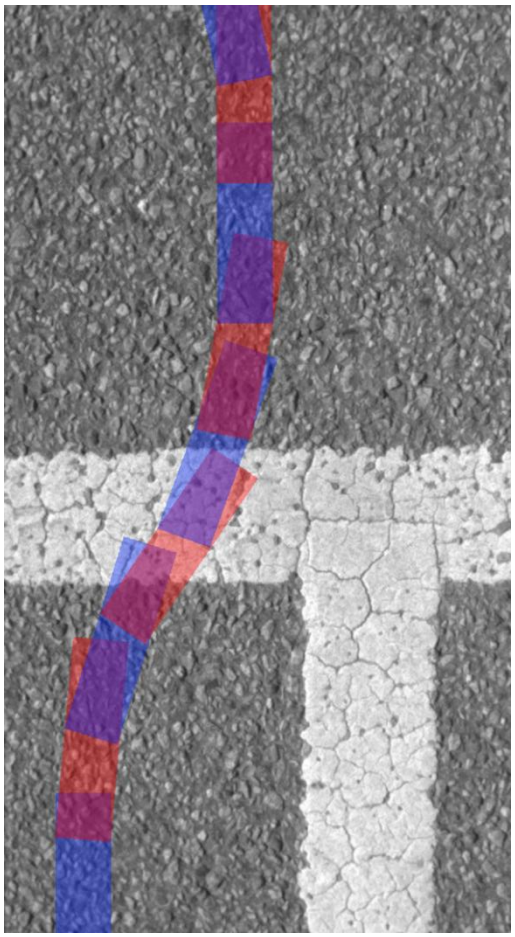
Master report, IDE 1238, May 2012
Embedded and Intelligent Systems

Movement sensor using image correlation on a multicore platform



Master thesis
School of Information Science, Computer and Electrical Engineering

Jonas Green, Christoffer Lind & Thomas Ingvarsson

Movement sensor using image correlation on a multicore platform

Master's thesis in Embedded and Intelligent Systems

School of Information Science, Computer and Electrical Engineering

Halmstad University

Box 823, S-301 18 Halmstad, Sweden

May 2012

Description of cover page picture: Image taken of a parking lot with blue and red transparent rectangles to illustrate photographed areas.

Preface

This report is the result of a thesis work within the master's program Embedded and Intelligent Systems at Halmstad University.

We would like to thank our supervisor Thomas Nordström for guiding us throughout the project. We would also like to thank Tommy Salomonsson and Björn Åstrand for providing us with a test vehicle and camera. Additionally we would like to thank Zain Ul Abdin and Johan Carlsson for supporting us with the hardware platform respectively the test vehicle.

This is the concluding part of our five years at Halmstad University.

Christoffer Lind, Jonas Green, Thomas Ingvarsson

Abstract

The purpose of this study was to investigate the possibility to measure speed of a vehicle using image correlation. It was identified that a new solution of measuring the speed of a vehicle, as today's solution does not give the *True Speed Over Ground*, would open up possibilities of high precision driving applications. It was also the intention to evaluate the performance of the proposed algorithm on a multicore platform. The study was commissioned by Halmstad University.

The investigation of image correlation as a method to measure speed of a vehicle was conducted by applying the proposed algorithm on a sequence of images. The result was compared to reference points in the image sequence to confirm the accuracy. The performance of the multicore platform was measured by counting the clock cycles it took to perform one measurement cycle of the algorithm.

It was found out that using image correlation to measure speed has a positional accuracy of close to a half percent. The results also revealed that one measurement cycle of the algorithm could be performed in close to half a millisecond and the achieved parallel utilization of the multicore platform was close to eighty-seven percent.

It was concluded that the algorithm performed well within the limit of acceptance. A conclusion about the performance was that low execution time of a measurement cycle makes it possible to execute the algorithm at a frequency of eighteen hundred Hertz. With a frequency that high, in combination with the camera settings proposed in the thesis, the algorithm would be able to measure speeds close to one thousand one hundred kilometers per hour.

The authors recommend that future work should be focused on investigating the camera parameters to be able to optimize both the memory and computational requirements of the application. It is also recommended to look closer at the algorithm and the possibilities of detecting transversal and angular changes as it would open up for other application areas, requiring more than just the speed.

Table of Contents

List of figures

List of tables

List of equations

List of code snippets

1 Introduction

This chapter will give an introduction to the problem that this project is aimed to solve as well as a proposed solution. This chapter starts with a brief introduction to the application area followed by a description of the problem as well as the proposed solution. After that the goals with the project and its requirements and limitations are described.

1.1 Application and Technology Area

The field of transportation has experienced a rapid development during the last century. With this development it is today possible to transport large amounts, goods as well as people, at high speeds. As the transport industry grew it also became cheaper and cheaper. Flying to exotic places far away has become a normal holiday for the ordinary family. The world we once knew has shrunk and today there is nowhere we cannot go. With speeds high enough to even make it possible to go around the world within a couple of days requirements on the safety has arisen.

To deal with these safety requirements computer systems have been introduced, where humans do not have the reactionary time or capability to do what is needed to avoid accidents. Vehicles such as large boats and airplanes already have hundreds of these computer systems to make it safe for their sometimes also thousands of passengers or tons of goods. A smaller vehicle such as the car, which has always been more dependent on the actual driver, has not so far experienced the same development but has recently become a very active field of research.

Key areas within the field are co-operative and autonomous automatic driving, two areas which require high precision driving to succeed at optimizing, simplifying and making the driving experience safer. However, the idea to have co-operative automatic driving is not new, already in the 30’s there was research done within the area [1]. One of the big restrictions to the idea back then, which still stands today and though it was just a prototype, was the weakness in sensors and sensor information.

1.2 Problem Statement

The modern car becomes more and more intelligent, dependent on highly advanced computer systems to make the driving experience both safe and optimal. To be able to do this these systems require precise and accurate real-time sensor data, information about both the vehicle and its surrounding.

Most important is the vehicle's movement, which has become source of information in need of modernization. A cars movement can be described as a set of features; velocity, acceleration, position, height over ground, wheel rotation, etc. Where some features are more important than others but where all of them can be used in some way.

This project will focus on the need of one specific feature, velocity, as this is a very commonly used feature. Velocity can be described as a speed in a given direction, whereas speed only describes how fast something moves velocity will also describes in which direction something moves. Speed is a physical quantity that is measured in a rather primitive way, especially in commercial cars. A common way to get the speed is by measuring the rotation of the wheel shaft. The rotation is proportional to the speed of the vehicle which makes it is easy to calculate the vehicles' speed. Using a Hall element to detect the magnetic field created when a magnet mounted on the axle shows up in front of the sensor is one way. This technique has been used for a very long time and is still good enough in many cases, for an example in a car. But as mentioned earlier, the future will bring more computer controlled vehicles and they need more accurate and failsafe sensors. By using the wheel shaft the measured value is affected by the current wheel dimension, the tire wear and by wheel spin. If a computer, controlling the vehicle, uses the speed value given by the Hall element when the wheels spin, it may make a wrong decision due to data fetched from a non-failsafe sensor.

To avoid this kind of problems, both to get more accurate data and data considered to be true in all situations, is to use a sensor that does not depend on anything else than the movement of the vehicle. Such sensor would provide so called True Speed Over Ground (TSOG).

1.3 Chosen Approach

The chosen approach, inspired by the precision of an optical computer mouse, to determine the TSOG is an approach using ground facing cameras and image analysis. A camera is used to capture images consecutively where the offset between each image is used to calculate a positional speed. This method should also theoretically be able to measure lateral and angle changes.

The chosen method shall, based on the high computational requirements due to the high frame rate required to be able to measure at the speed of 120 km/h, be parallelized and implemented on a multicore platform.

1.4 Goals

The projects goals have been to:

1. To evaluate and select a suitable image analysis method to measure offset in images.
2. Estimate the precision of the chosen method with test data.
3. Implement a sequential solution of the chosen method.
4. Measure the precision and timings of the sequential solution.
5. Implement the chosen method on a multicore solution to be tested as a prototype.
6. Measure the timings of the prototype.

1.5 Requirements and Limitations

This project involves many different areas. To define what areas that shall be investigated further, the following requirements are set for this project to fulfill;

1. Determine an image analysis method to calculate the movement of the analyzed surface.
2. Possible to measure speeds up to 120 kilometers per hour.
3. Implement the proposed solution on a parallel architecture.
4. The parallel implementation shall utilize the architectures' advantages where it is possible.
5. Evaluate the parallel architecture after the needs of the implemented application.

Some areas will not be taken into consideration by this project. The excluded areas, referred to as limitations, are;

1. Find the optimal camera for this application.
2. Decide optimal settings of the camera or optimal environmental needs like illumination.
3. Calculate or establish the optimal height of camera or overlapping distance between images.
4. Come up with a commercially usable product.

2 Background

The background addresses important theories for this project. Current solutions and works that are closely related, or similar, to this project are presented in section 2.1 and 2.2. This is followed by the section 2.3, 2.4 and 2.5 which explains algorithms and techniques related to images analysis based approaches that are relevant for this project. Section 2.6 gives an introduction to multicore systems and explains different design choices of this kind of systems. The multicore theory is followed by a preview of two developed multicore architectures in 2.7. This chapter ends with section 2.8 which explains different camera settings and shows examples of how the images are affected by these settings.

2.1 Current Solutions

There are several techniques and methods available to estimate speed. The by far most used method as of today is the hall sensor method which is used in most commercially available cars. This section will describe various methods used to measure speed.

2.1.1 Hall Sensor

The common way to measure speed of a car is to use a Hall sensor. It measures the repetitions of the rotation of the driveshaft. Combined with a gearing ratio a speed can be calculated. Further on dead reckoning can be used to measure the distance a vehicle has travelled.

This can be seen as a primitive but reliable way to measure speed. One big disadvantage is the inflexibility, as it requires a predetermined wheel size to translate driveshaft repetitions to the actual distance travelled. Changing wheels to a smaller size will result in an error linear to the speed which is not acceptable. Tire wear will also result in an incorrect measurement.

The theoretical solution to this is to measure the wheel size by measuring the radius of the wheel, from the wheel's center to the ground, and with it compensate for different wheel sizes. However,

another problem exists when a slip occurs between the wheel and the road. This will result in that the car thinks it goes faster than it does and it is not the TSOG.

2.1.2 GPS

A technique that becomes more and more generally recognized as a commercial navigation, and velocity estimator, is the Global Positioning System (GPS). There are commercial systems with a precision as low as 3 meters [2]. The error in precision is however too big to be ignored as it could cause dangerous situations where you require centimeter accuracy. Even though the precision of the position is far from the requirements of this project the velocity has far better precision. The velocity is calculated through derivation which gives the error less impact on the measurement. At low speeds the error will still exist but the higher speed the smaller the error will be because the error is static. There are also GPS systems which do not measure the altitude which makes the velocity wrong if the road tilts.

To sum up the GPS system has a measurement error when it comes to positioning. Its big advantage is that it is global and gives an absolute position all over the world, making it a good navigation tool.

2.1.3 Radar

Radar is a technique which has not been commercialized as a speedometer but can still be considered as an alternative to the previously mentioned techniques. The technique was before the 90’s considered too expensive to use commercially, but during that century there where several papers [3][4][5] discussing the opposite. The radar transmits a signal with a pre-determined frequency. The signal is reflected by the ground and detected by a receiver on the car. By utilizing the Doppler Effect, the change in frequency for a moving object relative its source, good accuracy can be achieved when determining the speed, TSOG to be exact. A more recent example of a radar solution utilizes two radars to measure both the ground speed and the angular change [6].

2.1.4 Optical

This sensor consists of a lamp lighting the surface beneath the vehicle and a light sensor detecting the light reflected by the ground. The reflected light passes an optical spatial filter and then reaches a photo detector array. This results in a time varying signal proportional to the speed.

This technique was described by Luigi Cocco and Sergio Rapuano in [7]. They evaluated two similar methods to measure accurate speed of a Formula One car. By using an infra-red light source and a photo detector array to measure the reflected light from the ground the speed can be determined. Before the reflected light reach the sensor it passes a two-phase optical grating system. It consist of two lenses and in between those a grating filter. A time-varying signal is created by this filter and is described in the paper as following; "When there is a relative movement between the surface and the sensor, the spatial filter selectively interferes with the particular spatial frequency component in the projected surface pattern, resulting in a time-varying signal". The speed of the vehicle is proportional to the frequency from this signal.

2.2 Closely Related Work

This section will give information about solutions that are related to the chosen approach for this project. Three different solutions will be introduced, one using an optical mouse sensor, one optical velocity sensor from a company named Correvit and one phase only correlation based solution implemented on an FPGA.

2.2.1 Optical Mouse Sensor

In the two papers [9][10] image correlation is used for positioning. There they used a commercial optic mouse sensor with a replaced lens. They achieved a very cost effective way to measure sub meter accuracy, but at a much lower velocity than required here. During their testing they had two major problems with their setup. The first problem was the height, where different heights above the ground gave incorrect measurements. To compensate for this a rangefinder was integrated so that the height could be measured continuously. This approach does only work within certain height differences as the big differences will make the lens go out of focus making the correlation harder to perform.

The second problem was more of a local sort as they had problem with transferring the data from the sensor to their system located in the vehicle, they solved this with a wireless radio connection. Even if these problems would be solved this system would not be able to measure a high enough speed required by this project.

There has also been some similar small scale work done with robots where the focus has been on precise detection of the TSOG in lower speed [10]. They did not have the need to scale it to higher speeds than 8.3 km/h, so also here they could use commercial mouse sensors to perform the correlation.

2.2.2 Non-Contact Optical Velocity Sensors

Corrsys-Datron is a company that sells non-contact optical velocity sensors, branded Correvit, that measures dynamic variables such as speed, distance, angle and height. In 1991 they introduced their first optical speed and distance sensor. The technique behind the sensor is based on correlation of the surface on micrometer level using an optical sensor [11].

A method where a Correvit sensor is used in combination with a camera to detect angular changes is presented in [12]. The camera is mounted in the front of the car and takes pictures of the surrounding environment in front of the car. The lateral and angular changes detected by the camera in combination with the speed of the Correvit sensor makes it possible to calculate the velocity.

2.2.3 Phase Only Correlation on FPGA

An FPGA solution was proposed in [13]. The proposed solution is based on the same principle as this project where an embedded system is used to calculate the offset between two consecutive images captured by a camera. The camera is a CMOS camera circuit which is connected to an FPGA. The camera is controlled from the FPGA with an I^2C interface. A "CMOS capture & pre-processing unit" is responsible for capturing two consecutive images from the camera and storing them in an external SDRAM. The images are then later grabbed from this SDRAM for further processing. The result of the velocity calculation is presented on a LCD monitor and with a LED.

The camera is capturing 60 frames per second with a resolution of 640 by 480. The active resolution that is used in the image processing step is however only 256 by 256. This resolution is used to optimize the FFT where a length that is a power of two is preferred. Figure 2.2-1 shows the image analyze algorithm in form of a flow chart.

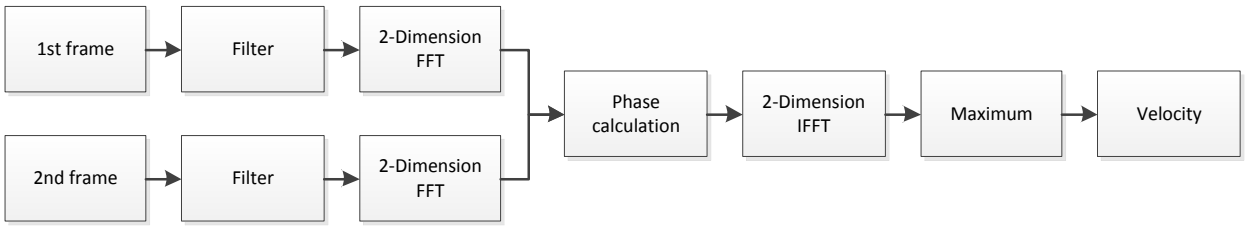


Figure 2.2-1: Phase correlation steps

As described before two consecutive images are captured by the FPGA. The two images are then filtered individually with a mean filter to decrease external interferences. Then the actual phase only correlation starts which is used to calculate the offset between two images with some overlap. The phase only correlation, which comprises the "2-Dimension FFT" block to the "Maximum" block, is explained in detail in 0 and will therefore not be explained here. The actual FFT algorithm, which is the most computational heavy block, is based on a pipelined approach. The output, velocity in this case, is finally calculated by using the offset, the frame rate and the actual size of each pixel on the ground.

2.3 Image Pre-processing

Images are often pre-processed before analyzed in later stages of an algorithm. This section will explain how normalization and filtering work. Another technique to represent an image, instead of the Cartesian coordinate representation, called Log-polar will also be introduced.

2.3.1 Normalization

Normalization within image processing is a process which changes the range of intensity in each pixel. It is very common that a picture does not use the whole range, the lightest parts are not pure white and the darkest parts are not pure black and if so normalization can be done so that the whole range is used. This improves the contrast in the image as the differences between light and dark increases, at the same time information is lost as the level of illumination in the picture is deformed. The result of normalization is that two images taken with the same camera but with different conditions, such as illumination, can have the same visible features afterwards. This way illumination can be ignored if needed as stated earlier.

The process is because of this sometimes called contrast stretching but within data processing the term dynamic range expansion is instead used. The term comes from when a signal needs to be within a certain range, a range expected by the application.

Two common applications for normalization is images with poor contrast due to glare or bad lighting or any newspaper having pictures in gray scale wanting to normalize them to the same intensity.

Another kind of normalization that is popular within computer vision is color normalization. As color images has three colors instead of only one as monochrome images it is possible to normalize the colors relative each other, this is called color normalization. The effect is that different levels of illumination within the image are filtered and pixels with the same color but different intensity are after normalization seen upon as the same.

Normalization is a linear process which can be divided into two parts; offset and scale. Offset is the difference between the starting points of two ranges while scale is the factor difference between the lengths of the ranges.

Image *A* have a range *a* and there is a goal range *b*, ranging from a_{min}/b_{min} to a_{max}/b_{max}. The offset and scale to normalize image *A* to range *b* is calculated as following;

$$Offset = b_{min} - a_{min}$$

Equation 2.3-1: Offset used by the normalization

$$Scale = \frac{b_{max} - b_{min}}{a_{max} - a_{min}}$$

Equation 2.3-2: Scale used by the normalization

The normalization is then applied to each pixel of the image as in Equation 2.3-3.

$$B_i = (A_i + Offset) \cdot Scale$$

Equation 2.3-3: Normalization of each pixel i

Color normalization is, just as monochrome normalization, a linear process but instead of having each pixel shifted and re-scaled each color is re-scaled based on its weight among the other colors as in Equation 2.3-4. *A* is the image before normalization and *c* is one of the colors *r*, *b* and *g* in pixel *i*.

$$B_{ci} = \frac{A_{ci}}{A_{ri} + A_{bi} + A_{gi}}$$

Equation 2.3-4: Color normalization

Equation 2.3-4 assumes that the colors red, green and blue all affect the illumination with the same weight which it does not do. The equation should therefore add the different weights for each color to be complete, see Equation 2.3-5.

$$B_{ci} = \frac{K_c \cdot A_{ci}}{K_r \cdot A_{ri} + K_b \cdot A_{bi} + K_g \cdot A_{gi}}$$

Equation 2.3-5: Weighted color normalization

2.3.2 Filtering

Image filters can be used for many different purposes. The purpose of a filter can be divided into two main areas; either the goal is to enhance something or to reduce something. With some filters both goals can be achieved at the same time. An image filter can be defined as a matrix of arbitrary size. This can result in many different filter shapes where the values in the matrix define the outcome after the filter has been applied. A filter is normally applied to an image using convolution.

In a lot of cases the goal with image filtering is to reduce noise. A commonly known method to achieve this is to use a low pass filter. The outcome from a low pass filtering is not only less noise but also reduced sharpness, an outcome that has to be taken into consideration before filtering. A simple low pass filter, or average filter, is shown in Figure 2.3-1.

$$F_{LP} = \begin{bmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{bmatrix}$$

Figure 2.3-1: Low pass filter

The low pass filter from Figure 2.3-1 is applied to an image in Figure 2.3-2. The figure shows, starting from the left; a non-filtered image, an excerpt of the image's pixel values, an excerpt of the resulting image's pixel values after convolution and the low pass filtered image. So what will happen because the filter size is 3 by 3 is that every pixel in the new image will be a result of 9 pixels from the original image. The resulting value 71 that is highlighted to the right is calculated by multiplying each value in the left matrix by its corresponding filter value and then adding them all together.

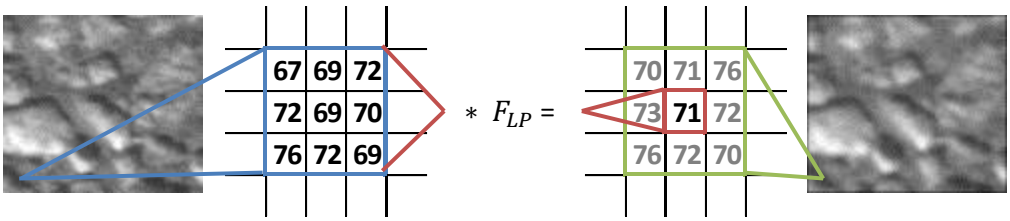


Figure 2.3-2: Example of low pass filtering result

Directional Filtering

Another commonly used filter type is directional filters. This kind of filter can be applied to images to find edges. An edge is defined as a large value change between two adjacent pixels in the image. This large value change can be identified by using a directional filter. There are many different versions of directional filters. This section will however mainly focus on one variant, Sobel filter, which is one of the most common directional filters. Figure 2.3-3 shows a Sobel filter in matrix form.

$$F_{Sobel} = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Figure 2.3-3: Sobel filter

If this specific filter is applied to an image the horizontal edges in the image will be highlighted. If instead the transpose of the same filter is applied the result will contain highlighted vertical edges. By then combining the result from the horizontal and vertical filtering it is possible to get an image with all edges highlighted. An example of this can be seen in the figures below; Figure 2.3-4 shows an image of a set of lockers, Figure 2.3-5 is the same image after the filter from Figure 2.3-3 has been applied, Figure 2.3-6 is the result after the transpose of the same filter has been applied and Figure 2.3-7 is the result from combining the previous two images by element wise addition.

Figure 2.3-4: Lockers

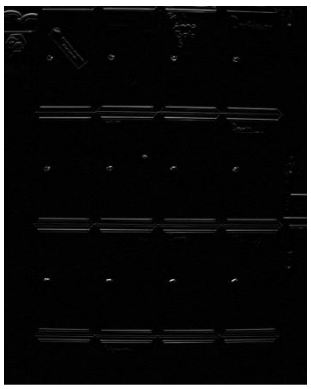


Figure 2.3-5: Horizontal sobel

Figure 2.3-6: Vertical sobel

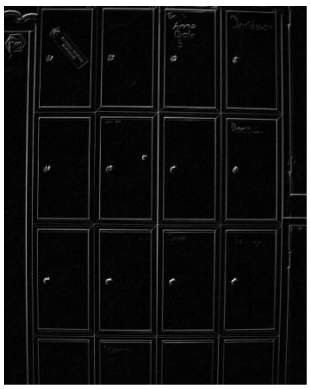


Figure 2.3-7: Combined sobel

The result that is achieved by an edge detection filter is very useful to distinguish objects and details in an image. Different objects often have different colors which results in an edge which this kind of filter is specialized for to find. An alternative to the Sobel filter is the Prewitt Filter which also has edge detection features. The shape of the filter is very similar and can be viewed in Figure 2.3-8.

$$F_{Prewitt} = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}$$

Figure 2.3-8: Prewitt filter

2.3.3 Log-polar

The common way to represent a coordinate is by using the Cartesian coordinate system. It uses two axes to localize each point in a plane. This is the normal way to describe the localization of a pixel in an image. The position of a pixel is defined by one value related to each axis, usually named x and y. Figure 2.3-9 shows a coordinate where both the x and y value are set to 2.

The Log-polar coordinate system is another coordinate system which can be used to describe an image, especially in computer vision. This system also uses two values to represent the location of each pixel but instead of the position along two axes, it uses the length p and the angle θ relative a reference point. The length p is represented in a logarithmic scale, hence the name Log-polar. Two Log-polar coordinates are shown in Figure 2.3-10. The coordinates have the lengths 1.8 respectively 0.7 from the reference point and the angels 33.7° respectively 315.0°.

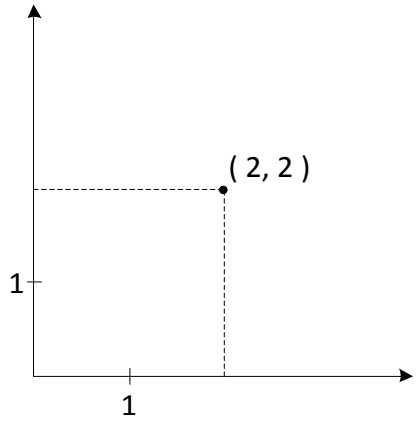


Figure 2.3-9: Cartesian coordinate system

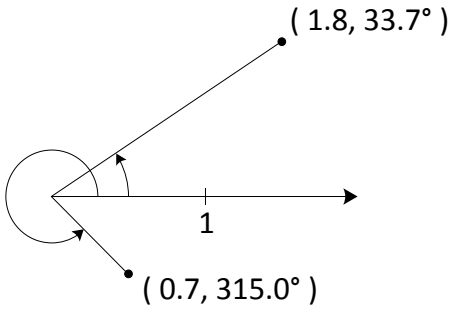


Figure 2.3-10: Log-polar coordinate system

To transform an image described in the Cartesian coordinate system to Log-polar coordinate system new pixels has to be sampled. How this is done is shown in Figure 2.3-11. A pattern is put on the Cartesian coordinate system with its center point equal to the chosen reference point.

