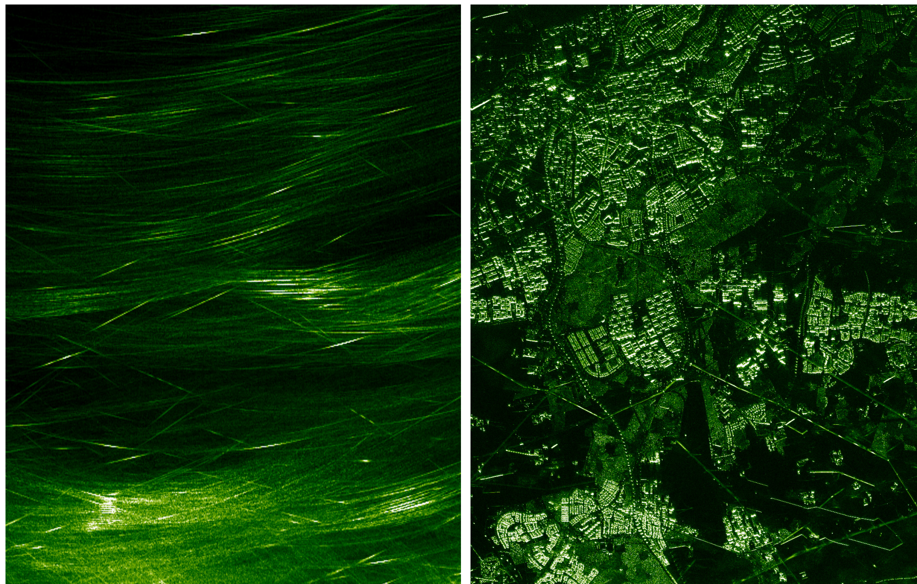
CHALMERS



Optimization of Fast Factorized Backprojection execution performance

Master of Science Thesis in Computer Science: Algorithms, Languages and Logic

Christian Lidberg and Johan Olin

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden, 18/06/2012

The Author grants to Chalmers University of Technology and University of Gothenburg the non-exclusive right to publish the Work electronically and in a non-commercial purpose make it accessible on the Internet. The Author warrants that he/she is the author to the Work, and warrants that the Work does not contain text, pictures or other material that violates copyright law.
The Author shall, when transferring the rights of the Work to a third party (for example a publisher or a company), acknowledge the third party about this agreement. If the Author has signed a copyright agreement with a third party regarding the Work, the Author warrants hereby that he/she has obtained any necessary permission from this third party to let Chalmers University of Technology and University of Gothenburg store the Work electronically and make it accessible on the Internet.

Optimization of Fast Factorized Backprojection execution performance

CHRISTIAN LIDBERG
JOHAN OLIN

©CHRISTIAN LIDBERG, June 18, 2012
©JOHAN OLIN, June 18, 2012

Examiner: SALLY MCKEE

Chalmers University of Technology
Department of Computer Science & Engineering
SE-412 96 Göteborg
Sweden
Telephone +46 (0)31-772 1000

SAR raw data and the backprojected image, used with courtesy of FOI

Deparment of Computer Science & Engineering
Göteborg, Sweden, June 18, 2012

Abstract

Real-time signal processing often requires high computational performance from the signal processing system. In order to increase performance computer systems have moved from the traditional one-core CPU to multi-core systems. This requires, however, parallel software to use all the available performance. It is therefore not only important to have efficient algorithms but also efficient parallel implementations of them.

The signal processing for a low-frequency synthetic aperture radar system, used to create high-resolution radar maps of the ground, is studied in this master's thesis. The datasets used to create the maps are often very large and therefore the computational burden is high. The efficient Fast factorized backprojection algorithm is used to create the images but still the images cannot be produced in real-time on a single core system.

This thesis describes how the Fast factorized backprojection is optimized and parallelized. OpenMP and vector instructions are used to reach real-time performance on a multi-core platform, for small and medium sized images.

Acknowledgements

We would like to thank our supervisors at Saab AB, Anders Åhlander, Jonas Lindgren and Hoai Hoang Bengtsson, and our supervisor at Chalmers University of Technology, Sally McKee, for their help and feedback throughout the project.

Contents

1 Introduction **1**
1.1 Background . . . 1
1.2 Scope of work . . . 2

2 Radar **3**
2.1 Radar basics . . . 3
2.1.1 Range determination . . . 3
2.1.2 Sampling . . . 4
2.1.3 Frequency . . . 4
2.1.4 Resolution and pulse compression . . . 5
2.2 SAR . . . 6
2.2.1 Geometry . . . 7

3 SAR signal processing **10**
3.1 Radar echo model . . . 10
3.2 Image creation in the time domain . . . 12
3.2.1 Global backprojection . . . 12
3.2.2 Fast factorized backprojection . . . 14
3.3 The SAR system used in this project . . . 16
3.3.1 Signal Processing . . . 17
3.3.2 Data structure . . . 18
3.3.3 Image creation . . . 19
3.3.4 Interpolation . . . 21
3.3.5 Data representation in memory . . . 21
3.3.6 Adressing pattern . . . 22

4 Parallelism **24**
4.1 Motivation . . . 24
4.2 Limitations . . . 25
4.3 Granularity . . . 26
4.4 Exploiting parallelism . . . 26
4.4.1 Instruction-level parallelism . . . 27

4.4.2 Data parallelism 27
4.4.3 Task parallelism 28

5 OpenMP **29**
5.1 Execution and memory model 29
5.2 Programming with OpenMP 29
5.2.1 Internal control variables 30
5.2.2 OpenMP directives 30
5.2.3 Example 32
5.2.4 Overhead 33
5.2.5 Behaviour of an OpenMP program 34

6 Optimizing and parallelizing the FFB **35**
6.1 Data collection 35
6.2 Merging 36
6.3 Vectorization 37

7 Test environment **39**

8 Result **42**
8.1 Small test case with delay 43
8.2 Small test case without delay 44
8.3 Medium test case with delay 45
8.4 Medium test case without delay 46

9 Performance analysis **47**
9.1 FLOPS 47
9.2 Cache utilization 48
9.3 Cycles per instruction 49

10 Discussion **50**
10.1 Performance 50
10.1.1 Small system 50
10.1.2 Medium system 51
10.2 Portability 51
10.3 Scalability 52
10.4 Engineering efficiency 52

11 Conclusion and further work **54**

Bibliography **56**

1

Introduction

This master thesis is carried out at Chalmers University of Technology and Saab AB. The goal of this thesis is to optimize and parallelize an image creation algorithm to reach real-time performance with respect to portability, scalability and engineer efficiency.

1.1 Background

Synthetic Aperture Radar (SAR) is a type of radar that simulates a long antenna, by using the antennas movement, to create high-resolution images of the ground. SAR can operate on lower frequencies, than conventional radar, which can penetrate foliage and detect camouflaged objects in forests.

The images for a SAR system can advantageously be created in the time domain to be able to compensate for non-linear flight-tracks. An effective algorithm to create the images in the time domain is the Fast Factorized Backprojection (FFB) algorithm that uses approximations and factorization to reduce the computational burden. However, the large data sets produced by a SAR system still makes it hard to reach the high performance that is required for real-time image creation, i.e. when the images is created during the flight. There is however, an unavoidable latency from when the last data is collected until the image is finished, so to create images in real-time the processing of the next image need to catch up the processing during the data collection.

1.2 Scope of work

Saab AB has developed and implemented a version of the FFB, utilizing one core, and a test bench written in C++, which will be used in this project. This version has been developed to obtain the desired image quality. Since the code produce an image with desired quality no optimizations that change the image has been done e.g. rearranging floating point operations.

The current version of the FFB does not meet the real-time requirements. The optimizations and parallelizations carried out are done to enable processing of data and image creation in real-time. This must be taken in consideration when parallelizing the algorithm since we do not have the whole data set at once but collect data continuously during the flight.

The project will include studies of the FFB algorithm, parallelization- and optimization techniques. The existing C++ implementation is rewritten and adapted to one or more techniques. The algorithm is then evaluated with respect to performance, portability, scalability and engineering efficiency.

Even though a big motivation for using image creation algorithms in the time domain is the ability to compensate for non-linear flight tracks this will not be applied in this scope of work. The evaluation of the result will only consider linear flight-tracks.

