, south, west, div2, neg
Empty shift (Copy) from slot 0 to slot 1		copy, nop

Table 4.2: Graph representation styles

The node numbers represent the slots in the state that we write to or read from. Note that the algorithms introduced later in this chapter never write to the same state slot twice, and monotonically increase the state slot they write to. Therefore, for all graphs in this thesis, the node (state slot) numbering follows the order in which the computation is carried out.

4.4.8 State Transformation Plans

In order to find a valid program for the convolution filter, we have to find a path of transformations from the initial state to reach a final state. Since the transformations described in Section 4.4.5 all map to elementary CPA instructions, it is then a relatively easy task to compile a program. There are multiple ways to achieve this. A straight forward, sequential algorithm is outlined below.

Sequential Approach

Algorithm 5 describes a sequential algorithm to compute a plan to reach a final state from the initial state. As defined above, the initial state has only the initial goal assigned to slot 0. The initial goal contains exactly 2^D atoms with coordinates $(0, 0)$.

In the first while loop, we scale down initial goal, until only a single atom remains. In every step, we remove half of the atoms, until we are only left with a single atom. Since the register originally contains 2^D atoms, this is always possible. Now, originating from this single atom, we can generate every other atom present in the final goal and add it to an accumulator. This again is always possible, as two single atoms can always be transformed into each other. We perform this for every atom in the final goal. In a last step, we add the generated atom to a sum of previously generated atoms, which in the end yields the final goal. Note that we always

Algorithm 5 Sequential, naive algorithm to generate code to achieve the final goal. The \ll operations signifies an "append" operation.

```

1: procedure SEQUENTIALPLAN(finalGoal, D)
2:   plan  $\leftarrow$  []
3:   prevAdd  $\leftarrow$  -1
4:   prevSlot  $\leftarrow$  0
5:   nextSlot  $\leftarrow$  1
6:   for  $i = \{0 \dots D\}$  do
7:     plan  $\ll$  (nextSlot  $\leftarrow$  transform(prevSlot : +(1)))
8:     prevSlot  $\leftarrow$  nextSlot
9:     nextSlot  $\leftarrow$  nextSlot + 1
10:  source  $\leftarrow$  prevSlot
11:  for  $a \in$  finalGoal do
12:    plan  $\ll$  (nextSlot  $\leftarrow$  transform(source :  $\rightarrow (a_u) \uparrow (a_v) \neg(a_{neg})$ ))
13:    prevSlot  $\leftarrow$  nextSlot
14:    nextSlot  $\leftarrow$  nextSlot + 1
15:    if prevAdd  $\geq$  0 then
16:      plan  $\ll$  (nextSlot  $\leftarrow$  add(+prevNr, +prevAdd))
17:    else
18:      plan  $\ll$  (nextSlot  $\leftarrow$  transform(prevSlot :  $\epsilon$ ))
19:    prevAdd  $\leftarrow$  nextSlot
20:    prevSlot  $\leftarrow$  nextSlot
21:    nextSlot  $\leftarrow$  nextSlot + 1

```

consider the lifetime of an assignment to the state to be infinite, we never overwrite a slot we have written to before. The slots get assigned to physical registers at a later stage, however, by simple static analysis one can see that programs generated by this simple algorithm will always require at most 3 physical registers. This algorithm is always feasible as long as we have 3 available physical registers and always generates a correct result. However, the runtime is generally far from optimal, as the algorithm does not make any smart choices.

Example 6. *A very simple toy example*

$$K = [0.5 \quad 1 \quad 0.5] \quad (4.14)$$

Yields the following initial goal and final goal:

$$IG = \{ ([0], 0, 0), ([1], 0, 0) \} \quad (4.15)$$

$$FG = \{ ([0], 0, 0), ([1], 0, 0), ([2], -1, 0), ([3], -1, 0) \} \quad (4.16)$$

Using the sequential algorithm to generate a plan yields the result outlined in Listing 4.2

Listing 4.2: Toy example with sequential algorithm

1	1 \leftarrow transform(0 : +(1))	6	6 \leftarrow transform(1 : ϵ)
2	2 \leftarrow transform(1 : \leftarrow (1))	7	7 \leftarrow add(+5, +6)
3	3 \leftarrow transform(2 : ϵ)	8	8 \leftarrow transform(1 : \rightarrow (1))
4	4 \leftarrow transform(1 : ϵ)	9	9 \leftarrow add(+7, +8)
5	5 \leftarrow add(+3, +4)		

The state obtained after the last step holds the final goal at slot 9 and is therefore a final state. While this plan is correct and works as intended, it is with 9 steps much longer than the optimal solution outlined in listing 4.3, which takes only 5 steps.

Listing 4.3: Optimal plan for toy example

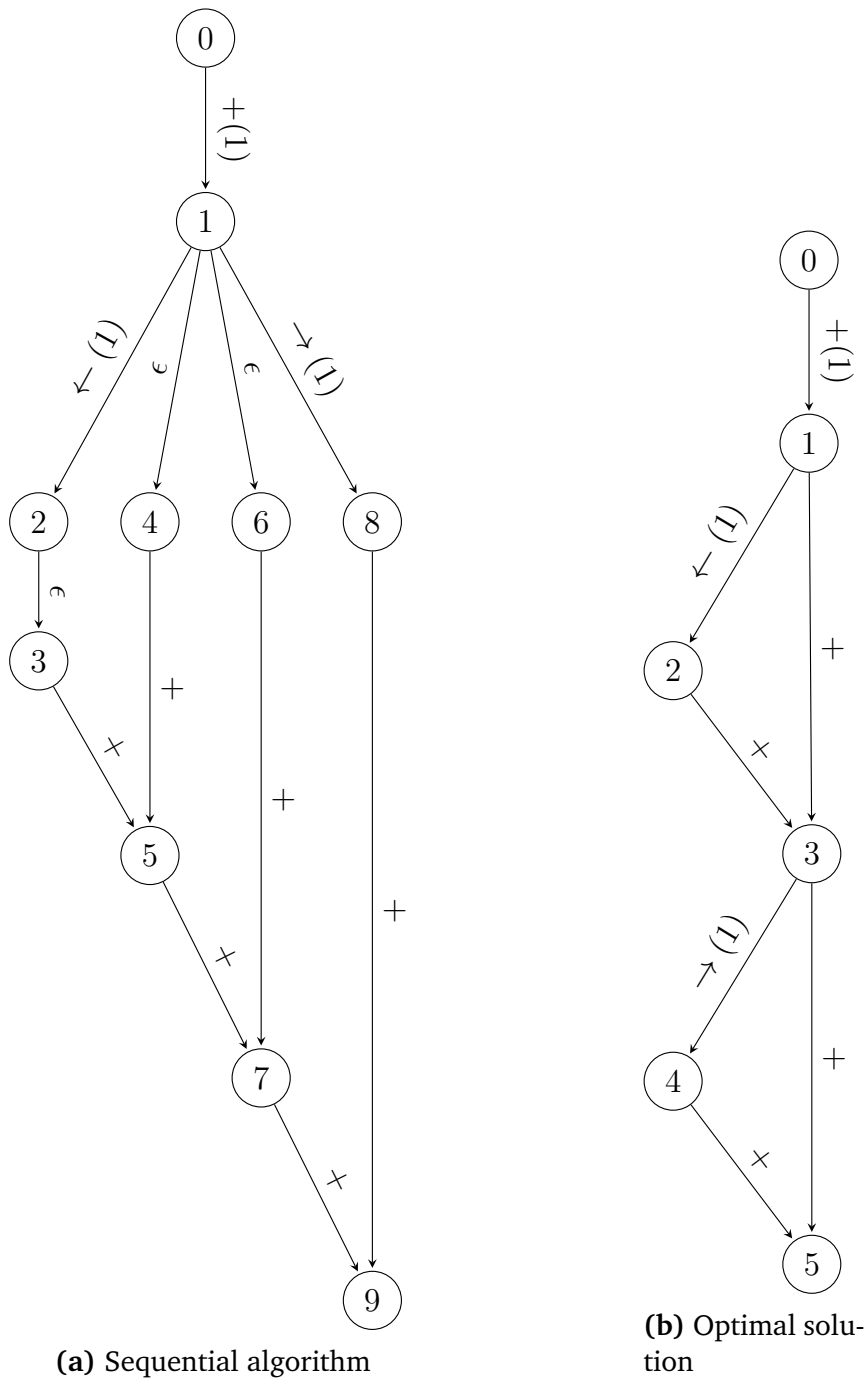
1	1 \leftarrow transform(0 : +(1))	4	4 \leftarrow transform(3 : \rightarrow (1))
2	2 \leftarrow transform(1 : \leftarrow (1))	5	5 \leftarrow add(+3, +4)
3	3 \leftarrow add(+1, +2)		

*Both plans are represented as a graph in **Graph 4.1**.*

The problem that example 6 shows is, that the sequential algorithm compiles a plan for every atom in the final set individually, without taking the whole set into account. Therefore, it completely fails to reuse sub expressions.

4.4.9 Reverse Splitting Algorithm

The sequential approach in Algorithm 5 showed weaknesses in finding good solutions even for very a small problem. A better approach is proposed here to find more optimal plans for assembling the final set from the initial set.



Graph 4.1: Graph representation of the toy example from example 6. The nodes represent sub results for the final results. Edges labeled with ϵ are empty operation. Edges with numbers represent a transformation of a value (shift or scale) where as edges with just a plus represent additions of sub results. The optimal graph is better because it contains fewer operations.

Motivation

As described, the final goal is a heterogeneous mix of different quantities of atoms with different coordinates. The initial goal however only contains atoms of a single coordinate $(0, 0)$. In order to generate the heterogeneous mix of the final goal, we have to assemble different sub goals which we then can add together.

Algorithm 5 does follow the same principle, however, it is only capable of generating and adding up sub goals with one atom in length. This is apparent in the graph, where every shifting edge originates from state slot 1 which only contains one atoms as required by the algorithm.

A smarter approach would be to not naively generate single atoms for the final solution, but to generate sub goals for the final solution which in turn can be transformed again to form other sub goals.

Example 7. *If we look at the final goal for the 3×3 Gaussian filter in example 1, the sequential algorithm would generate and assemble all the 16 atoms of the final goal separately. If we have a closer look at the final set, we can spot the following optimisation.*

Assume a program would generate state with the following sub goal (essentially the middle row) assigned to a slot $k \in \mathbb{N}_\neq$

$$\left\{ \begin{array}{l} \vdots \\ k : \left\{ \begin{array}{l} ([4], -1, 0), ([5], -1, 0), ([6], 0, 0), ([7], 0, 0), \\ ([8], 0, 0), ([9], 0, 0), ([10], 1, 0), ([11], 1, 0) \end{array} \right\} := S \subset FG \\ \vdots \end{array} \right. \quad (4.17)$$

If we look at the remaining parts of of the final goal, we can write them as

$$FG \setminus S = \underbrace{\left\{ \begin{array}{l} ([0], -1, -1), ([1], 0, -1), \\ ([2], 0, -1), ([3], 1, -1), \end{array} \right\}}_{R_0} \cup \underbrace{\left\{ \begin{array}{l} ([12], -1, 1), ([13], 0, 1), \\ ([14], 0, 1), ([15], 1, 1) \end{array} \right\}}_{R_1} \quad (4.18)$$

It is apparent that goals S and R_0 , S and R_1 and R_0 and R_1 all can be transformed into each other. This follows from the fact, that the coefficients of the lower and upper rows of the gaussian filter are just one half the coefficients of the middle row.

A smart algorithm can now easily compute the remaining parts R_0 and R_1 by reusing the result previously computed goal, saving massive amounts of work. Generating R_0 from S can be done in just two instructions.

As shown in example 7, the order in which we generate and add up sub goals is crucial to achieve a good result.

Assumptions

We assume, that it is never beneficial to generate a sub goal, which is not part of the final goal. Let S be any state in the execution of the program, G a goal in a slot of S , we can say that $\forall G \in S, G \subseteq FG$ for FG the final goal.

This is a sensible assumption as it ensures that the program never computes anything that is not part of the final solution.

Search Space and Optimality

The initial goal IG only contains 2^D atoms with coordinates $(0, 0)$. Any non empty final goal FG contains at least one or more atoms, with arbitrary coordinates. If the initial goal is directly transformable to the final goal, there is a defined, optimal way of doing so, given by the set of possible transformations.

If the final goal is not directly transformable from the initial goal, the final goal is an addition of two subgoals. There are $2^{|FG|}$ different possibilities of splitting the final goal into two sub goals. each of these two sub goals are again either directly transformable from the initial goal or a sum of yet two other sub goals. At some point, all the sub goals will be directly computable from the initial goal.

We can be sure that this search space contains all the possibilities to build a split tree to come up with the final goal. Note that this still does not take into account that we may have the possibility to reuse sub results. It just guarantees that we find the set of all possible splits. In order to optimise this, we would have to tie branches of the tree together to reduce computation.

Algorithmic discovery

Described here is the reverse split algorithm that explores the search space mentioned before. Let FG be the final goal. We start with the final goal and perform a split on it into three parts $FG = U + L + R$, where we call U the **upper goal**, L the **lower goal** and R the **rest goal**.

We require U and L to be transformable into each other. Therefore, $(|U| > 0) \leftrightarrow (|L| > 0)$. R always contains the rest that is not covered by U and L , in the extreme case where we can't find a U and L , $FG = R$. In the other extreme case, where we can perfectly split into U and L , we have $R = \emptyset$.

If we now build a program that can generate L and R , we know how to assemble FG as U can be generated from L . We therefore no longer need to consider U . One could say that we get U for "free" by generating L . From the final goal we had in the beginning we now generated two goals L and R that we have to generate. If the split was smart, generating L and R should be an easier task than generating FG , bringing us closer to the solution. We can now perform a similar step on L and R . Let L_i, U_i , denote the lower and upper set generated at step i , with $L_0 = L, U_0 = U$, $R_i^{(j)}$ denote the j -th rest step obtained at step i , with $R_0^{(0)} = R$. Step 0 was already performed above. For $i = 1$ we now have the following options:

1. Split L_0 : $L_0 = U_1 + L_1 + R_1^{(0)}$ have $\{R_0^{(0)}, L_1, R_1^{(0)}\}$ for next step
2. Split $R_0^{(0)}$: $R_0^{(0)} = U_1 + L_1 + R_1^{(0)}$ have $\{L_0, L_1, R_1^{(0)}\}$ for next step
3. Split L_0 and $R_0^{(0)}$: $L_0 = U_1 + R_1^{(0)}, R_0^{(0)} = L_1 + R_1^{(1)}$ have $\{R_1^{(0)}, L_1, R_1^{(1)}\}$ for next step

Assume for any step i we have n goals to split. Analogous to the case $i = 1$, we can either split any of those n goals individually or, we can choose any two of the goals to split together. This yields $n + \binom{n}{2}$ possibilities to choose sets to split. Since the R goals can be empty, we add or remove at at maximum one goal to the list of goals to obtain in every step. We perform the algorithm until we arrive at a single goal that can directly be transformed from the initial goal. A plan is then given by reversing the list of splits the algorithm performed.

Note that to execute the plan, we have to hold each of the goals in a register in the device. So in every step, we can omit splits that would yield more sub goals than the amount of available hardware registers. These cuts greatly improve search performance as well as ensure that every plan we find is actually executable with the amount of hardware registers available. In every step, we have $n + \binom{n}{2}$ possibilities to choose goals to split. This is potentially a very big number. However, since n is never bigger than the amount of hardware registers, this is generally not a problem. Especially on the SCAMP, we only have 6 registers of which usually only 3 are really available to perform the computation, the others being occupied by functionality outside the scope of the convolution kernel.

A recursive implementation of the algorithm is outlined in **Algorithm 6**.

Algorithm 6 Reverse split algorithm. The \ll operation signifies an "append" operation. `generatePairs` generates all the split-pairs that can be applied to the current goals. `isTransformable(a, b)` returns true, iff a and b are transformable into each other.

```

1: procedure REVERSE_SPLIT(goals, currentPlan)
2:   if |goals| == 1 and isTransformable(goals[0], IG) then
3:     plans  $\ll$  currentPlan + (goals, (initialGoal  $\mapsto$  goals[0]))
4:   return
5:   upGoals, lowGoals  $\leftarrow$  generatePairs(goals)
6:   for (upGoal, lowGoal)  $\in$  (upGoals, lowGoals) do
7:     newGoals  $\leftarrow$  []
8:     for goal  $\in$  goals do
9:       newGoal  $\leftarrow$  goal \ (upGoal  $\cup$  lowGoal)
10:      if |newGoal| > 0 then
11:        newGoals  $\ll$  newGoal
12:      newGoals  $\ll$  lowGoal
13:      ReverseSplit(newGoals, currentPlan + (goals, (lowGoal  $\mapsto$  upGoal)))

```

Algorithm 6 shows a minimalistic implementation of the reverse split algorithm. The algorithm works exhaustively by evaluating all the possible splits possible in every step. At every step, the algorithm adds a tuple with the goals to have in this step, as well as the pair of lower and upper goals that one needs to use in order to achieve this state from the previous one. Whenever the algorithm finds a solution, it adds it to the total list of solutions. Note that the reverse split algorithm does not directly generate a plan with additions and transformations. A second program generates the actual plan from the output of the reverse split algorithm.

This program figures out which goals have been splitted and works out the necessary add and transform operations to get a graph-able plan.

Branch Cutting

The reverse split explores the search tree in a depth-first fashion. If we have a good pair generation function, it is likely to find good solutions even early on. As soon as the algorithm finds a solution, the cost of that solution is stored as the current best. From now on, whenever the algorithm descends down a path that already has an accumulated cost higher than the already found minimum, the branch is cut of.

Provided a good pair generation function that leads the algorithm to good solution in the beginning, this can reduce the search space and allows the algorithm to prove the optimality of a result more quickly.

4.4.10 Pair generation

The reverse split algorithm as outlined in Algorithm 6 relies on the function `generatePairs((goals))` that essentially generates all the possible splits of the goals in the current step. In order to guarantee that we explore the whole search space, this function has to return a complete list of upper and lower goals that are applicable in this particular step. As mentioned before, if we have n goals, we have $n + \binom{n}{2}$ possibilities to select the goals we want to use in the splitting: Either select a single goal to split, or select two goals out of the n goals to split together (One getting the upper goal, the other getting the lower goal).

Two goals that are transformable into each other have to be disjoint. Therefore, it is sufficient to consider the case of splitting two goals, while considering the case of a single goal as splitting the goal and a copy of itself.

Overview

Assume we have two goals F and G , which may or may not be the same. We require $U \subset F$, $L \subset G$, $L \cap U = \emptyset$ and U and L transformable into each other. We have to find all possibilities of such sets U and L which fulfil subset and transformability requirements. The algorithm to generate pairs works essentially by sorting and grouping together atoms that share certain properties, followed by an exhaustive generating step.