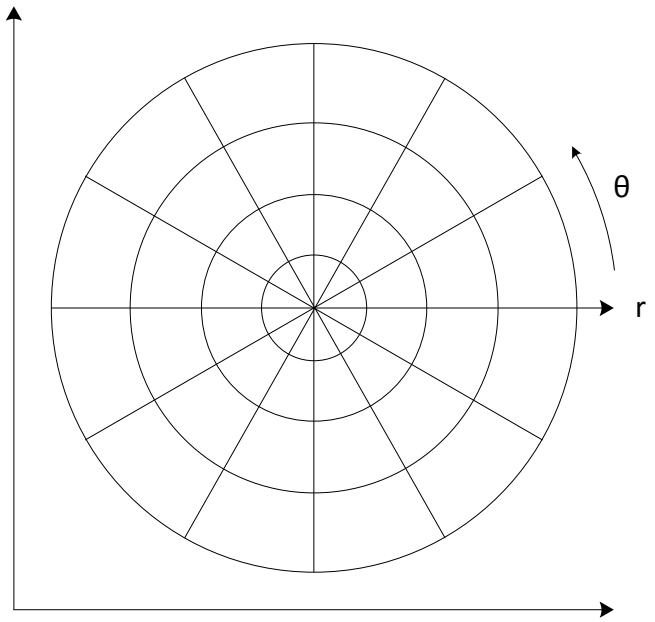


Figure 2.3-11: Log-polar transform sampling in Cartesian coordinate system.

To transform a coordinate to Log-polar coordinate the length p and angle θ has to be calculated. The length p is the radial distance which is calculated using Equation 2.3-6 where (x_c, y_c) is the center point.

$$p = \log_{base} \sqrt{(x - x_c)^2 - (y - y_c)^2}$$

Equation 2.3-6: Calculate the radial distance p, (x, y) – Cartesian coordinate to transform, (x_c, y_c) – Center point

The angle θ is the angular distance which is calculated using Equation 2.3-7.

$$\partial = \tan^{-1} \frac{y - y_c}{x - x_c}$$

Equation 2.3-7: Calculate the angular distance θ, (x, y) – Cartesian coordinate to transform, (x_c, y_c) – Center point

When all coordinates have been transformed to Log-polar representation the circular pattern can be shown as a squared image as in Figure 2.3-12. The radial distance is now represented in vertical direction and the angular distance is represented in horizontal direction.

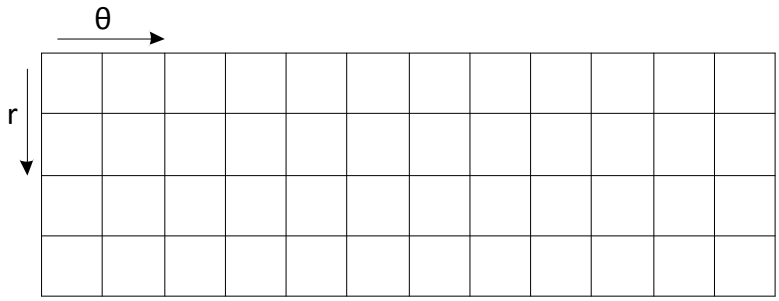


Figure 2.3-12: Image in Log-polar coordinate system.

The Log-polar coordinate system is useful in image-processing because of its scaling and rotation properties [14]. If an image is either scaled or rotated in the Cartesian domain this is represented by an offset between the two images when transformed to Log-polar representation. To illustrate these features an image together with its Log-polar transform, presented in Figure 2.3-13, will be used.

(a) (b)

Figure 2.3-13: (a) - Example image. (b) - Log-polar transform of the image.

In Figure 2.3-14 (a) the image has been rotated 45 degrees. If the Log-polar transform of the original image, Figure 2.3-13 (b), and the rotated image, Figure 2.3-14 (b), are compared it can be seen that the rotation has been translated to a pure horizontal offset.

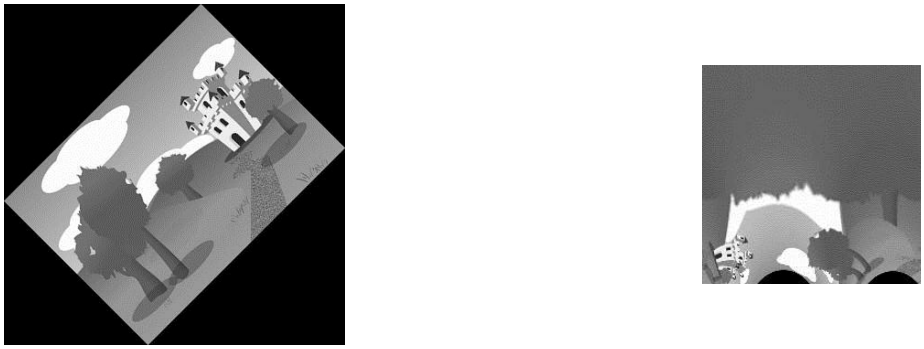


Figure 2.3-14: (a) - Image rotated 45 degrees. (b) - Log-polar transform of image.

To exemplify the scaling property the image in Figure 2.3-15 (a) has been scaled with a factor 0.5. As can be seen in Figure 2.3-15 (b) the Log-polar transform has turned the scaling into a pure vertical offset compared to the Log-polar transformation of the original image in Figure 2.3-13 (b).

(a) (b)

Figure 2.3-15: (a) – Image scaled with a factor 0.5. (b) – Log-polar transform of the image.

2.4 Fourier Transform

Jean Baptiste Joseph Fourier showed in 1807 that a periodic signal could be built up by a series of sine and cosine functions resulting in what today is called a Fourier series, Equation 2.4-1. Since sine and cosine are periodic functions with a frequency a Fourier series for a periodic signal is a series describing the frequencies belonging to that periodic signal. By extracting the coefficients of each sine and cosine a function of frequency is given.

$$f(x) = a_0 + \sum_{n=1}^{\infty} \left(a_n \cos\frac{n\pi x}{L} + b_n \sin\frac{n\pi x}{L} \right)$$

Equation 2.4-1: Definition of Fourier series

Fourier series can also be calculated for a non-periodic signal as long as it is finite, by repeating the finite signal infinitely it becomes periodic and its Fourier series can therefore be calculated.

Tightly connected to the Fourier series is the Fourier transform. The Fourier transform is a well-known tool, used for many different applications within physics as well as engineering, that translates a function of time, or space, in to a function of frequency. A function of space is dependent on distance or position instead of time, e.g. an image where there is a coordinate system of pixels. There are also cases where a function can be described as both a function of time and a function of space, the most common one a simple wave light signal as in Figure 2.4-1. Here the wave has a certain period in time but also a wavelength as the wave travels through space at the speed of light, making it possible to be described as both a function of time and a function of space.

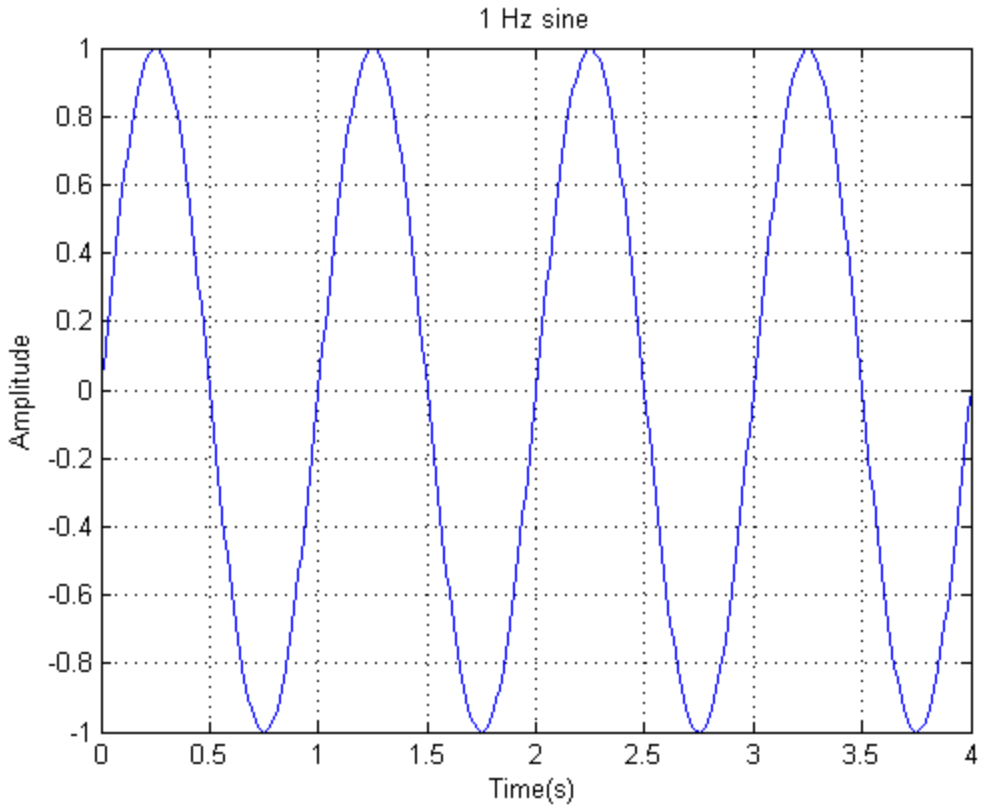


Figure 2.4-1: A sine wave as a signal of time

The result of a Fourier transform is often termed as a frequency domain representation or the functions frequency spectrum, where time or spatial domain representation is used for a function of time or space. Each value in the frequency domain represents a frequency and is expressed as a complex number, with both a magnitude, that describes the amount of that frequency in the function, and a phase that describes the initial angle of that frequency. Figure 2.4-2 shows the frequency representation of the sine wave in Figure 2.4-1, where a delta function at 1 Hz shows the only existing frequency in the given signal.

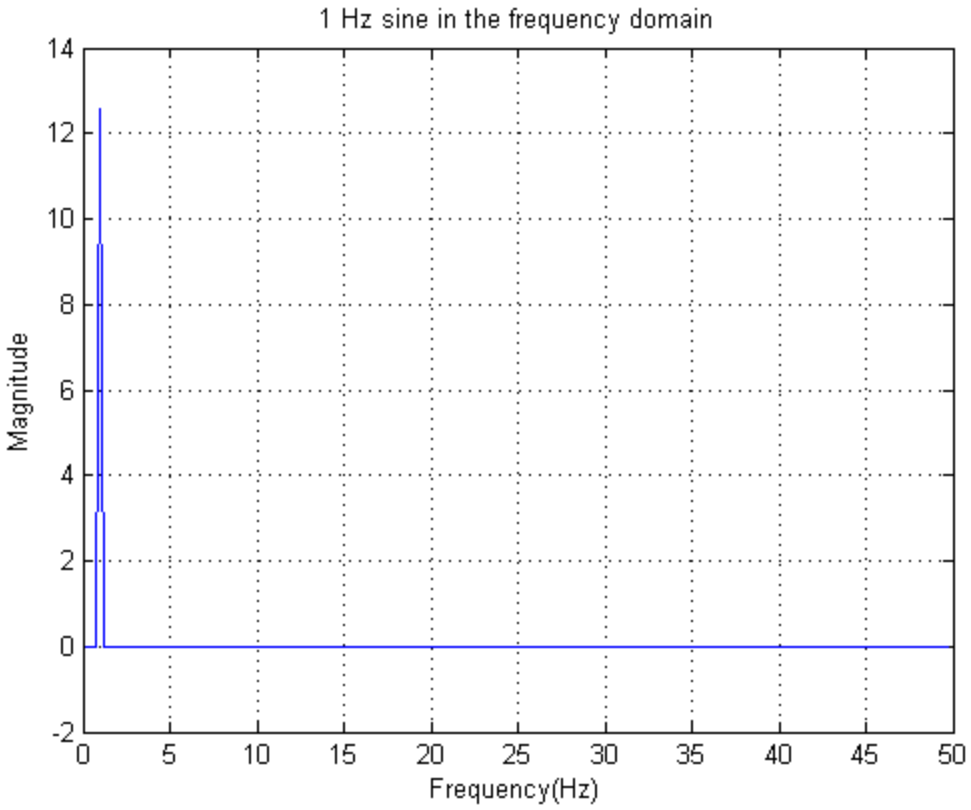


Figure 2.4-2: Frequency representation of a sine wave

Once in the frequency domain it is possible to go back to the time or spatial domain again through inverse Fourier transform, a method similar to the Fourier transform, together they make it possible to go from one domain to the other without any loss of information.

Fourier transform is widely used within almost all fields of science, whether it is physics, engineering, signal processing or mathematics. Wherever it is used it can be signal analysis, filtering, compression, differential equation or one of many other different applications.

The advantage with having a frequency domain representation of a function is besides the fact that you have the frequency representation, a set of properties that can be utilized. Since the Inverse Fourier transform gives the possibility to go back to the spatial domain again the properties can be utilized on a function that is later on transformed back to the spatial domain.

The definition of the continuous Fourier transform, Equation 2.4-2, is defined as the integral from negative infinity to positive infinity over the function of time, or space, multiplied with an exponential function. The notation used show the frequency function in terms of angular frequency, ω.

$$F(\omega) = \int_{-\infty}^{\infty} f(t) \cdot e^{i\omega t} dt$$

Equation 2.4-2: Definition of continuous Fourier transform

The inverse Fourier transform has smaller differences from the forward transform. With a change of the sign of the exponential function and a factor of 1/2π Equation 2.4-3, the inverse Fourier transform, is given.

$$f(t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} F(\omega) \cdot e^{-i\omega t} d\omega$$

Equation 2.4-3: Definition of continuous Inverse Fourier transform

Instead of a function in terms of angular frequency both the forward and inverse Fourier transform can also be defined in terms of oscillation frequency, *ν*.

For discrete signals there is an equivalent definition where a sum is used instead of an integral to summarize over the now finite signal. Equation 2.4-4 and Equation 2.4-5 show how a finite signal is transformed to the frequency domain and then back again if needed. In both equations *n* denotes a single sample of the *N* long finite signal and *k* a single frequency.

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-i2\pi \frac{k}{N} n}$$

Equation 2.4-4: Definition of discrete Fourier transform

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k \cdot e^{i2\pi \frac{k}{N} n}$$

Equation 2.4-5: Definition of discrete Inverse Fourier transform

Being able to go back and forth between the domains like this is a very usable property as it allows us to perform different operations in the frequency domain and then go back again. Some of the more important properties are listed in Table 2.4-1.

Linearity:	$\mathcal{F}\{ax(t) + by(t)\} = a\mathcal{F}\{x(t)\} + b\mathcal{F}\{y(t)\}$
Time/Frequency shift:	$\mathcal{F}\{x(t \pm t_0)\} = X(jw) \cdot e^{\pm j\omega t_0}$
Time/Frequency scaling:	$\mathcal{F}\{x(at)\} = \frac{1}{a} X(\frac{\omega}{a})$
Time reversal:	$\mathcal{F}\{x(-t)\} = X(-\omega)$
Symmetry:	If $\mathcal{F}\{x(t)\} = X(j\omega)$ then $\mathcal{F}\{X(t)\} = 2\pi \cdot x(-j\omega)$

Table 2.4-1: Important Fourier transform properties

The Fourier transform can also be applied to a multidimensional function, to deal with the multiple dimensions the transform is applied on each dimension across the other dimensions. For a two dimensional function the transform is applied to each row that spans across the columns and then for each column that spans across the rows.

An example of a two dimensional Fourier transform is given by first converting Figure 2.4-3, the famous Lena, to a monochrome image which is seen in Figure 2.4-4. This simplification is done to decrease the computation by only having to transform one color instead of three.

Figure 2.4-3: Lena

Figure 2.4-4: Monochrome Lena

The monochrome image is then transformed into the frequency domain shown in Figure 2.4-5, the amplitude of each frequency, and Figure 2.4-6, the phase of each frequency.

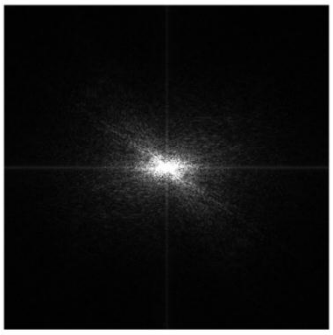


Figure 2.4-5: Amplitude of the monochrome Lena in the frequency domain

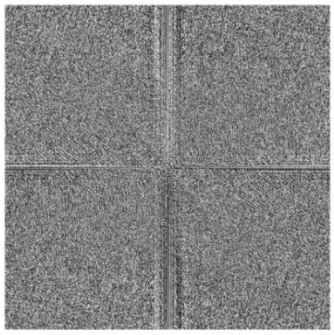


Figure 2.4-6: Angle of the monochrome Lena in the frequency domain

2.4.1 Fast Fourier Transform

The discrete Fourier transform is a very computational heavy process that requires $O(N^2)$ operations to be evaluated where N is the number of points to calculate. Through Fast Fourier transform, FFT, this number can be reduced to $O(N\text{Log}_2 N)$ operations resulting in a 100 times speedup for a 1024 point discrete Fourier transform. With $O(N\text{Log}_2 N)$ operations the FFT will not only be faster overall but vastly surpass it for larger numbers as it does not grow as fast, a comparison is made in Figure 2.4-7, where number of operations is plotted against the length of the input.

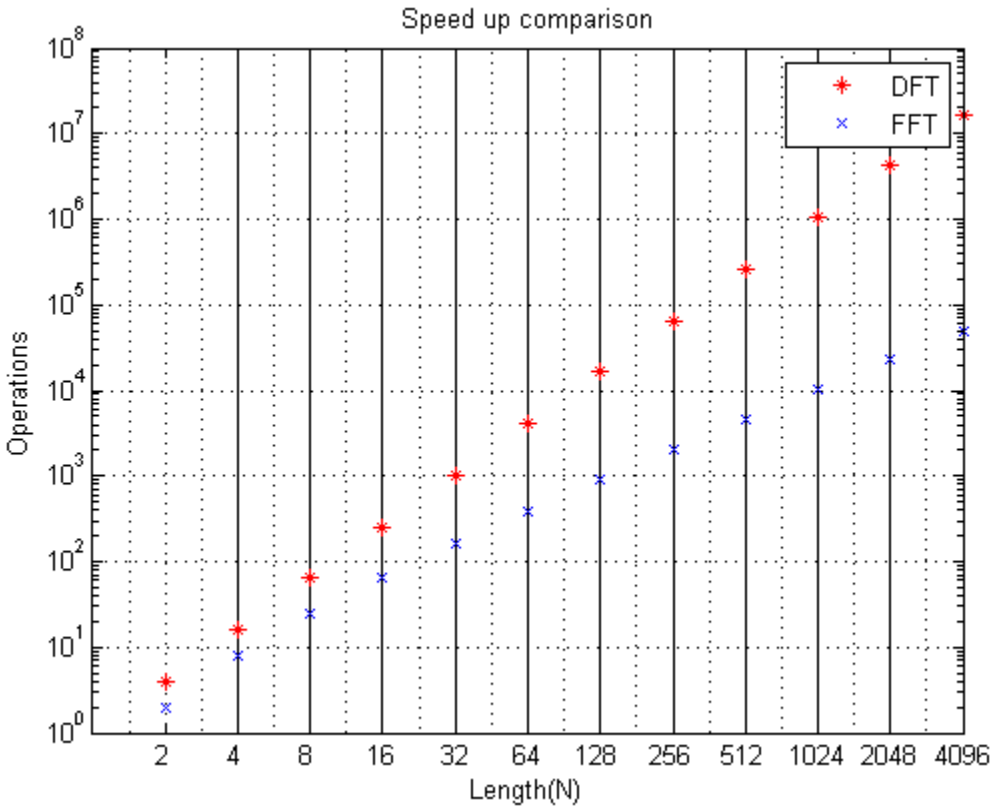


Figure 2.4-7: DFT and FFT speed comparison

There are several different ways to perform a FFT and the most common being the radix-2 Cooley-Tukey FFT algorithm which will be described here. The Cooley-Tukey FFT algorithm is not a different equation but a re-expression of the DFT. It re-expresses the DFT into smaller interleaved DFT's recursively. This way a reduction can be done on even and odd inputs separately resulting in not only a speedup but also in an increased precision due to reduced rounding errors. The rounding errors are decreased because of fewer operations. The reduction of the algorithm is mainly done by eliminating trivial operations, such as multiplication of one. The FFT has a restriction though due to the interleaving which is that the input has to have a length N, built on an arbitrary power of two, as in Equation 2.4-6 where *m* is any real number.

$$N = 2^m$$

Equation 2.4-6: Length N required for a FFT

When looking closer at the functionality of the FFT it is divided into three parts; interleaving, transforming and re-assembling into one frequency spectrum. The interleaving is done to prepare the signal for re-assembling and is done by recursively splitting the signal into odd and even pieces, seen in Figure 2.4-8. The prefix radix-2 of the famous Cooley-Tukey FFT algorithm comes from the fact that the interleaving is done by the base two, dividing each time into two new subsets.

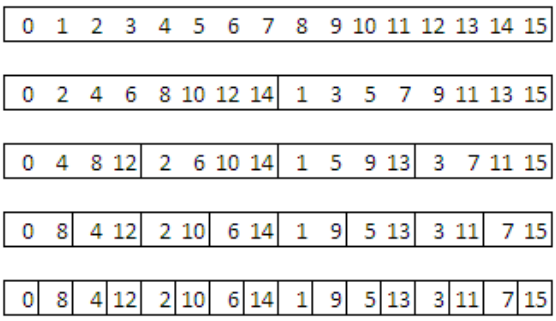


Figure 2.4-8: Interleaving of a 16 point signal

The interleaving of the signal in Figure 2.4-8 gives a re-arrangement of the original signal that can be visualized by the binary representation as in Figure 2.4-9. Here the given pattern of the radix-2 interleaving is truly shown; each sample's new position can be calculated through a bit reversal. A bit reversal is based on swapping the location of ones in a bit field, *0001* becomes *1000* etc.

Decimal	Binary	Decimal	Binary
0	0000	0	0000
1	0001	8	1000
2	0010	4	0100
3	0011	12	1100
4	0100	2	0010
5	0101	10	1010
6	0110	6	0110
7	0111	14	1110
8	1000	1	0001
9	1001	9	1001
10	1010	5	0101
11	1011	13	1101
12	1100	3	0011
13	1101	11	1011
14	1110	7	0111
15	1111	15	1111

Figure 2.4-9: Bit reversal of a 16 point signal

A 16 point signal, as in Figure 2.4-8, becomes 16 one point time domain signals after five recursive splits. These one point time domain signals are then easy to transform into frequency spectrums since it does not require any work as a one point time domain signal and its frequency spectrum is the same. This way the first two steps of the FFT are done by interleaving, i.e. bit reversing, and the only thing left to do is to re-assemble the frequency spectrums into one frequency spectrum.

The re-assembling is done by reversing the interleaving but with a so called FFT butterfly, Figure 2.4-10. The butterfly takes two complex points as input and has an output of two complex points. To match the two frequency spectrums the odd point is shifted. The shift is created by a multiplication with a sinusoid, represented by *S* in the figure.

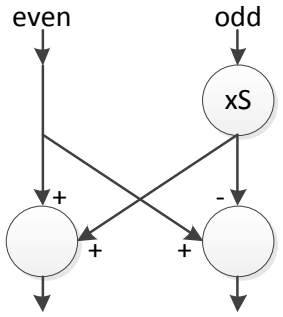


Figure 2.4-10: The FFT butterfly

The complete re-assembling is done by applying the butterfly in the same amount of steps as the interleaving, but instead of dividing the set by two for each step the process merges sets of two. Figure 2.4-11 shows how two frequency spectrums of each four points are merged together through a butterfly operation. This way the interleaving is reversed with the same amount of steps. At the end there is one frequency spectrum just as there was one signal before interleaving. This frequency spectrum is the final result of the Fourier transform.

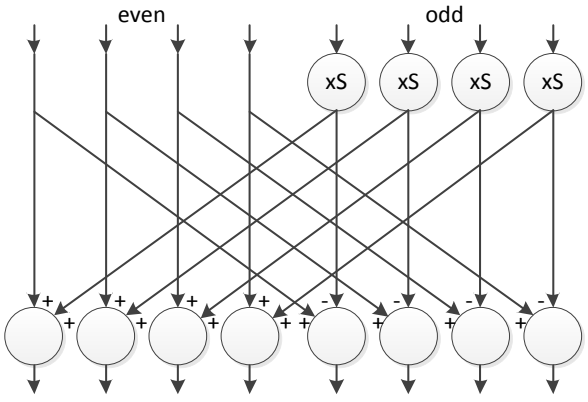


Figure 2.4-11: 8 point FFT butterfly

As inverse discrete Fourier transform differs very little from discrete Fourier transform so does the inverse FFT from FFT. The difference for discrete Fourier transform is an opposite sign on the exponent of the sinusoid and a factor $1/N$ on each sample, a difference that easily can be applied to the FFT without increasing the complexity.

2.5 Correlation

Correlation is used to find a relationship between two or more comparable objects. This section will show how correlation can be used to find the offset between two images that are shifted relatively each other. Cross correlation, calculated in the spatial domain, and phase correlation, calculated in the frequency domain, are introduced.

2.5.1 Cross Correlation

In signal processing cross-correlation measures the similarity between two signals as one of the signals is shifted with a time-lag. The result will be a signal showing the similarity in each shift. That makes cross-correlation good for applications such as pattern recognition where a known feature is searched for or when trying to measure an offset between signals.