The report starts with a chapter with basic theory for radar and SAR, and then signal processing for creating SAR images is described. Basic theory about parallelism, and how to exploit it, is followed by a description of an API to add parallelism to C++ code before it is described how the FFB is optimized and parallelized. Before discussion and conclusions the results for two test cases along with a performance analysis is given.

2

Radar

2.1 Radar basics

A Radio Detection and Ranging (radar) system emits electromagnetic pulses and detects the returned echo from reflecting objects. From the echo the radar can determine the distance and direction of the target. Distance is easily calculated since electromagnetic energy travels at constant speed, 300 000 km/s. The transmitter and receiver is usually, but not always, placed together. If that is the case, the power received by the antenna, P_r, is given by the following radar equation.

$$P_r = \frac{P_t G_t A_r \sigma F^4}{(4\pi)^2 R^4} \tag{2.1}$$

Where P_t is the transmitter power, G_t is the gain of the transmitting antenna, A_r is the effective aperture of the receiving antenna, σ is the radar cross section of the target, F is the pattern propagation factor and R is the distance between transmitter/receiver and the target. It is important to notice that the power will decrease severely with the distance to the target.

2.1.1 Range determination

The slant range R, i.e. the line of sight distance between the radar and the target, is calculated from the time, t, it takes the transmitted pulse to return and the wave velocity c.

$$R = \frac{ct}{2} \tag{2.2}$$

However, the radar cannot transmit and receive pulses at the same time because the receiver need to reset the timing system each time a pulse is transmitted to determine the distance. So if an echo returns after a new pulse has been transmitted it will incorrectly be identified as a target much closer to the radar than it really is. Therefore, to determine the range to a target the receiving time must be long enough for the echo to return. The receiving time depends on the pulse width, P_w, which is the duration the pulse is transmitted, and the pulse repetition frequency (PRF). The PRF is the number of pulses that are transmitted every second. The receiving time will be the pulse repetition time (PRT), which is 1/PRF, minus the pulse width. This gives that the maximum range, R_{max}, that can be unambiguously determined is:

$$R_{max} = \frac{c(PRT - P_w)}{2} \tag{2.3}$$

For the Equation (2.3) to be valid the power of the received signal, as described by Equation (2.1), must be larger than the noise in the receiver. For an ideal receiver the noise is kTB, where k is the Boltzmann's constant, T is the temperature and B is the receiver bandwidth[1].

2.1.2 Sampling

The receiver samples the received signal according to the Nyquist sampling theorem which states that if a signal is sampled with a frequency twice as large as the bandwidth the analog signal can be perfectly reconstructed (under ideal conditions). If the signal is sampled complex, i.e. both the amplitude and phase information is stored, the signal only have to be sampled with the bandwidth frequency. Each sample will then correspond to a range so each sample for a pulse will be stored in its range bin. This is done for all the pulses so the radar data will be stored in a matrix with pulses on one axis and range bins on the other.

2.1.3 Frequency

The frequency used in a radar system is determined by many factors. The size of the radar system is proportional to the wavelength because a longer wavelength needs a larger antenna to obtain a given lobe width. Thus, a high frequency radar system will be small and light which is an advantage for airborne systems. On the other hand, a larger system allows higher power because of better cooling and that increase the detection range. The frequency also decides how much an object of a certain size affect the signal. Shorter wavelengths will be more affected by small objects so for example weather radar need a wavelength that will be reflected by water drops which is around 6 cm (or 5 GHz). If it is undesirable for the radar to be affected by a certain object, e.g. clouds, the wavelength can be increased. If a long enough wavelength is used the signal

can remain unaffected by leaves and branches which make it possible to see through foliage which enables detection of hidden objects in forests.

2.1.4 Resolution and pulse compression

The resolution of a radar system is the ability to distinguish between two targets that are close to each other in either range or azimuth. The range resolution, Δr, is the smallest distance where two targets on the same azimuth can be separated. To separate two targets the time it takes for the pulse to travel back and forth between the two targets need to be larger than the pulse width[2].

$$\Delta r = \frac{cP_w}{2} \tag{2.4}$$

Using a lower pulse width will increase the range resolution but at the same time, the maximum detection range will be lower since the received signal will disappear in the noise faster than if a longer pulse width was used. To come around this problem pulse compression, which is a method to get the advantages of both the high energy of a long pulse width and the high resolution of a short pulse width, can be used. The transmitted pulse is changed so that each part of it has a unique frequency, which can be done by either frequency or phase modulation. The most commonly used modulation is linear frequency modulation, also called chirp pulse, where the frequency is changed with constant speed throughout the whole pulse. See Figure 2.1 for an example of a chirp pulse.

Figure 2.1: Chirp pulse

The receiver demodulates the signal with a compression filter that adjusts the relative phases so that a narrow pulse is created. The resulting pulse width is then $\frac{1}{B}$ where B is the transmitted pulse bandwidth.

The azimuth resolution in meter, Δa, is the smallest distance that two targets can be separated on the same range. It depends on the slant range to the target and the lobe width, which approximately is the wavelength, λ, divided by the length of the antenna, L.

$$\Delta a \approx \frac{\lambda R}{L} \tag{2.5}$$

Therefore, either a higher frequency or a larger antenna can obtain an increase in azimuth resolution.

2.2 SAR

The azimuth resolution for long distances is often insufficient since if a too high frequency is used the signal will be reflected by objects that are not of interest, e.g. clouds, or an unrealistically long antenna has to be used. Synthetic aperture radar (SAR) is a type of radar that circumvents that problem by simulating a long antenna by using the antennas movement. The synthetic aperture is created by using the pulses from the real physical aperture that has been collected during the flight. The radar operates on the same PRF the whole flight and the responses is stored in the same way as mentioned above, i.e. in a matrix with pulses on one axis and range bins on the other. All the pulses are processed together to create a high-resolution image.

There are three different modes for SAR, stripmap, spotlight and scan, see Figure 2.2. Since stripmap is the mode used in this project, the theory for the other modes will not be discussed.

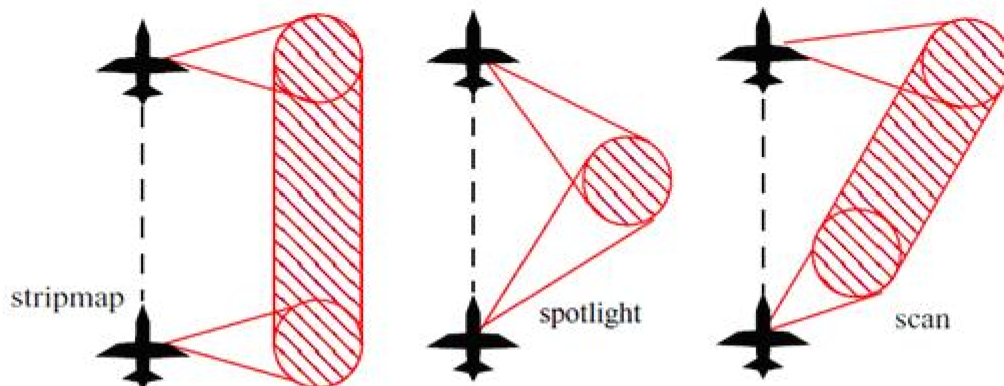


Figure 2.2: Different methods for SAR[3].

The echoes from different targets can be separated since they have different Doppler frequency because of the movement of the radar, see Figure 2.3.

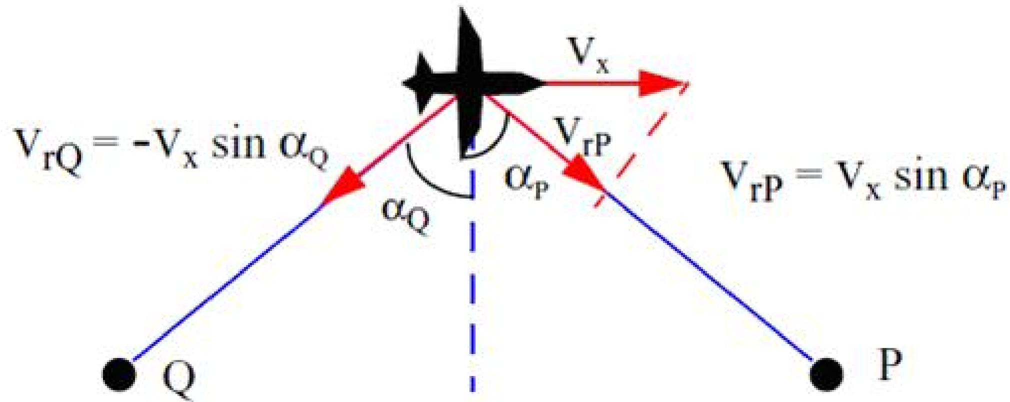


Figure 2.3: The targets Q and P will have different relative velocity, V_r, to the radar which lead to that they also have different Doppler frequency[3].