Figure 4.2 Shows a rough overview of the steps performed. At first, we calculate the distances between every atom from F to every atom in G . We then group together all pairs of atoms that share the same distance. Since there are usually more than one atoms at a specific coordinate location, there will be equivalent pairs in these groups. The next steps are all performed on each group individually. At first, we find and group pairs that share the same coordinates. From the pairs with the same coordinates, we can find all scalings (scale transformations) that are possible with these atoms at the specific coordinates.

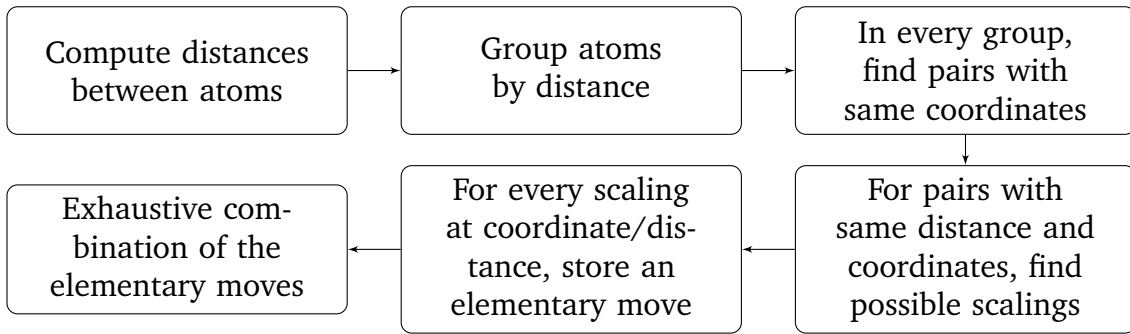


Figure 4.2: Steps performed for obtaining split pairs. The reverse splitting algorithm creates a plan to build a program by splitting up the set representation of the filter kernel. In order to so, it requires split pairs that are generated according to this flowchart.

Having obtained all this information, we assign an "elementary move" for every found possible transformation. In a last step, we group the elementary moves that share the same distances and scaling together in an exhaustive fashion.

Distances

For atoms $a \in F$, $b \in G$, we define the vector $d(a, b) = (a_x - b_x, a_y - b_y)$ as the **distance** between the two atoms.

In a first step, we compute the distances from every atom in F to every atom in G , and group the pairs together that share the same distance.

Let A be the set of all pairs (a, b) , $a \in F, b \in G$. We consider the sets $X \subseteq A$ such that $\exists D \in \mathbb{Z}^2. \forall (a, b) \in X, d(a, b) = D$

For two goals to be transformable into each other, all the atoms from the first set must have a correspondence to an atom in the second set and vice versa, with all those correspondences having the same distance. Therefore, for further analysis, we can consider the groups of pairs with same distance individually, without losing opportunities to find additional transformable sets U and L .

Proof. For a pair of atoms (a_1, b_1) with distance $d(a_1, b_1) = d_1$ and another pair of atoms (a_2, b_2) with distance $d(a_2, b_2) = d_2$, $d_1 \neq d_2$ and $d_1 \neq -d_2$ it follows that $\nexists m, n$ to fulfil the shift transformation condition for the two goals $U = \{a_1, a_2\}$ and $L = \{b_1, b_2\}$. \square

Example 8. For goals $F = \{([0], 1, 1), ([1], 1, 1), ([2], 0, 1)\}$ and $G = \{([3], 1, 0), ([4], 0, 0)\}$
Computing the distances, between all atoms we get (only ids printed)

F	G	Distance
[0]	[3]	(0, -1)
[0]	[4]	(-1, -1)
[1]	[3]	(0, -1)
[1]	[4]	(-1, -1)
[2]	[3]	(1, -1)
[2]	[4]	(0, -1)

Table 4.3: Distances between the atoms of F and G . The values in square brackets are the ids of the atoms.

Grouping the pairs by distance we get

Distance	Pairs
(0, -1)	([0], [3]), ([1], [3]), ([2], [4])
(-1, -1)	([0], [4]), ([1], [4])
(1, -1)	([2], [3])

Table 4.4: Grouped distances for pairs of F and G . The values in square brackets are the ids of the atoms.

Elementary moves

Only considering a single group of pairs of atoms with same distance, we can do further analysis. The pairs in the group differ either by coordinates (same shift, but at other coordinates) or just by the ids of the individual atoms. When two pairs in a group do not differ by coordinates, they provide evidence that there may be an opportunity for scale transformations as well.

We group together those pairs in the group, that have the same coordinates in the left atom (and since they all share the same distance, they also have the same coordinates in the right atom). We then extract the distinct ids from the left and right atoms of the pairs, and call this a **cluster**.

Let B be a set of pairs with the same distance. We consider all clusters $l \rightarrow r$ with $l = \{a_{id} : a_{id} \text{ distinct}\}$, $r = \{b_{id} : b_{id} \text{ distinct}\}$ for all pairs (a, b) in the $X \subseteq B$ such that $\exists x, y \in \mathbb{Z}$, $\forall (a, ?) \in X$, $a_x = x$, $a_y = y$. The ? represents an arbitrary atom.

Example 9. Reconsidering the problem from example 8, we build up the following clusters for the groups

Distance	Pairs	Clusters
(0, -1)	([0], [3]), ([1], [3]), ([2], [4])	$\{[0], [1]\} \rightarrow \{[3]\}$ $\{[2]\} \rightarrow \{[4]\}$
(-1, -1)	([0], [4]), ([1], [4])	$\{[0], [1]\} \rightarrow \{[4]\}$
(1, -1)	([2], [3])	$\{[2]\} \rightarrow \{[3]\}$

Table 4.5: Clusters formed for the individual groups. Atoms [0] and [1] end up in the same cluster, as they share the same coordinates.

A cluster $l \rightarrow r$ essentially describes the existence of transformations at a certain coordinate by a given distance. The transformation can start with any number in $|s| = 1 \dots |l|$ atoms, and can generate any number of $|t| = |s| \cdot 2^k, 1 \leq |s| \cdot 2^k \leq |r|$ atoms, $k \in \mathbb{Z}$. This property is once again defined by the allowed transformations, as only scalings of multiples of 2 are allowed. Furthermore, we have to operate on at least one atom and we can not use more atoms that are available in l and r respectively. As a next step, we can generate all the possible moves that are available per cluster. From cluster $l \rightarrow r$ we generate the **elementary moves** $s \mapsto t$ according to $s \subseteq l, |s| \geq 1$ and $t \subseteq r, 1 \leq |t| \cdot \exists k \in \mathbb{Z} s.t. |s| \cdot 2^k = |t|$

Example 10. Continuing example 9 we can generate the elementary moves for the clusters obtained

Distance	Pairs	Clusters	Elementary moves
(0, -1)	([0], [3]), ([1], [3]), ([2], [4])	$\{[0], [1]\} \rightarrow \{[3]\}$	$\{[0]\} \mapsto \{[3]\}$
		$\{[2]\} \rightarrow \{[4]\}$	$\{[0], [1]\} \mapsto \{[3]\}$ $\{[2]\} \mapsto \{[4]\}$
(-1, -1)	([0], [4]), ([1], [4])	$\{[0], [1]\} \rightarrow \{[4]\}$	$\{[0]\} \mapsto \{[4]\}$
			$\{[0], [1]\} \mapsto \{[4]\}$
(1, -1)	([2], [3])	$\{[2]\} \rightarrow \{[3]\}$	$\{[2]\} \mapsto \{[3]\}$

Table 4.6: Elementary moves generated from the clusters. As an example, the cluster $\{[0], [1]\} \rightarrow \{[3]\}$ yields two elementary moves. The first one $\{[0]\} \mapsto \{[3]\}$ involving no scaling, the second one $\{[0], [1]\} \mapsto \{[3]\}$ involving a division by two. We omitted $\{[1]\} \mapsto \{[3]\}$ as it is indistinguishable from $\{[0]\} \mapsto \{[3]\}$

Note that we omitted the elementary move $\{[1]\} \mapsto \{[3]\}$ as it is indistinguishable from the move $\{[0]\} \mapsto \{[3]\}$. The same holds for $\{[1]\} \mapsto \{[4]\}$ and $\{[0]\} \mapsto \{[4]\}$

As already noted in example 10, elementary moves generated from the same cluster that share the same coordinates and cardinalities are indistinguishable from each other. It turns out that only keeping one example of each combinations of cardinalities can reduce the search space enormously without altering the functionality.

Exhaustive Grouping of Elementary Moves

The last step that is left to do is to assemble the elementary moves from a group a pairs of atoms with same distance to arrive at the sets U and L .

Let X be a set of elementary moves, then $U = \bigcup_{s \mapsto t \in X} t$ and $L = \bigcup_{s \mapsto t \in X} s$

What is left to do is find valid sets of elementary moves X . Let $\text{scaling}(s, t) = \frac{|s|}{|t|}$ be the scaling of an elementary move $s \mapsto t$ and E be the set of all elementary moves for a group of pairs of the same distance, then any $X \subseteq E$ such that $\exists f \in \mathbb{R}. \forall (s \mapsto t) \in X, \text{scaling}(s, t) = f$ is a valid set of elementary moves.

The reasoning behind this is, that upon transformation all the all the atoms from various coordinates will be performing the same scaling transformation, therefore, the ratios of the number of atoms at the various positions has to be the same.

Example 11. *Table 4.7 shows an exhaustive set of all split pairs U , L that can be generated from the sets G and F . The prior results are taken from example 10.*

Distance	Elementary moves	U / L goals	
(0, -1)	$\{[0]\} \mapsto \{[3]\}$	$U = \{([0], 1, 1)\}$	$L = \{([3], 1, 0)\}$
	$\{[0], [1]\} \mapsto \{[3]\}$	$U = \{([0], 1, 1), ([1], 1, 1)\}$	$L = \{([3], 1, 0)\}$
	$\{[2]\} \mapsto \{[4]\}$	$U = \{([2], 0, 1)\}$	$L = \{([4], 0, 0)\}$
		$U = \{([0], 1, 1), ([2], 0, 1)\}$	$L = \{([3], 1, 0), ([4], 0, 0)\}$
(-1, -1)	$\{[0]\} \mapsto \{[4]\}$	$U = \{([0], 1, 1)\}$	$L = \{([4], 0, 0)\}$
	$\{[0], [1]\} \mapsto \{[4]\}$	$U = \{([0], 1, 1), ([1], 1, 1)\}$	$L = \{([4], 0, 0)\}$
(1, -1)	$\{[2]\} \mapsto \{[3]\}$	$U = \{([2], 0, 1)\}$	$L = \{([3], 1, 0)\}$

Table 4.7: Exhaustive grouping of the elementary moves to upper and lower goals

4.4.11 Non-Exhaustive Pair Generation and Heuristics

Motivation

The reverse splitting algorithm simply calls the `generatePairs` function to generate a list of split pairs applicable to the current state. In the previous subchapter we discussed a way to exhaustively generate all possible splits to make sure that the algorithm eventually finds the best solution. However, if we have a look at the size of the search space, it becomes clear that the exhaustive search space becomes intractable very quickly, even for small examples. Looking at a random 3×3 filter kernel with an approximation depth of 3, we end up with a search tree with a branching factor of around 300.

Therefore, finding the smart split pairs and presenting them to the algorithm first really is key to getting good performance out of the reverse splitting algorithm.

There are a couple of metrics implemented to guide the algorithm towards good solutions quickly which are outlined here. The idea is to generate pairs in batches, which we then sort based on an ordering metric.

1. Ordering
2. Maximum Pairs
3. Short Distance, Low Scaling
4. Row/Column Splitting

Ordering

We have a set of possible split pairs S for which we want to assign a value $b(U, L)$, $(U, L) \in S$ to each pair. A straight forward choice for this performance that is actually used in the implementation is

$$b(U, L) = \frac{|U|}{|d(U, V)|} \quad (4.19)$$

with $d(U, V)$ the (transformation) distance between the two sets.

The way the reverse split algorithm works is that it essentially allows us to get sub set $U \subseteq FG$ for the price of a transformation of sub goal L . The ratio defined by the function b quantises the benefit of applying this pair by computing the ratio of atoms we can generate from transforming L to the cost we have to pay for doing so.

An ideal pair would get us a large U goal by a very simple transformation from L .

A problem with this ordering is, that for it to be applied, it requires all pairs to be computed first. Because of this, a small batch of pairs thought to be good is generated initially on which the ordering is applied.

Maximum Pairs

If for a specific shift distance and scaling there are multiple pairs available, we only consider the pair with the longest upper and lower sets. The ordering rule 4.19 from the previous block already ensures this, but since the pairs with lower atom count will get sorted to the back of the list, we don't even have to compute them in the first place.

Consider a valid set of elementary moves to be grouped together to form U and L to be those $X \subseteq E$ such that $\exists f \in \mathbb{R}.\forall (s \mapsto t) \in X, \text{scaling}(s, t) = f$ and $\nexists (s \mapsto t) \in E. \text{scaling}(s, t) = f, (s \mapsto t) \notin X$

The reasoning for this is that if there is the opportunity to generate a bigger part of the of a sub goal with a transformation that is to be performed anyways, there is it is rarely sensible not to take it.

Short Distance, Low Scaling

Especially when computing solutions for big filters, the exhaustive set contains pairs of shifts over very large distances. As we know, transformation costs scale with the distance travelled, we prefer to generate pairs with a smaller distance.

This can be done in the pair generation part, by sorting the groups by distance and evaluate the groups with smaller distances first. Furthermore, groups with smaller distances tend to be bigger as the shift can be applied to more positions on the filter. This is beneficial, as larger groups generally yield larger pairs that can be used to generate larger parts of the final goal.

Row/Column Splitting

The order metric 4.19 assigns a value to choose the current split set based on the amount of "cheap" atoms we can get by choosing the pair.

The way the reverse splitting algorithm works is, that it can only generate the set U in a given step. The sets L and R remain and have to be computed in the next step. Therefore, it is also essential to choose the pairs in a way that results in well behaved L and R sets.

Example 12. Figure 4.3 represents a 5×5 box filter, with the black squares being present atoms of the final goal. When we compute all possible splits, and order them

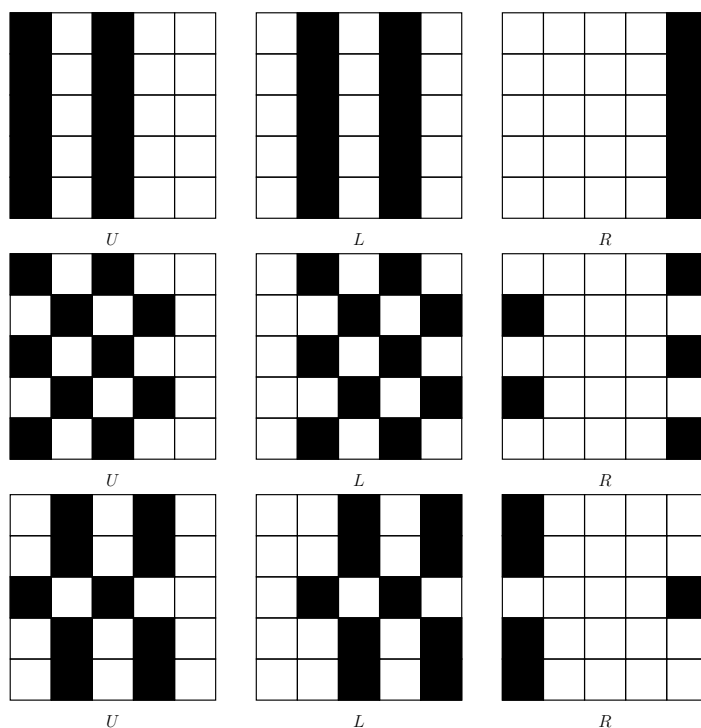


Figure 4.3: Effects of different split pairs with same metric and distance. When splitting a set into the three sub sets U , L and R , it is crucial to select a split that yields remaining sets that are easily computable in the next step. The first example shown here is smart choice for such a split, where as the others probably require more work in further splits. Even though they all have the same metrics, the fact that the first one is preferred is reflected in the ordering in which pairs are generated.

based on the order metric, we end up with multiple splits that have the same metric b . The splits in Figure 4.3 all have the same order metric. All splits have the same distance $(-1, 0)$, no scaling and yield an upper set of size 10. However when looking at the L and R sets, it is evident that the first split pair produces the best L and R sets for further splitting.

In general, goals in which the individual atoms are closer together are favourable, as they are assumed to be easier to split in the next step. Especially favourable are splits with sets that have the atoms nicely aligned in rows or columns. In the elementary moves combination step, we only generate these kinds of split sets in the first place, to explore these likely good candidates first.

4.4.12 Computation Graph Relaxation

Motivation

The reverse splitting algorithm manages to find optimal plans to assemble subsets of the final goal under the assumption that it never generates a subgoal that is not part of the final goal.

There are strong arguments for this assumptions. Consider a final goal FG and a sub goal $A \not\subseteq FG$

1. If $\nexists B \subseteq FG$ with A and B transformable into each other, A will never form a part of the final solution, and we may as well discard it.
2. If $\exists B \subseteq FG$ with A and B transformable into each other, we could just have generated B in the first place, without generating A first. This does not reduce generality, as every set A is transformable to, is also transformable from B .

While argument 2 is true, it assumes that all transformations incur the same cost. If this would be the case, it is true that there is never a benefit in computing A first, as we could just compute B directly and then continue from there. However, in practice, not all shifts do incur the same cost. Shifts from further away on the chip will be more expensive in general.

Example 13. A very simple example is the following one dimensional filter K

$$K = \left[1 \quad \frac{1}{2} \quad 1 \right] \quad (4.20)$$

with the final goal

$$FG = \{([0], 0, 0), ([1], -1, 0), ([2], -1, 0), ([3], 1, 0), ([4], 1, 0)\} \quad (4.21)$$

Graph 4.2 shows both the original and relaxed computation graph for the problem stated above. The original computation graph is directly obtained from the reverse split algorithm, whereas the relaxed graph is the result of an additional relaxation step.