To calculate the similarity between two signals, f and g, in one point the products of each matching pair in the whole signal is summarized. The summarization gives a value on the similarity. Cross-correlation is also known as sliding dot product as one signal slides along the other while the dot product is calculated for each point giving a value of similarity in each shift. A higher value means larger similarity and peaks in the signal will contribute the most to the result when they align. Negative peaks will because of the double negation give a positive result and contribute to the total result just as the positive peaks. A negative result on the other hand shows the similarity when one of the signals is mirrored over the x axis.

Two signals, Figure 2.5-1 and the same signal but shifted, Figure 2.5-2, can be cross-correlated to find the offset. When plotted against each other, Figure 2.5-3, the offset between these two very similar signals is easy to see with the eye. If the offset is not easy to see the cross-correlation in Figure 2.5-4 shows the offset with ease by finding the shift with the largest similarity. Beyond that the cross-correlation also shows that there are mirrored similarities in the result and that there is not only one point that is similar but many of different magnitude.

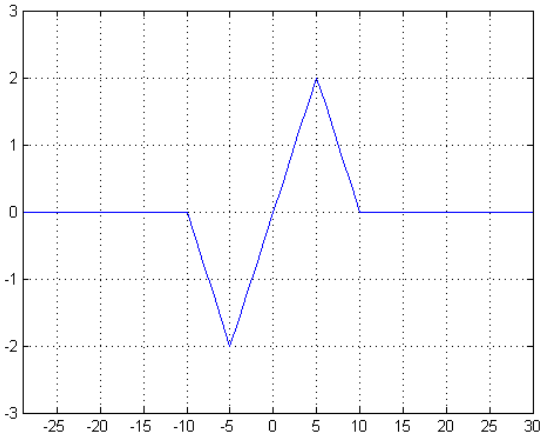


Figure 2.5-1: Signal A

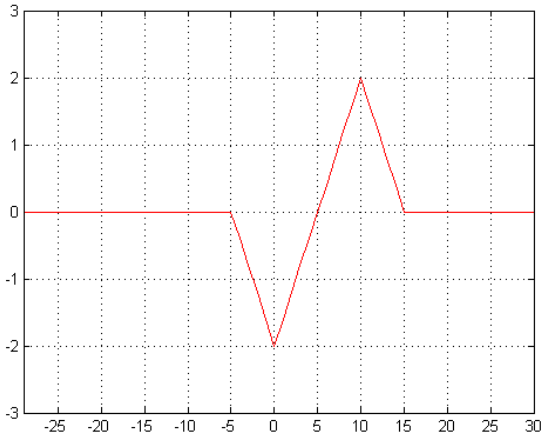


Figure 2.5-2: Signal B

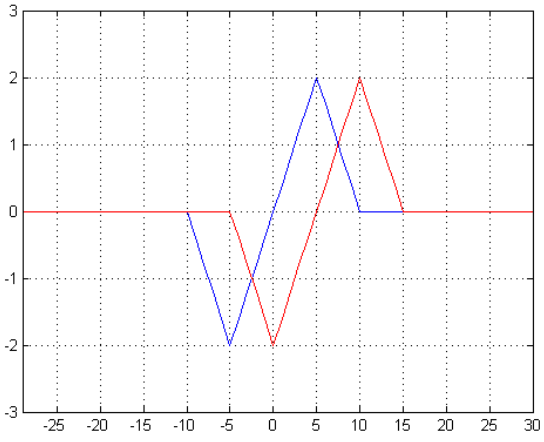


Figure 2.5-3: Signal A and B

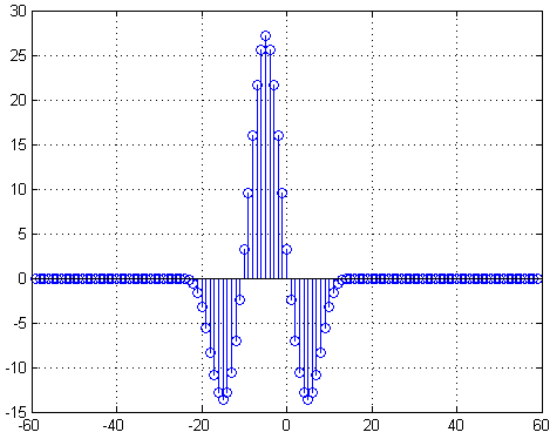


Figure 2.5-4: Cross correlation of signal A and B

If both signals are complex the cross-correlation is done on the conjugate of one of the two signals. This because of the imaginary unit squared is negative and will result in a negative value if two high imaginary components with the same sign are multiplied. With the conjugate of one signal the product of each imaginary pair is negated resulting in a double negation just like with the negative peaks.

The definition of cross-correlation between two signal *f* and *g* either it be continuously, Equation 2.5-1, or discrete, Equation 2.5-2, is based upon the integration, or summation, of the product in each sample for each shift of the signal *g*.

$$(f \star g)(t) = \int_{-\infty}^{\infty} f^*(\tau) g(t+\tau) d\tau$$

Equation 2.5-1: Continuous cross-correlation

$$(f \star g)[n] = \sum_{m=-\infty}^{\infty} f^*[m] g[n+m]$$

Equation 2.5-2: Discrete cross-correlation

To expand the cross-correlation to support two dimensional signals and filters the definition is expanded to integrate over two dimensions, or rather integrate each row on the integrations of the columns or vice versa. The complexity of the process stays the same and the number of operations is the same order per sample, O(*N*), as for a one dimensional cross-correlation.

In some cases there is a need to normalize the signal, most common among images which are classic examples of two dimensional signals, while doing the cross correlation. Normalized cross correlation, see Equation 2.5-3, can in many cases give an improved result if for example brightness vary in the two images.

$$(f \star g)[n] = \frac{1}{n-1} \sum_{m=-\infty}^{\infty} \frac{(f^*[m] - \bar{f})(g[n+m] - \overline{g)}}{\sigma_f \sigma_g}$$

Equation 2.5-3: Normalized Cross-correlation

Convolution is closely related to correlation, with the difference that convolution shifts backwards, Equation 2.5-4, compared to cross-correlation as in Equation 2.5-1.

$$(f * g)(t) = \int_{-\infty}^{\infty} f^*(\tau) g(t - \tau) d\tau$$

Equation 2.5-4: Definition of convolution

The advantage with convolution compared to correlation is associativity, Equation 2.5-5, which means that the order of multiple convolutions does not matter compared to multiple correlations for which it does. With this property instead of having several filter applied to one signal after each other they can be applied on each other and become one filter saving up processing time during runtime.

$$F \cdot (G \cdot I) = (F \cdot G) \cdot I$$

Equation 2.5-5: Associativity

Convolution can as correlation also be used for a two dimensional signal, or image. The difference, just like for a one dimensional convolution, is the shifting is done backwards but in this case in both dimensions. The two dimensional convolution has, as the one dimensional, the advantage of associativity.

2.5.2 Phase Correlation

Cross correlation can be performed in the frequency domain and is then called phase correlation as it correlates the phase between the two signals. When calculating the correlation in the frequency domain the computation required is heavily decreased as only a point wise multiplication is required, seen in Equation 2.5-6, compared to a summation of multiplications for each point.

$$\mathcal{F}\{f \star g\} = (\mathcal{F}\{f\})^* \odot \mathcal{F}\{g\}$$

Equation 2.5-6: Cross-correlation in the frequency domain

An inverse Fourier transform of the result in the frequency domain, Equation 2.5-7, gives the result of the cross correlation just as if it was calculated in the time, or spatial, domain.

$$f \star g = \mathcal{F}^{-1}\{(\mathcal{F}\{f\})^* \odot \mathcal{F}\{g\}\}$$

Equation 2.5-7: Cross-correlation through the frequency domain

Phase correlation is a correlation method based upon the Fourier shift theorem. The shift theorem states that a circular shift in the time domain will result in a linear phase factor in the frequency domain, mathematically described in Equation 2.5-8. In image processing this is used to determine the offset between two images, which can be seen as a circular shift, by calculating the linear phase factor. This becomes very advantageous as the calculation in the frequency domain can be done much faster than the estimation in the time domain.

$$\mathcal{F}(\{x_{n-m}\}) = X_k \cdot e^{-\frac{2\pi}{N}km}$$

Equation 2.5-8: A shift represented as a linear phase in the frequency domain

The calculation of the linear phase factor is the same as calculating the cross-power spectrum;

$$R = \frac{G_a G_b^*}{|G_a G_b|}$$

Equation 2.5-9: Cross-power spectrum

In the frequency domain the first image is multiplied, element-wise, with the complex conjugant of the second image. By normalizing the product element-wise the normalized cross correlation is obtained by going back to the time domain through an inverse Fourier transform, see Equation 2.5-10.

$$r = \mathcal{F}^{-1}\{R\}$$

Equation 2.5-10: Inverse Fourier transform

Here follows an example of a phase correlation between two images where the second is shifted relative the first. Figure 2.5-5 is in this case the reference image whilst Figure 2.5-6 is shifted 50 pixels both in x and y. White Gaussian noise with mean 0.1 and variance 0.02 has been added to both images to show how phase correlation performs when affected by noise.

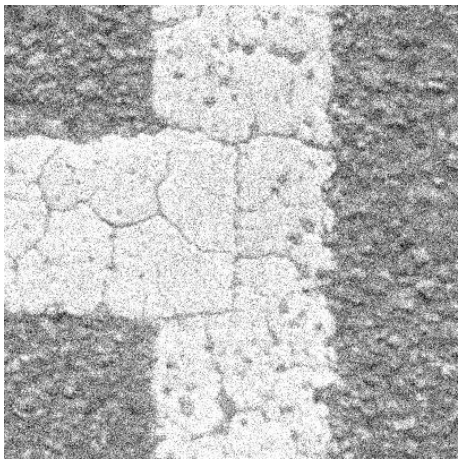


Figure 2.5-5: Parking marker without shift

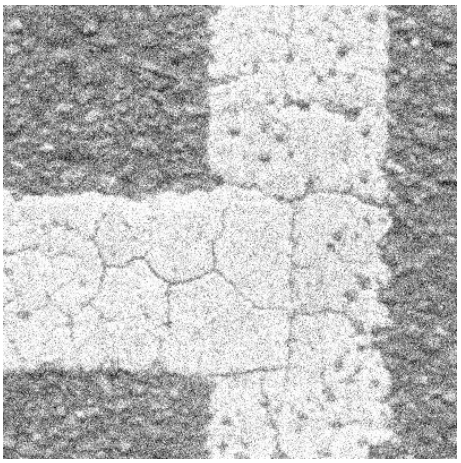


Figure 2.5-6: Parking marker with x and y shifted 50 pixels

The result from the phase correlation can be seen in Figure 2.5-7 with a distinct peak in (51, 51). The expected result was a peak in (50, 50) which actually equals (51, 51) in this case because Matlab®, which was used for testing, starts its indexing in (1, 1) instead of in (0, 0).

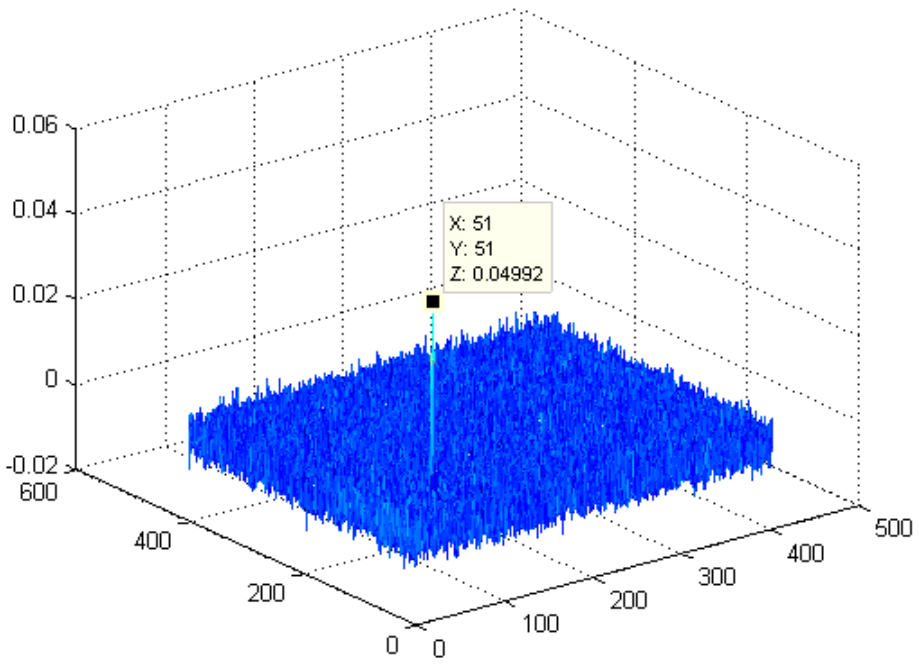


Figure 2.5-7: Phase correlation result

2.5.3 Sub-pixel Detection

The two previous sections have shown how the offset between two images that overlaps can be calculated. The two methods are however limited to determine the offset on pixel size level. In order to get even better precision there are methods that can be applied to the proposed methods, which then can give sub-pixel precision. The sub-pixel offset can be calculated either directly in the frequency domain [15] or in the spatial domain [16].

The method used in the spatial domain is based on the fact that there is not a distinct peak in the correlation result. The actual peak is hidden between the peaks that are shown in the result. The information in adjacent peaks is therefore possible to use to get an estimation of where the actual peak is. This phenomenon is illustrated in Figure 2.5-8, Figure 2.5-9 and Figure 2.5-10. The figures illustrate the result from a phase correlation between two images where one image has an offset of 2.8 in the x-axis and 4.0 in the y-axis.

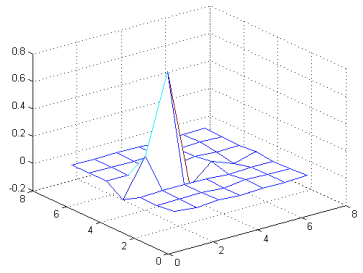


Figure 2.5-8: Sub-pixel 3D

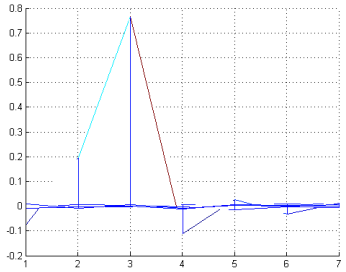


Figure 2.5-9: Sub-pixel x-z

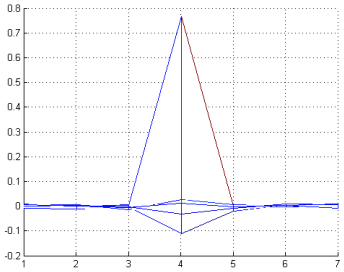


Figure 2.5-10: Sub-pixel y-z

The spatial domain based algorithm used in [16] is described as follows;

- Find the main peak from the phase correlation, from Equation 2.5-10, in the spatial domain (x_{peak}, y_{peak}) and the two side peaks $(x_s = x_{peak} \pm 1,\ y_s = y_{peak} \pm 1)$.
- Use the peak and the side peaks to calculate the Δx sub-pixel shift according to Equation 2.5-11. If $x_{peak} + 1$ is out of the image range, then use $x_{peak} - 1$ instead.

$$\Delta x = \frac{r(x_{peak} + 1, y_{peak})}{r(x_{peak}, y_{peak}) \pm r(x_{peak} + 1, y_{peak})}$$

Equation 2.5-11: x sub-pixel shift

- Δy is calculated according to Equation 2.5-12. If $y_{peak} + 1$ is out of the image range, then use $y_{peak} - 1$ instead.

$$\Delta y = \frac{r(x_{peak}, y_{peak} + 1)}{r(x_{peak}, y_{peak}) \pm r(x_{peak}, y_{peak} + 1)}$$

Equation 2.5-12: y sub-pixel shift

- The equations will give two solutions for both Δx and Δy. The correct solution is in the interval [0, 1] and has the same sign as $x_s - x_{peak}$ and $y_s - y_{peak}$ respectively.

The results from this algorithm if applied to the values in Figure 2.5-8 are presented in Table 2.5-1.

	Actual offset	**Calculated offset**
Δx	2.8000	2.8316
Δy	4.0000	4.0003

Table 2.5-1: Sub-pixel algorithm results

2.6 Multicore Techniques

This section gives and overview of multicore techniques. It starts with architecture classes used in multicore systems and how instructions and data are divided on multiple processing cores. This is followed by how memory strategies and how data are accessed from multiple cores without interfering each other. The last part is about interconnection networks, networks used to connect cores to each other. This to allow data exchange in a multicore system.

2.6.1 Architectures

A processing unit requires two streams of information to operate; instruction and data stream. Every instruction is executed once using data delivered by the data stream. In 1966 Michael Flynn classified four different computer architecture classes which states how these streams can be connected either using a single or multiple processing units. This is called Flynn's taxonomy and is presented in Table 2.6-1.

	Single instruction	**Multiple instruction**
Single data	SISD	MISD
Multiple data	SIMD	MIMD

Table 2.6-1: Flynn's taxonomy

The architecture classes combine single and multiple streams of instructions and data.

Single Instruction-Single Data Stream

This is a non-parallel architecture using only one processing unit, commonly referred to as a single core architecture shown in Figure 2.6-1.

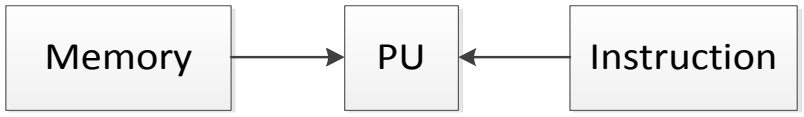


Figure 2.6-1: SISD architecture

Single Instruction-Multiple Data Stream

The architecture uses one control unit transmitting same instruction to all processing units. Every processing unit executes the received instruction on their own data stream. The design is shown in Figure 2.6-2. This kind of architecture is suitable when same instructions shall be executed on a great amount of data. Example application is image processing.

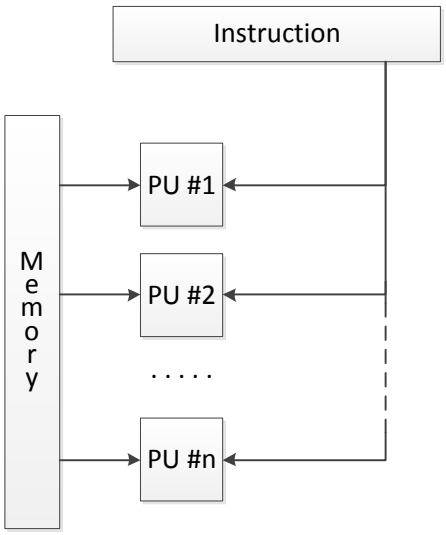


Figure 2.6-2: SIMD architecture

Multiple Instruction-Single Data Stream

This is an architecture where one data stream passes multiple processing units executing different instructions, shown in Figure 2.6-3. It can be viewed as a pipeline where the output stream from one processing unit is connected as input stream to the next one. This kind of architecture is rarely used; sometimes pipelined machines and systolic-array computers are mentioned as examples of this architecture.

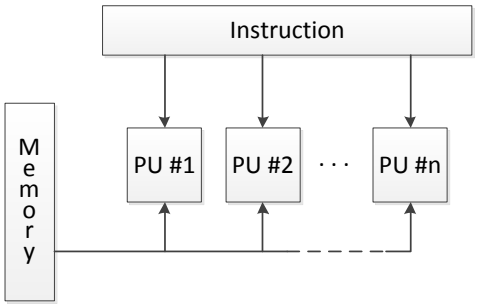


Figure 2.6-3: MISD architecture

Multiple Instruction-Multiple Data Stream

In a MIMD architecture every processing unit has their own control unit and data stream as shown in Figure 2.6-4. The architecture is divided into two categories according to how each processing unit access data in memory; shared memory or message passing. The categories are named by the way the processing units access memory and share data.

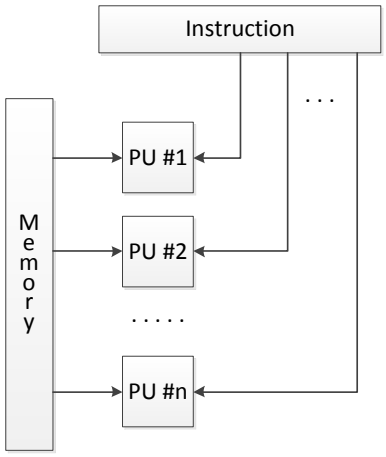


Figure 2.6-4: MIMD architecture

2.6.2 Data Access Methods

In a shared memory structure all processing units share a global memory. If not a multiport memory is used, the memory and the processing units are interconnected by either a bus or crossbar. With a global memory space it may be preferable to restrict memory areas to only local or adjacent processing units. By using an access control model memory accesses can be verified and can depending on the result either be allowed or denied.

A shared memory structure can have a single physical memory accessible through an interconnection network, called uniform memory access (UMA). The benefit with UMA is that all processing units have equal access time to memory. A shared memory structure may also have physically distributed memory among all processing units, called non-uniform memory access (NUMA), but still with a single address space. Contrary to UMA this design makes the access time dependent on the distance to the processor which the accessed memory belongs to. Cache-only memory architecture (COMA) uses cache-memories attached to each processor. But using cache-memories requires the data to be cached to the requesting processing unit before it can be used.

A message passing system distributes the memory to all processing units. Every unit got its own local memory to store data. To exchange data between processing units data has to be moved from local memory on one processing unit to the local memory on another processing unit. To make this possible all processors have to be interconnected to each other and a message passing protocol is needed. This category of MIMD is preferable when scaling to larger systems.

To utilize the advantages of both MIMD architectures distributed shared memory was introduced. This architecture distributes the memory as in a message passing system but the programming model abstract the physical hardware design and behave as a shared memory system.

2.6.3 Interconnection Networks

There exist many different solutions to interconnect processing units in a multicore architecture. The networks can either operate in synchronous or asynchronous mode. The control system responsible to handle access to the network and data passing between the processing units connected to the network is either designed in a centralized or a decentralized manner. Data is transmitted using circuit or packet switching mechanisms. Circuit networks establish a complete path to the destination when the transmission is initialized while packet switching networks transmits packets. Each packet is switched at each node on its way to the final destination. Networks may also differ in topology. They can either be static with fixed links between nodes or dynamic where switching elements creates temporary connections between nodes when requested.

Bus-based Networks

A simple way to connect multiple processing units to a global memory is to use a bus. With this design, every processing unit often has a local cache-memory to reduce the traffic on the bus. The limitation in this design is the bandwidth because only one processor can access the memory at a time.

One way to reduce this limitation is to extend the network with multiple buses. Multiple bus network can differ in design;

- All processing units and memories are connected to all buses. This design makes it easy for a processing unit to access all memories independently on which bus that is free to use.
- The memories may be distributed among the available buses because they do not support multiple connections. This requires a processing unit to wait until a specific bus is available to access the desired memory area.
- A combination of the other two designs where every memory module is be connected to a part of the available buses.
- These three designs can be combined to priorities important memories by connect them to all buses while less important memories only are connected to one or some of the buses.

The fourth design is visualized in Figure 2.6-5.

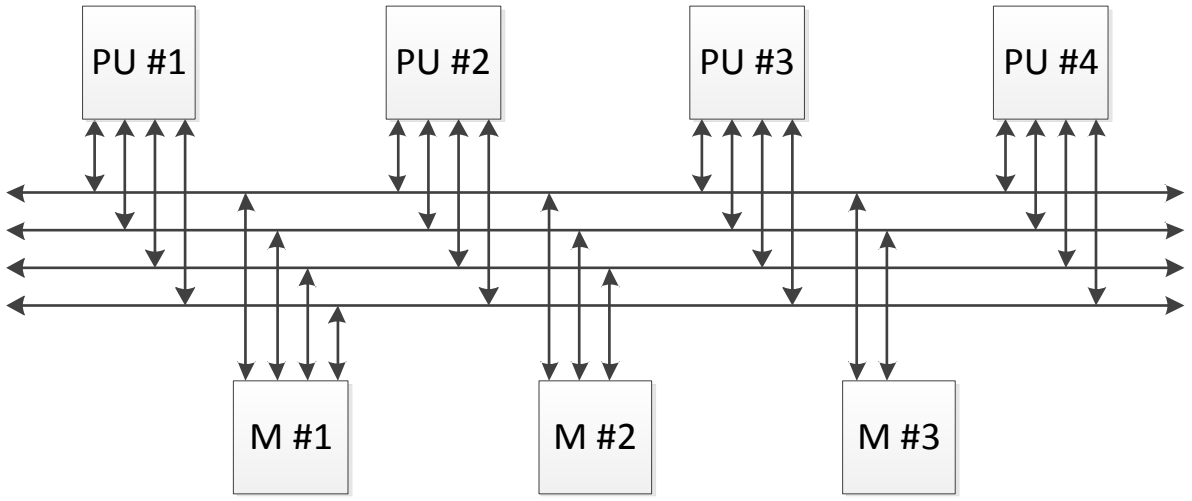


Figure 2.6-5: Multiple bus with class-based memory connection.

Except improving the bandwidth compared to a single shared bus, a multiple bus network is not as vulnerable as single shared bus if one bus is disconnected. This makes a multiple bus network more reliable.

Switched-based Networks

Three kinds of switching networks exists; crossbar, single-stage and multi-stage.

A crossbar network is built up by switching elements connecting rows and columns. Each processing unit in the system is connected to one row. Every column is connected to one memory module. Every switching element can either extend the signal on the current row or change direction of the signal to a column. Figure 2.6-6 visualizes a crossbar network. On the first row the signal from the processing unit (PU #1) is extended by the first switch and the second switch changes the direction of the signal towards the memory module (M #2). This design allows the network to provide simultaneous connections between all its processing units and memories. However, this design also requires many switching elements compared to the number of

processing units and memories. A system containing a high number of processing units gets a complex hardware design.