The length of the synthetic aperture is limited by the antennas physical lobe width since the region that is imaged need to be in the antenna beam for the entire time as the aircraft flies by it. The theoretical azimuth resolution obtainable is half the length of the antenna[4].

2.2.1 Geometry

The start-stop approximation is used so the movement of the radar between transmitting a pulse and receiving it is assumed to be zero. The reason for this simplification is to leave out the change in position in the geometry calculations and often the movement is negligible since the pulse travels at the speed of light. The different targets can still be separated with the Doppler frequency since it can be seen as a change in the range between the pulses. The echo from a target has a different distance to travel for each pulse so the phase for each pulse will be different.

The airborne SAR travels linear along the x-axis at a constant height $z = h$. The antenna has a lobe directed against the y-axis with a lobe width of 2θ. On the ground there is a single point target P located at (x_0,y_0,0), which can be seen in Figure 2.4.

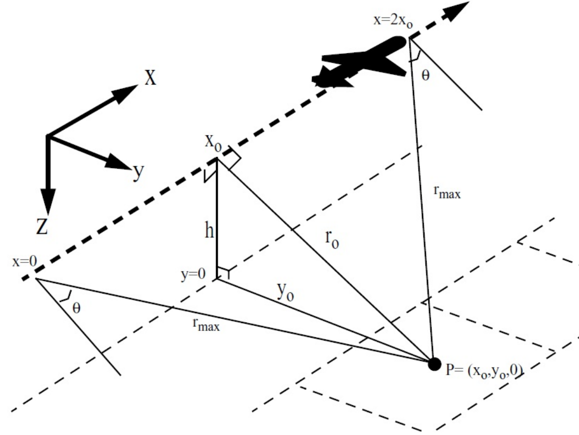


Figure 2.4: Geometry for a simple SAR-model[3].

The radar transmits a pulse, with centre frequency f_c and wavelength $\lambda_c = \frac{c}{fc}$, that reflects from P and is sampled in the receiver. When the antenna is at position $x = 0$ the target P is located at the edge of the lobe, i.e. at angle θ from the centre of the lobe and at range r_{max}. The echo is sampled in the receiver and the amplitude and the phase is stored in the range bin that corresponds to r_{max}. When the antenna moves along the track the distance to P will decrease until it reaches r_0, when the antenna is at position x_0, and then it will increase until it reaches r_{max} again at the other side. In the data matrix the echo from P will look like a hyperbola, see Figure 2.5.

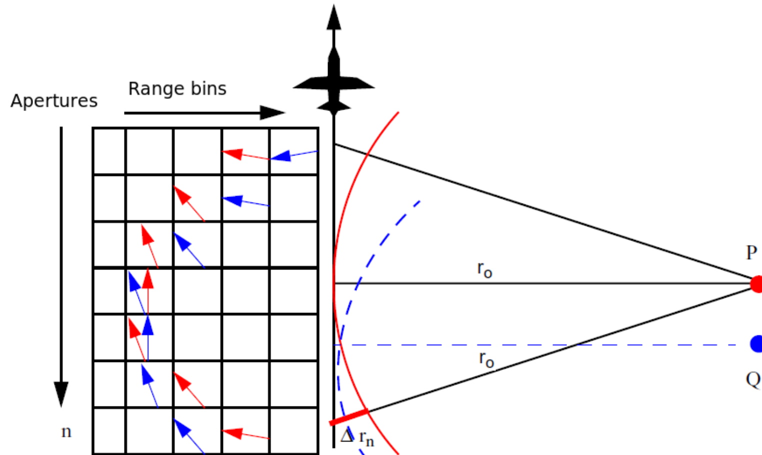


Figure 2.5: The behavior of the echo, from two targets, through the aperture positions. The arrows represent the phase for each aperture position[3].

The phase, ϕ_n, will vary between a pulse n and the centre aperture according to the formula[2]

$$\phi_n = 2\Delta r_n \frac{2\pi}{\lambda_c} \tag{2.6}$$

Focusing means that for each point, or pixel, the echo is phase shifted with the phase ϕ_n for each aperture position n. The phase shift for each echo will be different for each point so each point can be separated in azimuth. If there is a target located in the point the amplitude will be high since constructive interference will occur due to the echoes having the same phase. If no target is located in the point the amplitude will decrease since destructive interference will occur.

3

SAR signal processing

Signal processing for SAR has been solved earlier with methods in the frequency domain, e.g. the Fourier-Hankell- and Range-Migrationmethod that are very computational efficient. The drawback has been that they are developed for a linear flight-track and to handle a nonlinear flight-track the computational efficiency would be much lower. The fact that it could not handle a nonlinear flight-track is not an issue with microwave SAR since it uses a small lobe width. However, with VHF SAR where broader lobes are used this is a big issue[2][5].

SAR signal processing can also be solved in the time domain with the ability to handle nonlinear flight-tracks, which is discussed further in Section 3.2[6].

3.1 Radar echo model

A radar echo model can be created with a number of assumptions. It is assumed that it is a monostatic radar, i.e. the transmitter and receiver are located at the same place. It is also assumed that the wave velocity c is constant. The ground is assumed to be a collection of single-scattering objects so that superposition can be applied. This gives the model in Figure 3.1.

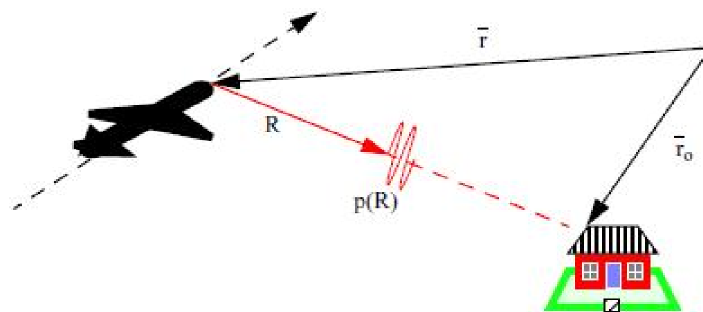


Figure 3.1: The radar echo model. The pulse $p(R)$ is reflected and received[3].

For a single object at position r_0 and with the radar antenna positioned at r we get the radar echo, $g(r,R)$, as a function of the antennas position and the pulse delay.

$$g(r,R) = \frac{p(R - |r - r_0|)}{|r - r_0|^2} \tag{3.1}$$

where $p(R)$ is the band-limited radar pulse after pulse compression and R is given by Equation (2.2).

The equation is limited to the case when the object and antenna is stationary to each other but since the start-stop approximation is used, as described in Section 2.2.1, that is not a problem. There are also some scaling factors of the echo, which is ignored in the equation that for simplicity are assumed equal to 1, however practically these corrections will be decided with stationary phase methods [7]. This model is applicable to most SAR-problems.

If a linear flight-track is assumed the equation can be simplified because then a cylindrical symmetry along the flight-track axis can be used. The cylindrical coordinates (ρ,θ,x) is used, where x is the flight-track axis, ρ is the radius and θ is the angle around the x-axis, see Figure 3.2.

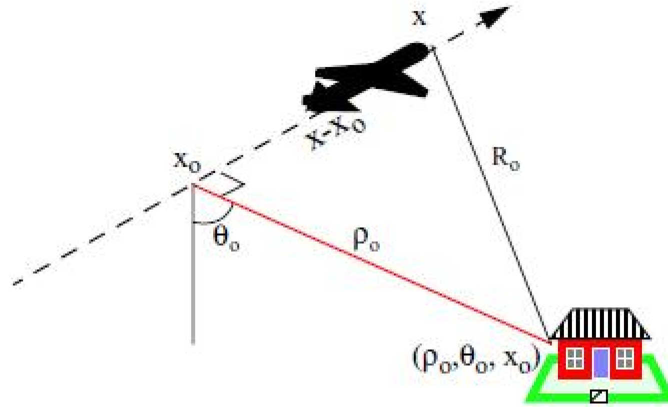


Figure 3.2: Model with cylindrical geometry[3].