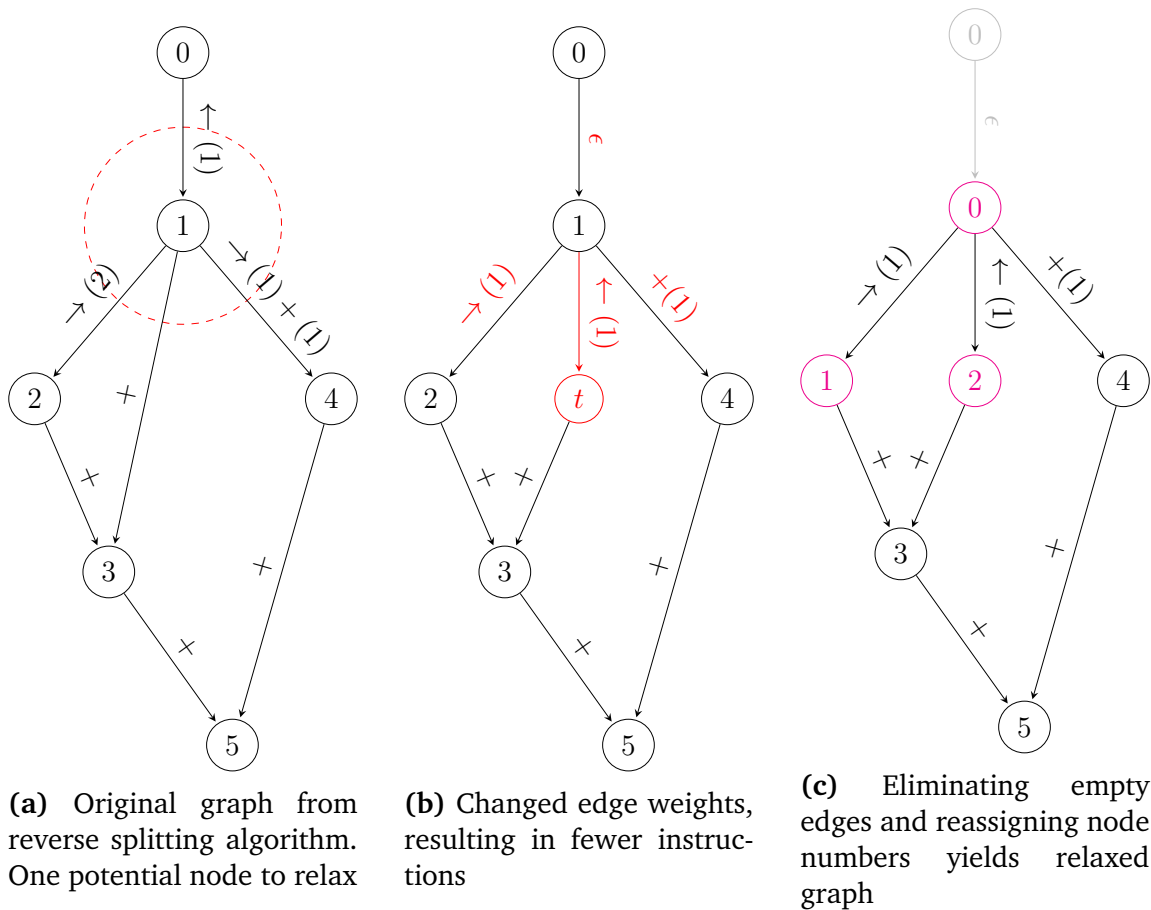
The original graph shows the problem well. The algorithm starts building the left entry (1), then shifts (copies) the left entry two pixels to the right to form the right entry (2). This is already a total shift of 3 pixels, where as we could just have copied the original value to the left, and then to the right. To make matters worse, the atom for the center entry (4) now has to be generated from one of the side entries (1), which adds yet another unnecessary shift.

As we have seen in Example 13, there are cases in which it is beneficial to compute intermediate results that are not directly part of the final goal, but somewhere in the middle of two parts of the final goal. This way, we can save some shift operations.

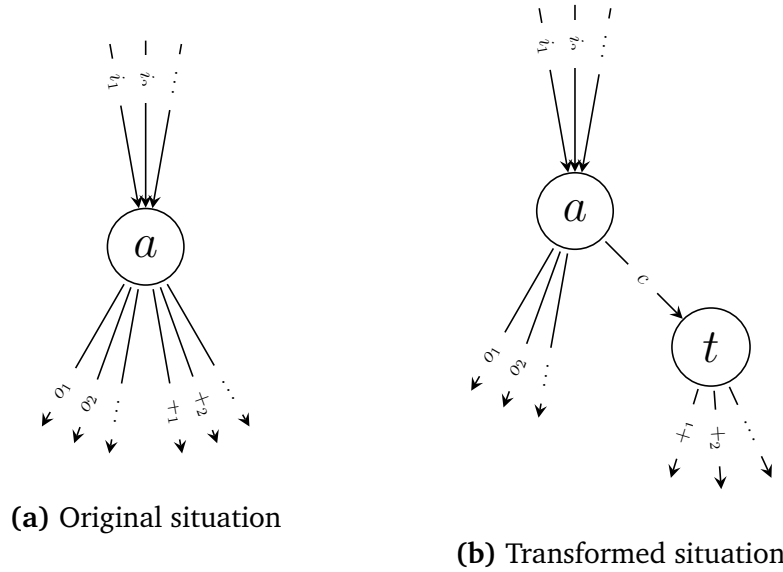
General Approach

Albeit quite simplified, we use an approach similar to retiming (Leiserson et al. (1983)), commonly used in integrated circuit design. The idea is, that we can see the shift operations in the computation graph as three dimensional edge weights consisting of horizontal shift, vertical shift and scaling.

If we draw a circle containing at least one node in the computation graph, but does not include the input or output node, we can change the graph according to the following rule. If we add a constant to the component of a weight to all edges that go into the circle, we have to subtract the same constant from the component of the weight of all edges that point out of the circle. The retiming theorem then says that by doing so, the original meaning of the program (or integrated circuit) remains the same. This gives us flexibility in changing the edge weights.



Graph 4.2: Original and relaxed graph for problem in **Example 13**. Node 1 in the original graph has the potential to be relaxed, as changing the weights on all of its edges reduces the number of shifts performed from 4 to 2. Changing the weights on edges at a node or group of nodes is allowed as we add the same amount to all outgoing edges we add to all incoming edges. Note that when considering the execution order, the relaxed graph requires at most 3 values to be live at the same time, whereas the original version only required two. Therefore, the relaxed version will need one more available hardware register to be performed.



Graph 4.3: Relaxation is only possible on transformation edges but not on *add* edges. Therefore, in order to allow relaxation of a , we introduce a new temporary node t connected with a (relaxable) edge c that connects it to a . All add edges of a are made such that they now originate from t .

Plus Edges

In our case, the computation graphs generated by reverse split algorithm do not only contain shift edges, but also add edges. Specifically, since the reverse split algorithm never produces anything that is not a subset of the final goal, every node except the start and end nodes will have at least one add edge pointing away from them. While we can perform retiming on shift edges, this is not possible on add edges. We have to perform an additional step to deal with these edges.

Let a be the (super)node we would like to relax. a being either a single node or a group of nodes with just inputs and outputs to the node exposed. **Graph 4.3** shows a representation of the general case of a retiming candidate a . We only consider nodes that only have shift edges as their parents, as the other cases are equivalent to the case where we just do not include the node providing the add edge. The edges i_k represent the k -th input shift edge, the edges o_k represent the k -th output shift edge and the edges $+_k$ represent the k -th output add edge.

If we were to retime the shift edges in this case, the output add edges would no longer be valid, as the next node would no longer find their expected result in node a . Therefore, we have to introduce a new node t and a shift edge from a to t in order to correct for the induced relocation of a .

Optimal Relaxation

With a (super)node a that only has shift edges (as ensured by the previous transformation), we can compute an optimal relaxation amount for every weight dimension.

We consider the a similar situation as before, with a node a having n input edges $i_k, k \in \{0\dots n\}$ and m output edges $o_l, l \in \{0\dots m\}$. As the correction edge c is a shift edge, it is considered to be in the output edges o . Let $w(e)$ be the weight of edge e in a particular dimension. As the computation of the optimal relaxation is equivalent and independent for each of the three dimensions (horizontal shift, vertical shift, scale), we omit the notion of the dimension. The total computational cost $C(a)$ of node a in the respective dimension is given as follows

$$C(a) = \sum_{k=0}^n |w(i_k)| + \sum_{l=0}^m |w(o_l)| \quad (4.22)$$

If we relax the weights by an integer $\lambda \in \mathbb{Z}$, the total computational cost of the node in the dimension is given by

$$C(a) = \sum_{k=0}^n |w(i_k) - \lambda| + \sum_{l=0}^m |w(o_l) + \lambda| \quad (4.23)$$

This follows from the fact that we subtract the relaxation from all incoming edges and add the relaxation to all outgoing edges to maintain the functionality

$$C(a) = \sum_{k=0}^n |w(i_k) - \lambda| + \sum_{l=0}^m |w(o_l) + \lambda| \quad (4.24)$$

Our goal is to minimise the computational cost C , therefore we can write this as

$$\min_{\lambda} C(a) = \min_{\lambda} \sum_{k=0}^n |w(i_k) - \lambda| + \sum_{l=0}^m |w(o_l) + \lambda| \quad (4.25)$$

Which is the simplest form of the well known geometric median problem in one dimension for which the solution is known as the median of the weights.

In our case, the value for λ that minimises the cost function is therefore given as

$$\min_{\lambda} C(a) = \text{median}(\{w(i_0), \dots, w(i_n), -w(o_0), \dots, -w(o_m)\}) \quad (4.26)$$

Equation 4.26 gives an efficient way of computing the optimal amount of relaxation to be applied to every (super)node. The optimal relaxation has to be obtained for all three dimensions independently.

Liveness analysis

The reverse splitting algorithm always produces nodes that have at least one add edge pointing away from node. Performing relaxation therefore almost always requires us to introduce the t node to fix the relaxation for the add nodes. Adding the t node generally leads to the need to store one additional parameter at a given time in the program, which increases the need for physical registers. Since we operate in an environment that is mostly restricted by the limited number of physical registers, finding opportunities to relax involves understanding how many values we have to maintain at a given time.

Let be k be a node in the graph. Let $\text{liveness}(k)$ be the set of nodes, including k , whose values are also required to be present at the time of computing k , in order to finish the computation of the graph. We assume k to be the latest computation performed in the graph, therefore all other nodes in $\text{liveness}(k)$ will have a lower value than k . $\forall n \in \text{liveness}(k), n < k$.

The reverse splitting algorithm only writes to each slot once, increasing the numbers of the slots as it passes along. Therefore, computing the liveness of a slot can be easily computed by iterating once over the plan (equivalent to traversing the graph in the order of the node numbers) as shown in algorithm 7.

Algorithm 7 Liveness analysis

```

1: procedure LIVENESSANALYSIS(plan)
2:   minMap  $\leftarrow$  {}
3:   maxMap  $\leftarrow$  {}
4:   for step  $\in$  plan do
5:     minMap[step.target]  $\leftarrow$  step
6:     maxMap[step.source1]  $\leftarrow$  step
7:     maxMap[step.source2]  $\leftarrow$  step
8:   livenessMap  $\leftarrow$  {}
9:   for node  $\in$  minMap.keys() do
10:    for i  $\in$  {minMap[node]...maxMap[node]} do
11:      livenessMap[i]  $\ll$  node

```

Algorithm 7 iterates once over the program, recording the step in which every node gets assigned for the first time. At the same time, it maintains a map of all the nodes storing the instructions in which the node was last referenced. This map gets updates in the same iteration over the program. The algorithm ends up with two maps, one containing the first occurrence of the nodes, the other map containing the last occurrences of the nodes. Since every node number only gets used once, we need to store the value of a node exactly from its first appearance until it is last referenced. We build up a map containing every node number as a key (= step number in the program) and a set of all the nodes that have to be live at the same time.

Note that *minMap* is technically just a map assigning each node number to itself, as all the nodes get first assigned in the step with the same number, (see Listings 4.2 and 4.3). However, in the actual implementation random node ids were used, while keeping the ordering external.

Finding relaxation candidates

After obtaining the liveness for every node, we can figure out which nodes or set of nodes can be relaxed without using more registers than physically available.

A (super)node can be relaxed safely iff

1. There are no add edges pointing out of the node

2. All the add edges are performed after last shift edge. If this happens, the a and t node can map to the same physical register and the need for physical registers does not increase
3. The length of the liveness of all child nodes that get computed from the first add edge to the last add edge is smaller than the number of physical registers. If this is the case, we can allocate a physical register to the t node.

If one of these cases is true for a (super)node, we then analyse if relaxation is beneficial and perform the graph transformation of inserting the t node and changing the weights. Note that we do not allocate registers in this step. We only take the physical registers bound into consideration to decide if the relaxed program is still feasible under the register constraint. Register allocation is performed at a later stage.

Eliminating Empty Shifts

After the relaxation step, it is likely that we end with shift edges of weight zero. Zero-edges are obviously not needed, as they suggest the need to store an intermediate result which is in fact the same as its parent, therefore opening up the possibility to mapping both values to the same physical register.

Eliminating empty shifts is a trivial operation of just reconnecting the empty edges to the parents, and removing the now unconnected nodes.

4.4.13 Register Allocation

In order to perform the computational graph on actual hardware, we have to map the nodes in the graph to physical registers. Since we do not have the possibility of register spilling into memory on the CPA, it is essential that the program only uses as many registers as there are physically available on the chip. The reverse splitting algorithm and the subsequent relaxation step already take this into account, and only produce code that is guaranteed to fit onto the number of physical registers. This is achieved by cutting branches that lead to too many sub results to be stored, but without actually allocating them. The goal of this step is therefore to find a mapping from the set of virtual registers (node numbers), to physical registers.

For every node in the graph, we compute the liveness of the node. The liveness is the set of other nodes, that are required to be available at the time we compute the node, in order to finish the program in the intended order. Since it was already used in the relaxation step, liveness analysis was already covered in Section 4.4.12.

Given the liveness of a program, we know for every node in the graph, which other nodes have to be available at the same time.

From this information, we build a bidirectional dependency graph DG from the computational graph G with the vertices $V(DG)$ being the vertices of G , $V(DG) = V(G)$. The edges $E(DG)$ being the dependencies obtained in the liveness analysis. $E(DG) = \{(a, b) : \forall a \in V(DG), b \in \text{liveness}(a)\}$

If $a, b \in V(DG)$ and $(a, b) \in E(DG)$, then a and b are live at the same time and can not be allocated to the same physical register. If we have n available physical

registers, the goal is therefore to find a colouring of the dependency graph DG using n registers. As in our case the number of physical registers is very small, a simple backtracking algorithm was implemented, which performed fast enough for this purpose.

Example 14. *We consider the following random filter kernel*

$$K = \begin{bmatrix} 0.25 & 0.125 & 0.25 \\ 0.125 & 0.25 & 0.25 \\ 0.25 & 0.25 & 0.25 \end{bmatrix} \quad (4.27)$$

The reverse splitting algorithm produces the computation graph depicted in Graph 4.4 together with the dependency graph. The dependency graph shows the graph colouring in the node shapes, where every distinct node shape represents a physical register.

Note that there are edges in the dependency graph that are not expected at first. For example there is an edge between nodes 1 and 5, even though it is not evident from the computational graph how these nodes are connected. However, the dependency stems from the ordering of operations. Since node 6 is dependent on node 1 and gets computed after node 5 (ordering according to node values), the value of node 1 remains live until the completion of node 6 and is therefore also live when we compute node 5. Another thing to note is that nodes 0 and 15 are absent from the dependency graph. This is due to the fact that these nodes are unconstrained and can be assigned to any arbitrary register without conflicts.

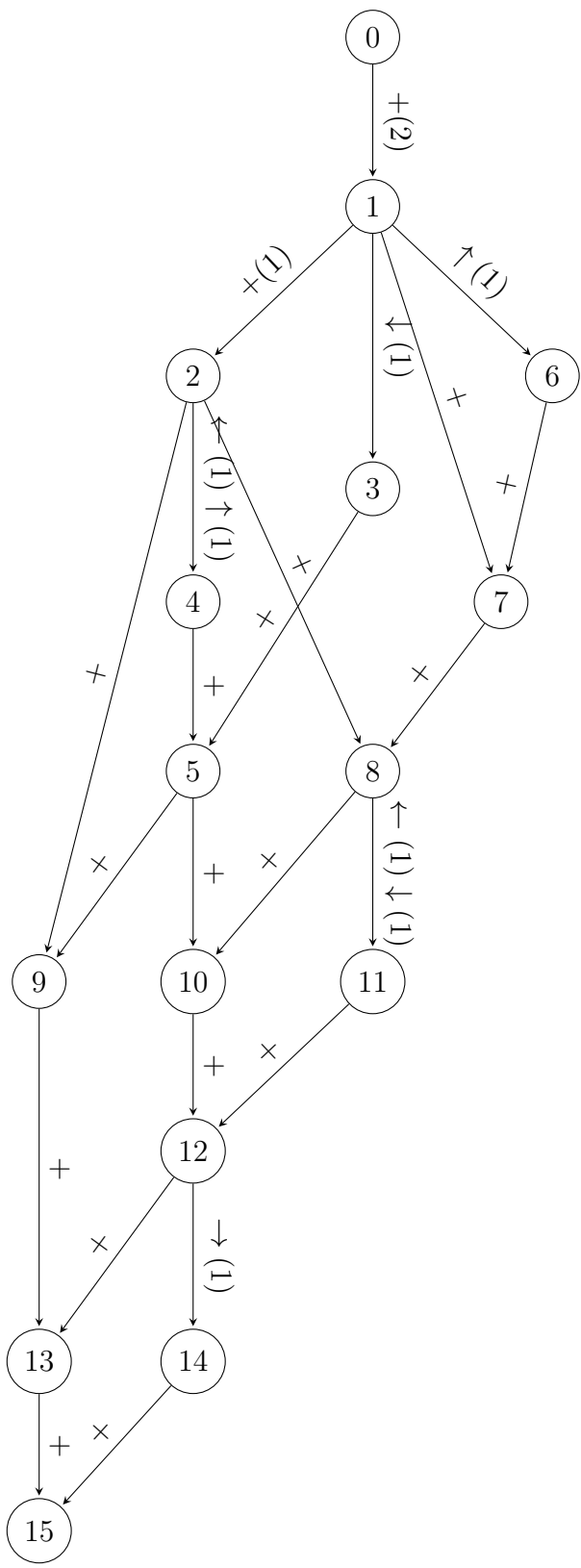
4.5 Performance Evaluation

The algorithm produces meaningful results for reasonably sized filters. This section shows some metrics that show the performance of the search algorithm and its pair generation function

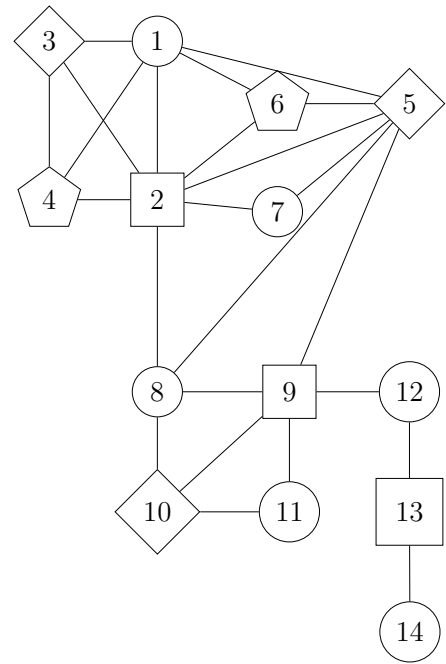
Test System and Methods

The test system runs on an Intel i7-4960HQ processor at 2.60 GHz, together with 16GB 1600 MHz DDR3 RAM. Even though the processor has multiple physical cores, only one of them is used as the software is not designed for multiple processors. The system runs macOS 10.12 Sierra.