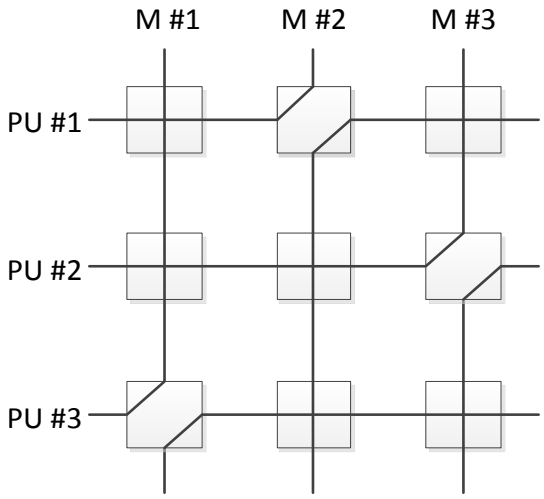


Figure 2.6-6: A 3x3 crossbar network interconnection processing units with memory units.

Single-stage and multistage networks are built up by switching elements with two inputs and two outputs. Figure 2.6-7 contains two switching elements where the left one forwards both inputs to their respective output. The right one exchanges the signals and connects the upper input to the lower output. As the name indicates, Single-stage only has one stage with switches. This makes it impossible for one node to access all the other nodes if there is more than one switch in the network. Data has to be shuffled through other nodes to reach the destination.

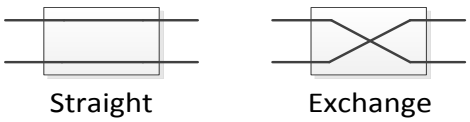


Figure 2.6-7: 2x2 switching elements visualizing straight and exchange modes.

The problem with Single-stage networks, with limited addressability, can be solved by adding additional stages of switches which creates a Multi-stage network. This design allows multiple connections but an established connection can still block simultaneous connection requests. There are solutions to rearrange currently established connections if they block a requested connection. The different switch stages can be connected in different patterns. One example is the Banyan network.

To route messages through a Banyan network the destination address is used. Each bit determines how the switch on each stage should be configured to reach correct destination. A banyan network with a route between node 001 on the left side and node 101 on the right side is shown in Figure 2.6-8. As the figure visualize, the destination address 101 determines the first and third switch to use the lower output while the second switch use the upper one. The benefits with this network is that it requires less switching elements than a crossbar and the addressing technique is simple but on the other hand it cannot establish a connection for each processing unit simultaneously.

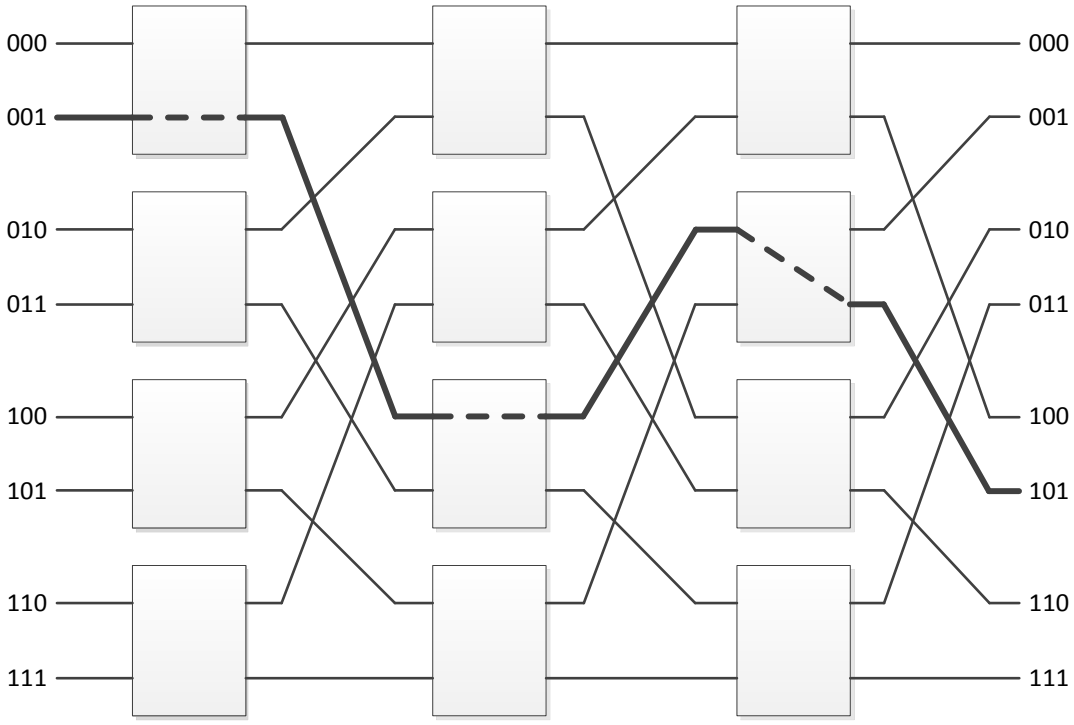


Figure 2.6-8: 8x8 Banyan network is a common multi-stage interconnection network.

Static Networks

A static network does only have fixed paths. The paths connect processing units to each other and can either be unidirectional or bidirectional.

A completely connected network connects every node to all the other nodes. This guarantees fast delivery of data because only one link has to be passed. But this advantage is at the expense of the complexity, the large amount of links results in when the number of processing units are increased. An illustration of this network is presented in Figure 2.6-9.

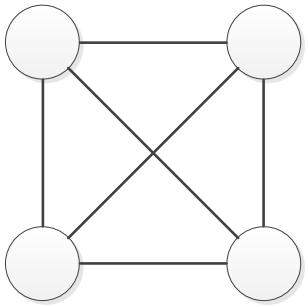


Figure 2.6-9: Completely connected network with 4 nodes.

Limited connection network is an alternative choice to get a less complex design. But this network does not connect every node to each other. Instead, data transmission between some nodes has to be routed through other nodes to reach its destination. Two issues have to be solved because of this; interconnection pattern and routing mechanisms.

The most common interconnection patterns are; linear array, ring, tree, mesh and cube networks. All these networks have their own advantages and disadvantages. Linear array, Figure 2.6-10 (a), has low complexity but tends to be rather slow when data must be routed through many nodes. The maximum number of nodes to traverse is reduced if the nodes at the edges are linked; this action creates a ring network, shown in Figure 2.6-10 (b). The tree network, Figure 2.6-10 (c), reduces the transmission time but it results in a more complex design.

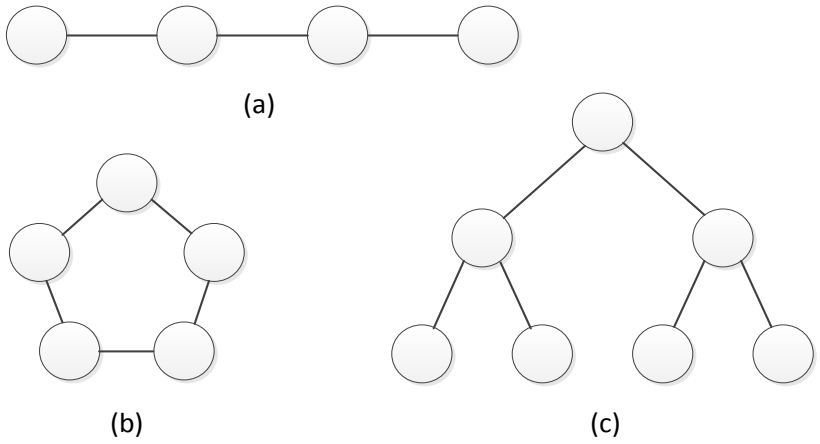


Figure 2.6-10: Interconnection patterns in Limited connection networks; linear array (a), ring (b) and tree (c).

A mesh-connected network is built up by *n* dimensions where every dimension consists of a number of nodes. In a mesh with *n* dimensions every node connects to *n* other nodes. One method to transmit data is to route the data in one dimension at a time and change direction when the data has arrived at the correct coordinate in every dimension. This results in only two turns when data is transmitted from one node to another. The longest path in a mesh is $\sqrt{N}$ where N equals the number of nodes in the mesh. Figure 2.6-11 illustrates a mesh network and shows a routing path between two nodes using the method described above.

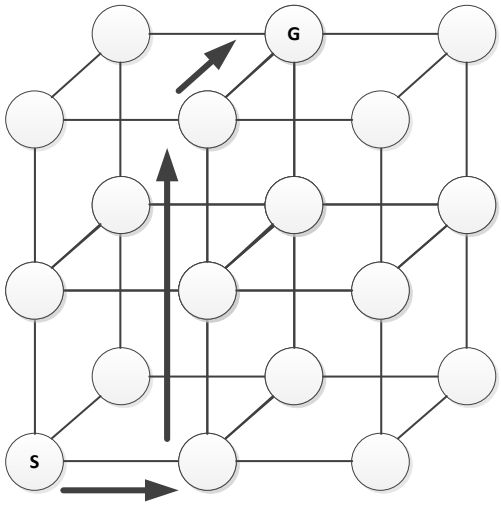


Figure 2.6-11: A 3x3x2 mesh network including a routing path between two nodes.

The cube network can also be designed in *n* dimensions. An *n*-cube consists of 2^n nodes where every node has *n* connections to other nodes. Except a low number of links this design also results in short paths to transmit data to any node. Only *n* links has to be traversed to reach any connected node. A 3-cube is shown in Figure 2.6-12.

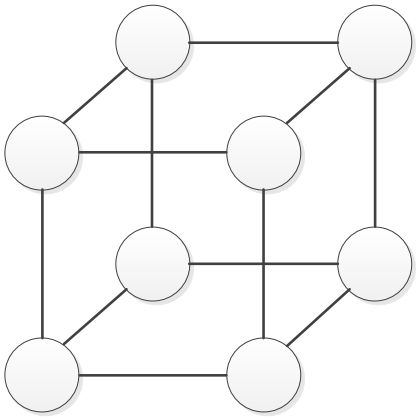


Figure 2.6-12: A 3-dimensional cube network.

2.7 Multicore Architectures

This section goes through two different multicore architectures using technologies described in section 2.6. The first system, the Ambric architecture, is developed by Ambric and the second architecture, Epiphany, is developed by Adapteva. Both systems are MIMD solutions but still differ much in hardware design and in software development.

2.7.1 Ambric

The Ambric architecture [17] is built up by blocks named brics. Every bric consist of two computation units (CU) and two RAM units (RU). One pair of a CU interconnected to a RU is visualized in Figure 2.7-1.

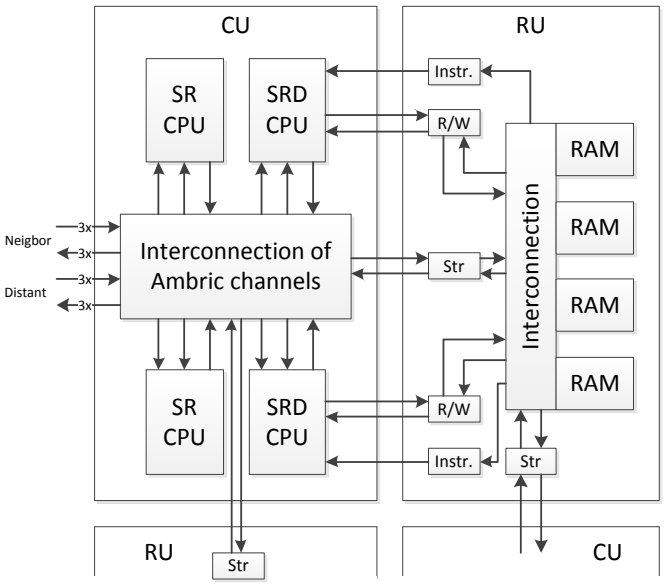


Figure 2.7-1: Two pairs of CU and RU units creates a bric.

The AM2045 chip is built up by 45 brics which result in totally 360 CPUs. A language called aJava, which is a strict subset of Java, has been developed to be used with this chip.

Bric

Each bric consist of two pairs of a CU interconnected with a RU. This gives 8 central processing units (CPU) and totally 13 kilobytes of RAM memory in every bric. The brics are interconnected to their neighbors to form an array of cores. To communicate between the cores channels are used. There are dedicated channels to the bric's neighbors and to reach distant brics. The distant interconnection network is designed as a 2D circuit-switched network with four channels in each direction. Every hop is one bric long. There are also four channels in each direction to interconnect the switch to the compute units in every bric.

Compute Unit

A CU consists of four CPUs. Every CPU has one crossbar to direct data to their input channels. Connections between CPUs within the CU are configured dynamically by instruction fields. The output channel of each CPU is connected to a statically configured crossbar. This crossbar links the CPUs output channels and the CU's input channels to the output channels of the CU. The CU has input and output channels which can be configured to interconnect with neighboring CUs, to the distant channel switch and to adjacent RU.

RAM Unit

The RU has four memory banks containing 1 kilobyte each. Every RU has six engines which can be configured to stream data over channels to the SRD CPUs in the CU. The RU engines can dynamically connect a channel to the requested memory region.

Central Computing Unit

There are two kinds of CPUs named SRD and SR in the CUs. Both are 32-bit RISC CPUs with slightly different design. The main difference is that the SRD has DSP extensions to process complex and math-extensive objects. The SRD has two ALUs in serial and a third one in parallel. It can either process 32 bit, dual 16 bit or quad 8 bit operations. There is a local 256 word memory available for instructions and data and more RAM is accessible from the RAM unit which will be discussed later. To communicate with other nodes in the architecture, it has two

input channels and one output channel. The SRD has also one read and one write channel connected to the adjacent RU together with an instruction channel from the RU.

The SR is simpler with only one ALU mainly used to process small and fast tasks. It can process both 32 bit and dual 16 bit operations. As the SRD it has 8 registers but only 64 words of local memory. The CPU also has two input channels and one output channel but it has no channels towards any RAM unit. Block schematics of both CPU types are shown in Figure 2.7-2.

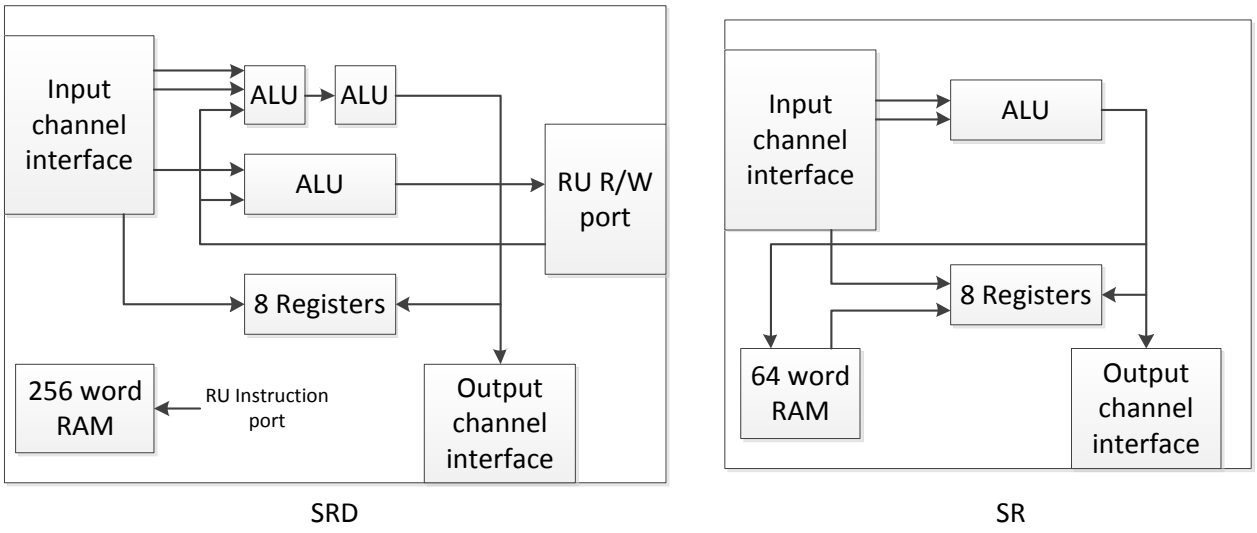


Figure 2.7-2: SRD and SR are the CPUs used in the Ambric architecture.

2.7.2 Epiphany

The Epiphany architecture [18] is a MIMD design where the cores are interconnected by a 2D mesh network. These design choices simplify the scalability. The architecture is scalable up to 4096 cores due to addressing limitations. To develop software to this system the common C language is used.

Epiphany Core

The core in the Epiphany architecture is built up by a central processing unit and a router for the interconnection network as in Figure 2.7-3. The CPU has both one arithmetic logic unit (ALU) and one Floating-point Unit (FPU). This design makes it possible to execute both an integer and a floating-point operation per cycle. Load or store a floating-point is done by an integer operation. Therefore, while the CPU executes a floating-point calculation it can also prepare a new floating-point calculation by loading data. The ALU and FPU have access to a 64-entry register file where every 32-bit entry can hold either an integer or a single-precision floating-point value.

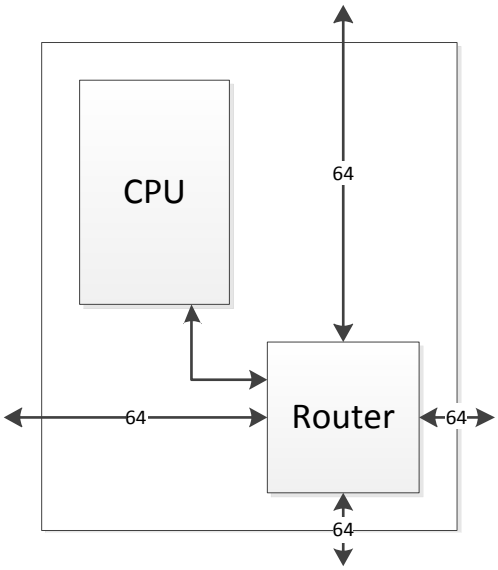


Figure 2.7-3: Epiphany core with CPU and router.

A detailed view of the CPU is shown in Figure 2.7-4. Its register file has a 64-bit path to the SRAM. Every core has a SRAM that can store 32 kilobytes of data. The memory is divided into four banks. Each bank may be accessed individually by different operations in each cycle. An ideally example is when an instruction fetch, a data load, a DMA access and an external core access is treated in one cycle. The choice of memory can be derived to the desire to develop an energy efficient architecture. Normally cache memories are preferable because it simplifies the software but SRAM has benefits in terms of power consumption. This choice does however put the memory management load of the SRAM to the software. The local memory in each core shares a common address space which makes it possible for a core to do read and write operations to any core's memory.

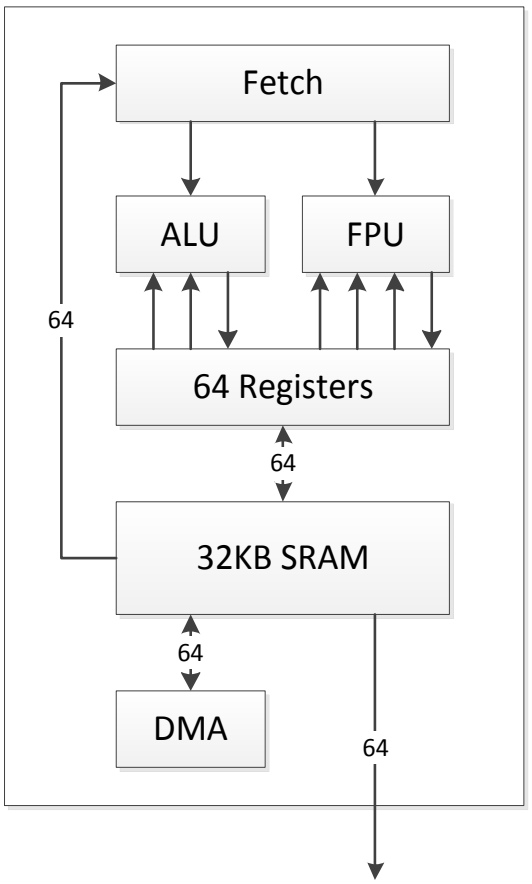


Figure 2.7-4: Block view of Epiphany CPU.

2D Mesh Network

The choice of interconnection network is motivated by the desire to get a scalable and a low power consuming architecture. By using a mesh network the signal paths between cores are kept short. The benefits of a mesh network are that the drive current is kept low and the clock speed is the same as for the cores. The router included in the core is connected to the four adjacent cores and to the core's CPU. In every cycle, every core can transfer 64 bits of data to any of its neighbors or between its CPU and router. For every core the data has to pass to reach its destination the transmission time increases with one cycle.

2.8 Camera Settings

There are many different camera parameters that are important when choosing or configuring a camera for a specific task. This section will give an introduction to the parameters, and environmental needs, that are considered important for this project. The parameters will be introduced in the following order; shutter type, exposure time and aperture.

2.8.1 Shutter Type

The shutter type determines the method used to procure the image frame. There are two types of shutters for digital image sensors, global shutter and rolling shutter. Global shutter is when the full frame is captured at one time, i.e. all pixels are captured at the same time. If a rolling shutter is used the full frame is not captured at one time, the frame is instead captured line by line with a short capture delay between the lines as a result. This shutter technique can lead to some interesting motion effects when photographing an object in motion.

Figure 2.8-1 and Figure 2.8-2 show images of the same spinning fan captured with both global and rolling shutter. Figure 2.8-1 which is captured with global shutter shows no track of distortions. Figure 2.8-2 however shows a fan whose blades look distorted which is a result of the rolling shutter used in this example. The frame is captured line-wise vertically starting from the top. By knowing that the fan is spinning counter clock-wise it is possible to explain why the image looks like it does. The left part of the fan is moving in the same direction as the rolling shutter which makes it look like the fan blade is bigger than it actually is. This happens because the same blade is a part of several consecutive captures since it moves along the direction of the rolling shutter.

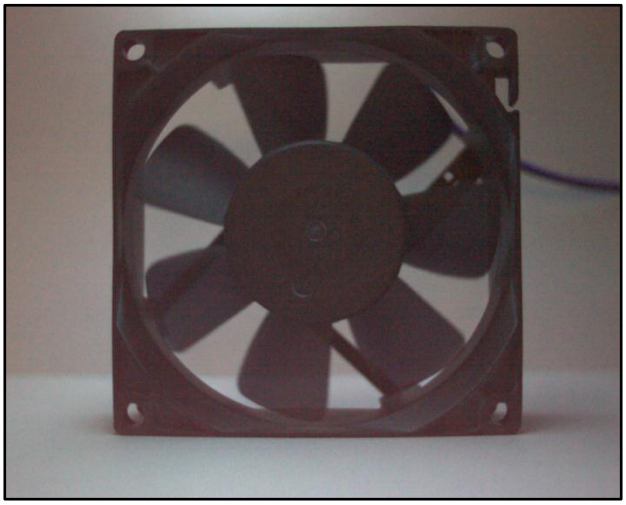


Figure 2.8-1: Global shutter

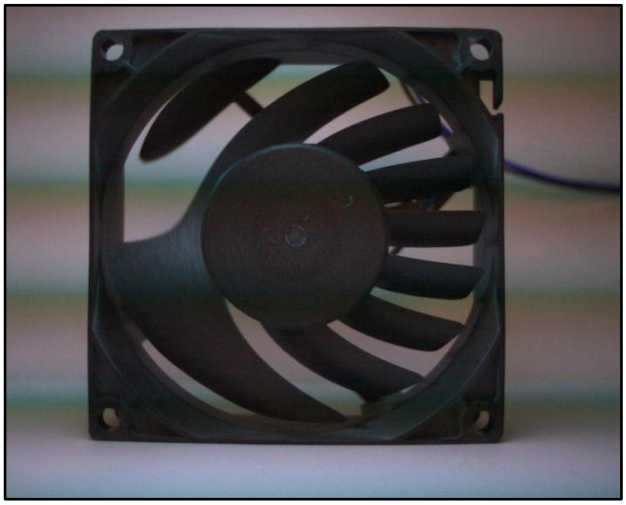


Figure 2.8-2: Rolling shutter

To avoid the rolling shutter phenomenon in the following examples global shutter will be used.

2.8.2 Exposure Time

Exposure time corresponds to the time the image sensor is exposed to light. The longer the exposure time, the more light hits the sensor. There are however problems that emerges if the exposure time is too long, depending on the application. One problem is overexposure which makes bright parts of an image look all white. A long exposure time can also result in motion blur that emerges when capturing an image of a moving object. On the contrary if the exposure time is too short, also application specific, called underexposure makes dark parts of an image look all black. The image also tends to look noisy if not enough light is captured during the exposure time.

Figure 2.8-3 and Figure 2.8-4 are images of the same spinning fan as in 2.8.1 but with the exposure time altered instead of the shutter type. Figure 2.8-3, which is captured with an exposure time of 100µs, shows no sign of fan movement, but the image looks a bit noisy and dark compared to Figure 2.8-4. This can however be solved with better background lighting. Figure 2.8-4, which is captured with an exposure time of 1millisecond, shows the motion blur effect.

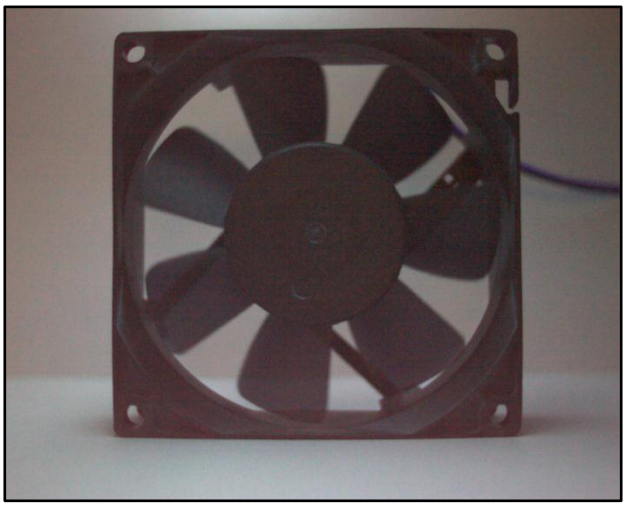


Figure 2.8-3: 100 µs exposure time

Figure 2.8-4: 1 millisecond exposure time

2.8.3 Aperture

Aperture size is the size of the mechanical opening that let the light in to the image sensor. A large aperture results in more light and a small aperture results in less light. The aperture can be compared to the human's pupil which adapts to the level of present light. If you enter a dark room the size of the pupil will increase in order to capture more light and vice versa if the room is bright. Not only is the amount of light that is received affected by the aperture but also the focal length. A small aperture results in a longer focal length compared to a large aperture which results in a more limited focal length. The trade-off for more light is basically a shorter focal length.