This lead to the following function for the radar echo data:

$$g(x,R) = \frac{p(R - \sqrt{(x-x_0)^2 + \rho_0^2})}{(\sqrt{(x-x_0)^2 + \rho_0^2})^2} \tag{3.2}$$

The range history for a single object is hyperbolically dependent on x. The single objects angle coordinate θ is not included in the formula, which means that objects on both sides of the antenna is contributing to the same pixel which lead to right-left as well as topographic ambiguities. The advantage with the rotation symmetry is though that we can produce a SAR-image from a linear flight-track without any knowledge of the ground topography[2][5].

3.2 Image creation in the time domain

The images are created with backprojection in the time domain. Global backprojection is the most basic backprojection method but it is computationally heavy, so fast factorized backprojection, which uses approximations, can be used instead.

3.2.1 Global backprojection

Using two cylindrical coordinates, a radar map can be created where every object has the position (x, ρ) where x is the flight-track position and ρ is the distance to the object.

With the SAR radar echo data Equation (3.2) this gives the following backprojected signal $h(x,\rho)$.

$$h(x,\rho) = \int_{-\infty}^{\infty} g(x',R)R dx' \tag{3.3}$$

Where $g(x,R)$ is the radar echo data as a function of the flight-track position x and range R, $R = \sqrt{(x'-x)^2 + \rho^2}$. The radar echo data has been sampled from the original continuous signal according to the Nyquist criteria and can therefore be fully reproduced. The correct value for every position (x, ρ), in the resulting image, is given by summarizing the integral over the aperture positions x', with the interpolated and phase shifted value, at the specific range R. R corresponds to the range between the antenna and the image position.

Figure 3.3 illustrates how one image position, $h(x, \rho)$ is created by integrating over the contributing values.

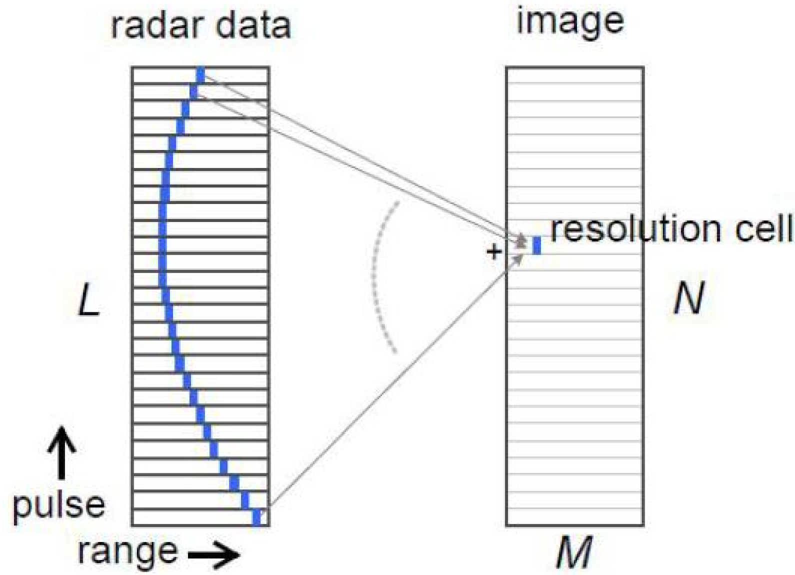


Figure 3.3: Illustration over how GBP integrates over the contributing aperture positions to create one image position[8].

When the backprojection is modified to handle a non-linear flight track the focusing becomes dependent on the ground topography if a one-dimensional aperture is used. The reason for this is that that the data inversion becomes a three-parameter problem that requires a two-dimensional aperture to solve exactly. Therefore the backprojection integral is expressed as a function of the three-dimensional image position vector r_0 according to

$$h(r_0) = \int g(x',R)Rdx' \tag{3.4}$$

where $R = R(x') = |r(x') - r_0|$ is the range between the antenna and the image position. This equation can handle all track geometry but exact inversion is only possible for a linear flight track.

The global backprojection is very inefficient considering the number of operations. Consider an image with $M * N$ pixels and an aperture with L positions. For every received aperture and pixel position, the range between the antenna and the pixel need to be calculated, then the radar echo need to be interpolated to find the correct value to add to the image. This is proportional to $L * M * N$ operations which with large images is very computationally heavy. Global backprojection is still efficient for small images

and has the advantage that it only need to hold the radar echos until they have been backprojected and therefore saves memory[2][5].

3.2.2 Fast factorized backprojection

To speed-up global backprojection other backprojection methods can be used. If another coordinate system is used it can be seen that it is unnecessary to backproject the aperture data to all image pixels for every new aperture position. This can be seen by looking at the geometry in Figure 3.4.

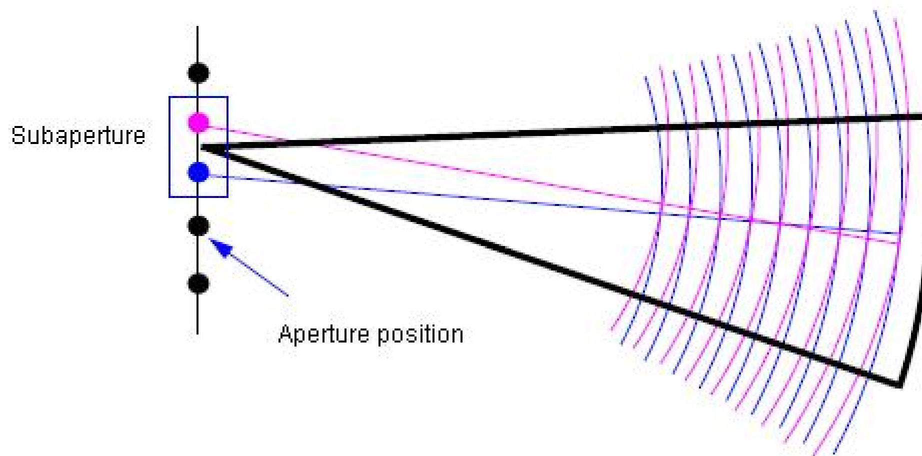


Figure 3.4: Illustration of the data redundancy if all aperture data is stored[8].

The figure shows concentric range circles centered in two different aperture positions. It is along these circles that the radar echo data is projected over the image. The circles are matched along the beam centre line, which corresponds to the beam focusing in this direction. Successive aperture positions have almost the same circular pattern inside an angle sector. This means that one single range data line can be used to represent the angular sector with minor error. For points further away from this line the circular pattern differ, because of increasing phase errors. Polar coordinates are used to represent the data, in the image plane, in an effective way. The bigger the aperture, the smaller is the angular sector that can be represented by a single data line. On the other hand, the amount of data lines that is needed to represent a subimage is increased with the subapertures length.

The principle is to split up the calculations, using polar coordinates, in several iterations instead of computing the Cartesian image with full resolution at once. In every iteration, the angular resolution increases by combining more and more apertures and creating smaller and smaller lobes, which represents the different angles out from the aperture[2][5].

If raw data with L aperture positions are used with a base n, that is the number of subapertures that is combined in every processing stage, the number of subapertures after the first processing stage will be L/n. If every L subaperture represent different ranges in a large lobe at approximately θ degrees then every new L/n subaperture will have n data lines, each representing a smaller lobe of θ/n degrees in different directions. This goes on until the full image is created with L data lines each representing a small lobe of θ/L degrees in different directions. In Figure 3.5, it can be see how the subapertures are merged together and for every iteration creating a larger subaperture. In every iteration, each subaperture get better angular resolution.