All of the algorithms have been implemented in Python 3.6. As Python is an interpreted scripting language, it is expected that one could get boost in performance when implementing the algorithms with a lower level language. When not assuming a particular ordering of the pairs, the order of the pairs has been randomised eliminate side effects from a possible unexpected ordering from the way pairs get generated in the first place. The randomisation of the pair ordering makes the process stochastic, which is why multiple runs were performed with different random orderings to obtain comparable results.



(a) Computational graph G



(b) Dependency graph DG and register allocation. The node numbers are the same as in the computational graph. Every edge between two nodes represents a dependency meaning that both values have to be live at the same time in order to successfully compute G . The node shapes represent the colouring of the graph, with every distinct node shape representing a hardware register. Nodes that do not appear in the dependency graph can be mapped to any hardware register.

Graph 4.4: Computation graph and register allocation for the random kernel in Example 14

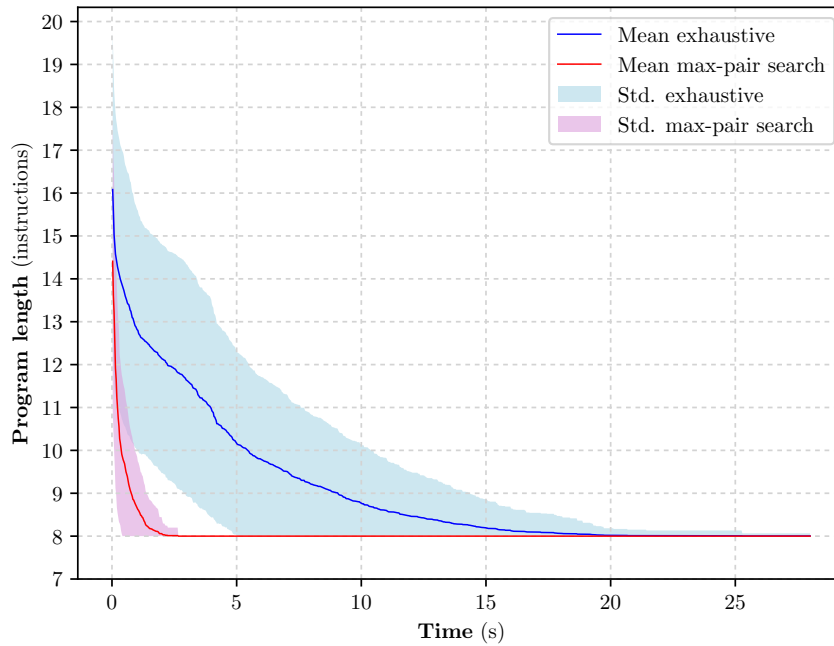


Figure 4.4: Comparison between the maximum pair approach and the full exhaustive approach. Sampled over 256 independent runs.

4.5.1 Heuristics vs. Exhaustive Search

Maximum Pairs vs. Exhaustive Search

We consider a 3×3 Sobel filter kernel

$$K = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \quad (4.28)$$

The best code for this specific kernel, both the algorithm and the author could come up with, is a solution consisting of 7 instructions. A possible, best solution is given by the computation graph 4.5. This graph is the result of an (automatic) relax operation. The original, optimal graph obtained by the reverse split algorithm has a length of 8. After register allocation and translation to a SCAMP program, we would get the program listed in Listing 4.4.

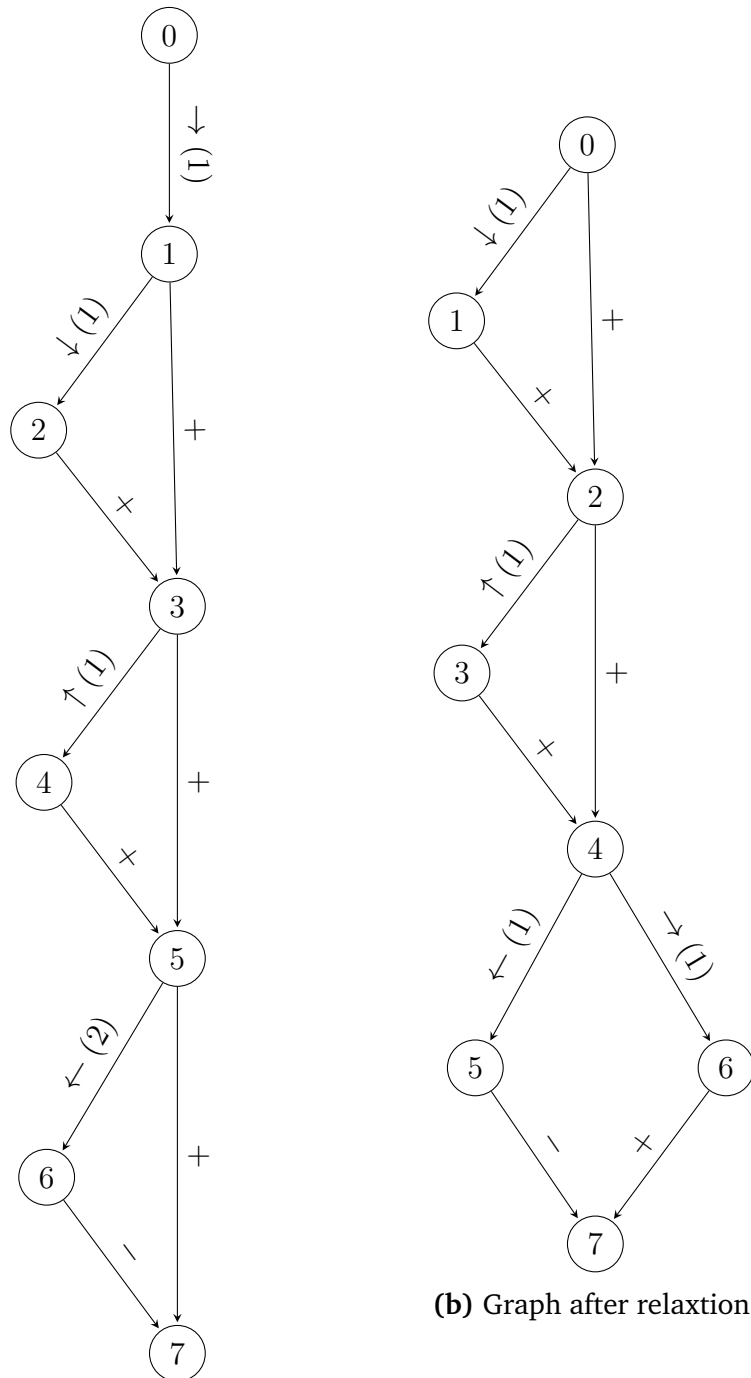
Listing 4.4: Algorithm to perform Sobel filter in x-direction

```

1 B = north(A)
2 A = add(A, B)
3 B = south(A)
4 A = add(A, B)
5 B = east(A)
6 A = west(A)
7 B = sub(A, B)

```

Figure 4.4 shows the mean and the standard deviation of the program lengths obtained at time t , sampled from 256 independent runs. One can see the way the



Graph 4.5: Best computational graph for the Sobel kernel. The solution found by the reverse split algorithm contains 8 operations. This solution then gets relaxed to a solution using the minimum of 7 operations.

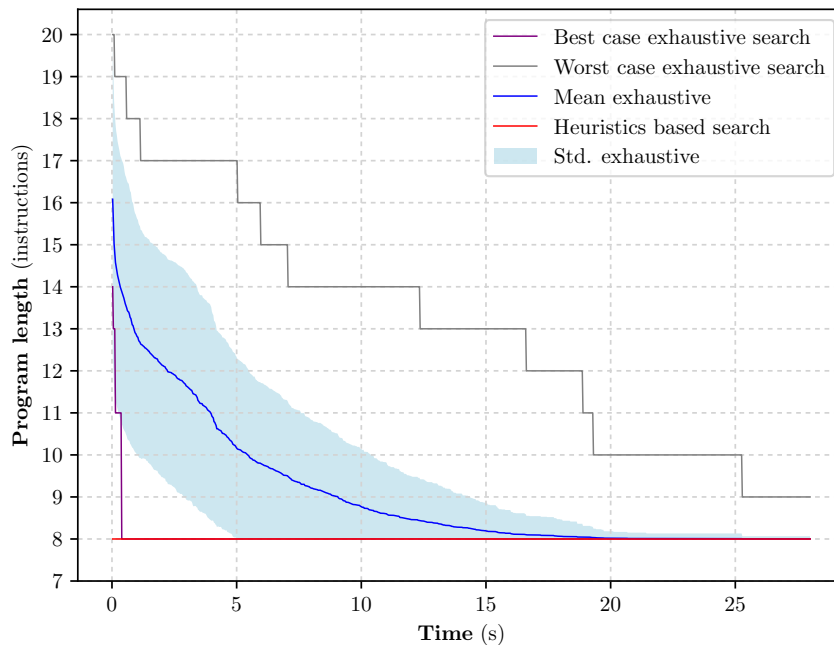


Figure 4.5: Full exhaustive search compared to heuristic search on Sobel 3×3 filter. Sampled over 256 independent runs.

reverse splitting algorithm finds better and better solutions as time goes on, but no solution exceeds the apparent limit of 8 instructions (7 after relaxation). A full, randomised exhaustive search can be performed on all split pairs in approximately 30 seconds. However, in more than half of the runs, the optimal solution is already found after approximately 10 seconds. By only considering maximum split pairs (always take all atom that can be shifted and scaled a particular distance), we can improve the performance of the algorithm a lot. Even an exhaustive search of all possible, maximum pairs finishes in under 5 seconds, with a high probability of finding the optimum after two seconds. We can see that only considering maximum pairs indeed improves performance by a fair bit.

Full Heuristic vs. Exhaustive Search

Figure 4.5 compares the fully exhaustive search to the heuristic search.

The heuristic search has the following heuristics enabled:

1. Only considering maximum pairs
2. Only considering row/column Splits
3. Ordering based on length of upper set and shift distance

Since we order all the split pairs based on length of upper set and shift distance, the order in which we generate the pairs does not matter. Therefore, we do not care about short distances and low scalings.

One can see from the figure, that the heuristic search manages to find the optimum of 8 instructions almost immediately while the exhaustive search takes (in general) a much longer time to reliably find it. The figure also shows best and worst case runs from the sampling of 256 performed exhaustive randomised runs. In the best case, the exhaustive runs finds the optimal solution almost immediately as well. This however is just a matter of chance, whereas the heuristic search yields the immediate result in every run.

4.5.2 Code for Well Known Filters

In this section we show the algorithms performance on some other well known filters in addition to the Sobel filter covered in the previous section.

Gaussian Filter

We assume a 3×3 Gaussian like filter of the form

$$K = \frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \quad (4.29)$$

Figure 4.6 shows both the exhaustive search using the maximum pair assumption as well as with all heuristics enabled. One can see, that the last of the 256 runs only converges to the minimum after 28 minutes and 20 seconds. Even the best case from the exhaustive search converges to the minimum only after 1 minute and 50 seconds. This is still far longer than the heuristic approach, which finds the minimal solution in just 7.4 ms. The convergence of the heuristic approach is shown in Figure 4.7.

The code generated by the algorithm contains 12 instructions and is outlined in Listing 4.5. To compare this with a hand crafted result, the authors solution to the same problem is listed in Listing 4.6.

```

1 A = div2(A)
2 A = div2(A)
3 A = div2(A)
4 A = div2(A)
5 B = north(A)
6 A = add(A, B)
7 B = south(A)
8 A = add(A, B)
9 B = east(A)
10 B = add(A, B)
11 A = west(B)
12 B = add(B, A)

```

Listing 4.5: Autogenerated code for gauss 3×3 filter

```

1 A = div2(A)
2 B = div2(A)
3 C = south(B)
4 A = add(A, C)
5 C = north(B)
6 A = add(A, C)
7 A = div2(A)
8 B = div2(A)
9 C = east(B)
10 A = add(A, C)
11 C = west(B)
12 A = add(A, C)

```

Listing 4.6: Handcrafted code for gaussian 3×3 filter

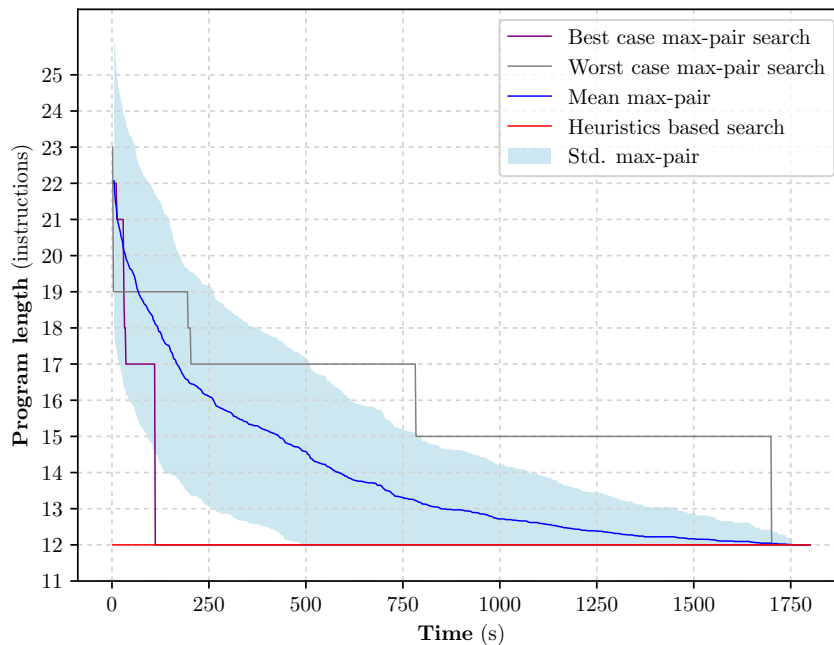


Figure 4.6: Heuristics search and max-pair exhaustive search for the 3×3 gaussian filter $N = 256$

The code generated by the algorithm has the same length as the one generated by the author and uses every type of instruction the exact same number of times. However, the ordering of the instructions and the register allocation makes the code generated by the algorithm use one register less. This trick was unexpected, as the author believed it was impossible to achieve a solution in 12 instructions using only two registers.

A problem associated with the automatic filter code generation is, that it does not take any hardware noise or value range limitations into account. For example, the code generated for the 3×3 gaussian filter does all the divisions right in the beginning, before it assembles the values together to form the final filter. This is unfortunate, as the signal values get very small and the effect of noise might get larger. A better idea would be to spread the divisions further over the program.

For the 3×3 Sobel filter, full exhaustive searching even without the maximum pair assumption was still tractable in reasonable time. The gaussian kernel, which is only moderately more complicated gets prohibitively expensive to compute exhaustively. The heuristic version however still performs great, finding the minimal code almost instantly.

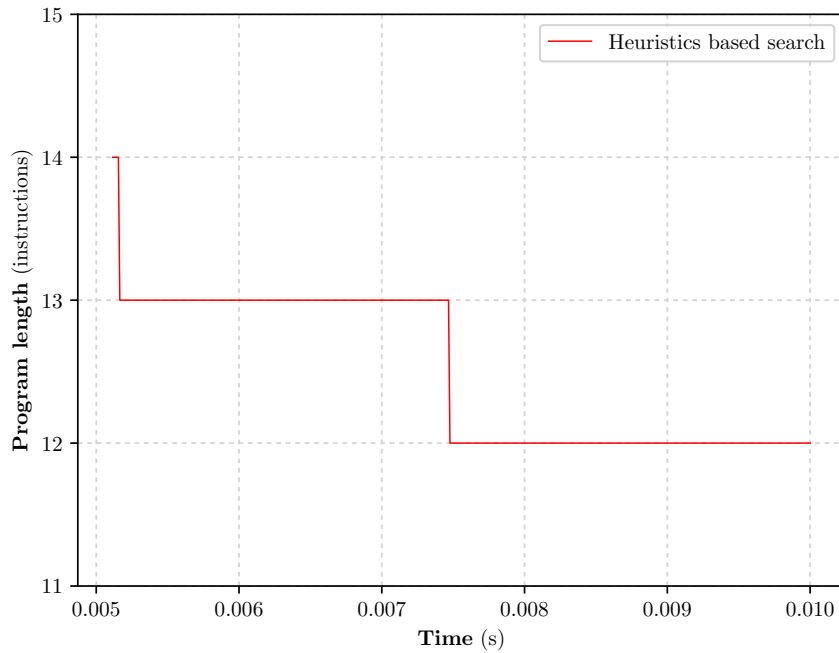


Figure 4.7: Convergence of the heuristics based approach for the 3×3 gaussian filter

Box Filters

We consider common box filters with dimensions $n \times m$ of the form

$$K = \begin{bmatrix} \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \end{bmatrix} \in \mathbb{Z}^{n \times w} \quad (4.30)$$

These filters do not need any value approximation and expose exactly one atom per coordinate in their final goal. This makes them especially well suited for optimisation as there is a large potential of reusing previously computed values.

For example, for the 5×5 box filter, the algorithm produces the code shown in Listing 4.7. As we only have one atom per coordinate, we can represent individual goals graphically. **Graph 4.6** shows the computational graph for the 5×5 box filter, annotated with a graphical representation of the goals.