An example of two different aperture settings is shown in Figure 2.8-5 and Figure 2.8-6. Both images have the same focus setting with the focal point in the right part of the images. The difference between the aperture settings can easily be distinguished by looking at the left part of the images. In Figure 2.8-6 where the aperture is large the left part is all blurry while the text is still somewhat readable in Figure 2.8-5.

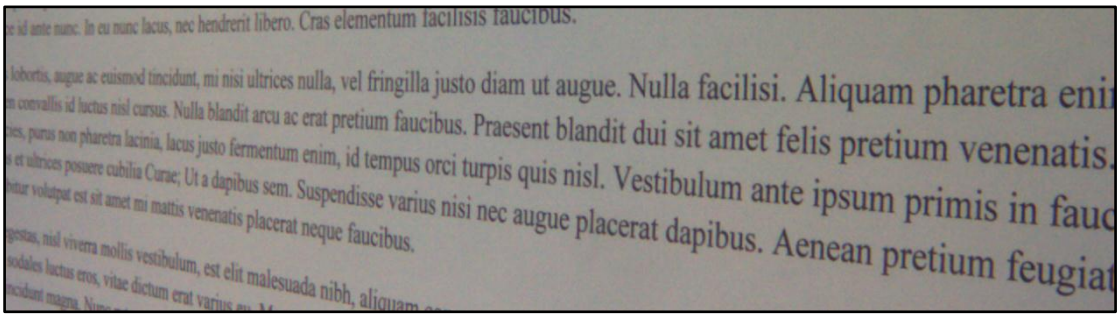


Figure 2.8-5: Small aperture

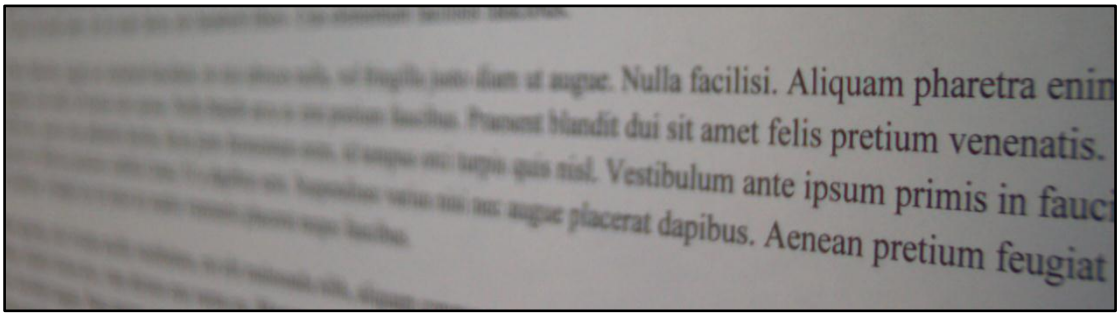


Figure 2.8-6: Large aperture

3 Solution

This chapter presents the solutions that have been chosen for this project. This includes the chosen hardware; camera and parallel architecture, algorithm to process the generated data and the design of the parallel implementation.

3.1 Overview

The main idea in this project is to use a camera to capture images of the surface below a vehicle. If the images are captured frequently enough they will overlap each other. By analyzing two consecutive images it should be possible to find some similarity in the images and by that calculate the offset. The offset is a result of the distance the vehicle has traveled in the time elapsed between the two consecutive images. Using a multicore solution to do the image analysis makes it possible to process images to measure the speed of a vehicle in real-time. In addition to calculate the speed it would also be possible to track a relative position of the vehicle. The complete system is visualized in Figure 3.1-1.

Figure 3.1-1: Overview of the system.

The images are loaded into the multicore architecture where the image analysis is done. The chosen algorithm is Phase correlation with an addition in form of a filter as pre-processing. Calculations included in the algorithm are; fast Fourier transform, matrix operations and inverse fast Fourier transform. All of them are executed in the multicore system.

The interpreted result from this algorithm shall then be forwarded to a control system that is using the measured values to control any other systems in the vehicle. What kind of interface needed to do this has to be investigated.

3.2 Camera

Quantities such as speed and acceleration set requirements the camera has to fulfill. How large area of the last image that has to be available in the next image, to find the offset between them, sets the requirement of the time period between every image from the camera. The image period can be calculated using Equation 3.2-1. The length *L*, visible distance on the ground, depends both on the distance from the camera to the ground and the kind of optics attached to the camera. How much of the image that has to overlap the next image in the sequence can depend on parameters such as the surface and light conditions. This is set by *x* in percent of the image length in the equation. The maximum speed the sensor should manage is v_{max}.

$$T = \frac{L * (1 - x)}{v_{max}}$$

Equation 3.2-1: Image period T, L – true length of image (m), v_{max} – speed (m/s), x – image overlap.

When analyzing two images it is important to have details in the images so they can be matched with a good result. As described in 0 a low exposure time is preferred in this application to avoid the motion blur effects when traveling in high speeds. But this also results in a limited amount of light reaching the camera sensor.

To get around this problem a large aperture could be set in the camera. This will in turn reduce the focal length as described in 2.8. However, this application only photographs an area on a certain distance from the camera and it is therefore not necessary to have a wide focal length. Probably it is not enough to set preferable settings in the camera to manage all situations this sensor may be exposed to. To make sure enough light is provided in all situations an external light source should be used. What kind of light source and how powerful it should be is not in the scope of this project.

To get an image that is as clear as possible global shutter has to be used. Else the image would be distorted as described in 2.8.1 because the photographed area is moving relatively the camera while exposing the CMOS sensor to light.

This project has used a camera made by IDS, model UI-1204SE, with USB connection. This camera satisfied the project's requirements and was available at the time of this project. Therefore no thorough analysis has been done if this camera is the best suited for this application.

The sensor will always be in the same position relative to the vehicle's driving direction. The movement of a vehicle is basically only in the driving direction and the lateral movements are comparatively very small. The image width may therefore be much smaller relative to the length of the image in the driving direction.

The camera has a specified frame rate of only 25 frames per second. This is unfortunately too slow if the system shall be able to measure the speed of a car. But during experiments with the camera higher frame rates could be achieved by reducing the width of the image. This made the camera to exclude columns on the CMOS sensor and the frame rate could be increased to 200 frames per second using the resolution 1280x80.

3.3 Proposed Algorithm

The image analysis block from Figure 3.1-1 is displayed in greater detail in Figure 3.3-1. The different steps are distinguished in the flow chart and are explained more thoroughly in the following sections. Two example images will be used to show every step of the algorithm. One thing to note is that it is not necessary to calculate two FFT's for every pair of consecutive frames, as illustrated in the figure. Instead it is possible to store the previous image's frequency representation, *n-1:th image*, which is needed twice, both in the previous and the current phase calculation.

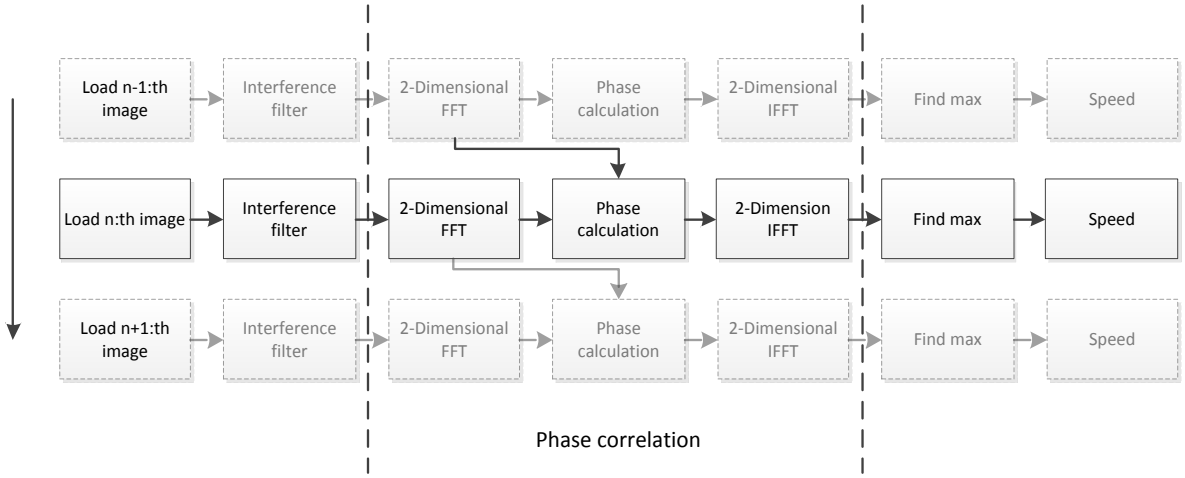


Figure 3.3-1: Image analysis overview

The algorithm starts with two consecutive images which in this example have been captured with a moving camera pointing towards the ground with a frame rate of 200 frames per seconds. The images are shown in Figure 3.3-2 where a small offset in the horizontal direction can be seen.

Figure 3.3-2: Two consecutive images with a horizontal offset

3.3.1 Filter

In order to reduce background interference, in form of noise, as much as possible some pre-processing was found out to be needed. A filter was decided to be the most convenient pre-processing method. In order to choose the best filter several commonly used filters were evaluated using a video test sequence. The video was recorded by a camera filming the ground from a moving vehicle where the actual distance the vehicle moved during the test sequence was used as reference when evaluating the filters. The filters were applied to the images in the video before the phase correlation algorithm was applied. The performance of each filter was then evaluated by comparing the resulting total offset from the first to the last frame in the video sequence. The filters that were evaluated and the results are presented in Table 3.3-1. All filters are of size 3 by 3 except the last average filter that is of size 4 by 4.

Filter type	Distance (m)
Reference	80.0
Sobel	79.55
Prewitt	79.47
3x3 average	79.43
4x4 average	79.42
No filter	78.48

Table 3.3-1: Filter result

As the table shows a Sobel filter turned out to be the best solution. The Sobel filter gives good noise reduction as well as a distinct enhancement of details in each image. These two attributes combined resulted in very good image offset estimation when used in combination with the phase correlation algorithm.

The two images from Figure 3.3-2 are displayed in Figure 3.3-3 after a Sobel filter has been applied. As can be seen in the image the horizontal edges have been highlighted which is an expected result from a Sobel filter.

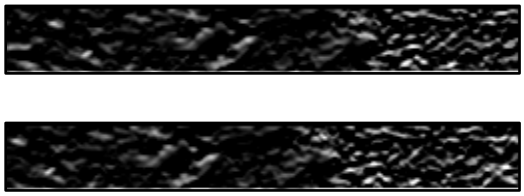


Figure 3.3-3: Two consecutive images after filtering

3.3.2 Phase Correlation

Phase correlation was found out to be the most efficient way to calculate the offset between two images, both due to its relatively low computational requirements and its resilience to noise. Its noise resilience is shown in section 0 where it is clearly shown that the noise did not have a significant impact on the result. To give an understanding how much faster phase correlation is compared to normal cross correlation a comparison is displayed in Figure 3.3-4. The test sequence was run on a PC using Matlab® so the actual execution times are not relevant, but rather the difference in time between the two correlation techniques.

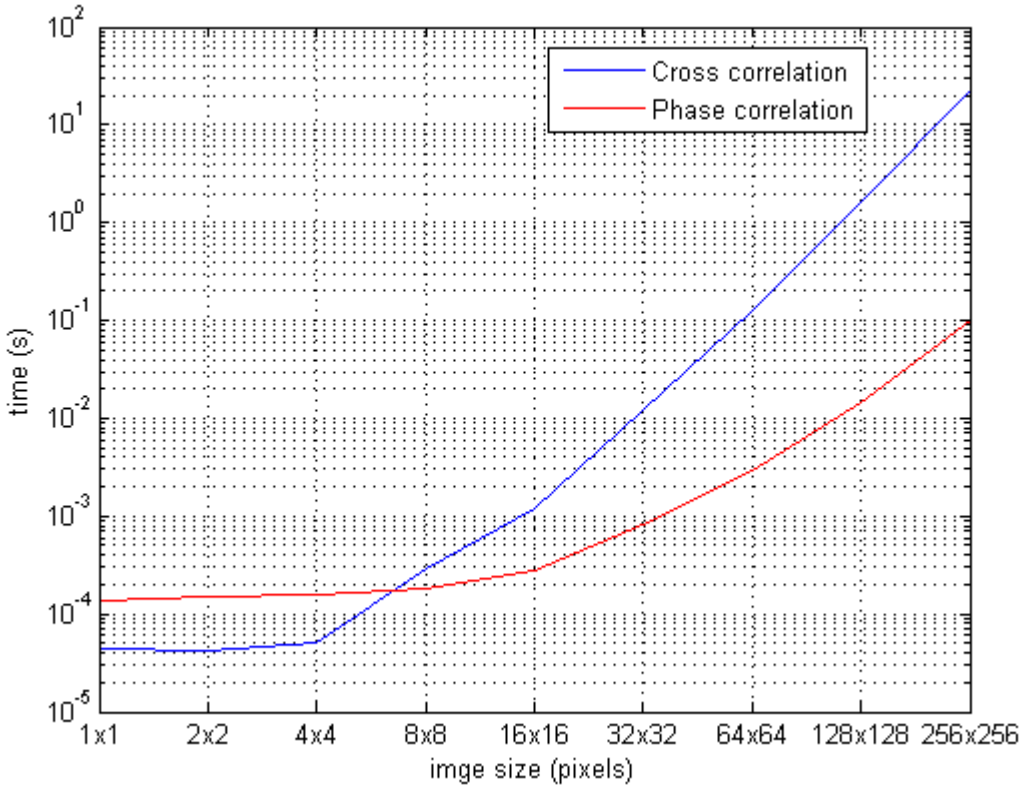


Figure 3.3-4: Cross correlation vs. phase correlation

As the figure shows cross correlation is faster for really small images while phase correlation is faster when the image is of size 8 by 8 or bigger. For the final test size, 256 by 256, phase correlation is as much as 200 times faster than cross correlation. The reason why the phase correlation is much faster is because a correlation in the frequency domain translates to a simple element-wise matrix multiplication while a correlation in the spatial domain is a number of summations.

In order to improve the precision of the phase correlation, which is limited by the size of each pixel, a sub-pixel addition was used. The method described in [16], which is applied in the spatial domain, was used for this purpose due to its low complexity compared to the method proposed in [15] which is frequency domain based.

The result from the phase correlation algorithm between the images from Figure 3.3-3 is displayed in Figure 3.3-5.

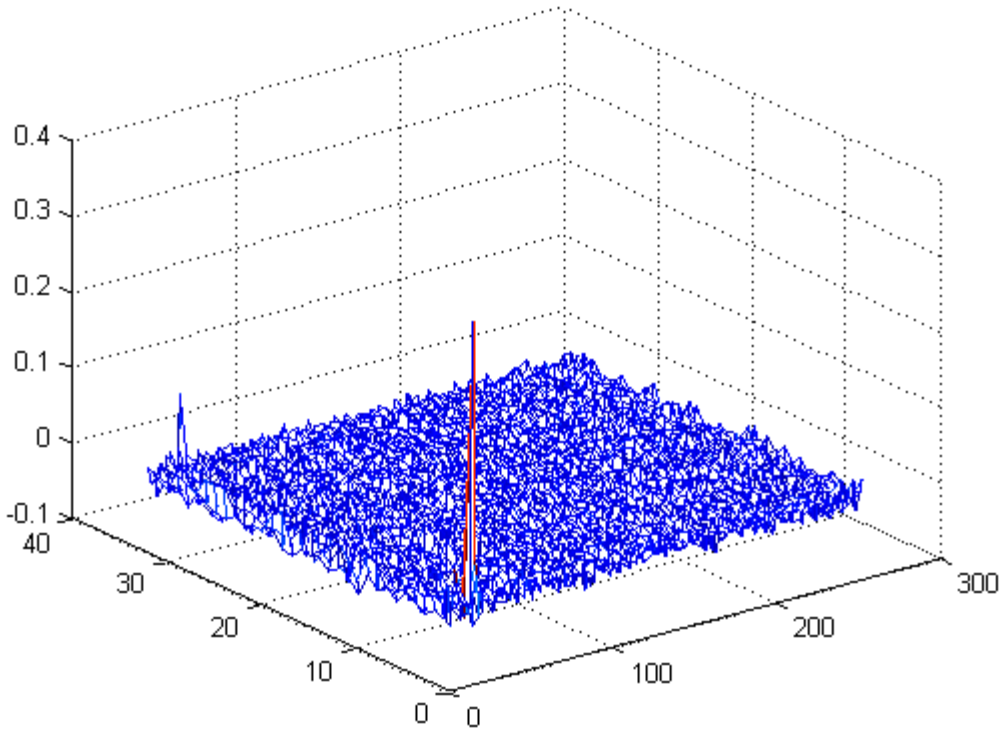


Figure 3.3-5: Phase correlation result

To calculate the offset from the phase correlation result the proposed sub-pixel method was used to produce the results presented in Table 3.3-2.

	Calculated offset
Δx	19.13 pixels
Δy	-0.08 pixels

Table 3.3-2: Offset with sub-pixel precision

3.3.3 Position Estimation

To get an estimation of where the vehicle is in a plane the speed in the vehicle's driving direction is not enough. To calculate how the vehicle is moving in the plane the vehicle's current heading relative some reference heading is needed. Figure 3.3-6 illustrates how the vehicle is moving with speed *v* in direction *α* in the plane *x*, *y*. Previous sections have already explained how *v* is calculated, which leaves *α* as the only unknown factor for position estimation.

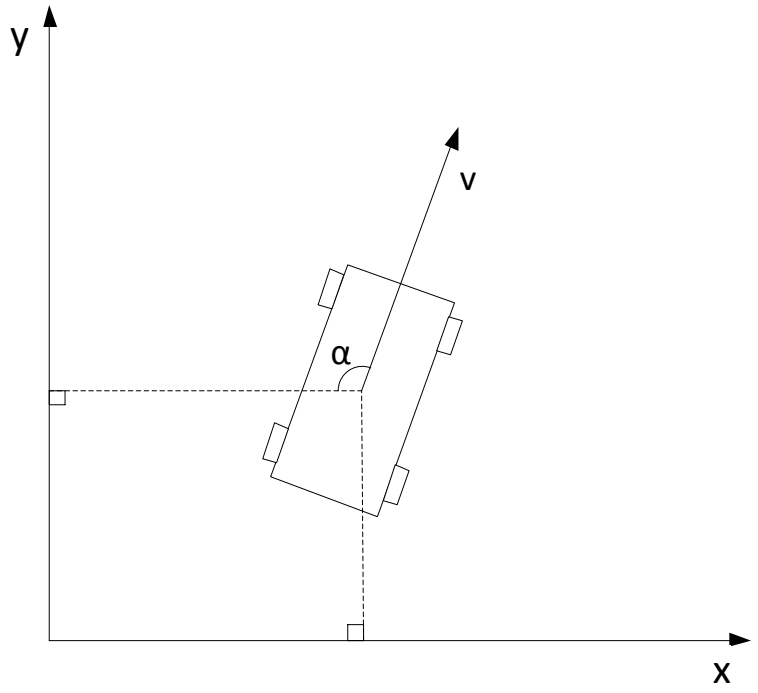


Figure 3.3-6: A vehicle moving in a plane (x, y) with speed v and direction α

It turns out, however, that α is not that easy to calculate under the current circumstances. A high frame rate will result in small changes in α from one frame to the next. By using the already introduced Equation 3.2-1 and the maximum speed requirement of 120 km/h, along with some assumptions it is possible to calculate the minimum frame rate needed. With the camera's distance to the ground used for the tests, 35 centimeters, the resulting *L* was measured to 23.5 centimeters. Let's then assume that at least 30% overlap between two consecutive frames is

needed in order for the phase correlation to give a good result. Using the proposed parameters the resulting period (*T*) ends up at around 4.9 milliseconds or a frequency of 200 frames per second (Hz). With the calculated frame rate it is then possible to estimate an angle change from one image to the next by looking at a typical turn. In accordance to Trafikverket, the Swedish Transport Administration, a normal turn on a highway should have a turning radius of at least 800 meters [19]. To calculate the angle change ($\Delta\alpha$) between two consecutive frames Equation 3.3-1 can be used.

$$\Delta\alpha = \frac{180 \cdot v}{F \cdot r \cdot \pi}$$

Equation 3.3-1: Angle change between two consecutive frames. F – frame rate, r – turn radius (m), v – speed (m/s)

With the proposed parameters the equation gives $\Delta\alpha \approx 0.012°$ which is the image rotation that has to be identified between two consecutive frames. This result is just to be seen as an example since it only reflects the angle change based on a Swedish highway, it is very likely that the angle change is much smaller than this. Log polar was evaluated to detect these small angular changes but failed to obtain such high precision. This resulted in that position estimation was not possible to achieve and the final solution will only covers speed measurement.

3.3.4 Interpret the Result

The next step after the offset has been calculated is to translate the offset to speed. In order to do this the offset in pixels have to be translated to distance in terms of meters. The actual pixel size when capturing an image from a certain distance was calculated by photographing a measuring tape. The measuring tape was put on the floor parallel with the camera's *x*-axis. The pixel size was calculated by dividing how much of the measuring tape that was visible along the *x*-axis by the number of pixels on the *x*-axis. But to calculate the speed one more parameter is needed; the time between two consecutive images. The frame rate was set to 200 frames per second during testing which results in five milliseconds between every image. The actual output will then be the average speed the last 5 milliseconds with an update rate of 5 milliseconds.

For the example images from Figure 3.3-2 the pixel size was calculated to $0.23 \cdot 10^{-3}$ m/pixel using the described method above. The resulting speed was calculated using Equation 3.3-2 which gives $v = 0.92\ m/s$.

$$v = \frac{\Delta x \cdot m/pix}{\Delta t}$$

Equation 3.3-2: Speed calculation formula, v. Δx – image offset (m), m/pix - pixel size (m), Δt - image period

To give an idea of how the output looks like over a longer period of time, 2000 frames in this case, an example is displayed in Figure 3.3-7. The image sequence used to produce this figure was not recorded with a vehicle with constant speed so a perfectly straight curve is not to be expected.

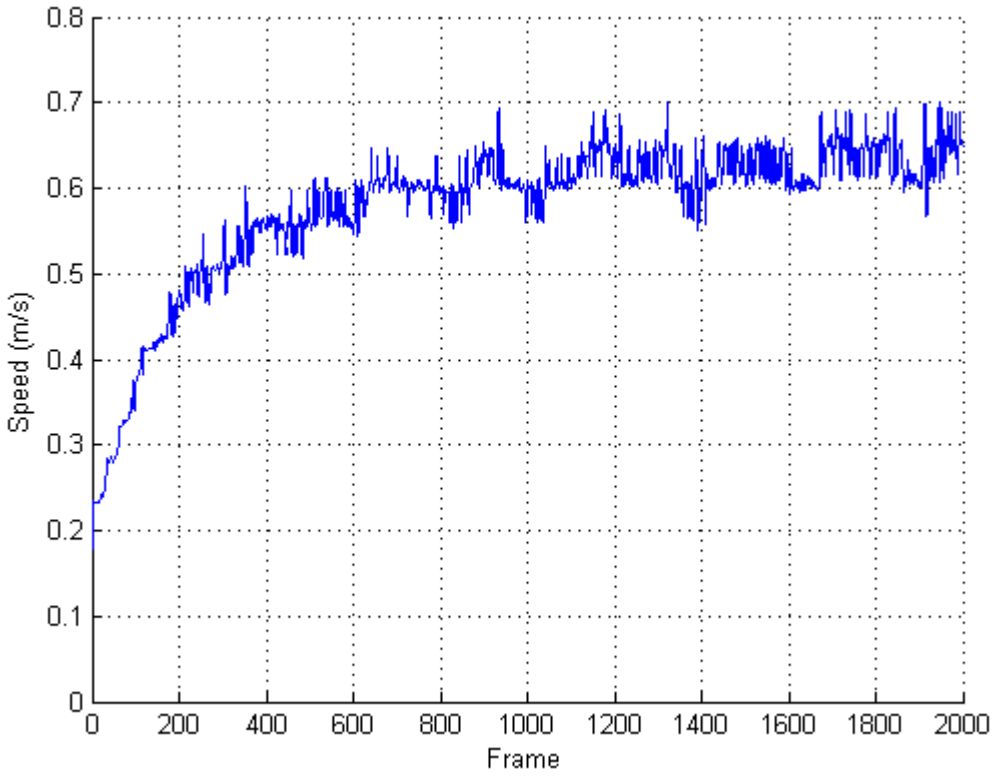


Figure 3.3-7: Output speed example

3.4 Parallelization

So far the solution has been focused on an algorithm that can solve the problem. To be able to run this algorithm in real-time with an update frequency of 200 Hz a parallel based approach was chosen. The algorithm has almost no need for sequential code sections making it possible to implement a parallel solution with good speedup results.