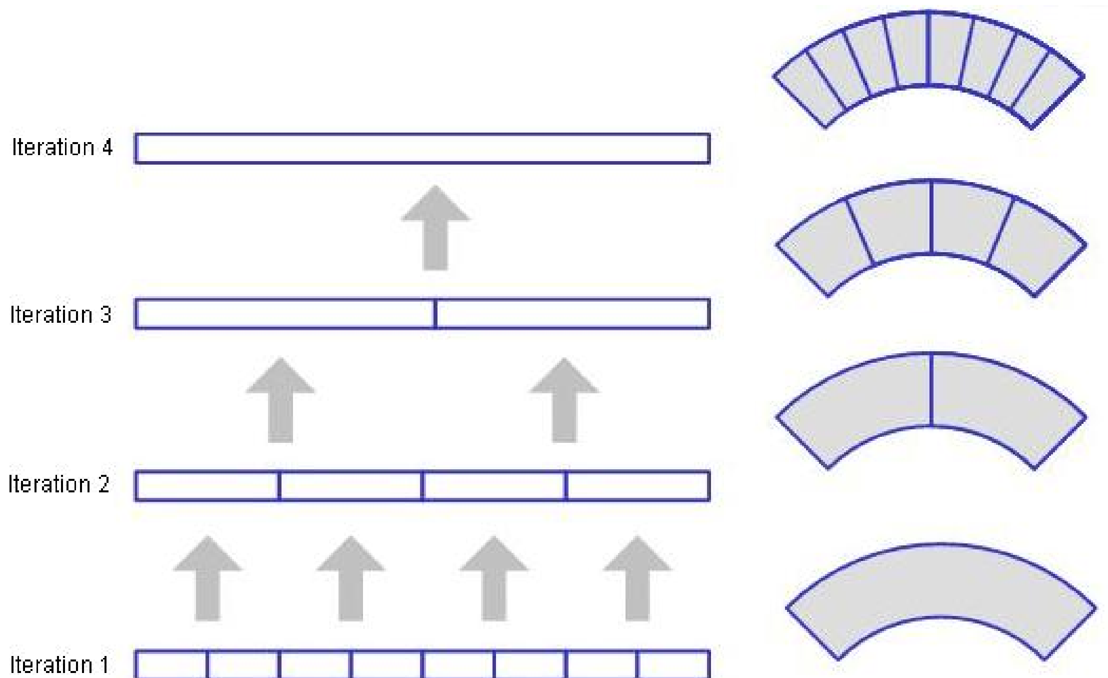


Figure 3.5: As the subapertures are merged together and become larger, the azimuth resolution improves[8].

In Figure 3.6 it can be see how an image position is created with FFB compared to the GBP in Figure 3.3.

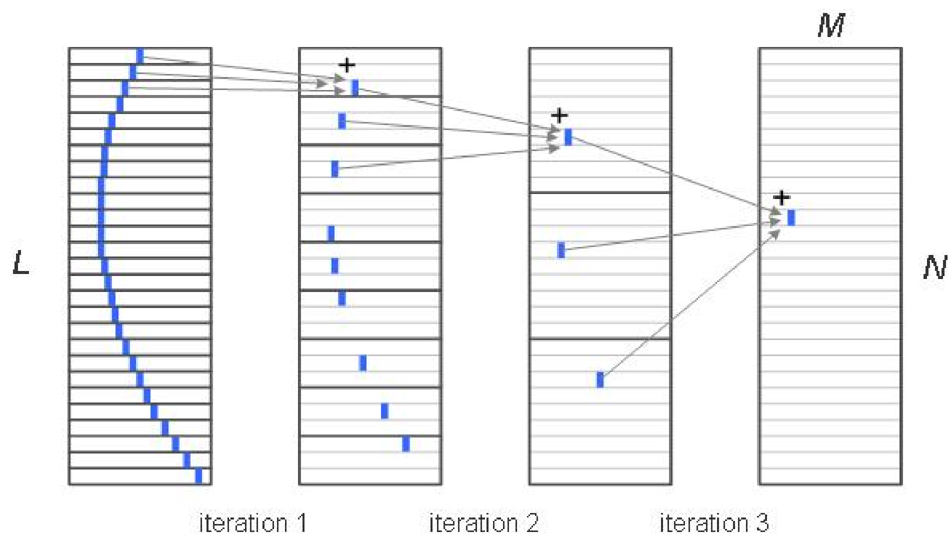


Figure 3.6: Illustration over how FFB creates the resulting image position over the iterations[8].

The computational burden compared to global backprojection is much less. Considering the same case as with the global backprojection where an image with $M * N$ pixels and L positions were used. In each iteration, K, n subapertures are merged to a larger subaperture. Every $M * N$ pixels in the resulting image need n calculations in each iteration and there are $K = logn(L)$ iterations. This gives a total of $nMNlogn(L)$ operations..

The minimum of this expression is reached when $n = e$ but since the base needs to be an integer the minimim is at $n = 3$. The final number of operations is then $3MNlog_3(L)$. Compared to the GBP, this is a huge speedup. Worth noticing is that a base of 2 or 4 gives approximately the same result i.e. 6% more operations than with a base of 3.

3.3 The SAR system used in this project

The system investigated in this thesis is a SAR system that operates on the low VHF-band (20-90 MHz) with wavelengths between 3-15 m. This gives it the ability to penetrate foliage and detect concealed objects with good spatial resolution. It is also useful for mapping forest biomass and, to some extent, finding targets in the ground. The SAR system has two bi-conical wideband antennas placed on an aircraft. Since the system is located between 20-90 MHz strong radio-frequency interference is present such as FM-radio and communication for emergency services, which need to be suppressed.

3.3.1 Signal Processing

The FFB algorithm is used for image creation. It is much faster than the GBP but still requires a good amount of memory and bandwidth. Together with the FFB calculations, the geometric calculations for the flight-track and ground topography compensations need to be considered. Figure 3.7 shows the signal processing chain used for the SAR system.

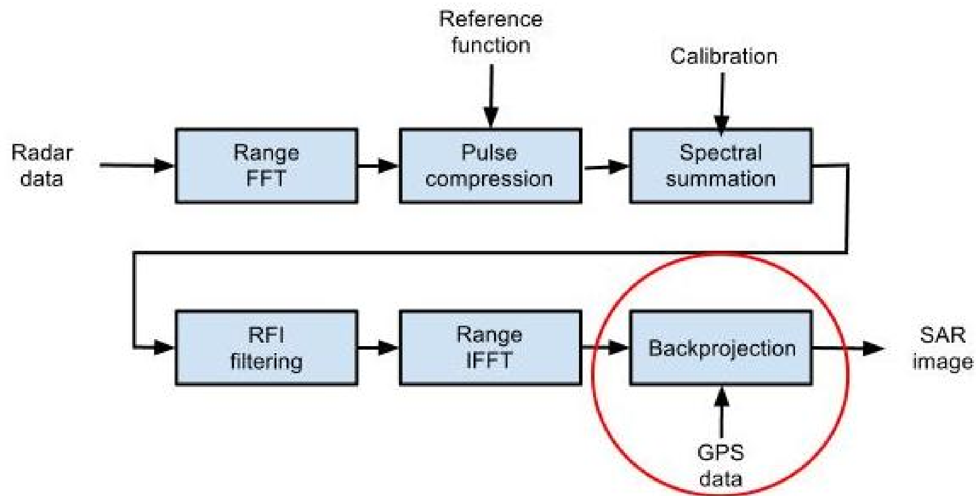


Figure 3.7: A typical signal processing chain. The scope of this project highlighted, excluding the handling of GPS data.

The radar system continuously processes incoming data while working in stripmap mode, see Figure 3.8, and outputs the full resolution image after processing all data from the integration length L. The time for processing all data is at least the time for flying along the integration length.

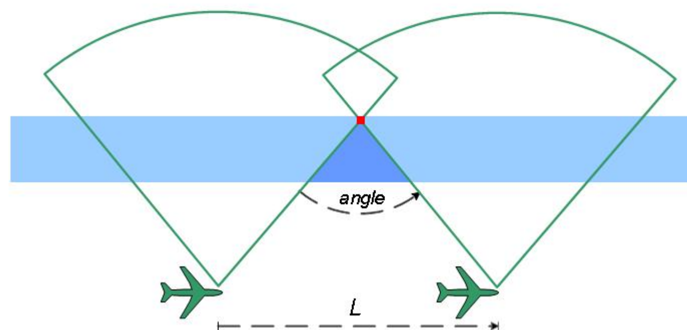


Figure 3.8: The blue triangular area is fully integrated when using stripmap processing[8].

The processing of data is done during flight but the iterations are dependent of each other as seen in Figure 3.9. We collect the data at different times and as we can see, the processing of data from t_0 cannot start until $t_0 + 1$ has been received. The dependency is most visible when we have collected all data along the integration length and have a delay at $t_0 + 7$ where all iterations need to be processed to get the full resolution image. However, already at $t_0 + 3$ its possible to produce a low resolution image.