Listing 4.7: Generated code for a 5×5 box filter

1 A = north(A)	7 A = add(B, A)
2 A = east(A)	8 B = south(B)
3 B = south(A)	9 A = add(A, B)
4 B = south(B)	10 B = west(A)
5 B = add(A, B)	11 B = west(B)
6 A = north(A)	12 B = add(B, A)
	13 A = east(A)

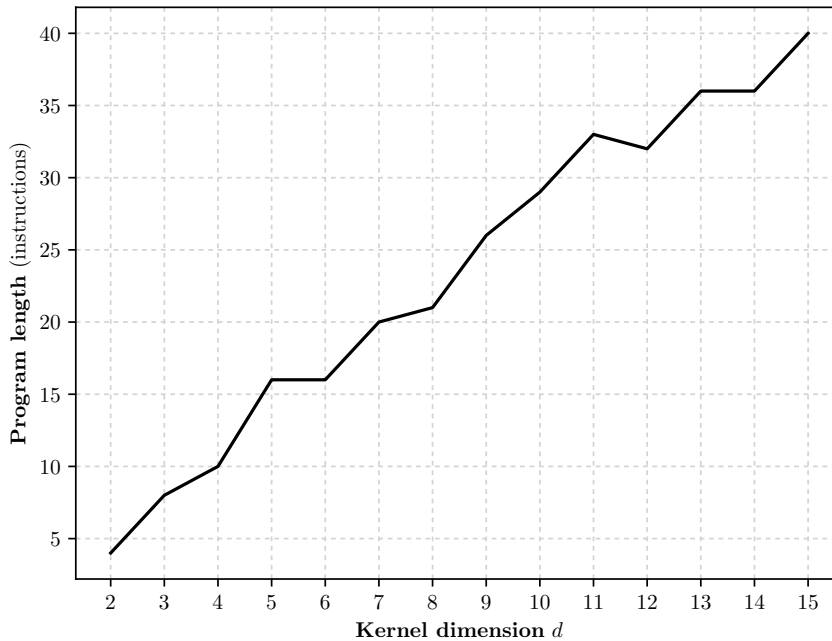


Figure 4.8: Program length for box filters of size $d \times d$. Note that the program length increases linearly.

```

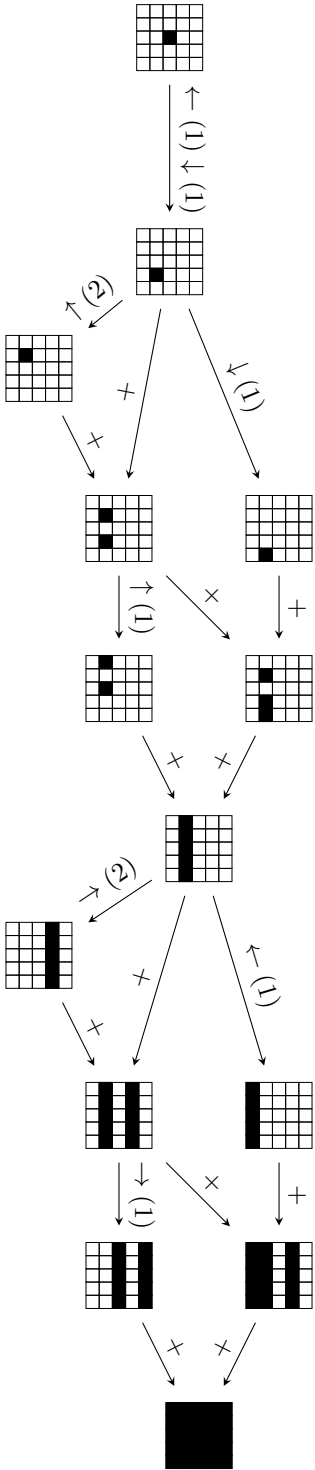
14 A = add(B, A)
15 B = west(B)
16 B = add(A, B)

```

The graph gives an interesting insight in the structure of the found program. The program starts with the initial atom at position $(0, 0)$ and successively doubles the amount of atoms by shifting and accumulating the previous result. Since the width and height of the filter are odd, it also keeps factors to fix the remaining parts that can not be obtained by doubling previous results.

Another interesting property of the CPA is apparent, when comparing program lengths of box filters to the dimensionality of the filter as depicted in Figure 4.8. The graph shows that the program length for box filters of dimensionality d increases linearly with d and not, as one would expect, quadratically. This stems from the fact that the algorithm implicitly separates the filter into two linear sums $K = [1, \dots, 1]^T * [1, \dots, 1]$.

On a conventional computer, we would have to compute the vertical sum for every column to be added by the horizontal sum, requiring us to access every pixel at least once and therefore scaling with d^2 . However, since we perform every instruction on every pixel at the same time on the CPA, the vertical sums all get computed at the same time anyways. Therefore, the horizontal sum can add an arbitrary amount of horizontal sums, as they are all already computed anyways.



Graph 4.6: Graphical representation of the computation graph of a 5 × 5 box filter

4.5.3 Comparisons with CPU and GPU implementations

To compare the execution times of filter kernels on the CPA hardware in comparison with standard CPU and GPU implementations, various test runs have been performed.

Testing Environment

A selection of eight well-known filter kernels was executed on a 256×256 8bit grayscale image. The image resolution and colour was chosen to achieve comparable results to the CPA implementation, as the SCAMP chip features the same resolution. The chosen algorithms for CPU and GPU are the default algorithms shipped with *OpenCV 3.3.0*. The CPU version of *OpenCV* was compiled with both *Threading Building Blocks (TBB)* as well as *Integrated Performance Primitives (IPP)* enabled. The GPU algorithms were compiled for *CUDA V8.0.61* to run on *nVidia CUDA* equipped GPUs. To measure the time, the same algorithm was applied 10000 times to the same image. The reported time is therefore the average runtime of these filter applications. Only computation time was measured. Transfers between harddisk and system memory as well as from system memory to GPU memory were performed outside the timing loop. A number of different *Intel* CPUs from different generations were tested as well as three *nVidia* graphics cards.

The execution times reported for the SCAMP CPA chip are based on the reported 10Mhz instruction rate (Carey (2017)) as well as the length of the programs generated by the algorithm presented in this chapter. The number of cycles for individual instructions is taken from Table 3.4.

The CPU power consumption was read out using the *RAPL* interface present on most newer *Intel* CPUs. The CPU accumulates the used energy in a hardware register which can get read out. While the values provided by this interface are shown to be fairly accurate, it is worth noting that they do not stem from an actual physical measurement. Instead, the values are computed based on a built-in energy model that takes into account various hardware counters (Hackenberg et al. (2013)). Furthermore, the CPU only reports energy consumed instead of current power consumption, as there is no time associated with the data. Power estimates can therefore only be performed accurately over a larger period of time (Hackenberg et al. (2013)). A sample of the accumulated energy was taken before the batch of 10000 filter applications and after.

The GPU power consumption was read out using the *nVidia NVML* API, for which *nVidia* claims 5% accuracy. Unfortunately, this API is only present on more powerful professional devices, which is why the energy measurement could only be carried out on the *TITAN X* graphics card. The SCAMP measurements are estimated based on the reported 1.23W (Carey et al. (2011)) power consumption of the chip under full load (10Mhz). The power and energy consumption was then computed using the estimated runtimes of the generated kernel algorithms.

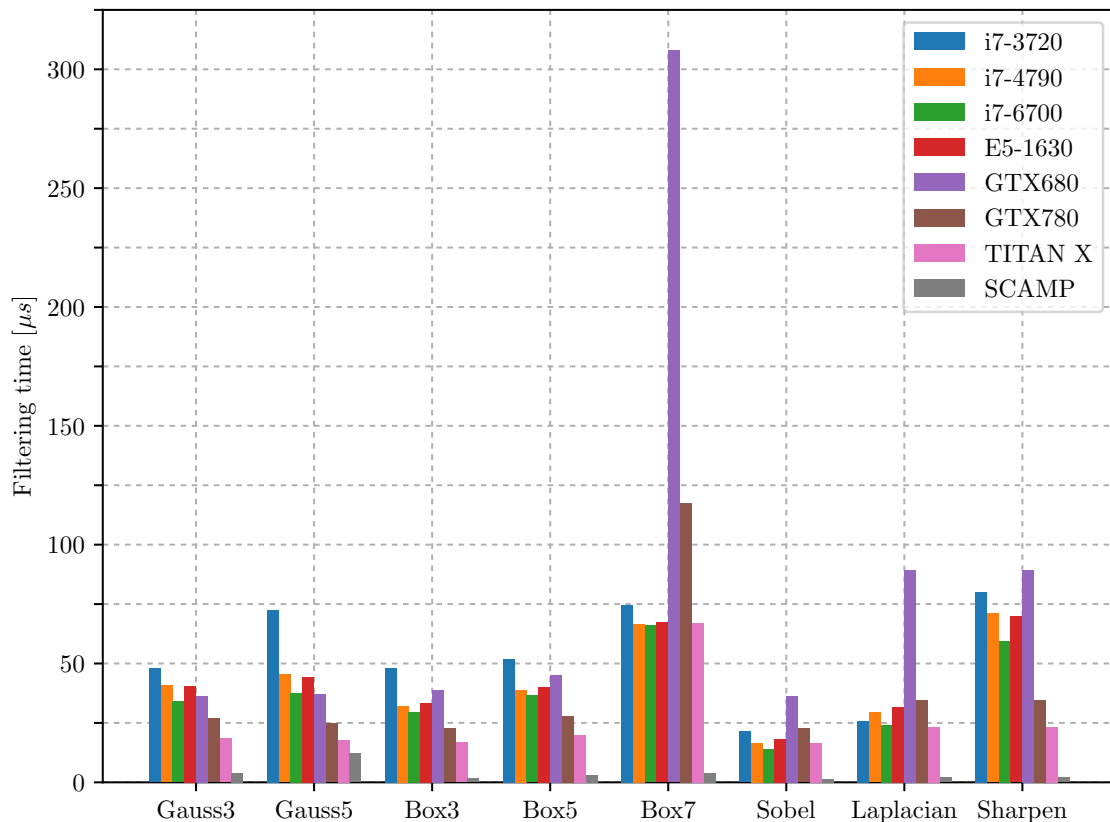


Figure 4.9: Runtime of various well known filter kernels on different types of CPU and GPU hardware (averaged samples from 10'000 runs), as well as the estimated runtime for the generated algorithm for the CPA. For the CPU and GPU values, the algorithms are the default algorithms shipped with *OpenCV*.

Execution times

Figure 4.9 and **Table 4.8** show the execution times of a single application of various filter kernels on different hardware components.

One can see, that the parallel nature of the CPA (SCAMP) allows it to perform all of the tested filter kernels in a fraction of the time needed by the other devices. This is a direct consequence of having a dedicated processing element available for every pixel, building up the filter on the whole image at the same time. As for the other devices, we see that for dense kernels (*gauss*, *box*) GPUs usually perform better than CPUs, where as for sparse kernels (*sobel*, *laplacian*, *sharpen*) CPUs seem to have an advantage. An outlier case being the 7×7 box filter, at which only the most powerful graphics card manages to get a result comparable to the CPUs. It is assumed that the CPU implementation follows a more suitable algorithm than the GPU implementation, even though both implementations are based on their vendors performance libraries (*Intel IPP*, *nVidia NPP*). Another reason could be the fact, that the *GTX680* and *GTX780* are based on a hardware architecture that is less suitable for this type of filter than the *TITAN X*'s architecture.

	i7-3720	i7-4790	i7-6700	E5-1630	GTX680	GTX780	TITAN X	SCAMP
Gauss3	47.9	40.7	34.0	40.5	36.2	26.9	18.4	4.0
Gauss5	72.5	45.4	37.5	44.3	37.1	24.8	17.7	12.1
Box3	48.2	32.0	29.5	33.4	38.8	22.7	16.8	1.6
Box5	51.9	38.9	36.5	40.2	45.3	27.9	19.8	3.2
Box7	74.7	66.5	66.0	67.2	307.9	117.5	67.0	4.0
Sobel	21.4	16.6	14.0	18.1	36.3	22.7	16.5	1.4
Laplacian	25.9	29.4	24.2	31.7	89.3	34.6	23.3	2.1
Sharpen	80.0	71.0	59.5	69.8	89.3	34.6	23.3	2.0

Table 4.8: Runtimes of various well known filter kernels on different hardware configurations. Values corresponding to **Figure 4.9**

<i>Units: [μs]</i>	With TBB/IPP	Without TBB/IPP
Gauss3	40.5	65.4
Gauss5	44.3	76.7
Box3	33.4	74.6
Box5	40.2	80.0
Box7	67.2	74.4
Sobel	18.1	419.9
Laplacian	31.7	65.0
Sharpen	69.8	72.4

Table 4.9: Effect of *Integrated Performance Primitives (IPP)* and *Threading Building Blocks (TBB)* on the performance. Evaluated on *Intel E5-1630*.

Upon inspection it was found that the CPU implementations of the filters heavily profit from the *Intel IPP* library. To investigate the effect of the *IPP* and *TBB* libraries, tests were performed with these libraries disabled. **Table 4.10** shows the comparison between runs with *IPP* and *TBB* compiled into *OpenCV*, and runs without the libraries.

Power Consumption

The average power consumption at performing several well known filter kernels on standard hardware as well as on the SCAMP CPA is reported in **Figure 4.11**. The energy spent on the CPUs was obtained by computing the difference of the energy consumed after the filter application and the energy consumed before the application, as reported by the CPU accumulator. The power value was then computed by dividing this value by the time the CPU spent on computing the filter kernels.

For the *TITAN X* GPU, a continuous measurement of the current power consumption was carried out, as depicted in **Figure 4.10**.

Figure 4.11 and **Table 4.10** show the average continuous power consumption of the devices at computing well known filter kernels. One can see, that most CPUs

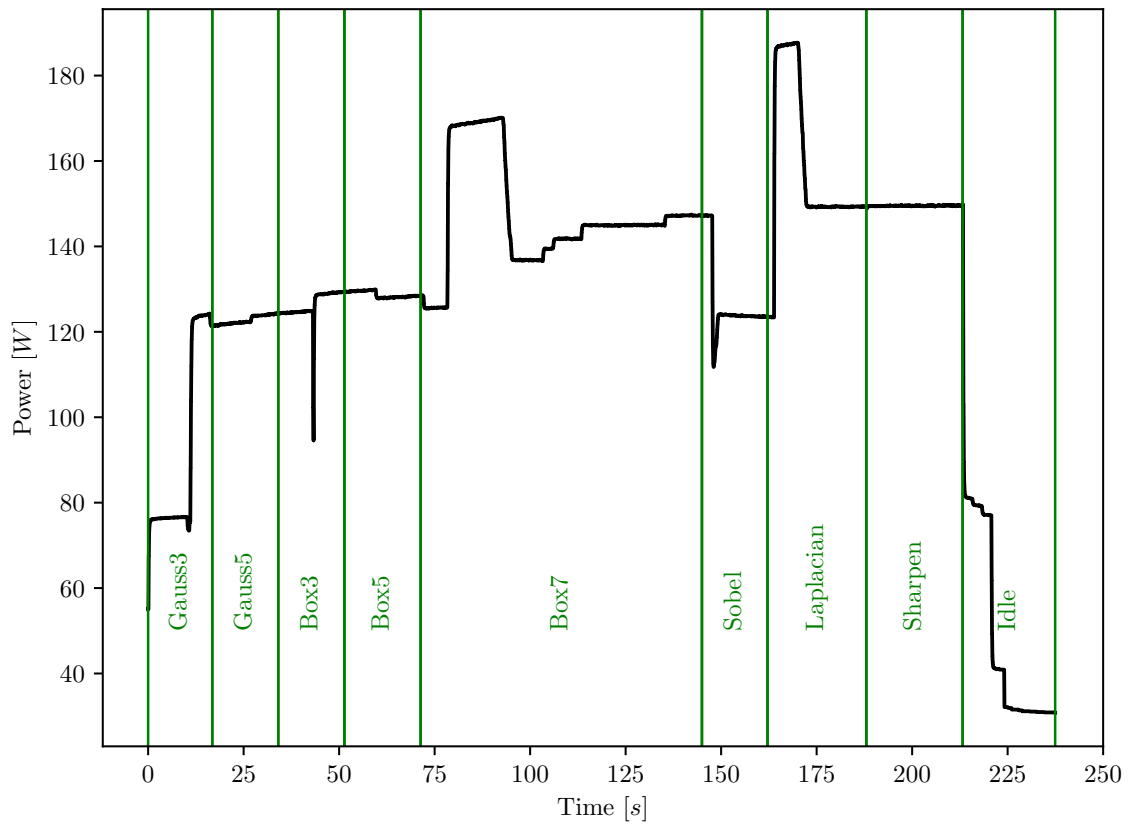


Figure 4.10: Power consumption measured on the *TITAN X* GPU while performing one million applications of various filter kernels. One can see, that the GPU seems to have multiple discrete power levels. These power levels are most probably caused by the fact, that the GPU changes its clock frequency according to the work it is doing.

Units: [W]	i7-3720QM	i7-4790	i7-6700	E5-1630	Titan X	SCAMP
Gauss3	19.128	25.755	22.445	24.836	91.788	1.230
Gauss5	19.573	25.288	22.734	30.835	122.772	1.230
Box3	18.115	28.577	20.256	29.856	126.239	1.230
Box5	18.160	29.778	20.131	33.143	128.744	1.230
Box7	18.092	28.347	19.694	31.163	146.925	1.230
Sobel	19.147	32.693	24.517	34.719	126.857	1.230
Laplacian	19.332	31.820	22.385	33.594	158.172	1.230
Sharpen	17.767	23.887	21.585	32.814	149.515	1.230

Table 4.10: Average power consumption while performing various well known filter kernels on different hardware components. Averaged over 10000 applications of the filter kernels.

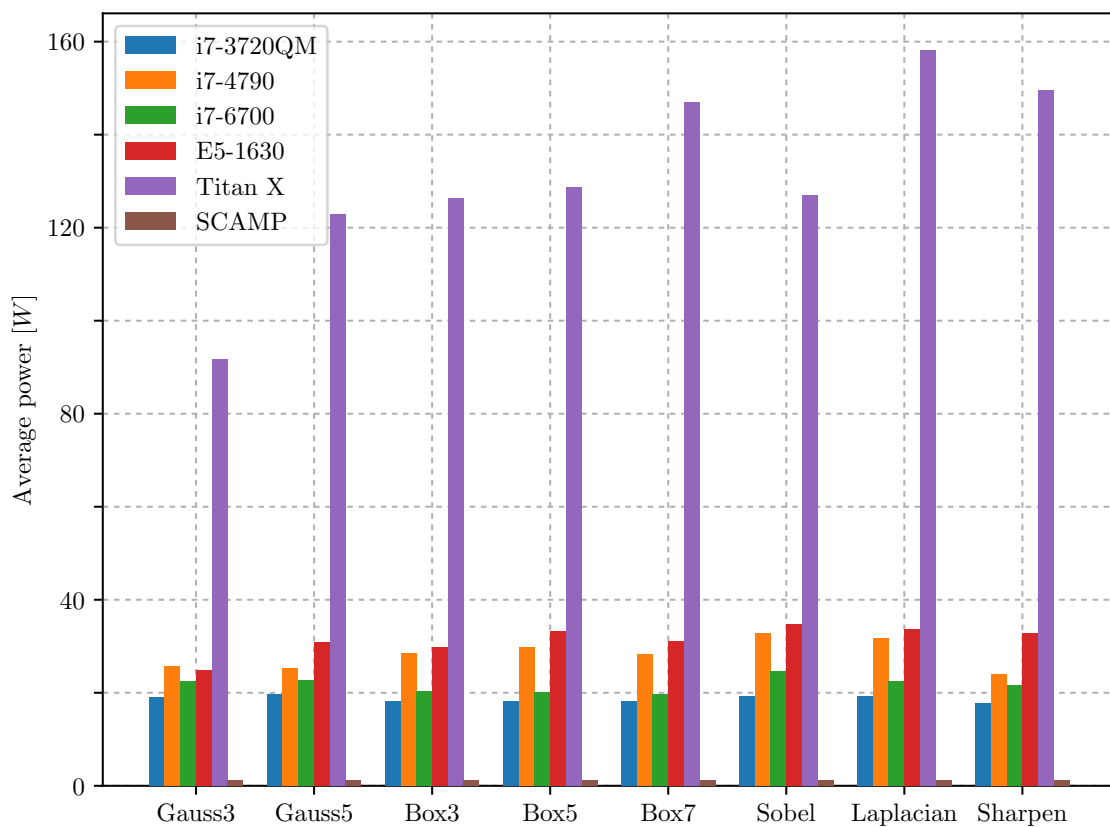


Figure 4.11: Average power consumption while performing various well known filter kernels on different hardware components. Averaged over 10000 applications of the filter kernels.