3.4.1 Multicore Platform

The Epiphany multicore system from Adapteva was chosen as hardware platform. The main reasons were the hardware support for floating-point operations, which the Ambric platform lacks, and the simpler hardware design making it easier to adapt the software to the architecture. The chosen algorithm was preferably implemented as a SIMD application where all cores were assigned unique data and executed the same instruction stream. In an application where all cores are synchronized the shared memory structure offered by the Epiphany system is preferred as it allows easy move of data between the cores. A disadvantage with the Epiphany system is its lack of memory protection compared to the Ambric. However, for the chosen application the need of memory protection can be handled in the software through barrier synchronization and dedicated memory areas for each core.

The chosen system has 16 cores and every core has 32 kilobytes of local memory. The resolution of the images to analyze was set to 256 by 32 pixels to fit in the available memory.

Development Board

The Adapteva Epiphany chip is attached to an evaluation board developed by BittWare named Anemone104. This evaluation board is a daughter board connected to an Altera Stratix III FPGA development board. The Stratix III FPGA board contains, except the FPGA, a 32 megabyte external memory and an USB-connection as PC interface. The development board is visualized in Figure 3.4-1.

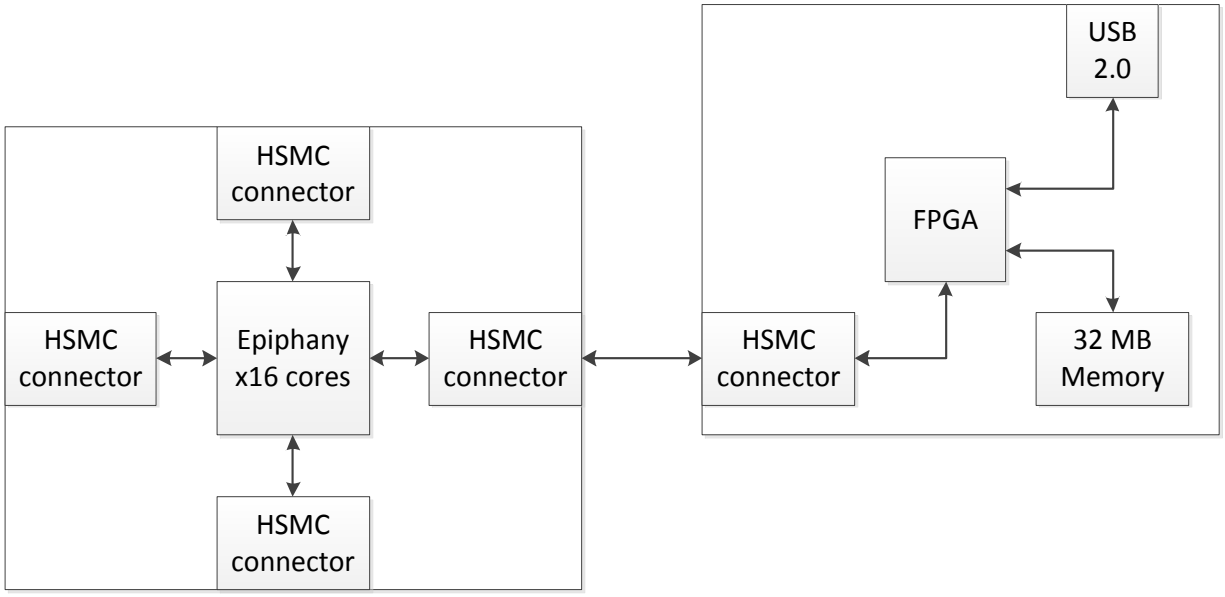


Figure 3.4-1: Development board.

The boards are connected to each other using an interface named High Speed Mezzanine Card (HSMC) developed by Altera. This interface is developed to achieve a high-performance I/O interface between FPGA motherboards and add-on cards with different functionality [20].

The development board does not have any support to connect external hardware. For this reason it was not possible to connect the camera to deliver images in real-time to the Epiphany chip. Instead, a sequence of images recorded to a computer was downloaded to the external memory attached to the FPGA board. The Epiphany chip was then able to load the images from this memory and perform the calculations.

Development Environment

An integrated development environment (IDE) based on Eclipse, specially developed for the Epiphany, was used to develop, debug and download code to the Epiphany chip. The IDE gives the possibilities to create, manage and navigate C based multicore projects as well as compiling, linking and debugging.

Each multicore project has a main project combined with an individual project for each core giving the possibility to a complete MIMD solution. The main project combines all output files from the cores into a single executable output file. To run a SIMD solution or a SIMD look-alike solution, e.g. code compiled to behave similar with only small differences depending on the core id, a common library can be used. The library is compiled with each core project resulting in the same code in each core. With a common library the only requirement on each project is a main function that calls a common main in the library.

A drawback of this project structure is the lack of general settings among cores, default all the settings are the same but if some compilation or linking settings is changed it has to be changed in each individual core project. Though this is a drawback if the cores should be treated similar it gives an advantage when core specialization is wanted.

Compilation of C code is done by *e-gcc*, an Epiphany compiler based on the well-known GNU GCC, and supports out of the box ANSI C code. For assembly parsing *e-as*, the Epiphany assembler, is used and for linking *e-ld*, the Epiphany linker, is used.

Bundled with the *e-gcc* is a set of libraries available to be included. This set is based on Newlib, a distribution of standard C and standard math libraries for embedded systems. Also included is the Epiphany Hardware Utility library (*eLib*) providing functionality to configure and accessing hardware resources.

To debug the multicore project *e-gdb*, the Epiphany debugger based on the well-known GNU GDB, is used. One *e-gdb* debugger has to be run for each core which even though it gives the possibility to debug each core individually also requires a lot of resources to be run. On a standard PC debugging 16 cores makes the PC slow and hard to work with. As each of the core's debug session requires the same resources the work load on the debug PC will scale linearly with the number of cores making it difficult to debug more cores.

3.4.2 Memory Usage

To make the images to analyze available to the multicore chip they are stored in the external memory on the FPGA board. This memory is divided into two memory banks, bank 0 and 1. The first bank contains synchronization bytes, bytes to store the final results and some of the images to analyze. The second bank is completely filled with more images.

Every core in the Epiphany chip has 32 kilobytes of memory. The memory is divided into 4 banks. The first two banks and a part of the third are dedicated to the interrupt vector and handler, code libraries, firmware and data. On the third and fourth bank the images are stored while doing the calculations. Memory maps of both the external memory on the FPGA board and the internal memory in the Epiphany core are shown in Figure 3.4-2.

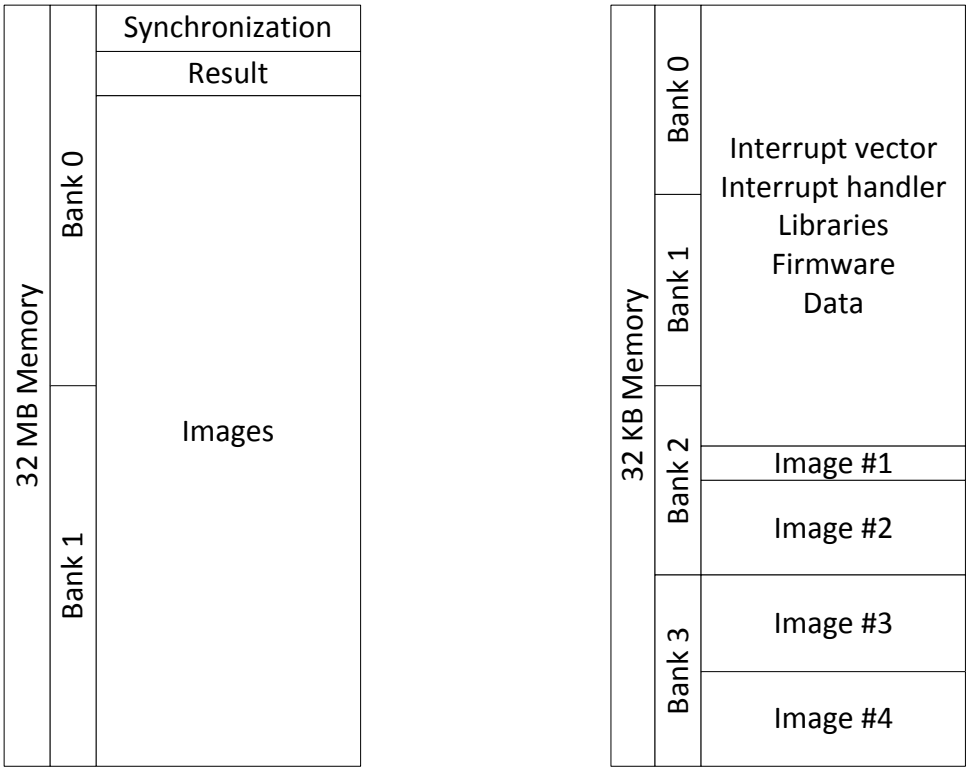


Figure 3.4-2: Memory map of the memory on the FPGA board and the internal memory in an Epiphany core

Every core has a sixteenth of the image stored locally. There is memory allocated to store four copies of this part. These copies are used to store the original image, the filtered image, the transformed image and the previously transformed image. The first image, where the original image loaded from the external memory is stored, is smaller due to it is represented by 8-bit real numbers. The other images are represented with complex numbers of single-precision floating-point type (32-bits).

During the development of the parallel solution it was observed that some parts of the program were executed really slow. After some analyzes the reason of this was discovered. Some code libraries were put in the external memory attached to the FPGA. A parallel program, where all cores execute the same code, turns out not to be parallel anymore when all cores try to execute code from the same memory. Each core has to queue to access the instructions stored in this external memory. By moving these code libraries into the local memory on the Epiphany chip a great execution speed-up was observed.

3.4.3 Synchronization

When using many cores to run a single application in parallel some kind of synchronization mechanism may be needed. It is for example needed to make sure that the cores do not use data that is not yet valid, in case another core has to update the data first. This project involved testing of two different barrier synchronizing methods which both included *ready* and *go* flags. The basic concept of this principle is that when a core has finished its task it sets a ready flag and waits for synchronization with the other cores. Each core then has to wait until its go flag is set before it can continue. One of the synchronizing methods involves the host computer and the external SDRAM while the other only uses the internal resources within the Epiphany chip.

The method which involves the SDRAM is illustrated in Figure 3.4-3. The idea here is that each core sets a *ready* flag in the SDRAM when it is ready and needs to wait for synchronization with the other cores. After the *ready* flag has been set the core waits for its *go* flag to be set. While the cores are doing this the host application is polling the *ready* flags and waits for all of them, one for each core, to be set. When all the *ready* flags are set the host application will set all the *go* flags which will trigger all cores to continue executing. Before the cores continue they will clear

both its *ready* and *go* flag to prepare for next synchronization. This procedure is then repeated every time synchronization is needed.

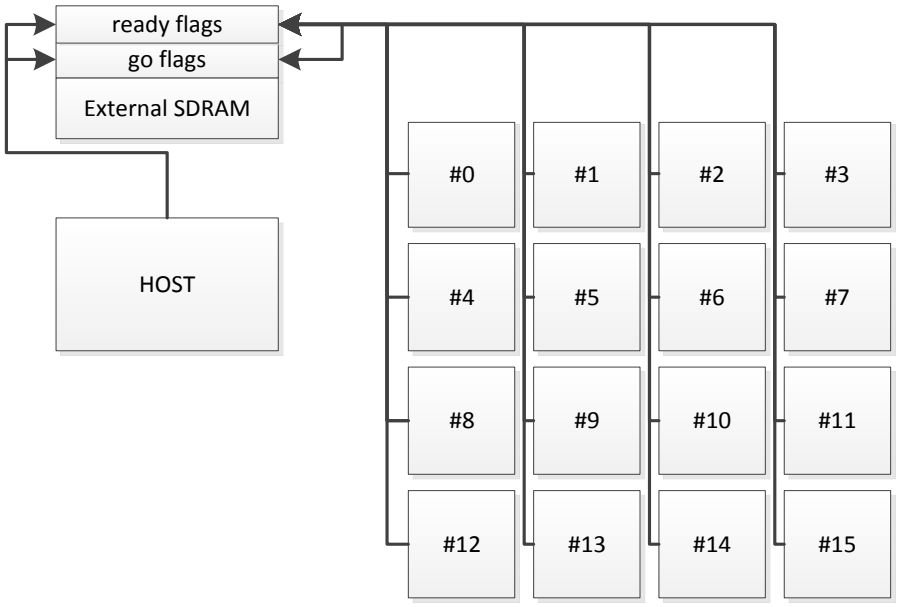


Figure 3.4-3: Synchronization through SDRAM

A drawback with this solution is however that the synchronization is handled through the external SDRAM which has higher access time compared to internal memory. But it is an advantageous solution for debug purposes since there are better output possibilities when the host application, which is running on a PC, is involved.

The other proposed synchronization method is not relying on external memory or an external host application but instead completely internally handled. The method is illustrated in Figure 3.4-4 which also shows that each core has a *ready* and a *go* flag allocated in its local memory. This method is based on the principle that each core sets a *ready* flag, when it is ready for synchronization, in the next core according to the pattern in the figure. The pattern, which always results in writes to an adjacent core, has been chosen to minimize write delays. To make sure that the *ready* flags are set in a certain order no core is allowed to set the next core's *ready* flag before its own *ready* flag is set, with the exception of core #0 which starts the synchronization. When

core #0 reads a 1 in its *ready* flag it means that all cores are ready to go. Core #1 then starts the same procedure with the *go* flag as with the *ready* flag according to the same pattern as before. Each core then starts execution when its *go* flag is set and is responsible for clearing its *ready* and the *go* flag afterwards.

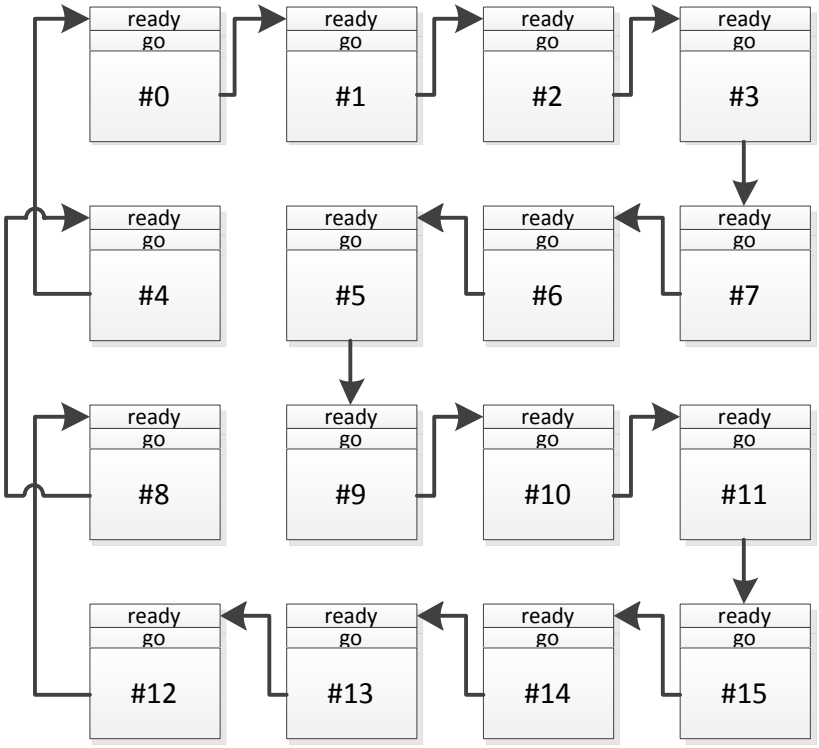


Figure 3.4-4: Internal synchronization

This method has lower access times compared to the first method but requires internal resources. The two bytes that are needed for the synchronization flags are however only a fraction of the total memory available locally to each core. Custom code is required for core #0 as it starts the synchronization and does not have to wait for the initial *ready* flag to be set. The *ready* flag is in core #0 used to notify that all cores are ready.

A benchmark test was run to compare the two methods to each other. The presented result is the number of clock cycles needed after the first image grab, which require synchronization afterwards, from the external memory until all cores are synchronized again. The result is presented as a mean value because the time differs from core to core. The reason that the synchronization times differs are because all cores do not reach the synchronization point at the same time. The results are presented in Figure 3.4-5.

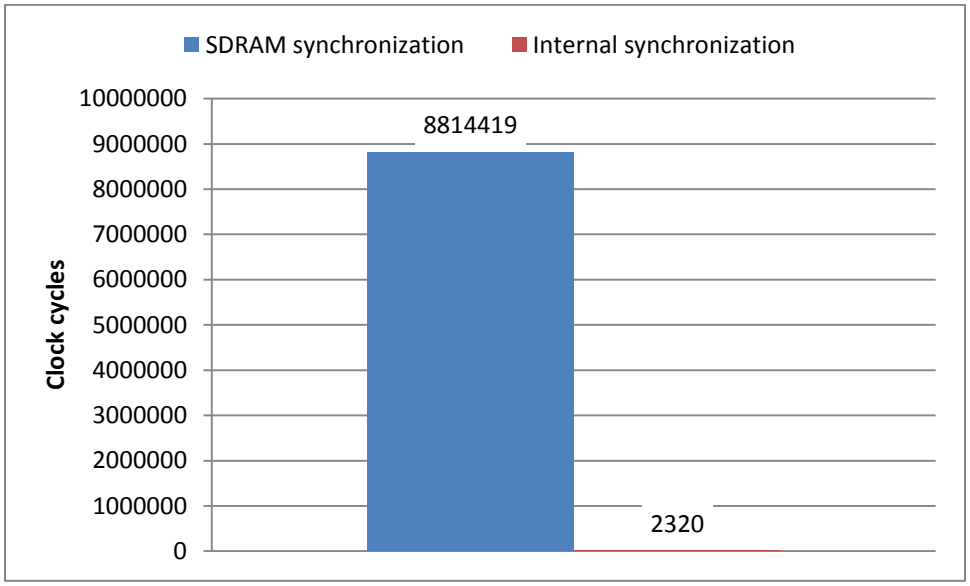


Figure 3.4-5: Synchronization comparison

As expected there is a huge gain in handling the synchronization internally. Not only is the access time from the cores to the external SDRAM memory greater compared to internal memory access but the access time from the host application in the PC to the SDRAM is also slow and becomes a bottleneck.

3.4.4 Data Movement

Because every core only has a part of the complete image, data has to be exchanged between the cores to get through all calculations. This subsection will explain how the data movement between the cores is done while executing the complete algorithm described in section 3.3.

At startup the first image is loaded from the external memory and is divided equally between all cores. The image is separated in columns and is stored in memory bank 2 as image #1 shown in the left memory map in Figure 3.4-6. When the image has been loaded each core sends the adjacent pixels needed by every other core to apply the filter. To make sure each core has received the adjacent pixels, needed for the filtering, the cores synchronize with each other before continuing the execution.

Every image after the first one will be read from the external memory using the DMA channels, along with the write of adjacent pixels for the filtering. This makes the time it takes for an image to be read non-existent as it will be done in parallel to the execution of the application for each new measurement. The time saved by not having to read the image, as well as redistributing adjacent pixels, combined with the required synchronization measures to 137720 clock cycles. The cost of having the next image read in parallel is that the measurements will be lagged one measurement compared to reading the new image just as it is taken.

The first thing to do when a new image has been loaded and all cores are synchronized is to apply the filter. To calculate the new value for an arbitrary pixel of the image the values from adjacent pixels in the original image are required. For this reason, it is not possible to overwrite the old pixel values in the current image. Instead, the filtered image is stored as a new copy in image #2; this is shown in the right memory map in Figure 3.4-6. To prepare for the Fourier transform the image is stored in complex form.

After the image has been filtered image #1 is available for a new image to be read, along with the writing of the adjacent pixels for the next filtering, using the DMA channels while the execution of the application continues on undisturbed. Before the next measurement starts the core confirms that the DMA read has finished.

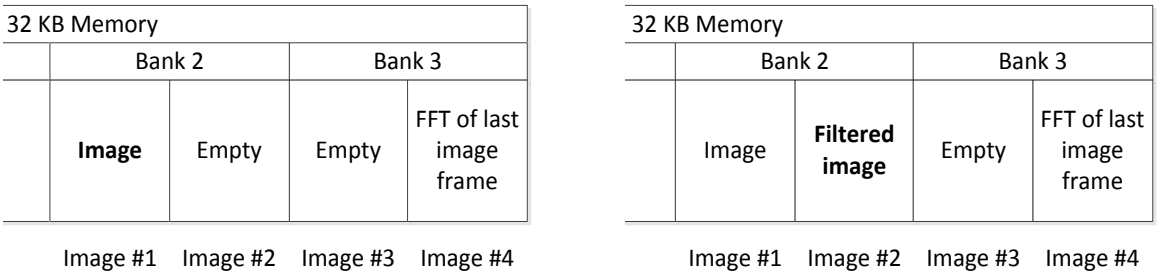


Figure 3.4-6: Memory map when an image has been loaded to the left and after the filtering to the right

The Fourier transform does not depend on the adjacent pixels to calculate the new pixel values. The result of the transformation of each pixel is therefore possible to store in the same memory area, shown in the left memory map in Figure 3.4-7. The image is a two dimensional signal and the Fourier transform calculations are performed on both columns and rows of the image. A data shift between the cores must occur when the first step of the Fourier transform is finished so that each core instead of whole columns has whole rows. This means that every core exchange data to get a part of the image represented by rows instead of columns.

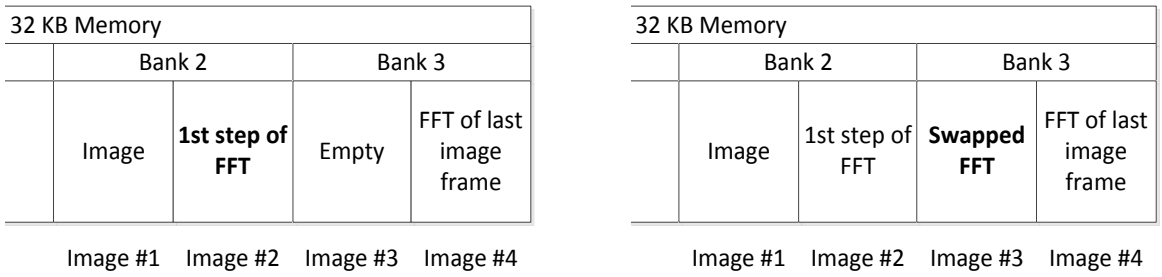


Figure 3.4-7: Memory map after the first step of the FFT to the left and after the data swap to the right

When going from columns to rows the data of one core has to be distributed among all the other cores as one column includes parts of each row. This re-distribution utilizes the shared memory map by locating the destination matrix in each core, given by the hardware function *e_address_from_coreid()*, and then writes directly to its address. The write operation is then handled by the interconnection network minimizing the effort to write the code. By scaling up the amount of cores the time for the swap scales non-linearly as the distance to the furthest core increases for each added row, or column.

If all cores start from the beginning of their columns they will all want to write to the first core as the beginning of each column will result in the new first row located in the first core. One could think this would create conflicts as multiple cores write to the same core at once compared to each core writing to a separate core but what is shown by a simple test is that this is not the case. A simple test where each core starts in their columns with an offset based on their core number causing all cores to write to a separate core throughout the swap. The simple test shows that it is actually more demanding to perform the swap where each core has an offset due to the extra calculations that are needed to handle this offset and the memory management itself is not effected by the change.

All cores do the swap simultaneously and the new image part, represented by rows, is stored in memory bank 3 as either image #3 or image #4. Which memory area to use varies for every other frame loaded into the system. This behavior, to vary the location, is chosen to save memory space as two images are the least required to be able to compute the correlation but only the latest frequency representation is required for the next measurement. The older one is therefore overwritten for each new image. In the right memory map in Figure 3.4-7 the image is stored as image #3.

Before the computation of the Fourier transform can continue the cores have to synchronize with each other to make sure that the memory swap between the cores are finalized.

The result of the second and final step of the Fourier transform is stored in the same memory area, similar to how the first step of the Fourier transform worked, which is shown in the left memory map in Figure 3.4-8.

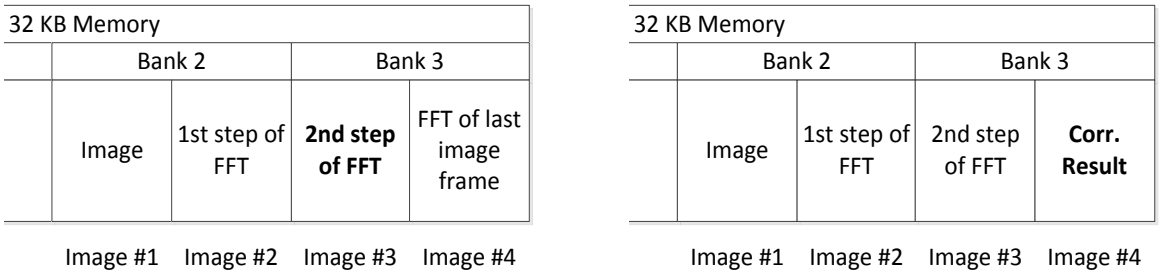


Figure 3.4-8: Memory map after second FFT to the left and after the correlation to the right.

Assumed that the transformation of the previous image is stored in image #4, the result of the correlation is written in the memory area holding the transformation of the oldest image, in this case image #4. The memory map after this action is shown to the right in Figure 3.4-8.

The inverse Fourier transform has the same behavior as the forward Fourier transform described earlier. But now every core begins with an image part represented in rows. The result is saved in the same memory area as the correlation was saved to, viewed in Figure 3.4-9.