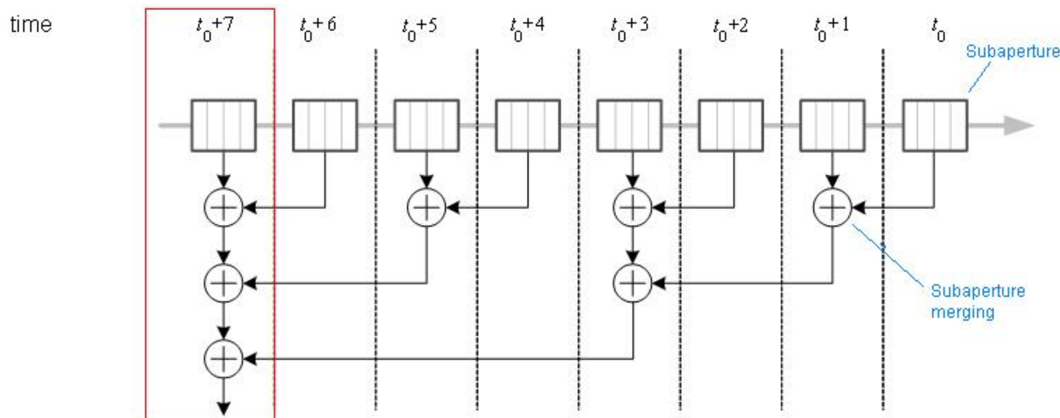


Figure 3.9: Graph over the FFB data processing and its dependencies[8].

3.3.2 Data structure

To handle the data in this project a ping-pong technique is used. This technique uses two buffers to create the image, storing data from every two iterations. The received range data lines are stored in the pingbuffer ordered by the time they were received. When enough data has been received in the pingbuffer, to merge the first iteration, the algorithm starts to merge them together to a new subaperture in the pongbuffer. To be able to merge the next iteration, enough subapertures need to be created in the pongbuffer. Then the algorithm starts to merge the contributing subapertures from the pongbuffer to a new subaperture in the pingbuffer.

The principle is seen in Figure 3.10, though in this example the processing of data is not done in real-time, all data is already received when the merging begins. This proceed until one buffer only have one subaperture and then we got the final image.

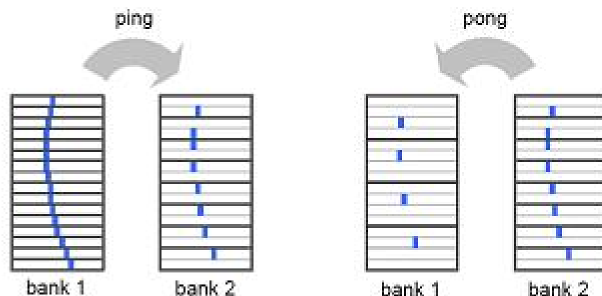


Figure 3.10: Merging two contributing subapertures from the ping- to the pongbuffer and then merging two larger subapertures from the pong- to the pingbuffer. The blue dot in the subaperture represents a specific target in the picture, which for every step is represented with a better angular resolution. In this figure the data isn't processed until all data is in the pingbuffer[8].

3.3.3 Image creation

When creating an image using the FFB there are many calculations to take in consideration. For every element calculated in the merged subaperture the contributing element need to be located from each contributing subaperture. This is not trivial since finding the right element is dependent on both flight-track geometry and ground topography. Additionally, an interpolation method has to be used to find the right element.

A polar coordinate system is used to represent positions in a subaperture according to Figure 3.11 where r is the range to the target from the center of the subaperture, and θ is the Doppler angle.

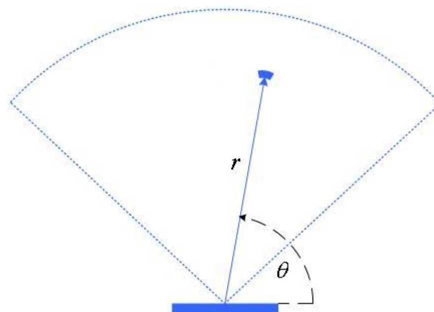


Figure 3.11: An image position represented with a polar coordinate system[8].

To merge two subapertures is simple when the fligh-track is linear as picture 1 in Figure 3.12. Though in reality that is not the case, see picture 2 in Figure 3.12, so compensations for the nonlinear flight has to be done with e.g. GPS coordinates.

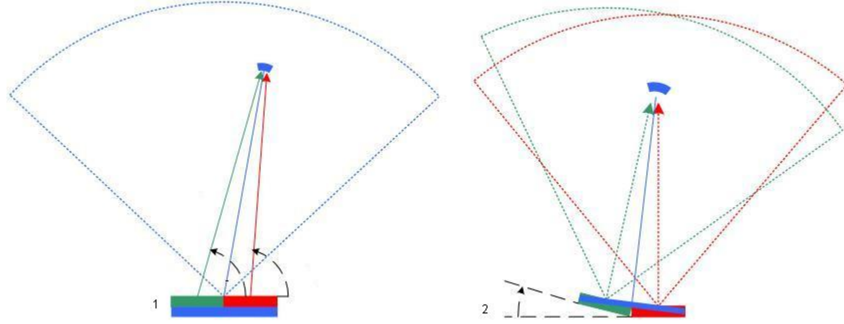


Figure 3.12: An image position is created using elements from two contributing subapertures, both with a linear flight-track and a nonlinear flight-track[8].

The ground topography also need to be taken into consideration. In the case with a linear flight-track the ground topography is irrelevant for focusing. However, it is determining when the image data is projected on a ground grid. The data need to be placed on the correct ground range by angling down the received data with respect to its slant range. This is computationally heavy, but it can be approximated.

This is done by assuming that the topography does not change considerably in a fixed range interval δr_{limit}. By using constant difference in range and angle within this interval only a fraction of the geometrical calculations for the ground topography has to be done, see Figure 3.13.

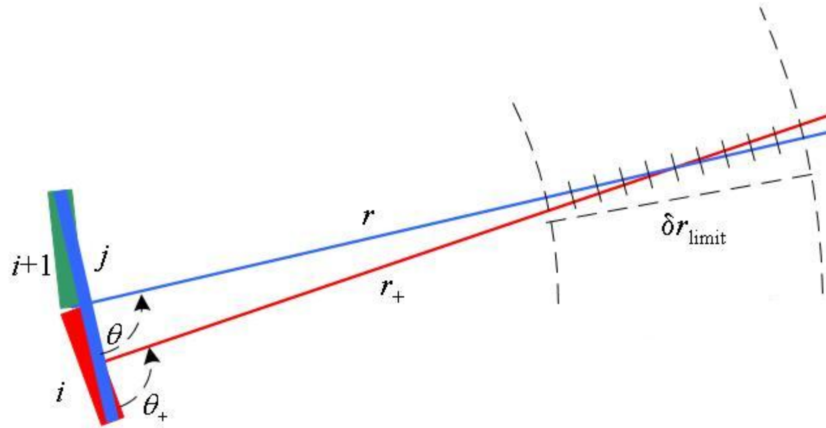


Figure 3.13: When finding the correct element from the contributing subaperture, a constant range and angle difference is assumed within a range interval δr_{limit}[8].

3.3.4 Interpolation

The calculated element to pick from the contributing subaperture does not exist very often because of the sampled signal. Therefore, an interpolation method has to be used to approximate the value to use in the merge. The simplest method is to use nearest-neighbor interpolation where the element closest to the calculated is used, but this comes with a reduction in resolution. To obtain a better resolution more complex interpolation methods can be used, e.g. sinc or polynomial interpolation. However, a more complex interpolation method will increase the computational burden so in this project nearest-neighbor interpolation is used since the given resolution is good enough.

3.3.5 Data representation in memory

The radar data is represented in the computer memory according to Figure 3.14. Every subaperture consists of a number of range data lines, each having a number of range bins. Each pair (r,θ) represent a floating point complex value in every subaperture. r corresponds to the range bin and the angle θ represent a specific range data line.

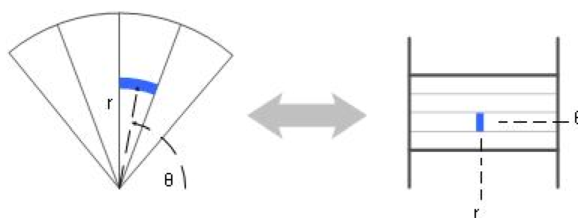


Figure 3.14: Representation of data in memory using range and angle[8].