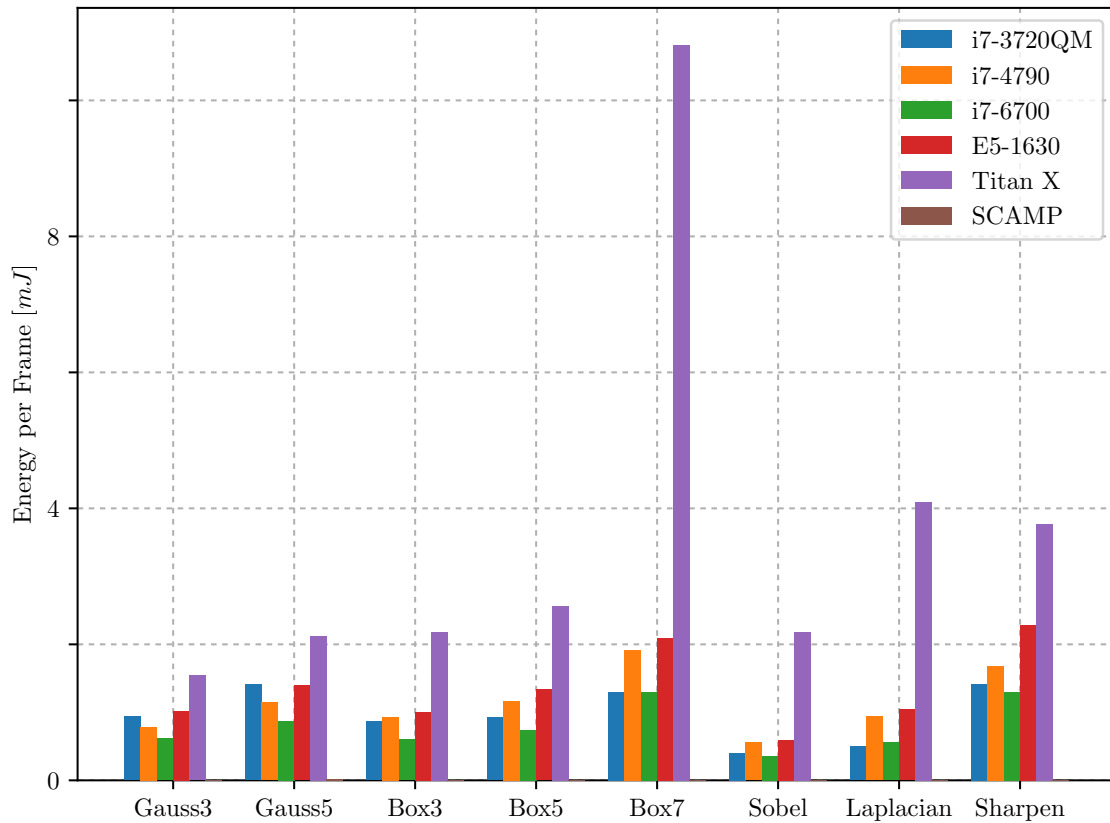


Figure 4.12: Energy spent per filter application for various well known filters on various hardware components.

consume about $20W$ to $30W$, with the *i7-3720QM* using slightly less. This is most probably due to it being a mobile CPU while the others are desktop class CPUs. The *TITAN X* GPU consumes a considerable amount more power than the CPUs, with values ranging up to more than $150W$. The *SCAMP* chip in contrast, consumes only $1.23W$ under full load, which is a lot less than any of the other tested devices. The benefit in energy ranges from a 180 times improvement (*Sobel*, *i7-6700*) all the way to a 2100 times less energy per frame (*Box7*, *TITAN X*).

The continuous power consumption is informative from a systems designer point of view, however, to compare the performance of the devices at computing filter kernels, a different metric was computed. **Figure 4.12** and **Table 4.11** report the energy the devices spend per single filter application. This value was computed by dividing the continuous power consumption by the frame rate.

Still, the *TITAN X* GPU has the highest power consumption for all filters. However, the difference to the CPUs is considerably smaller than it was for the continuous power consumption. This follows naturally from the fact, that the GPU manages to perform more filter applications in the same time. The results look different for the *SCAMP* CPA. Since it has not only the smallest computation time per filter, but also the lowest continuous power consumption, the energy spent per frame is much smaller than for any device.

	i7-3720QM	i7-4790	i7-6700	E5-1630	Titan X	SCAMP
Gauss3	0.941	0.786	0.617	1.021	1.542	0.005
Gauss5	1.411	1.143	0.878	1.397	2.120	0.015
Box3	0.868	0.932	0.602	1.000	2.186	0.002
Box5	0.926	1.167	0.737	1.336	2.568	0.004
Box7	1.297	1.922	1.296	2.094	10.820	0.005
Sobel	0.406	0.556	0.360	0.594	2.178	0.002
Laplacian	0.497	0.939	0.555	1.041	4.092	0.003
Sharpen	1.418	1.677	1.295	2.281	3.768	0.002

Table 4.11: Energy spent per filter application for various well known filters on various hardware components.

Approx. Depth	Mean Abs. Error	Std. Abs. Error
2	0.26398	0.14141
3	0.11558	0.06348
4	0.05251	0.03038
5	0.02932	0.01627
6	0.01399	0.00827
7	0.00576	0.00324
8	0.00354	0.00197
9	0.00149	0.00085

Table 4.12: Absolute pixel intensity value errors for pixels in range $[0 - 1]$ sampled on a set of 1000 random images with a set of 100 random filter kernels. The filter kernels have been approximated to various depths. The blue shaded region shows the standard deviation in absolute pixel intensity error.

4.5.4 Effects of Approximation

At the beginning of Chapter 4, Figure 4.1 showed the theoretical maximal approximation accuracy for different approximation depth. While this gives an idea about how good a coefficient can be approximated, it is unclear what effect this approximation will have on the final image.

Testing Methods

A set of 100 random 3×3 filter kernels was generated. Each filter coefficient is drawn independent and identically distributed from a constant distribution between $[0, 1]$. All coefficients in the filters were approximated with the algorithm presented in Algorithm 4. A test set of 1000 random images from the *Caltech101* (Fei-Fei et al. (2007)) was chosen. Each filter was applied to the images, once in original (perfect) configuration and once approximated. The reported errors are then the differences in pixel intensities after the application of the filters.

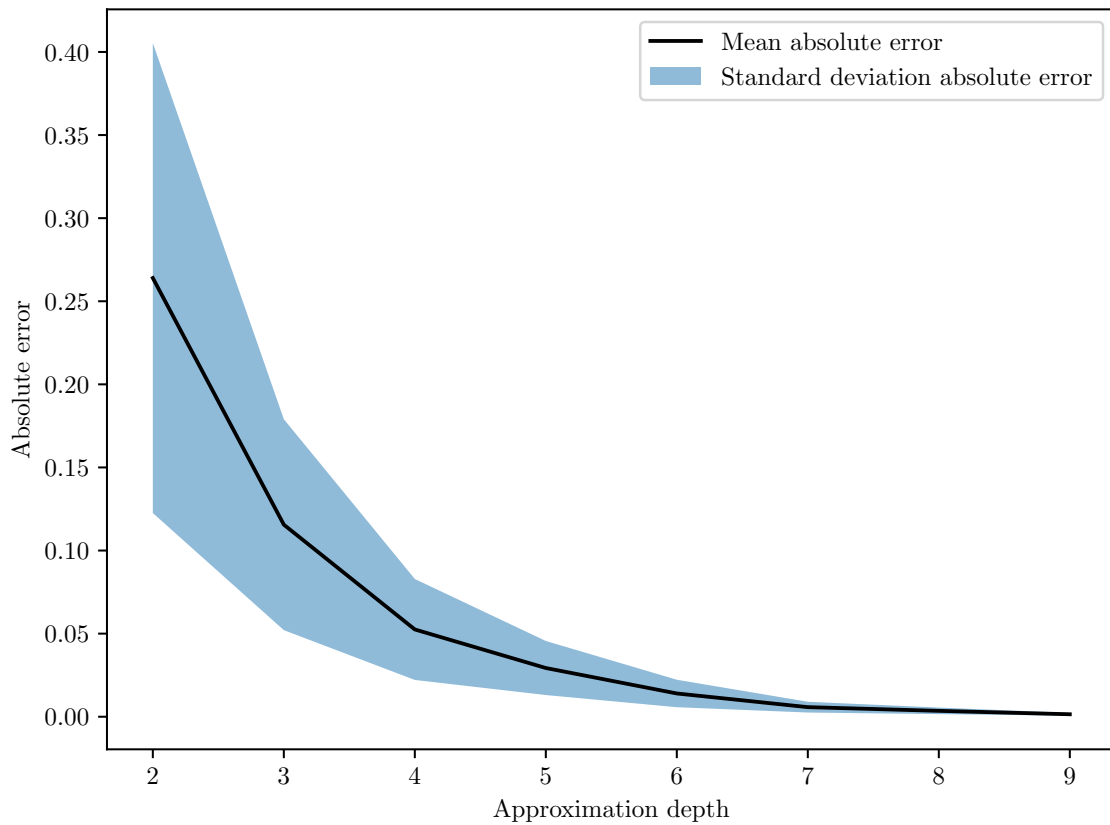


Figure 4.13: Absolute pixel intensity value errors for pixels in range $[0 - 1]$ sampled on a set of 1000 random images with a set of 100 random filter kernels. The filter kernels have been approximated to various depths.

Results

Figure 4.13 and Table 4.12 shows the real world absolute pixel intensity errors that result from approximating the filter kernels to various depths using Algorithm 4. One can see, that every approximation step yields around half the error rate of the previous approximation. This is expected, since we are approximating at powers of two. Another thing to note is, that when approximating to a depth of 8, and assuming an 8-bit input image, the filter kernel is approximated to the same depth as the input image.

4.6 Face Detection

This section describes the implementation and evaluation of a simple but powerful face detector similar to the one described in (Viola and Jones (2001)). The theoretical background to this method is discussed in the background section of this thesis. An in-depth discussion of the algorithm is provided by Wang (2014).

4.6.1 Motivation

Face detection is a commonplace problem, both in Computer Vision research as well as in industry. The face detector described by Viola and Jones (2001) is quite simple in design, while being powerful enough to have found widespread industry adoption. The simple design of the detector allows it to run almost in real-time, reaching frame rates from about 2 fps on a slower CPU up to about 100 fps on a GPU (Acasandrei and Barriga (2011)). However, achieving good performance at a low power consumption is still an unsolved problem where CPAs could be applied to.

The algorithm is based on a set of Haar features which we can see as simple box-filter like convolution kernels. As we know from Section 4.5.2, box filters are especially well suited to the CPA architecture, which makes the algorithm an ideal candidate to implement.

4.6.2 Implementation

A crude version of a Viola and Jones (2001) based face detector has been implemented on the APRON simulator. To ease the process and to obtain comparable results, the implemented face detector is based on the pre-trained values of the *FrontalFace* Haar cascade contributed by Lienhart (2013). This is the default face detection cascade that is shipped with *OpenCV*.

Parallelisation

While the CPU implementation of the algorithm detects object by shifting a 24×24 pixels detection frame serially over the image, the CPA implementation can detect objects on every pixel simultaneously. To do this, we assign every processing element to be responsible for the detection of images in its 24×24 pixel neighbourhood.

Since every possible location on the image is covered by a processing element, we can apply the features to the whole image simultaneously. This also eliminates the need for early rejection, as there is no gain in processing speed by disabling some processing elements.

Features as Filter Kernels

As mentioned above, every processing element is responsible for the detection of objects in its 24×24 neighbourhood. In order to do so, the processing element has to compute the features which are essentially thresholded differences of partial sums, of the pixels in its neighbourhood. A straight forward approach would be to just to represent the features as a large 24×24 filter mask. However, looking at the provided features by *OpenCV*, one can see that there are a lot of small features at considerable offset from the center of the detection frame. **Figure 4.14** shows an example of a feature with two sums to be evaluated. Since shifting is a very natural and cheap operation on the CPA, and every pixel performs the same computation simultaneously, it does not matter where exactly inside the detection frame a processing element computes the kernel, as the result of the computation can easily get shifted to the right place. This allows us to relocate the computation of the sums for the feature to a location that is most suitable for the specific processing element. This is, naturally, the position where the center of the feature coincides with the processing elements location. The very sparse 24×24 pixel kernel of the straight forward approach therefore gets replaced by a smaller, dense kernel of the same size as the area covered by the features sums.

Reusing Filter Kernels

Looking at the individual features present in a stage of the algorithm, one can see that often features do only differ in the position of their summing windows inside the 24×24 detection frame. Since we compute the kernels on every position on the image simultaneously, we can use this fact to reduce the amount of sums (filter kernels) we have to perform in the first place. As discussed above, it is not important to where exactly on the image we perform the pixel summation, as long as it is done at every pixel at the same time, we can easily shift the results to the right place. In the case of the pre-trained face detector, this technique allows to reduce the number of individual features one has to compute from 2913 to 1656, which is a reduction by 43%.

Complete Algorithm

The complete algorithm consists out of 1656 filter kernels, distributed over 25 stages. Every filter application gets followed by a series of thresholds, representing the thresholds of the features the filter kernel implements. Different constant values are then accumulated by the processing elements, depending if they passed the feature threshold.

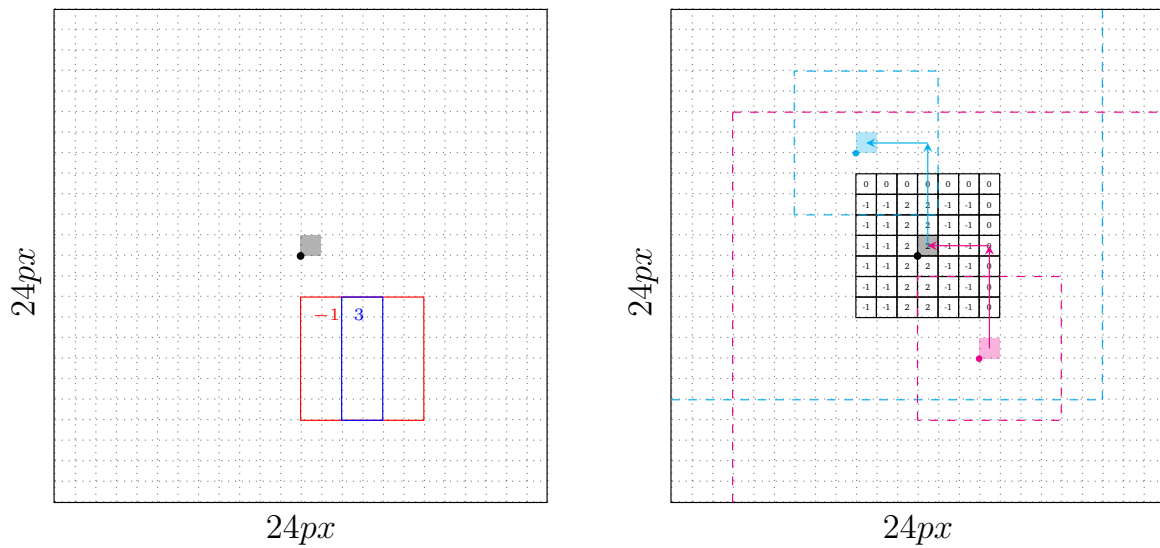


Figure 4.14: An example of a feature consisting of two sums. The large 24×24 square is the detection frame of the processing element at coordinates (12, 12) in the middle, shaded in gray. This processing element has to obtain the values of the weighted sums of the pixels in the red and blue rectangles. To facilitate the problem, the algorithm translates the feature to a filter kernel, aligned at the center of the detection frame. After computing the kernel and thresholding, the (binary) result gets shifted to the top-left to be evaluated by the processing element of which the features center coincides with the center of our detection frame. At the same time, we get the result from the processing element on the bottom-right.

Stage	Total kernels	Distinct kernels	Reduction
1	9	9	0.00 %
2	16	15	6.25 %
3	27	24	11.11 %
4	32	26	18.75 %
5	52	40	23.08 %
6	53	40	24.53 %
7	62	46	25.81 %
8	72	50	30.56 %
9	83	52	37.35 %
10	91	61	32.97 %
11	99	58	41.41 %
12	115	65	43.48 %
13	127	73	42.52 %
14	135	77	42.96 %
15	136	80	41.18 %
16	137	76	44.53 %
17	159	83	47.80 %
18	155	87	43.87 %
19	169	90	46.75 %
20	196	109	44.39 %
21	197	98	50.25 %
22	181	95	47.51 %
23	199	104	47.74 %
24	211	104	50.71 %
25	200	94	53.00 %

Table 4.13: Features that only differ in the position, but otherwise have similar summing areas only have to be computed once on the CPA, as the result can get easily shifted to the right place. This table shows the number of distinct kernels that remain to compute after this technique has been applied

After each stage, a stage threshold decides which pixels pass get to the next stage. **Algorithm 8** shows a pseudocode implementation of the detection algorithm described in this chapter. Note that every instruction gets executed by every processing element at the same time. Every processing element computes the kernels and thresholds locally, on their own local data, even though the underlying feature requires the sum to be calculated at an offset of the processing elements positions. This is then corrected by shifting the boolean result of the thresholding operation to the correct location. Only after that, the leaf value of the specific features get added up. Every processing element keeps an accumulator for all the features of a stage. After all features of a stage have been applied, the value present in the accumulator gets evaluated against the stage threshold in order to decide if the processing elements location remains a candidate for a positive detection in the next stage.

Algorithm 8 Pseudocode for the face detection algorithm as executed on the CPA. Note that every instruction gets executed by every processing element at the same time. The *apply* function applies the convolutional kernel belonging to a group of features to the image, the code is generated by the algorithm presented in Chapter 4. *shiftDecisionToRightPlace()* shifts the boolean value, denoting if a feature passes the threshold at the current pixel, to the correct position in the 24×24 detection frame.

```

1: procedure FACEDetect
2:   for stage  $\in$  stages do
3:     success  $\leftarrow$  true
4:      $\alpha \leftarrow 0$ 
5:     for kernel  $\in$  kernels(stage) do
6:       evidence = apply(kernel)
7:       for feature  $\in$  features(kernel) do
8:         decision  $\leftarrow$  evidence > threshold(feature)
9:         shiftDecisionToRightPlace(decision)
10:      if decision then
11:         $\alpha \leftarrow \alpha + \text{positiveLeaf}(feature)$ 
12:      else
13:         $\alpha \leftarrow \alpha + \text{negativeLeaf}(feature)$ 
14:      success  $\leftarrow$  success && ( $\alpha > \text{threshold(stage)}$ )

```

Datasets

The *OpenCV* model used in this implementation was trained by Lienhart (2013) on an unknown set of 24×24 face images. These are the same image dimensions used by Viola and Jones (2001) in their original paper. To evaluate the performance of the algorithms, the *MIT CBCL Face Database #1* dataset was used in this thesis. The original images of size 19×19 pixel were upscaled to 24×24 pixel using bicubic interpolation. Test frames of size 256×256 pixel were assembled by pasting 100 24×24 pixel images on a 10×10 grid, leaving a border of 8 pixel on each side.