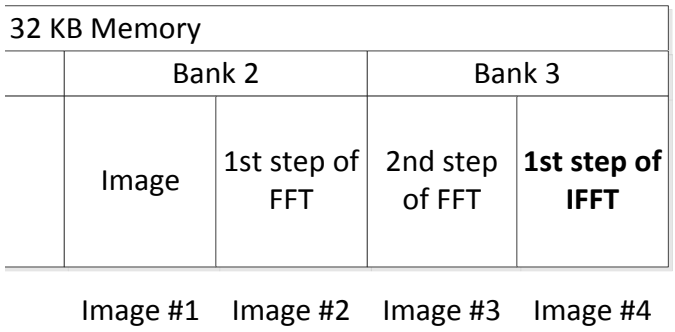


Figure 3.4-9: Memory map after the first step of the IFFT

To finalize the inverse Fourier transform the cores have to do a data exchange again. This to get an image part represented in columns. As previously this is done simultaneously between the cores. The new image part, represented in columns, is stored in memory bank 2 as image #2. To continue the cores has to synchronize just as for the forward Fourier transform.

The result of the second step of the inverse Fourier transform is stored to the same memory area. The memory state after the data swap is shown to the left and after the second step of the inverse Fourier transform is shown to the right in Figure 3.4-10.

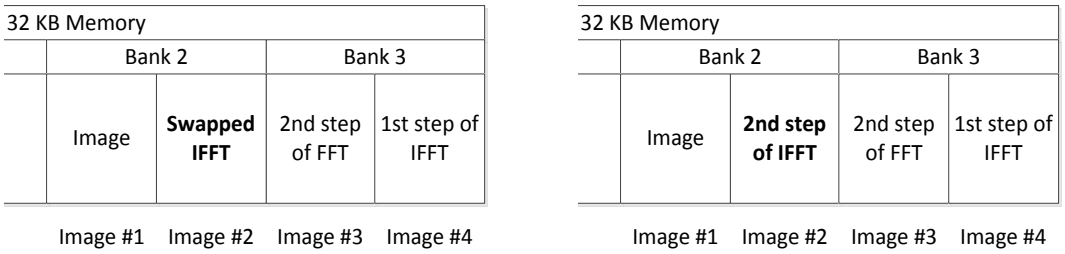


Figure 3.4-10: Memory map after the data swap to the left and after the second step of IFFT to the right

The highest peak value, along with its coordinate, from each core is written to the external memory to be analyzed by the host application.

3.4.5 Core Implementation

The code is written in a way that makes it possible for all cores to run the same code. Each core has its own unique core identifier which is used to handle cases where the expected behavior is different depending on which core it is. One example of this is the filtering step where each core has to use two columns from its neighboring cores to complete the filtering. For the first and last core there is however only one column to copy pixel values from while the other non-existing column will have to be filled with zeroes. The core identifier is also used to calculate from where in the external memory its part of the image is. No special treatment is needed in this case though since all cores use the same formula to calculate the memory offset by using its identifier. Since the same code is executed on each core it is possible to add more cores without big code modifications. More cores could mean faster execution time as the image is divided into even smaller parts.

MOV Instruction

Since the chosen algorithm uses a frequency representation of the image complex numbers are used. The complex numbers are described as a *struct* with an imaginary and a real part which both are defined as single-precision floating-points. To increase performance when moving complex numbers the code has been written to explicitly force the compiler to use 64-bits *MOV* instructions when moving complex numbers. A move of a complex number could otherwise have resulted in two 32-bit *MOV* instructions.

BITR Instruction

To increase the calculating speed for the FFT there is an assembly instruction, *BITR* (bit reversal), which is used in the implementation. *BITR* has to be written explicitly with inline assembly in order to make the compiler use it. The benefit with using *BITR* is that the bit reversal can be done with one instruction instead of writing C code for the same purpose which would have generated several instructions including loops. The performance of the *BITR* instruction was tested by comparing it to a C code implementation. The result from the comparison can be viewed in Figure 3.4-11. The y-axis shows number of clock cycles for the different implementations across 512 samples. The measured speedup was from this calculated to 174%.

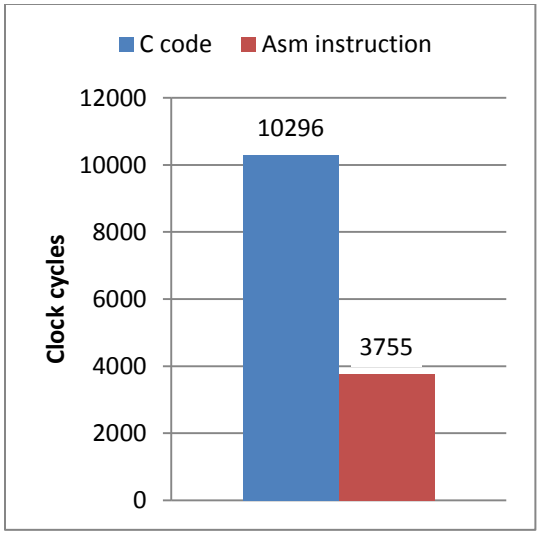


Figure 3.4-11: Bit reversal comparison

Floating Point Division

The proposed algorithm includes floating-point division both in the normalized cross power spectrum calculation and in the inverse FFT. Since there is no hardware support for floating-point division in the Epiphany chip these divisions result in time consuming loops. In some cases division can be replaced by multiplication which is supported by the hardware. Code snippet 3.4-1 shows an example of two floating-points which both are divided by a third floating-point. This is the case when calculating the normalized cross power spectrum where both the imaginary and the real part are divided by a floating-point.

```
void DivExample( float *a, float *b, float c){

   *a /= c;
   *b /= c;
}
```

Code snippet 3.4-1: Floating-point division example including two divisions

Instead of doing the two floating-point divisions from the previous example it is possible to do one division and two multiplications. An example, which will produce the same result, of this can be seen in Code snippet 3.4-2. This solution is much less time consuming because of the previously mentioned hardware support for floating-point multiplication.

```
void OptimizedDivExample( float *a, float *b, float c){
   float f;

   f = 1/c;
   *a *= f;
   *b *= f;
}
```

Code snippet 3.4-2: Floating-point division example including one division

The speedup from using the first method versus the second method was tested in the normalized cross power spectrum implementation. The results from the test are presented in Figure 3.4-12 in number of clock cycles required to calculate real division across 512 complex numbers, i.e. division of both the real and the imaginary part. The speedup in this case is approximately 86%.

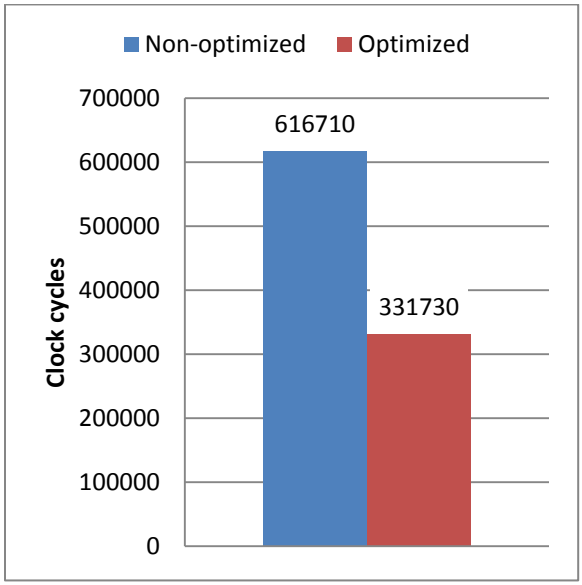


Figure 3.4-12: Division versus multiplication comparison

Square Root

To calculate the square root is another very time consuming operation which also includes a large amount of iterations just as division. Instead of using the standard *sqrtf()* from the standard math lib, which results in the mentioned iterations, there are other faster alternatives. The alternatives often suffer from worse precision but the approximation can in many cases be good enough. An example of such alternative is a square root function that was popularized when the Quake III source code was released [21]. The original author of the code is however Greg Walsh according to [22]. The code is presented in Code snippet 3.4-3.

```
float QuakeSqrt(float x){
   float number = x;
   float xhalf = 0.5f*x;
   int i = *(int*)&x;
   i = 0x5f3759df - (i>>1);
   x = *(float*)&i;
   x = x*(1.5f - xhalf*x*x);
   return x * number;
}
```

Code snippet 3.4-3: Quake square root

The Quake method is also an iteration based approach, with limited iterations, but revolves around the "magic number" 0x5f3759d which is used to calculate the first approximation. The last line before the return can be repeated to increase the precision even more.

To test the performance of the mentioned methods a test was performed calculating the absolute value across 512 complex numbers, which is done when calculating the normalized cross power spectrum. The results are presented in Figure 3.4-13 in number of clock cycles.

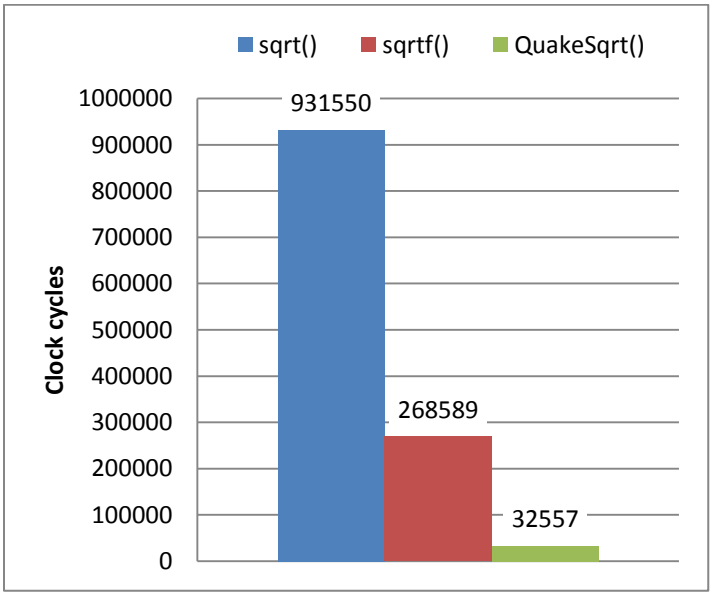


Figure 3.4-13: Square root comparison

The normal *sqrt()* function has been added to show how important it is to know what data type that is being used. When using the *sqrt()* function the floating-points will be casted to double-precision floating-points and therefore the decreased performance. The speedup from using the *QuakeSqrt()* compared to *sqrtf()* is in this case 725%.

If one takes a closer look at Equation 2.5-9, which is the place where the square root function is used in the implementation, it can be noted that the division can be replaced with a multiplication of the inverse absolute value. The new cross-power spectrum formula can be seen in Equation 3.4-1.

$$R = G_a G_b^* \cdot |G_a G_b|^{-1}$$

Equation 3.4-1: Cross-power spectrum without division

This rewriting of the formula results in a need of the inverse square root rather than the square root. Such implementation can also be found in the Quake III's source code. By simply replacing the last line, as seen in Code snippet 3.4-4, the function will return the inverse square root. Because of this, the inverse square root function is actually even faster than the previously mentioned *QuakeSqrt()*.

```
float InverseQuakeSqrt(float x){
   float xhalf = 0.5f*x;
   int i = *(int*)&x;
   i = 0x5f3759df - (i>>1);
   x = *(float*)&i;
   x = x*(1.5f - xhalf*x*x);
   return x;
}
```

Code snippet 3.4-4: Inverse Quake square root

3.5 Interface

This system aims to be used in vehicles such as cars. Different interfaces may be considered, for instance a serial link or CAN network may be used. A serial link enables the sensor to be connected to another system such as a control system. On the other hand, CAN network is common in especially cars and makes it possible for any system connected to the vehicle's CAN network to access the information this sensor provides.

As it showed up that no external hardware could be connected to the development board, the work with finding a suitable solution to the interface was discarded.

4 Test Setup

4.1 Preparation

To verify the proposed solution test data had to be captured. This was done by recording a video using a robot built by bachelor students at Halmstad University. The camera was mounted on the robot facing downwards the ground. The robot traveled a specified distance which will be used as reference when evaluating the algorithm. A laptop was used to store the images taken by the camera to create the video sequence. The setup is shown in Figure 4.1-1.

Figure 4.1-1: The vehicle with the camera mounted to the left and close-up of the camera installation to the right.

The camera was configured with the settings presented in Table 4.1-1.

Camera setting	Value
Frame rate	200
Exposure time	100 µs
Image resolution	1024*64

Table 4.1-1: Configuration of camera in test sequences

A distance of 80 meters was measured to record the test sequence. Every 10 meter was marked with a piece of tape on the ground to get references of the traveled distance in the image sequence as shown in Figure 4.1-2. These marks are intended to be used to compare the result from the algorithm with the actually traveled distance.

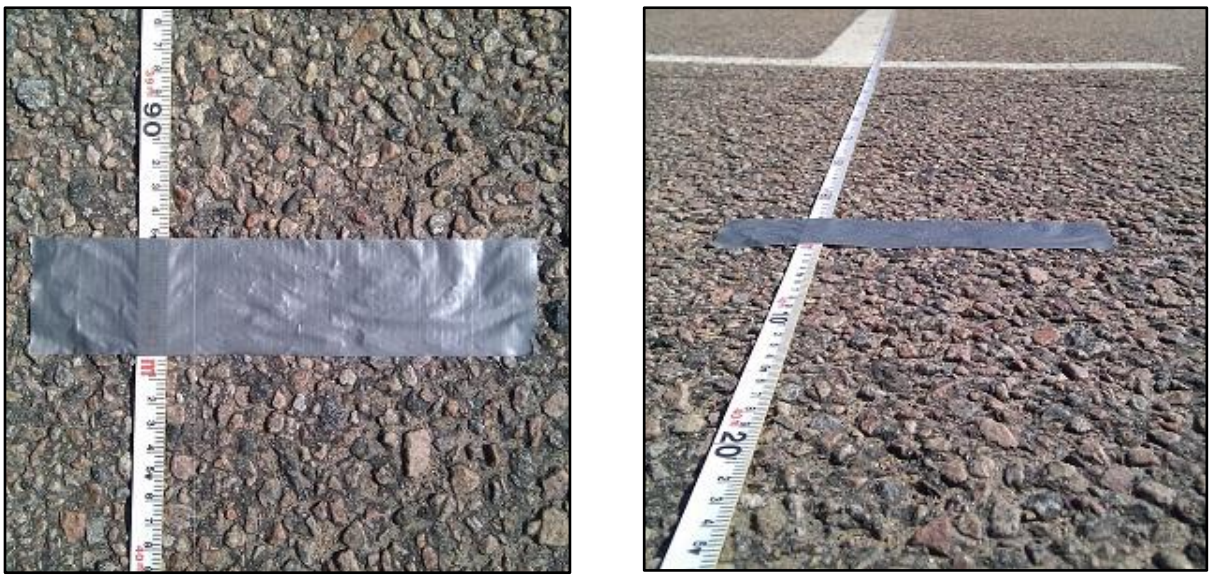


Figure 4.1-2: Measured test distance with reference points

The test sequence was recorded in daylight with direct sunlight and the temperature was 15 degrees Celsius. No extra light sources were used.

4.2 Software Verification

The first implementation of the chosen algorithm was written in Matlab®. The reason why the first implementation was done in Matlab® was because of the simplicity compared to writing the code in C directly. Functions such as FFT, image filters, matrix operations and IFFT are already built-in which enabled the focus to be put on the complete functionality rather than the implementation. Another advantage with doing the first implementation in Matlab® was the good output possibilities, such as easy graph plotting.

To verify the functionality of the actual algorithm all the images from the recorded test sequence were extracted from the video and used as input to the Matlab® program. The output was then given as the total offset from the first to the last image in the video sequence. To evaluate the result the output was compared to the measured distance of 80 meters. While recording the test sequence frame drops were encountered which resulted in consecutive images with no similarity.

The Matlab® implementation was then used as a reference for a sequential C implementation. The image filter and the matrix operations were implemented from scratch while the FFT and IFFT functions are modified versions of Paul Bourke's C implementation [23]. Each step of the algorithm in C was compared to the result from Matlab® with two reference images as input with a known offset.

The final implementation, also written in C, was a parallel solution for the Epiphany chip written with the sequential C code as reference. Each step of the calculation in the parallel code, including the final result, was uploaded to the external SDRAM and compared with the result from the sequential C code. With all the described tests the behavior of the parallel code was proved to produce the same result as the original Matlab® code.

4.3 Epiphany Performance

When the algorithm had been verified and both the sequential and parallel implementation was executed properly it was time to analyze the performance of the solutions. To verify the performance of the chosen multicore platform, Epiphany, the execution time was measured while executing both the parallel and sequential implementations.

The test was prepared by the host application which downloaded a number of images from the recorded test sequence to the external SDRAM memory. Then the host application made the system start. The time is measured both including the loading of images from external SDRAM and without this loading stage. How well the interface between the multicore chip and the external SDRAM is unknown. By using DMA the loading time does not exist because the image is loaded in the background by a DMA-channel while the application is executed. However, the interface must perform good enough to load the image within the time limit determined by the execution time of one measurement cycle. Even better would be to set the time limit equal to the execution time of a part of the algorithm that is between two data swaps, this to not let the DMA-channel interfere the data swap which can increase total execution time.

The sequential solution was tested in the same manner as the parallel but only using one core in the Epiphany chip. Lack of memory in a single core made it impossible to analyze a complete image. Therefore only a sixteenth of the complete image was analyzed and the results of this were then used to theoretically calculate the complete execution time. Since the actual calculation times of the FFT, the phase calculation and the IFFT scales linearly as the number of columns or rows increases the calculation time for one sixteenth of the image was multiplied by 16.

The results of the parallel and the sequential solution were compared to the requirements set by the project. It is also of interest to measure how much better performance the parallel implementation resulted in and the parallel utilization. The theoretical performance gain is affected both by communication delays and the how well the software is parallelized. To calculate how much the performance was affected by these factors the parallel utilization was calculated using Equation 4.3-1.

$$E(N,P) = \frac{1}{P} \cdot \frac{T_{seq}(N)}{T(N,P)}$$

Equation 4.3-1: Parallel utilization. N - Size of problem, P - Number of nodes.

5 Results

The results produced from the tests described in chapter 4 are presented in this chapter. Section 5.1 presents the result of the algorithm and section 5.2 the performance of the Epiphany platform.

5.1 Position

Figure 5.1-1 shows the distance measured by the proposed algorithm using the recorded video sequence described in section 4.1. The distance is presented as the calculated offset between every consecutive image in the video sequence. Every 10 meter a red cross marks the reference distance.

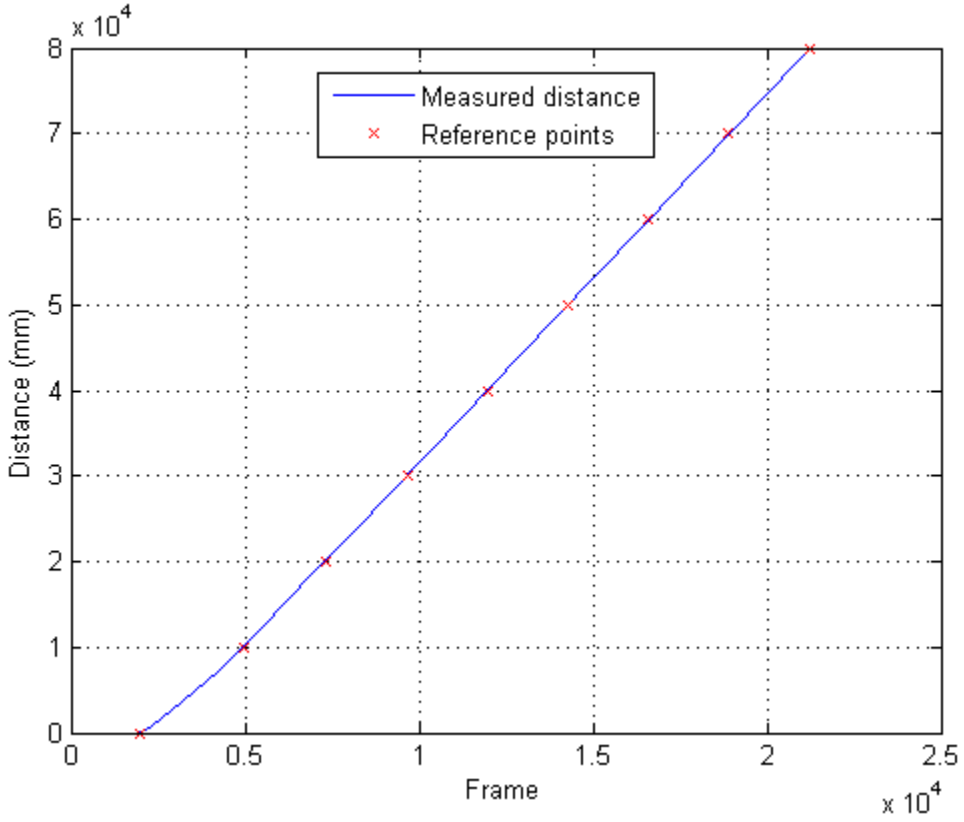


Figure 5.1-1: Calculated distance from recorded test sequence

In Table 5.1-1 the error between the measured distance and the reference distance are presented. A positive error indicates a greater measured value and a negative value indicates a lower measured value compared to the reference value. As mentioned in 4.2 frame drops were experienced when recording the test video. A result of this can be seen in the figure from the 50000 millimeter mark to the 60000 millimeter mark where the error has changed from 25 to -266.

Reference distance (mm)	Measurement error (mm)	Error rate
10000	50	0.50%
20000	39	0.20%
30000	63	0.21%
40000	50	0.13%
50000	25	0.05%
60000	– 266	0.44%
70000	– 255	0.36%
80000	– 231	0.29%

Table 5.1-1: Measurement error at reference points

Using the results of the measured distance and the known image period the positional speed was calculated and is presented in Figure 5.1-2. The video sequence used was not recorded with a vehicle with constant speed so a perfectly straight curve was not to be expected.

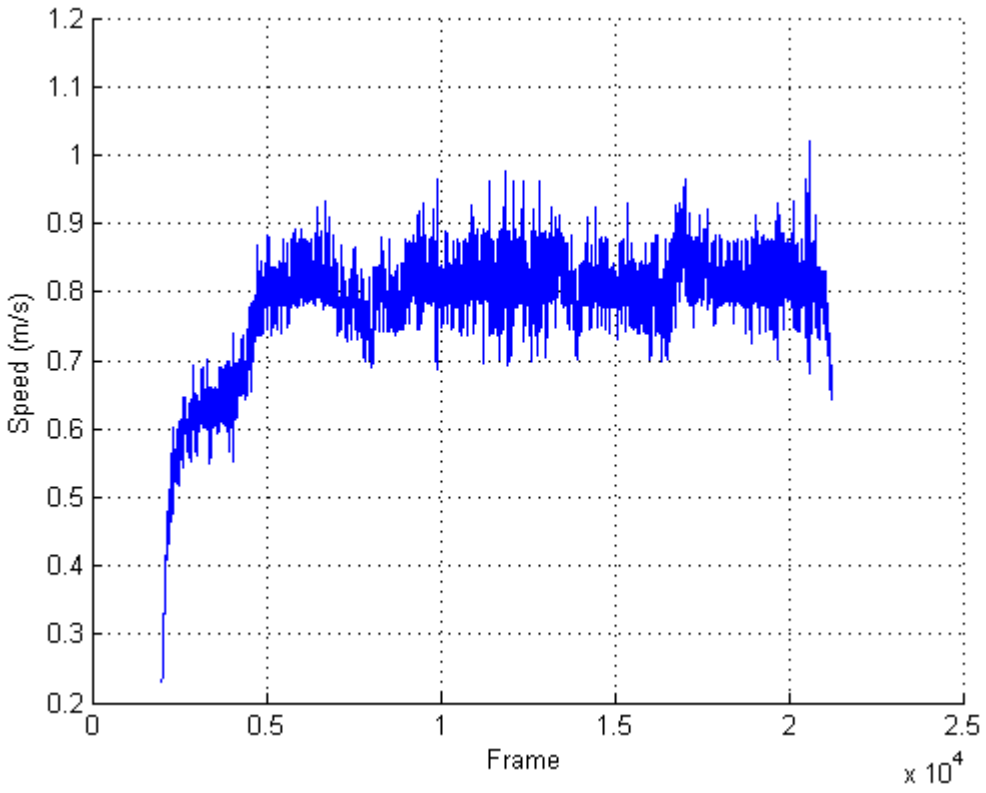


Figure 5.1-2: Calculated speed from recorded test sequence

5.2 Epiphany Performance

In this section the performance of the Epiphany platform is presented. The result is divided into three subsections; 5.2.1 and 5.2.2 where the parallel respectively the sequential performance is presented and 5.2.3 where the two solutions are compared. All measurement results are presented in number of clock cycles. With the used clock frequency of 1 GHz, the number of clock cycles can be divided with 10^6 to get the result in milliseconds.

5.2.1 Parallel Solution

The performance of the parallel solution is presented in this subsection. The algorithm has been divided into smaller parts to show how much each part take of the total computation time. Brackets are used in the figure to show the sub-parts that are included in the FFT, the phase correlation and the IFFT. The result is divided into two parts, one with the performance of the original implementation and one with the optimizations proposed in section 3.4. The result from the first implementation is presented in Figure 5.2-1.