This way of representing data in memory is not very economic which is shown in Figure 3.15 where a linear strip is mapped to memory.

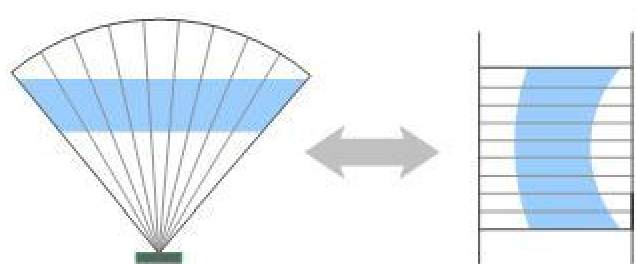


Figure 3.15: Projection of data in memory using a polar coordinate system[8].

Since the number of range data lines will increase when the subapertures are merged together, for every iteration, the subapertures will grow and so will the amount of memory for representing a subaperture[9].

3.3.6 Adressing pattern

The resulting data row is created from elements that are fetched in contributing sub-apertures, however the correct contributing element is found along non-linear paths in memory as seen in Figure 3.16.

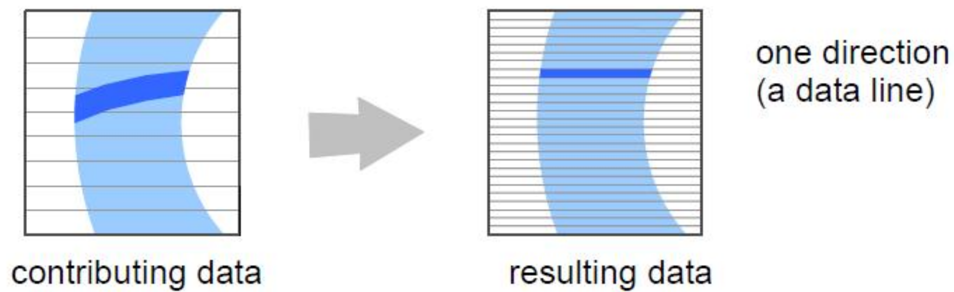


Figure 3.16: Contributing data is collected along nonlinear paths[8].

Finding these paths requires lots of geometrical calculations, especially since the paths get more and more distorted for every merging iteration as seen in Figure 3.17.

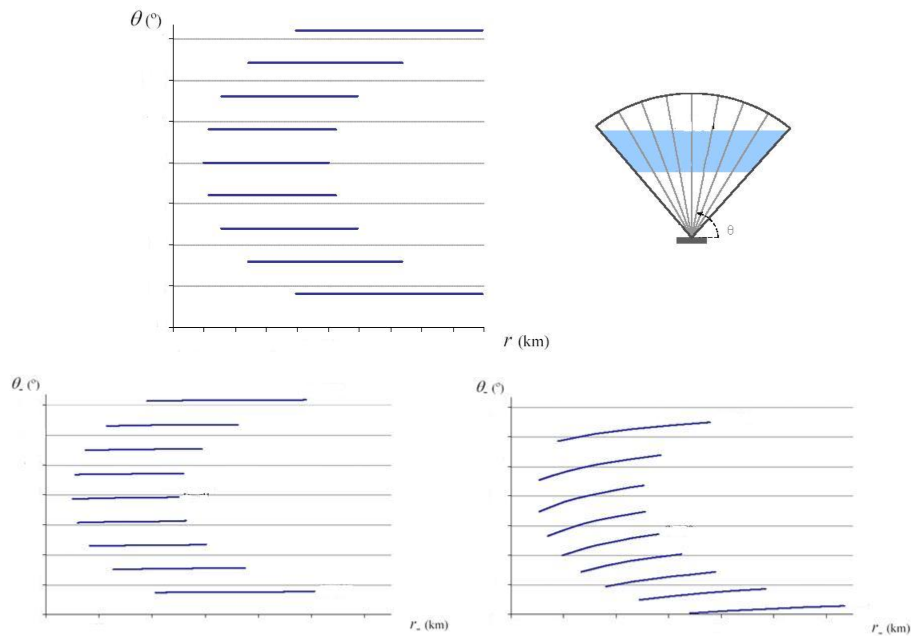


Figure 3.17: 8 range data sectors represented in memory, for every merge the data get more and more distorted[8].

However, this can be approximated if we assume fixed values (Δr,$\Delta\theta$) inside an interval δr_{limit}. I.e. we approximate the curved path with a linear path in that interval as seen in Figure 3.18.

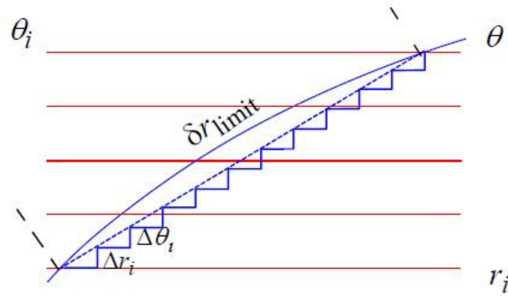


Figure 3.18: Approximation of the memory read path[8].

Along these linear paths, an interpolation kernel is used to pick the correct contributing element[9].

4

Parallelism

The high computational burden in a typical SAR system prevents the FFB algorithm to reach real-time requirements. Parallelization is needed to increase speed and meet the timing constraints. Parallelization can be applied with different level of granularity. The problem is though not easy to fully parallelize because of dependencies, which is illustrated in Figure 3.9, and other factors.

4.1 Motivation

Historically the development of integrated circuits have been following Moore's law, which says that the number of transistors that can be placed on a single die is doubled approximately every two years[10]. When the number of transistors increased, the clock frequency increased, so the performance of an application increased without the need to make any changes in the code. The development of transistors will still follow Moore's law but the increase of clock frequency is limited to the power consumption of a chip, P:

$$P = CV^2F \tag{4.1}$$

C is the capacitance switched per clock cycle, V is the voltage and F is the processor frequency. When this peak was reached, the industry switched to parallel techniques in the shape of multicore architectures.

4.2 Limitations

Parallelization of code puts the programmer in a hotspot since the sequential code need to be rewritten to some extent, except in some cases when the compiler parallelizes the code. The programmer need to take into consideration, when parallelizing code, that there could be race conditions, mutual exclusion and synchronization problems. Nevertheless, just because the code is parallelized and a dual-core processor is used the program will not always run twice as fast. There is a limitation due to Amdahl's law, which says, given the number of processing elements and the portion of code that can be parallelized, what possible speed-up can be gained, see Figure 4.1. If 10% of the code is not possible to parallelize it is not possible to get more than a ten times speedup disregarding how many extra cores are used[11].

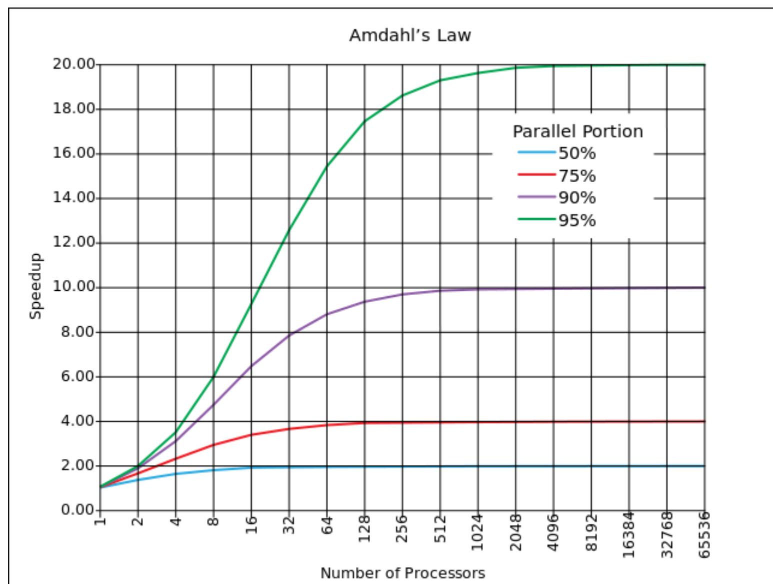


Figure 4.1: Graph illustrating Amdahl's law[12].