The set used to evaluate the algorithms consists of 2329 examples of faces and 4671 examples of non-faces. The dataset turns out to be quite a challenge, as the stock *OpenCV* implementations only manages to find 180 faces after 25 stages.

4.6.3 Results

Due to code size limitations in the APRON simulator, we were unable to perform a full 25 stage face detector. The largest implementation found to work without technical issues is a 7 stage implementations.

Testing Methods

The 7-stage implementation of the algorithm was applied on the *MIT CBCL Face Database #1*. To achieve a fair comparison, the source code of *OpenCV* was modified in a way, that it only takes the first 7 stages of the full 25 stage face detector into account. Since the face images are of the same size as the trained base window size, (24×24 pixel), detection was only performed on a single (base) scale. *OpenCV* was restricted to use only the base scale. Both algorithms output pixel coordinates of the center point of faces. A coordinate is considered a match, if the coordinate lies within the boundaries of an actual face image. in the grid of test images.

Detection Rates

Figure 4.15 shows the evolution of the detections on a sample testing frame, by the CPA algorithm on the APRON simulator. The white dots represent positions where it is believed that this is the center of a face. One can see, that the algorithm starts out with a very rough estimate, which gets successively refined as the algorithm reaches later stages.

Figure 4.16 shows performance measure for both the *OpenCV* reference implementation as well as for the CPA implementation. Due to technical limitations, only 7 stages could be implemented on the APRON simulator, allowing only a comparison of the first 7 stages. One can see, that the performance measures of the CPA implementation follows the measures of the reference implementation. The algorithm is very biased in classifying something as a positive example of a face in early stages, only later stages effectively reduce false positives. This can be seen in the rather low precision and the high recall rates in early stages.

Processing Time / Energy Consumption

CPU processing times were obtained the same way as described in Section 4.5.3. Only single scale detection was performed on 70 frames, frames of 256×256 pixels in size, each frame containing up to 100 examples. For a more reliable time estimate, the classification of the set was done 10 times, the reported timing and energy figures are averaged over the 10 runs. Unlike in Section 4.5.3, the timing figures for CPU include transfers from disk to memory, however, due to the small image size, the

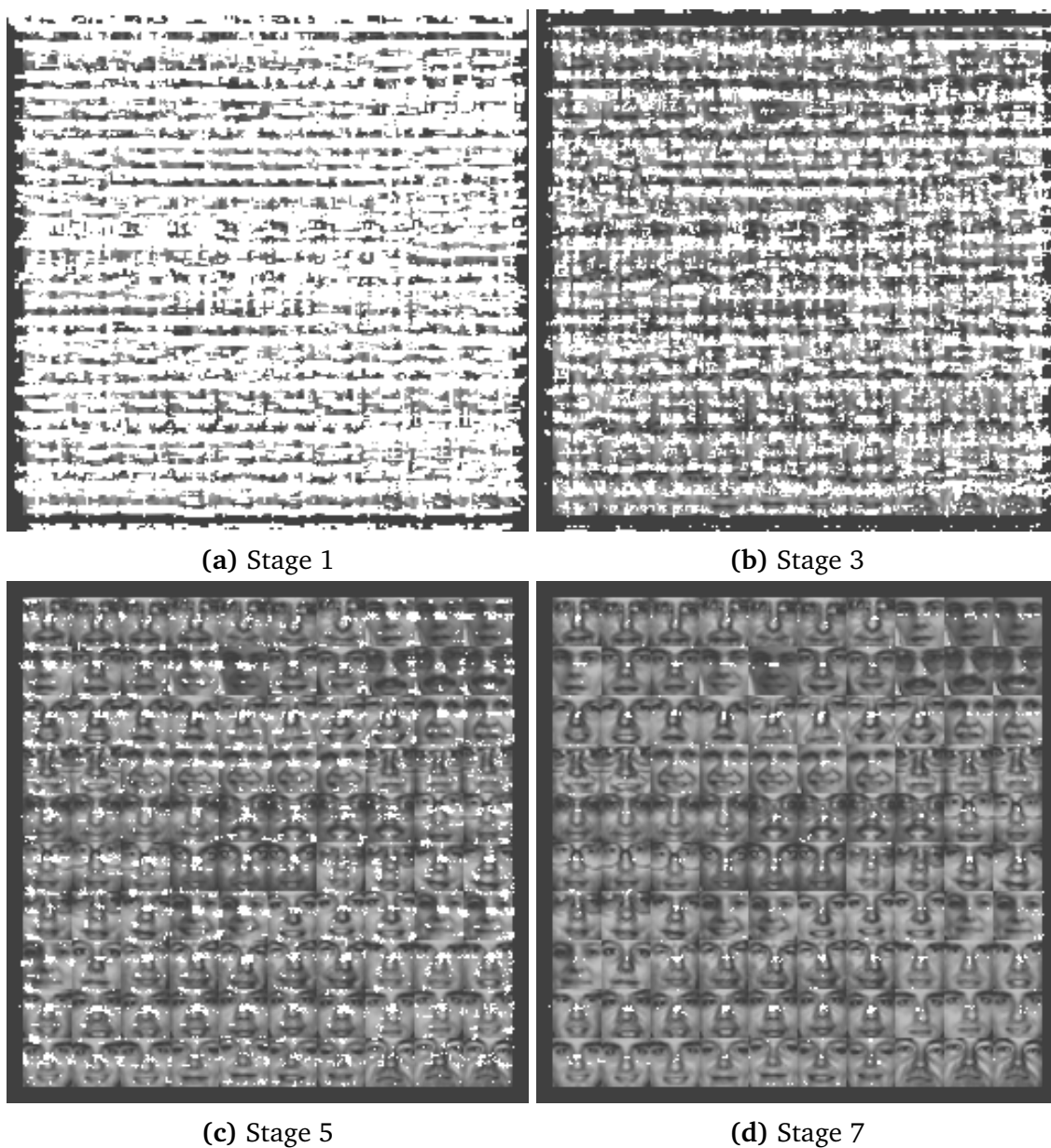


Figure 4.15: Detections of the CPA face detection algorithm on a grid of 10×10 faces from the CBCL dataset. The white dots represent a pixel where the algorithm believed that there is a face. The more stages we apply, the better the estimate becomes.

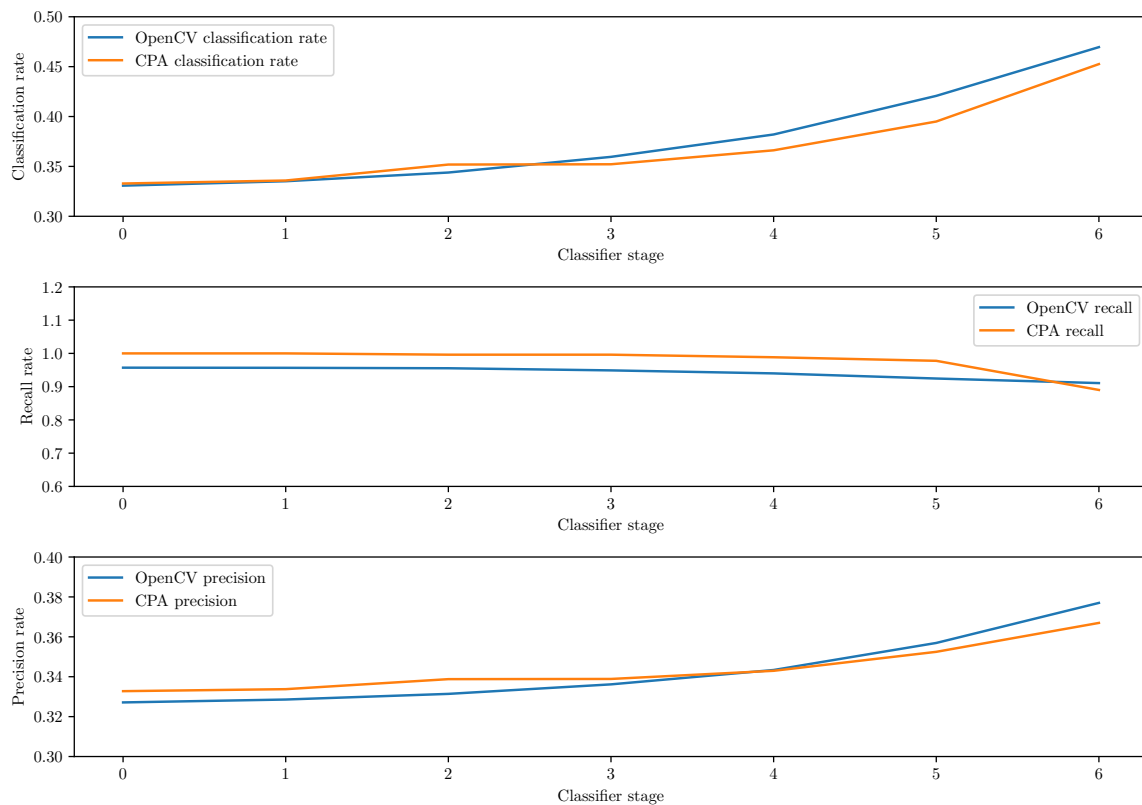


Figure 4.16: Classification rate, recall and precision of the CPA implementation compared to the stock OpenCV implementation on CPU. Note that due to code size limitations, it was only possible to successfully implement the 7 first stages of the algorithm on CPA.

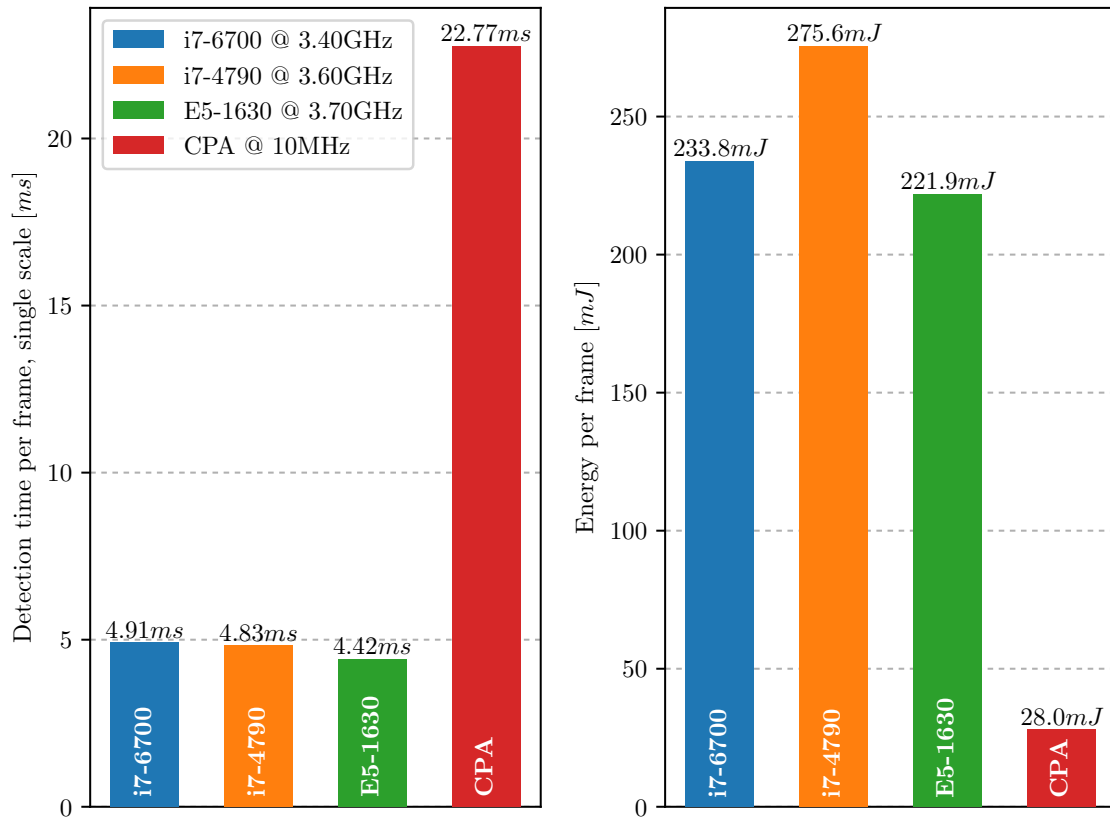


Figure 4.17: Speed and energy consumption of CPUs (measured) and CPA (estimated) at performing the 2full 25 stage *OpenCV* face detector. Note that detection was only performed at a single scale on a 256×256 image, hence the good CPU performance. One can see that although having a longer processing time, the CPA can yield significant energy savings. Note also that these values assume a relatively slow CPA running at 10 *Mhz*.

images are most likely retained in memory by the operating system. All the system used for evaluation feature SATA mounted Solid State Drives (SSD).

The figures for CPA are estimates based on the published figures for the SCAMP chip (Carey et al. (2011)). A clock frequency of 10 *MHz* is assumed with a peak power consumption of 1.23 *W* at peak usage. The clock cycle counts for individual instructions are taken from Table 3.4. Note that the current SCAMP platform is unable to run this algorithm (see Section 4.6.4). Nevertheless, the figures from the SCAMP chip were taken into account to get a sensible estimate of the performance of a CPA device.

Table 4.14 shows the estimated instruction counts, processing times per frame and energy spent per frame for a CPA running at 10 *Mhz* with 1.23 *W* of power consumption, for the different stages of the algorithm. Note that the more stages we perform, the longer the runtime as well as the energy per frame gets. **Figure 4.17** puts these runtimes into relation with the reference implementation for CPU. One can see, that the CPA in our current configuration performs about five times worse

Stage	# Instr.	Runtime [ms]	Energy per Frame [mJ]
1	1011	0.10	0.12
2	2808	0.28	0.35
3	5859	0.59	0.72
4	9094	0.91	1.12
5	13974	1.40	1.72
6	19112	1.91	2.35
7	25077	2.51	3.08
8	31321	3.13	3.85
9	38404	3.84	4.72
10	46356	4.64	5.70
11	54360	5.44	6.69
12	63316	6.33	7.79
13	73068	7.31	8.99
14	83309	8.33	10.25
15	93957	9.40	11.56
16	104545	10.45	12.86
17	116297	11.63	14.30
18	128190	12.82	15.77
19	140690	14.07	17.30
20	156085	15.61	19.20
21	170161	17.02	20.93
22	183887	18.39	22.62
23	198528	19.85	24.42
24	213543	21.35	26.27
25	227703	22.77	28.01

Table 4.14: Estimated runtimes and energy consumptions per frame for a 10MHz cellular processor array, related to the SCAMP class of chips. The estimated number of cycles for the different instructions is taken from Table 3.4. Note that although the SCAMP chip would be capable of running this algorithm in real-time, its analogue architecture prevents us from performing this many operations without a significant degradation of the image even at early stages.

time wise, compared to the CPU implementations. However, this is still a promising result, given the fact that the CPU runs at a clock frequency 300 times higher than the CPA. One can imagine a future CPA device running at a higher clock frequency that could easily outperform current CPU implementations. The CPA energy consumption is about 7.8 times lower than the most energy efficient CPU, where as the CPU does not even include energy spent on DRAM and disks.

4.6.4 Practicality on Current Hardware

Figure 4.17 shows that a CPA with specification such as the SCAMP would be capable of performing the full face detection algorithm at competitive speeds. However, due to fact that the SCAMP chip is implemented as an analogue device, every operation performed introduces an irreversible error on the data. According to Carey (2017), the number of operations one can reliably perform is in the order of magnitude between $10^2 - 10^3$, depending on the quality requirements of the application. With the full face detection algorithm requiring two orders of magnitude more operations to complete, it is currently not possible to perform the algorithm on current hardware.

Chapter 5

Conclusion

This thesis contains contributions to the areas of Visual Odometry, automatic code generation and face detection focussed on Cellular Processor Arrays (CPA). As of now, there are not many publications about the application of Computer Vision algorithms to Cellular Processor Arrays. For higher level algorithms like Object Detection or Convolutional Neural Networks there are no known publications as of now. This thesis represents a first approach on mapping these higher level applications to CPAs. Starting with arbitrary convolutional filters, it was possible to show that an efficient face detection algorithm is feasible on CPA.

5.1 Contributions

This thesis contains contributions to pose estimation as well as automatic code generation. This section summarises the various contributions that have been made.

5.1.1 Pose Estimation

Chapter 3 introduced multiple novel ways to perform pose estimation tasks on Cellular Processor Arrays. The novel algorithms introduced in Section 3.4 form a working 2DoF Visual Odometry approach for CPA. The simple structure, high achievable frame rate and low power consumption achievable with the algorithms allow for promising future applications of these algorithms. It was shown that the performance of the algorithms is highly dependent on the availability of enough variance in the images the camera captures. The algorithm presented in Section 3.5 extends the 2DoF approach with roll rotation and translation into view direction. This is the first algorithm to use a tiling method for pose estimation. It was shown, that the global motion of the camera can be detected using only the vectors measured in the individual tiles by means of a model fitting approach. An Ordinary Least Squares and a RANSAC based approach for the model fitting was presented. The evaluation of the algorithm showed good tracking performance on a synthetic dataset. Four degrees of freedom are ideal for applications where the agent is mechanically restricted to move into one directions, such as wheeled robots and cars.

5.1.2 Automatic Code Generation

Code Generation

Since a lot of Computer Vision problems can be expressed by means of convolutional kernels, having the possibility to automatically generate code for convolutional kernels is essential. The implementation of convolutional kernels on Cellular Processor Arrays is non-trivial, especially for devices with very limited hardware capabilities such as the SCAMP chip. This thesis presents a formalism to write convolutional kernels as sets of approximation factors. Furthermore, a formalism is presented to encode the chips hardware functionalities into operations on sets. An algorithm, called the *reverse splitting* algorithm, which uses this formalism to find a minimal plan for building up the convolutional kernel. The algorithm makes use of heuristics, which in every step provide it with likely good choices for creating a good program.

Subsequent software is presented that can perform local optimisations for cases the *reverse splitting* is known to be unable to find the optimal solution. This is done by retiming edge values in the computation graph by a method known as *retiming* commonly applied in integrated circuit design. Known algorithms such as graph colouring for register allocation have been used in subsequent steps to create a real, runnable CPA program from the intermediate representations as well as validating the result against the input.

The system has proved to be very reliable in generating code for arbitrary filter kernels. Especially the heuristics has proven to be essential in speeding up the algorithm to acceptable levels. It has been shown, that the code generated by the algorithm is in most cases equivalent, or outperforms code written by human experts for the same convolutional filter.

Code has been generated for a set of commonly used convolutional filters. It was shown that running the generated code on a SCAMP CPA yields significant performance and energy gains compared to applying the same filters on standard hardware.