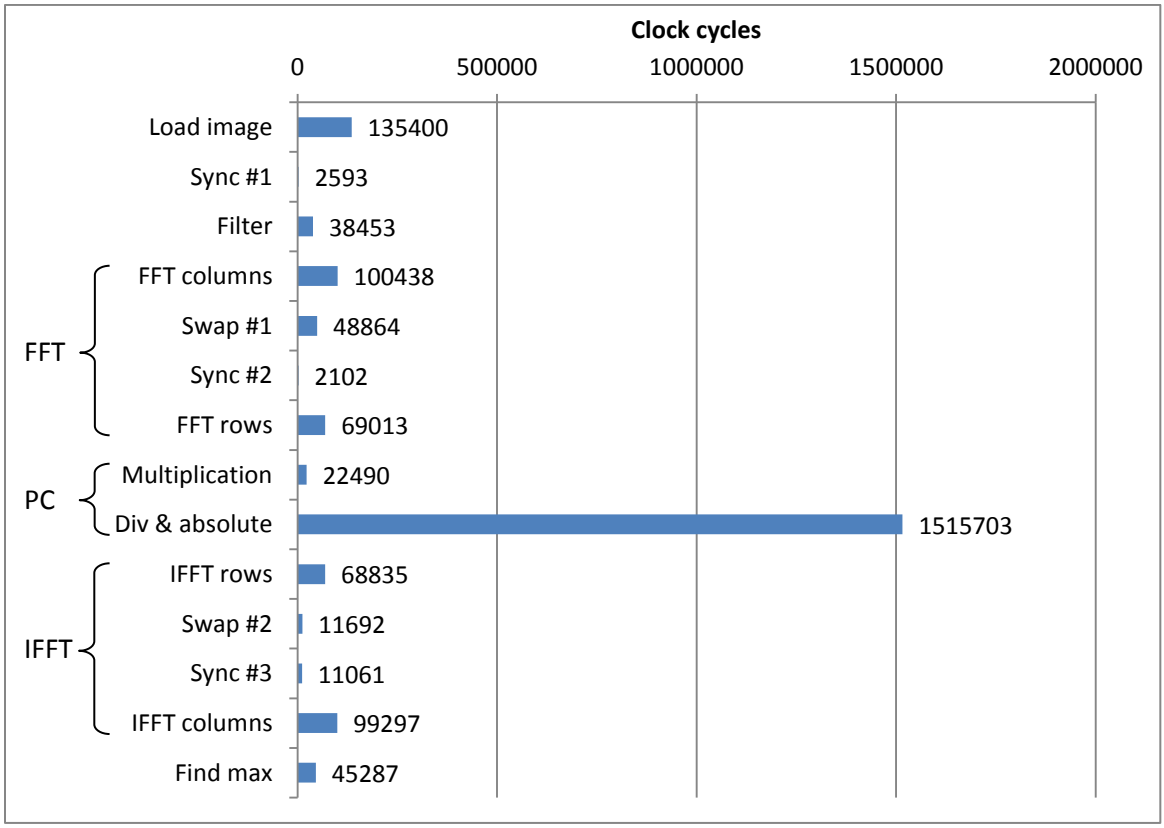


Figure 5.2-1: Original parallel performance divided into sub-tasks.

As can be seen in the figure the two big bottlenecks were *Load image* and *Div & absolute*. The result after optimizing those two parts is presented in Figure 5.2-2. The difference from the previous version is that this version uses the DMA channel as proposed in 0 and the square root optimization as proposed in 3.4.5. All other mentioned optimizations are implemented in both results.

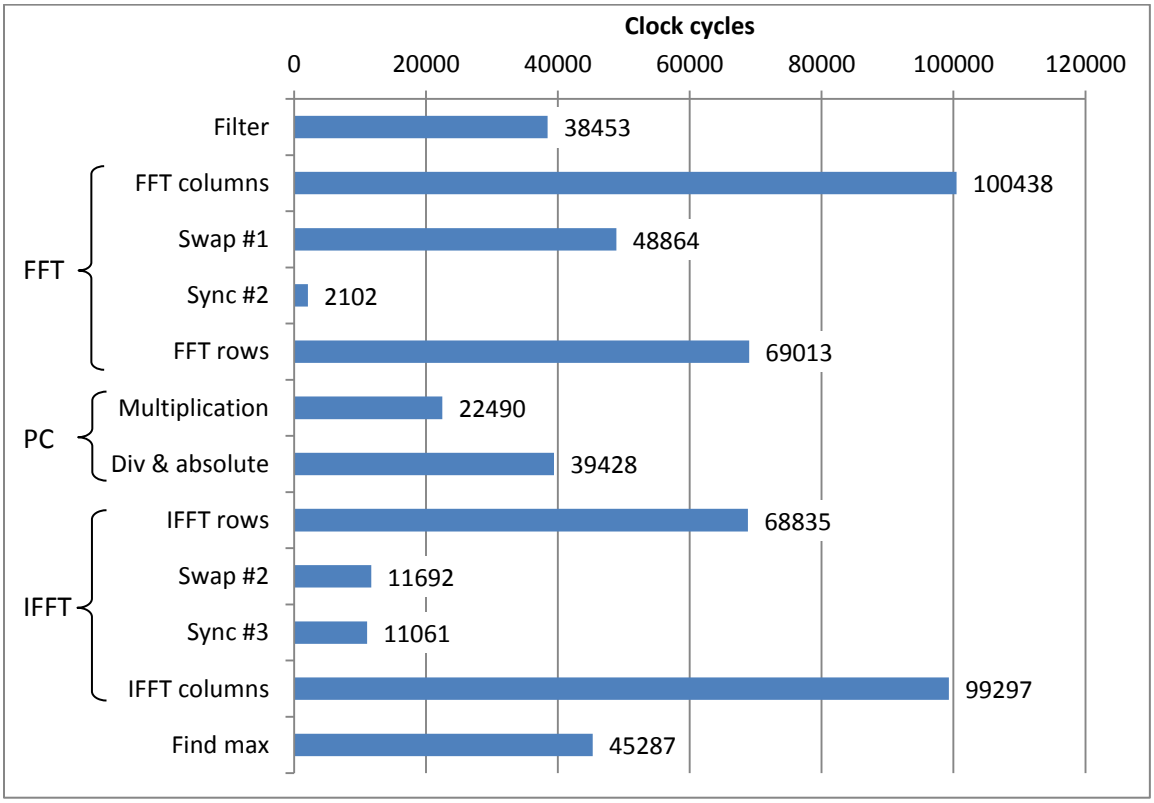


Figure 5.2-2: Parallel performance chart divided into sub-tasks.

In the bullet list below some results have been calculated using the values in figure. The values will be used to calculate parallel utilization as well as for further discussion in the conclusion chapter. The same values have been calculated for the sequential solution and are presented in the next subsection. Some results are however not presented in the sequential performance since these parts do not exist in the sequential solution. These parts are the data movement between the cores, referred to as swap in the figure, plus the synchronization times. The synchronization and

the swap are results from the overhead that occur when the cores are dependent on each other's data.

- In the FFT the columns take 45.5% more time to calculate compared to the rows.
- In the IFFT the columns take 44.3% more time to calculate compared to the rows.
- The FFT calculation takes 0.22 milliseconds, 39.6% of the total time.
- The phase calculation takes 0.06 milliseconds, 11.1% of the total time.
- The IFFT calculation takes 0.19 milliseconds, 34.3% of the total time.
- The synchronization and the swap take up 13.2% of the total computation time.

5.2.2 Sequential Solution

The performance result for the sequential solution is presented in Figure 5.2-3. The algorithm has been divided into smaller parts to show how much each part take of the total computation time. Brackets are used in the figure to show the sub-parts that are included in the FFT, the phase correlation and the IFFT.

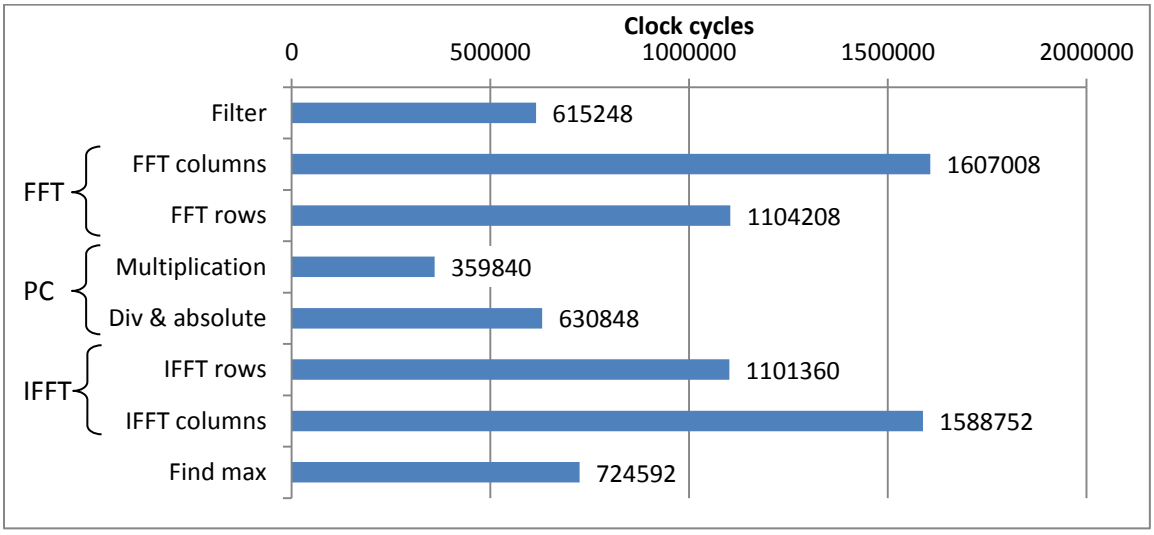


Figure 5.2-3: Sequential performance chart divided into sub-tasks.

The same results that were extracted from Figure 5.2-2 are here extracted from Figure 5.2-3 and presented in the bullet list:

- In the FFT the columns take 45.5% more time to calculate compared to the rows.
- In the IFFT the columns take 44.3% more time to calculate compared to the rows.
- The FFT calculation takes 2.7 milliseconds, 35.1% of the total time.
- The phase calculation takes 1.0 milliseconds, 12.8% of the total time.
- The IFFT calculation takes 2.9 milliseconds, 34.8% of the total time.

5.2.3 Parallel vs. Sequential

When it comes to the overall performance of the parallelization it outperforms the simulated sequential code as seen in Figure 5.2-4. The sequential code executes one measurement cycle in just above 7.7 milliseconds while the parallel executes in just below 0.56 milliseconds.

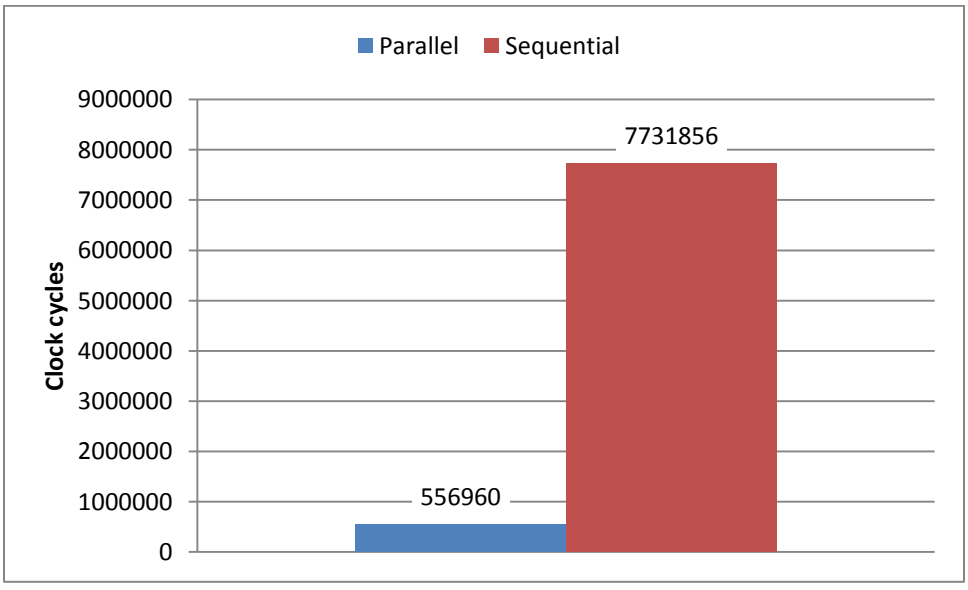


Figure 5.2-4: Comparison of number of cycles used to execute the algorithm in parallel and sequential

The parallel efficiency calculated based on the execution time and number of cores is 86.76%. Figure 5.2-5 shows the parallel efficiency for selected parts of the code. 100% means that the highest possible speedup was achieved. Only the parts that involved interaction between the cores, swap and synchronization, did not achieve a 100% speedup.

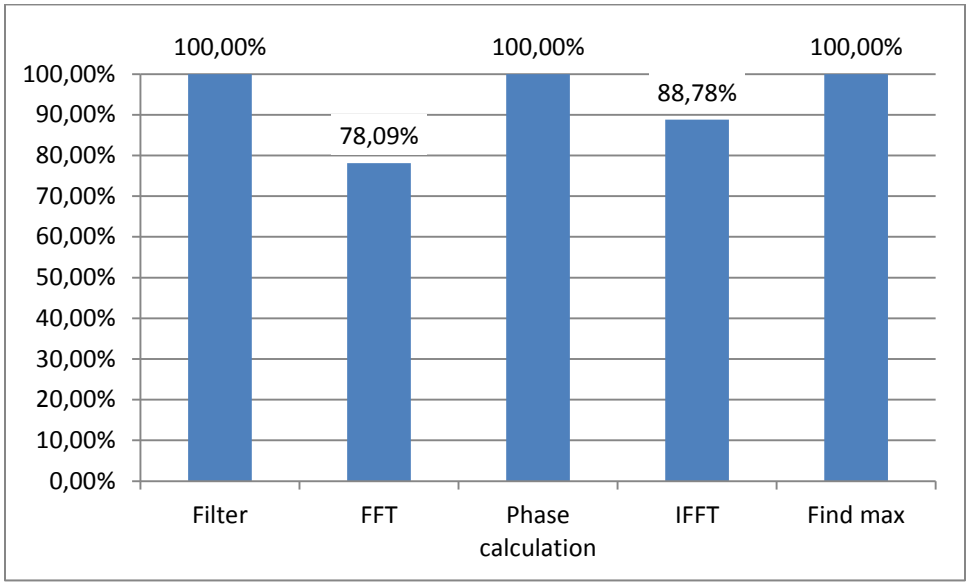


Figure 5.2-5: Parallel efficiency

6 Conclusions and Suggestions to Future Work

In this thesis the Adapteva Epiphany processor was evaluated running a phase correlation algorithm to determine the offset between two consecutive images with sub-pixel accuracy. The chosen application utilizes most of the Epiphany features, such as the shared memory map, interconnection network mesh, DMA and 64 bit floating-point ALU, to ensure a great evaluation.

6.1 Algorithm

The application is based on an algorithm consistent of an image filtering to enhance, phase correlation to find the offset and a sub-pixel estimation to increase the accuracy. This application achieved over an 80 meter test distance a result of 0.5% accuracy. To put in context makes a fault of 5 meters on a distance of 1 kilometer.

Unfortunately the result does not speak the whole truth as there were frame drops detected during the test. These frame drops can have a significant impact on the end result if all the dropped frames happened in a row creating a hole in the measurement.

The result is also worsen because of the uncertainty in the measured pixel size, due to the height of the camera not being confirmed to be the same during the whole test as well as the actual measurement being made on one arbitrary image containing a yardstick. To escape the problems with the pixel size it should be calculated during runtime using the height as an argument.

Transversal Movement and Angular Change

The angular change between two consecutive images, when using a frame rate of 200 Hz, in a typical turn on the highway is $\Delta\alpha \approx 0.012°$, a value likely to be even lower in most cases. The high frame rate itself is required to be able to measure high enough speed, in this case the maximum speed limit in Sweden which is 120 km/h. The method evaluated to detect angular changes, Log polar, does not achieve high enough precision and has in most of the tests a non-existent accuracy. One possible way that may work is to create better conditions for detecting small changes, as of a hundredth of a degree, would be to increase the image resolution lowering the pixel size making it more possible to detect a change between two images. This does on the other hand increase the computational requirements of the system.

Sub-pixel

The chosen sub-pixel method proved to give good results during internal testing. Other sub-pixel methods were however not evaluated and it is possible that other methods would perform even better. An interesting approach which was mentioned in 2.5.3 is to determine the sub-pixel shift in the frequency domain. This along with other methods could be evaluated for further improvements of the algorithm.

6.1.1 Future Work

Variable Frame Rate

A possibility that would decrease the computational requirements at least momentarily is to have a variable frame rate. As the frame rate is dependent on the speed, images must be taken often enough that they overlap, a lower speed requires a lower minimum frame rate. This phenomenon opens up the possibility to not only lower the computational requirements momentarily but also make it possible to detect transversal and angular changes.

Having the ability to detect transversal and angular changes at lower speeds will perhaps open up the possibilities for systems requiring high precision position estimation such as automatic parking systems.

Absolute Position

Since the developed application results in a relative position it would be interesting to look at possibilities where the result is combined with an external sensor that provides an absolute position, e.g. GPS. This would result in an overall system that does not require the GPS to be activated all the time. If this would be a better system, either it be cost or energy wise, will have to be evaluated.

Image Optimization

Within this thesis work optimization of the application settings such as image size, required overlap, camera height, etc. was just briefly touched. What was evaluated was that the settings we used worked but however the limits of which the settings could be pushed to were never tested. It would be interesting to evaluate and determine these limits as they would show what the minimum computational requirements are as well as the requirements off the camera, perhaps resulting in a low-end and low cost product.

6.2 Hardware and Implementation

Hardware Design

The Epiphany chip's simple hardware design makes it easy to understand the functionalities of the architecture while also simplifying the software design. The shared memory map simplifies the C code when moving memory between cores as no extra communication between the cores is needed. When building more complex applications, utilizing the MIMD architecture, the *mutex* functionality, included in *eLib*, can be used to ensure the data is protected.

The Epiphany chip had 32 kilobytes of memory in each core. The variant with 64 kilobytes of memory would enable the application to support higher resolution images and still perform on millisecond level.

Environment

The developed application was completely written as common code using a core identification variable to change behavior of the software to each core which resulted in a SIMD application. The possibility to use the project structure where every core has one dedicated Eclipse project was not used. But in projects were cores run different applications, which would have resulted in a MIMD structure, it may be hard to get a good overview of all cores. The offered debug possibilities used during the development of the software were good. Both stepping in all cores at once and in one core at a time was possible. Also to visualize current state of memory was possible. Unfortunately the debugging tool was a bit unstable at times due to high system resource demands when debugging 16 cores at the same time.

Development board

The development board only supported a connection to the PC to download code and for debugging. In order to allow this platform to be a part of more sophisticated systems it would be good to have support for more external connections. For this project the generous amount of SDRAM was an adequate replacement instead of connecting the camera directly to the board.

Real-time requirement

The final results show that the required calculation time is much less than the previously calculated limit of 4.9 milliseconds. This gives many possibilities to either extend functionality or broaden the application areas. The time left can be spent to process more data or apply other calculations to the currently loaded data. On the other hand, using the current algorithm and data sizes, it would be possible to increase the frame rate up to almost nine times which could open up for new application areas. This solution can then support a frame rate up to 1800 frames per second resulting in a maximum speed of around 1060 km/h using the same assumptions as in 3.3.3. The current solution however relied on pre-loaded images to the external SDRAM. To allow for such high frame rates the images also have to be loaded to the processor in real-time.

Fast Fourier Transform

The transformation is favorable to parallelize. This because the two dimensional transform shall be applied to both rows and columns where every row or column is interpreted as an independent signal as mentioned in 2.4. Rows and columns can be distributed among the available cores and only one data exchange need to occur when switching from rows to columns in each core. This parallelization is easy to scale up and adapt to a larger chip. The only limitation is that the number of rows and columns shall be greater or equal to the number of cores. Else the transform of a single row or column has to be parallelized as well.

When transforming by columns each core has 2 columns, 256 pixels long, and when transforming by rows each core has 16 rows where each row contains 32 pixels. The theoretical time difference between columns and rows can be calculated as the fast Fourier transform requires $O(N\text{Log}_2 N)$ operations. Based on the number of rows and columns the following amount of operations is needed;

$$2 \cdot 256 \cdot \log_2 256 = 4096$$

$$16 \cdot 32 \cdot \log_2 32 = 2560$$

Calculating by columns will therefore theoretically require 60% more operations than calculating by rows.

The results does however show that it takes around 45 % more clock cycles, as can be seen in Figure 5.2-2, to transform the image by columns than by rows. The difference from the theoretical value can be explained by the overhead created by the extra function calls required when calculating by rows.

DMA

Using a DMA channel to load image data to the core's local memory as a background process removed the need to allocate processing time for this and reduced the total execution time. The disadvantage with this method is that the image loaded by DMA will first be used next calculation cycle. This will introduce a delay of the result with one image period of 5 milliseconds. If DMA is preferable depends on if it is most important to keep the execution time or the delay of the results as short as possible.

6.2.1 Future Work

A final solution must include interface to the camera, some kind of host system managing everything except the heavy calculations done by the Epiphany chip and interface to deliver sensor data.

To interconnect with a camera there are three interesting alternatives; Camera Link, GigE Vision and USB. Both Camera Link and GigE are dedicated to vision applications while USB is a well-known standard used by many applications.

The Epiphany chip is built to assist an application by processing large amount of data in parallel, not to run the entire application. A final solution should have a combination of a host core and the Epiphany chip. The host core shall handle tasks such as initialization, controlling and handling results.

DMA

Currently DMA is only used to load a new image for next calculation. But DMA could also be used in the swap needed between the transform of columns and rows. When a core has transformed one column, the pixels of this column could be transferred to their new destinations by a DMA channel while the core continues to transform the next column. However, this requires multiple DMA transactions to be created because the pixels have different destination cores. How much extra calculations this requires and if this will reduce the total execution time is unknown. However, it will open up resources to run other applications simultaneously.

7 References

[1] O'Toole, Randal. Gridlock: Why We're Stuck in Traffic and What to do About It. Washington: Cato Institute, 2009.

[2] Marinescu**,** A., Dragos C.. Towards improving positioning with the use of GPS and EGNOS. Electronics and Telecommunications (ISETC). 2010; 9, pp. 245-248.

[3] Lobur, M., Darnobyt, Y.. Car speed measurement based on ultrasonic Doppler's ground speed sensors. CAD Systems in Microelectronics (CADSM). 2011; 11, pp. 392-393.

[4] Beasley, P.D.L., Simmons, I.M., Stove, A.G.. High accuracy radar speedometer. Automotive Sensors, IEE Colloquium. 1992, pp. 8/1-8/5.

[5] Automobile Doppler speedometer. Kleinhempel, W. Ottawa : Deutsche Aerospace AG, Ulm, 1993. Vehicle Navigation and Information Systems Conference. ss. 509-512.

[6] Nagasaku, T., Kogo, K., Shinoda, H., Kondoh, H., Muto, Y., Yamamoto, A., and Yoshikawa, T.. 77GHz low-cost single-chip radar sensor for automotive ground speed detection. Compound Semiconductor Integrated Circuits Symposium. 2008, pp. 1-4.

[7] Cocco, L., Rapuano, S.. Accurate Speed Measurement Methodologies for Formula One Cars. Instrumentation and Measurement Technology Conference Proceedings. 2007, pp. 1-6.

[8] Horn, J., Bachmann, A., Dang. T.. A Fusion Approach for Image-Based Measurement of Speed Over Ground. Multisensor Fusion and Integration for Intelligent Systems. 2006, pp. 261-266.

[9] Jackson, J.D., Callahan, D.W., Marstrander, J.. A Rationale for the use of Optical Mice Chips for Economic and Accurate Vehicle Tracking. Automation Science and Engineering. 2007, pp. 939-944.

[10] Jackson, J.D., Callahan D.W., Marstrander, J., Richardson, J.A., Hakima, I.K.. Low-Cost Precision Tracking of Vehicles using Optical Navigation Technology. WSEAS Signal processing, computational geometry and artificial vision. 2008; 8, pp. 164-171.

[11] Correvit Optical Sensor Technology. Last updated June 23 2010, viewed May 27 2012. http://www.corrsys-datron.com/technology.htm.

[12] Nagai, I., Watanabe, K., Nagatani, K., Yoshida, K. Noncontact position estimation device with optical sensor and laser sources for mobile robots traversing slippery terrains. Conference on Intelligent Robots and Systems (IROS). 2010, p. 3422 - 3427.

[13] Xie W., Zhou T., Li L.. Application of phase only correlation in velocity measurement based on FPGA. Image and Signal Processing (CISP). 2011; 4, pp. 1301-1304.

[14] Matungka, R.. Studies on Log-polar transform for image registration and improvements using adaptive sampling and logarithmic spiral. Columbus: The Ohio State University, 2009.

[15] Balci, M., Foroosh, H.. Subpixel estimation of shifts directly in the fourier domain. IEEE Transactions on Image Processing. 2006; 15 (7), pp. 1965-1972.

[16] Foroosh, H., Zerubia, J.B., Berthod, M.. Extension of Phase Correlation to subpixel registration. IEEE Transactions on Image Processing. 2002; 11 (3), pp. 188-200.

[17] Butts, M., Jones, A.M., Wasson, P.. A Structural Object Programming Model, Architecture, Chip and Tools for Reconfigurable Computing. Annual IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM. 2007; 15, pp. 55-64.

[18] Gwennap, L.. Adapteva: More Flops, Less Watts, Epiphany Offers Floating-Point Accelerators for Mobile Processors. Mircoprocessor Report. June, 2011.

[19] Horisontalkurvor. Vägverket publication 2004:80, pp. 59-62.

[20] Altera. High Speed Mezzanine Card (HSMC). Specification. Version 1.7. San Diego, June 2009.

[21] Origin of Quake3's Fast InvSqrt() – Part One(2006). Last updated March 21 2007, viewed May 18 2012. http://www.beyond3d.com/content/articles/8/.

[22] Origin of Quake3's Fast InvSqrt() – Part Two(2006). Last updated March 21 2007, viewed May 18 2012. http://www.beyond3d.com/content/articles/15/.

[23] Fast Fourier Transform, June 1993. Viewed April 10 2012. http://paulbourke.net/miscellaneous/dft/.