Face Detection

It was shown, by experiments with the automatic code generation code, that box filters are especially well-suited to be performed on CPA devices. This is even up to a point where straight forward linear-time area summations might be more efficient than the constant-time *integral image* approach introduced by Viola and Jones (2001). To show this, a simple face detector based on the works by Viola and Jones (2001) was described. It was shown that, by using the CPAs parallelism, early rejection, a performance increase feature of the original algorithm, is no longer necessary on CPA. Furthermore, it was shown that formerly location dependent sums become independent of actual location when summing on the CPA, allowing the possibility of reusing results from previous filter applications. This method cut the amount of filter kernels that the chip has to perform by 43%. It has been shown that a complex object detection algorithm can work on a CPA, with most likely increased frame rates and better energy consumption than CPU/GPU implementations on fu-

ture, faster CPA devices. Unfortunately, due to technical code size limitations, we were unable to evaluate the full algorithm on current simulation tools. Also, there is no hardware device currently in existence that could run the generated algorithm as current analogue devices suffer from noise.

5.2 Future Work

This thesis marks a first step towards high level Computer Vision algorithms on Cellular Processor Arrays. Starting with arbitrary kernel code generation, a number of possibilities for the implementation of well-known algorithms open up. Furthermore, it shows the limitations of current CPA devices and provides guidance on how to better target CPAs at higher level Computer Vision algorithms.

5.2.1 Pose Estimation

The 2DoF approaches presented in Section 3.4 work surprisingly well, but are limited by only capturing motion in two directions. A system containing multiple CPA cameras pointing in different directions would allow to extend the system with more degrees of freedom as well as redundancy. This system could use the presented algorithm running on each CPA device. A similar approach could be followed with the 4DoF algorithm. With multiple cameras in a system, one would get 16 measurement vectors from each CPA device for which a RANSAC based approach could estimate the global agent motion. All these works would be performed in an attempt to get accurate tracking at high frame rates on energy limited devices, such as small aerial vehicles or virtual-reality headsets. The massive frame rate provided by the CPA allows for simpler tracking algorithms using larger approximations. This opens up an interesting design space involving the trade off between frame rate and complexity. As of now, it is unclear where the optimal solution lies in this design space.

5.2.2 Automatic Code Generation

Convolutional Neural Networks such as *ImageNet* (Krizhevsky et al. (2012)) contain large amounts of convolutional filters in their first layers. Arbitrary kernel code generation as presented in this thesis would allow the computation of these layers completely pixel parallel in hardware, significantly speeding up recognition performance. Arbitrary filter code generation is a basic building block for many more sophisticated high-level Computer Vision applications such as edge detection, image segmentation or object recognition. The complete 25-stage Viola and Jones (2001) based algorithm shown in this thesis was unable to run on the APRON simulator due to technical limitations. However, we managed to run a 7-stage implementation. CPA prototyping and simulation tools better suited for automatically generated code bases would allow researchers to explore the CPA design space more efficiently.

While this thesis introduces automatic generation of code from the abstraction level of filter kernels, one can imagine a compiler that can directly compile efficient

CPA code from a high-level language such as C. Also imaginable is a novel programming language better aimed at the pixel-parallel execution capabilities of CPA hardware.

5.2.3 Hardware

It has been shown in this thesis, that the raw performance figures of the SCAMP chip compare very favourable with CPU and GPU for convolutions and even for object detection. However, since the chip is implemented as an analogue circuit, image degradation is a major problem when applying many operations on the images. As an example, **Figure 3.16** shows the effect of a simple rotation operation on the chip. While this rotation can still be executed at frame rates magnitudes higher than real-time, severe image degradation is visible. This puts the current SCAMP chip in a very special position of the CPA design space. While it would be possible to perform large amounts of operations, enabling high level image processing algorithms, the designer is restricted to very short programs because of signal degradation. Due to this limited possible program length, using a CPA is only beneficial in very high frame rate applications or very low power applications with the possibility to put the chip to sleep most of the time. A digital CPA would provide the advantage of perfect signal representation. This would allow more complicated code to be compiled for the chip. A digital CPA could also lead to further advantages like the possibility to fabricate it in a much smaller fabrication process, further reducing energy consumption and allowing for more hardware to be available per pixel. 3D-stacking of image sensor and the processing unit would allow significantly more hardware to be placed per pixel. It is even imaginable to stack multiple layers of processing units on top of each other, to represent a fully programmable neural-network-like structure. This system would allow the programmer to shift a signal into 3-dimensions, which could make applications such dynamic network training using gradient descend possible on the CPA.

Bibliography

- CBCL Face Database #1. <http://www.ai.mit.edu/projects/cbcl>. pages
- Acasandrei, L. and Barriga, A. (2011). Accelerating Viola-Jones face detection for embedded and SoC environments. In *Distributed Smart Cameras (ICDSC), 2011 Fifth ACM/IEEE International Conference on*, pages 1–6. IEEE. pages 90
- Barr, D. R., Carey, S. J., Lopich, A., and Dudek, P. (2006). A control system for a cellular processor array. In *Cellular Neural Networks and Their Applications, 2006. CNNA'06. 10th International Workshop on*, pages 1–6. IEEE. pages 6, 7, 8, 9
- Barr, D. R. and Dudek, P. (2008). A cellular processor array simulation and hardware prototyping tool. In *Cellular Neural Networks and Their Applications, 2008. CNNA 2008. 11th International Workshop on*, pages 213–218. IEEE. pages 9, 21, 32
- Campbell, J., Sukthankar, R., Nourbakhsh, I., and Pahwa, A. (2005). A robust visual odometry and precipice detection system using consumer-grade monocular vision. In *Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on*, pages 3421–3427. IEEE. pages 10
- Carey, S. J. (2017). Personal Communication. pages 7, 26, 32, 82, 100
- Carey, S. J., Barr, D. R., Wang, B., Lopich, A., and Dudek, P. (2012). Locating high speed multiple objects using a SCAMP-5 Vision-Chip. In *Cellular Nanoscale Networks and Their Applications (CNNA), 2012 13th International Workshop on*, pages 1–2. IEEE. pages 1, 5, 8
- Carey, S. J., Lopich, A., Barr, D. R., Wang, B., and Dudek, P. (2013). A 100,000 fps vision sensor with embedded 535GOPS/W 256×256 SIMD processor array. In *VLSI Circuits (VLSIC), 2013 Symposium on*, pages C182–C183. IEEE. pages 1, 2, 3, 4, 8, 9, 26
- Carey, S. J., Lopich, A., and Dudek, P. (2011). A processor element for a mixed signal cellular processor array vision chip. In *Circuits and Systems (ISCAS), 2011 IEEE International Symposium on*, pages 1564–1567. IEEE. pages 4, 5, 82, 98
- Chen, J. (2016). SCAMP-5c Vision System. <https://github.com/jianingchen/scamp5c>. pages 9, 26, 28, 44

- Cheng, Y., Maimone, M., and Matthies, L. (2005). Visual odometry on the Mars exploration rovers. In *Systems, Man and Cybernetics, 2005 IEEE International Conference on*, volume 1, pages 903–910. IEEE. pages 10
- Chiuso, A., Favaro, P., Jin, H., and Soatto, S. (2000). 3-D motion and structure from 2-D motion causally integrated over time: Implementation. *Computer Vision—ECCV 2000*, pages 734–750. pages 11
- Chua, L. O. and Yang, L. (1988). Cellular Neural Networks: Theory. *IEEE Transactions on Circuits and Systems*, 35(10):1257–1272. pages 3
- Clemente, L. A., Davison, A. J., Reid, I. D., Neira, J., and Tardós, J. D. (2007). Mapping Large Loops with a Single Hand-Held Camera. In *Robotics: Science and Systems*, volume 2. pages 11
- Corke, P., Strelow, D., and Singh, S. (2004). Omnidirectional visual odometry for a planetary rover. In *Intelligent Robots and Systems, 2004.(IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*, volume 4, pages 4007–4012. IEEE. pages 10
- Davison, A. J. (2003). Real-time simultaneous localisation and mapping with a single camera. In *null*, page 1403. IEEE. pages 11
- Deans, M. C. and Hebert, M. (2005). *Bearings-only localization and mapping*. PhD thesis, Carnegie Mellon University, The Robotics Institute. pages 11
- Dominguez-Castro, R., Espejo, S., Rodriguez-Vazquez, A., Carmona, R. A., Foldesy, P., Zarándy, Á., Szolgay, P., Szirányi, T., and Roska, T. (1997). A 0.8- μm CMOS two-dimensional programmable mixed-signal focal-plane array processor with on-chip binary imaging and instructions storage. *IEEE Journal of Solid-State Circuits*, 32(7):1013–1026. pages 1, 3
- Dudek, P. (2003). A flexible global readout architecture for an analogue SIMD vision chip. In *Circuits and Systems, 2003. ISCAS'03. Proceedings of the 2003 International Symposium on*, volume 3, pages III–III. IEEE. pages 1, 4, 7, 8
- Dudek, P. (2004). Accuracy and efficiency of grey-level image filtering on VLSI cellular processor arrays. In *Proc. CNNA*, pages 123–128. pages 3
- Dudek, P. (2005). Implementation of SIMD vision chip with 128/spl times/128 array of analogue processing elements. In *Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on*, pages 5806–5809. IEEE. pages 1, 4, 5
- Dudek, P. and Hicks, P. J. (2000). A CMOS general-purpose sampled-data analog processing element. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 47(5):467–473. pages 1, 3
- Dudek, P. and Hicks, P. J. (2001). An analogue SIMD focal-plane processor array. In *Circuits and Systems, 2001. ISCAS 2001. The 2001 IEEE International Symposium on*, volume 4, pages 490–493. IEEE. pages 3

- Dudek, P. and Hicks, P. J. (2005). A general-purpose processor-per-pixel analog SIMD vision chip. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 52(1):13–20. pages 1, 3, 8
- Engel, J., Schöps, T., and Cremers, D. (2014). LSD-SLAM: Large-scale direct monocular SLAM. In *European Conference on Computer Vision*, pages 834–849. Springer. pages 11
- Fei-Fei, L., Fergus, R., and Perona, P. (2007). Learning generative visual models from few training examples: An incremental bayesian approach tested on 101 object categories. *Computer vision and Image understanding*, 106(1):59–70. pages 88
- Fischler, M. A. and Bolles, R. C. (1981). Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 24(6):381–395. pages 10, 11, 32
- Forster, C., Pizzoli, M., and Scaramuzza, D. (2014). SVO: Fast semi-direct monocular visual odometry. In *Robotics and Automation (ICRA), 2014 IEEE International Conference on*, pages 15–22. IEEE. pages 11
- Freund, Y., Schapire, R., and Abe, N. (1999). A short introduction to boosting. *Journal-Japanese Society For Artificial Intelligence*, 14(771-780):1612. pages 12
- Goecke, R., Asthana, A., Pettersson, N., and Petersson, L. (2007). Visual vehicle egomotion estimation using the fourier-mellin transform. In *Intelligent Vehicles Symposium, 2007 IEEE*, pages 450–455. IEEE. pages 10
- Haar, A. (1910). Zur Theorie der Orthogonalen Funktionensysteme. *Mathematische Annalen*, 69(3):331–371. pages 12
- Hackenberg, D., Ilsche, T., Schöne, R., Molka, D., Schmidt, M., and Nagel, W. E. (2013). Power measurement techniques on standard compute nodes: A quantitative comparison. In *Performance analysis of systems and software (ISPASS), 2013 IEEE international symposium on*, pages 194–204. IEEE. pages 82
- Handa, A., Whelan, T., McDonald, J., and Davison, A. (2014). A Benchmark for RGB-D Visual Odometry, 3D Reconstruction and SLAM. In *IEEE Intl. Conf. on Robotics and Automation, ICRA*, Hong Kong, China. pages 11, 12
- Kerl, C., Sturm, J., and Cremers, D. (2013). Dense visual SLAM for RGB-D cameras. In *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*, pages 2100–2106. IEEE. pages 11
- Kirby, M. and Sirovich, L. (1990). Application of the Karhunen-Loeve procedure for the characterization of human faces. *IEEE Transactions on Pattern analysis and Machine intelligence*, 12(1):103–108. pages 12

- Klein, G. and Murray, D. (2007). Parallel tracking and mapping for small AR workspaces. In *Mixed and Augmented Reality, 2007. ISMAR 2007. 6th IEEE and ACM International Symposium on*, pages 225–234. IEEE. pages 11
- Kohonen, T. (1989). *Self-organization and Associative Memory: 3rd Edition*. Springer-Verlag New York, Inc., New York, NY, USA. pages 12
- Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105. pages 44, 103
- Lacroix, S., Mallet, A., Chatila, R., and Gallo, L. (1999). Rover self localization in planetary-like environments. In *Artificial Intelligence, Robotics and Automation in Space*, volume 440, page 433. pages 14
- Leiserson, C. E., Rose, F. M., and Saxe, J. B. (1983). Optimizing synchronous circuitry by retiming. In *Proceedings of the 3rd Caltech Conference on VLSI*, page 87. pages 66
- Leonard, J. J. and Durrant-Whyte, H. F. (1991). Mobile robot localization by tracking geometric beacons. *IEEE Transactions on robotics and Automation*, 7(3):376–382. pages 11
- Lienhart, R. (2013). Haarcascade Frontalface Default. https://github.com/opencv/opencv/blob/master/data/haarcascades/haarcascade_frontalface_default.xml. pages 90, 94
- Linan, G., Espejo, S., Dominguez-Castro, R., and Rodriguez-Vazquez, A. (2002). Architectural and basic circuit considerations for a flexible 128×128 mixed-signal SIMD vision chip. *Analog Integrated Circuits and Signal Processing*, 33(2):179–190. pages 1
- Matthies, L. and Shafer, S. (1987). Error modeling in stereo navigation. *IEEE Journal on Robotics and Automation*, 3(3):239–248. pages 10
- Mei, C., Sibley, G., Cummins, M., Newman, P., and Reid, I. (2011). RSLAM: A system for large-scale mapping in constant-time using stereo. *International journal of computer vision*, 94(2):198–214. pages 11
- Milford, M. J. and Wyeth, G. F. (2008). Single camera vision-only SLAM on a suburban road network. In *Robotics and Automation, 2008. ICRA 2008. IEEE International Conference on*, pages 3684–3689. IEEE. pages 10
- Moravec, H. P. (1980). Obstacle avoidance and navigation in the real world by a seeing robot rover. Technical report, STANFORD UNIV CA DEPT OF COMPUTER SCIENCE. pages 10
- Mur-Artal, R., Montiel, J. M. M., and Tardos, J. D. (2015). ORB-SLAM: a versatile and accurate monocular SLAM system. *IEEE Transactions on Robotics*, 31(5):1147–1163. pages 11

- Naroditsky, O., Zhou, X. S., Gallier, J., Roumeliotis, S. I., and Daniilidis, K. (2012). Two efficient solutions for visual odometry using directional correspondence. *IEEE transactions on pattern analysis and machine intelligence*, 34(4):818–824. pages 10
- Newcombe, R. A. and Davison, A. J. (2010). Live dense reconstruction with a single moving camera. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, pages 1498–1505. IEEE. pages 11
- Nistér, D., Naroditsky, O., and Bergen, J. (2004). Visual odometry. In *Computer Vision and Pattern Recognition, 2004. CVPR 2004. Proceedings of the 2004 IEEE Computer Society Conference on*, volume 1, pages I–I. Ieee. pages 10
- Olson, C. F., Matthies, L. H., Schoppers, H., and Maimone, M. W. (2000). Robust stereo ego-motion for long distance navigation. In *Computer Vision and Pattern Recognition, 2000. Proceedings. IEEE Conference on*, volume 2, pages 453–458. IEEE. pages 14
- Paeth, A. W. (1986). A Fast Algorithm for General Raster Rotation. In *Proceedings on Graphics Interface '86/Vision Interface '86*, pages 77–81, Toronto, Ont., Canada, Canada. Canadian Information Processing Society. pages 39
- Poikonen, J., Laiho, M., and Paasio, A. (2009). MIPA4k: A 64× 64 cell mixed-mode image processor array. In *Circuits and Systems, 2009. ISCAS 2009. IEEE International Symposium on*, pages 1927–1930. IEEE. pages 1
- Saeedi, S., Paull, L., Trentini, M., Seto, M., and Li, H. (2014). Map merging for multiple robots using Hough peak matching. *Robotics and Autonomous Systems*, 62(10):1408–1424. pages 11
- Scaramuzza, D. and Fraundorfer, F. (2011). Visual odometry [tutorial]. *IEEE robotics & automation magazine*, 18(4):80–92. pages 10, 11
- Scaramuzza, D., Fraundorfer, F., Pollefeys, M., and Siegwart, R. (2009). Absolute scale in structure from motion from a single vehicle mounted camera by exploiting nonholonomic constraints. In *Computer Vision, 2009 IEEE 12th International Conference on*, pages 1413–1419. IEEE. pages 10
- Smith, R., Self, M., and Cheeseman, P. (1990). Estimating uncertain spatial relationships in robotics. In *Autonomous robot vehicles*, pages 167–193. Springer. pages 11
- Steinbrücker, F., Sturm, J., and Cremers, D. (2011). Real-time visual odometry from dense RGB-D images. In *Computer Vision Workshops (ICCV Workshops), 2011 IEEE International Conference on*, pages 719–722. IEEE. pages 10
- Sturm, J., Engelhard, N., Endres, F., Burgard, W., and Cremers, D. (2012). A Benchmark for the Evaluation of RGB-D SLAM Systems. In *Proc. of the International Conference on Intelligent Robot Systems (IROS)*. pages 11

- Tange, O. (2011). GNU Parallel - The Command-Line Power Tool. ;login: *The USENIX Magazine*, 36(1):42–47. pages
- Tardif, J.-P., Pavlidis, Y., and Daniilidis, K. (2008). Monocular visual odometry in urban environments using an omnidirectional camera. In *Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on*, pages 2531–2538. IEEE. pages 10
- Turk, M. and Pentland, A. (1991). Eigenfaces for recognition. *Journal of cognitive neuroscience*, 3(1):71–86. pages 12
- Viola, P. and Jones, M. (2001). Rapid object detection using a boosted cascade of simple features. In *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, volume 1, pages I–I. IEEE. pages 2, 12, 13, 44, 90, 94, 102, 103
- Wang, Y.-Q. (2014). An analysis of the Viola-Jones face detection algorithm. *Image Processing On Line*, 4:128–148. pages 13, 90
- Whelan, T., Johannsson, H., Kaess, M., Leonard, J. J., and McDonald, J. (2013). Robust real-time visual odometry for dense RGB-D mapping. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pages 5724–5731. IEEE. pages 14
- Whelan, T., Leutenegger, S., Salas-Moreno, R., Glocker, B., and Davison, A. (2015). ElasticFusion: Dense SLAM without a pose graph. *Robotics: Science and Systems*. pages 11
- Zarandy, A., Dominguez-Castro, R., and Espejo, S. (2002). Ultra-high frame rate focal plane image sensor and processor. *IEEE Sensors Journal*, 2(6):559–565. pages 3