In contradiction to Amdahl's law, that assume a fix data set, Gustafson's law says that given a data set that increases with the number of processors the sequential portion of the code will decrease and the speed-up will approach the number of processors, see Figure 4.2[13].

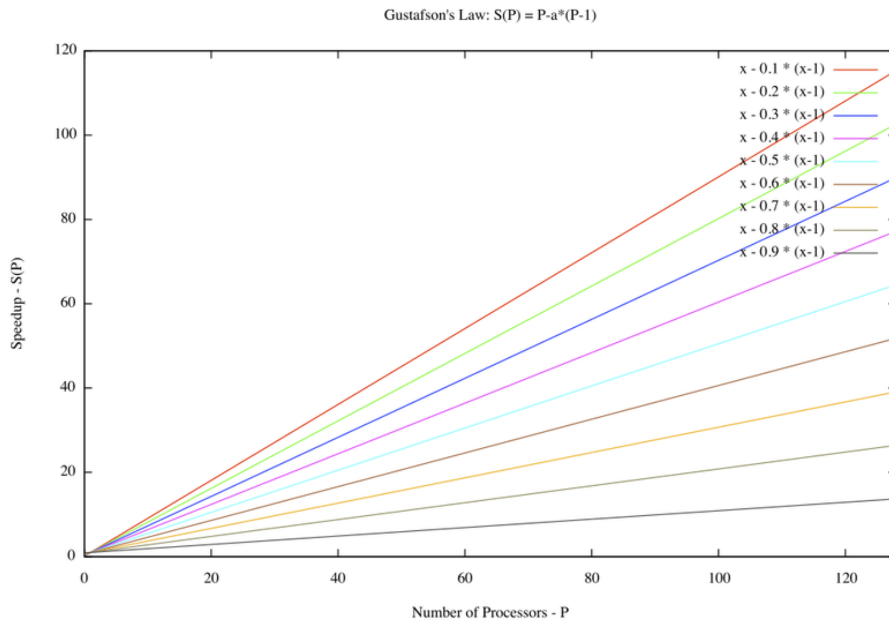


Figure 4.2: Graph illustrating Gustafson's law[14].

4.3 Granularity

Parallelizable applications can be classified into three groups. Fine-grained is the case when the created subtasks are small and need to synchronize often, this kind of parallelization requires some overhead for communication and synchronization. Coarse-grained is when the subtasks are bigger and does not need to synchronize that often, which leads to less overhead. Embarrassingly parallel is when the subtasks hardly ever need to synchronize which also means little or almost no overhead. This need to be taken into consideration by the programmer so the overhead does not exceed the performance gain when the application is parallelized.

4.4 Exploiting parallelism

Exploiting parallelism can be done in several ways depending on the application and the hardware used.

4.4.1 Instruction-level parallelism

Instruction-level parallelism makes use of the pipeline and re-orders the instructions so they can be performed in parallel, without changing the result of the program. Processors that have N-stage pipelines can hence execute N different instructions at the same time. There are also superscalar processors, which can, if there is no data dependency, execute more than one instruction at once.

4.4.2 Data parallelism

Data parallelism can be used in loops where the different iterations can be divided upon several threads. This is natural for many loops where the same operations will be executed over big data sets.
Though if the data is dependent throughout the loops and an iteration rely on results from the previous iteration then the loop can't be parallelized. Though the programmer can sometimes rewrite the loop and make it data independent. However, if the data is dependent throughout the loops and an iteration relies on results from the previous iteration, the loop cannot be parallelized. The programmer can sometimes rewrite the loop and make it data independent.

Data parallelism can also be exploited with vector instructions, sometimes called Single-Instruction-Multiple-Data (SIMD) instructions. Vector instructions are applied to a vector register, available on most modern CPUs, which contain several elements so the same operation is performed on all elements simultaneously. Using vector operations can be done explicitly by using intrinsic functions, i.e. primitive functions that are translated to one or a few machine instructions, to tell the compiler which vector operations to use. Some modern compilers can also automatically vectorize the code to some extent. The parallelism has to be obvious to the compiler and the data accesses or the expression cannot be too complicated. The speed up of using vector instructions depends on the size of the vector registers that depend on the architecture and range from 64-bit to 256-bit.

The following code example will be vectorized with a good compiler.

```
const int size = 512;
int a[size], b[size];
for (int i = 0; i < size; i++) {
    a[i] = b[i] + 3;
}
```

A 128-bit vector register will be used to put four elements of b which will be added with another vector register holding four elements (3,3,3,3). The result will be stored in a. Optimally the size of the array is divisible by four in this case. Otherwise, it is preferable to add dummy values to the end of the array to achieve this. This operation will be done 512/4=128 times with a speed-up up to four times.

4.4.3 Task parallelism

Task parallelism could be compared to data parallelism with the exception that it can also be applied to different calculations and not only the same calculations as in a loop. I.e. one task can be assigned one part of the code and another task can be assigned a completely different part of the code.

5

OpenMP

Open Multi-Processing (OpenMP) is an Application Program Interface (API) that supports shared-memory parallel programming in C, C++ and Fortran. The OpenMP Architecture Review Board (ARB) is a non-profit corporation that oversees the specification, produces, and approves new versions of it. The majority of the ARB consists of representatives from major software and hardware vendors, which in turn create OpenMP products. This makes OpenMP a portable and scalable model that can be used to develop applications for platforms ranging from laptops to supercomputers[15].

5.1 Execution and memory model

OpenMP uses the fork-join method, when a program is run, a single thread executes it until it encounters a parallel region. When a parallel region is reached, the thread creates a workgroup, of zero or more new threads, and the encountering thread becomes the master thread of the workgroup. Each thread in the workgroup is assigned its own implicit task, which is defined by the code in the parallel block.

Everything declared before the parallel region will be shared memory between the threads, unless specified otherwise in the parallel construct, and everything declared inside the parallel region will be private to each thread. Each thread is allowed to have a temporary view of the shared memory, i.e. it does not have to fetch the shared variables from memory each time they are used but instead it can store them in registers, the cache or other local storage. While introducing the need to use synchronization directives manually to ensure the behavior of the program it avoids unnecessary synchronization to occur each time a shared variable is used.

5.2 Programming with OpenMP

The OpenMP API gives a simple and flexible way to develop parallel applications. The programmer uses pre-processor directives and runtime library routines to control the program. Much of the work, such as creating and deleting threads, is hidden from the

programmer.

The simplicity makes it easy to modify a sequential program to run in parallel if the program is in fact parallelizable, i.e. it does not have any real data dependencies, and it does not contain artificial dependencies. An artificial dependency is a dependency that is not required for the correctness of the program but rather the programmer creates it.

5.2.1 Internal control variables

The Internal Control Variables (ICVs) control the behavior of an OpenMP program. They can be modified by changing environment variables or through API routines and can only be retrieved by the program with API routines. Two ICVs, needed to control how the parallel regions behave and ways to modify them, are given in Table 5.1. The first controls the number of threads that shall be used for the next parallel region and the second is used to set if nested parallelism is allowed or not. If nested parallelism is enabled a thread that is already in an existing workgroup with other threads is allowed to create its own new workgroup if it encounters a new parallel region. Each thread has its own copy of the ICV for the number of threads to use for the next parallel region, allowing each thread to use different sizes for new workgroups.

Environment variable	Routine	Retrieve
OMP_NUM_THREADS	omp_set_num_threads()	omp_get_num_threads()
OMP_NESTED	omp_set_nested()	omp_get_nested()

Table 5.1

5.2.2 OpenMP directives

The directives and their properties that were considered important for the project will be presented. For a complete description of all directives and their properties the reader is referred to the OpenMP 3.0 specification[16].

All OpenMP directives start with *#pragma omp*, which allow the compiler to identify it as an OpenMP directive. Then a construct follows, that specifies what OpenMP should do, which can have optional clauses.

The parallel construct

When a thread encounters a parallel construct, it creates a new workgroup to execute the parallel region. The number of threads in the workgroup depends on the ICV that control the number of threads but it can be overridden by a *num_threads(n)* clause. The encountering thread becomes the master thread for the workgroup and the number of

threads in the workgroup is constant for the duration of the parallel region. The task region the master thread executed before encountering the parallel construct is suspended and instead each thread in the new workgroup, including the master thread, is assigned a new implicit task based on the code in the parallel region. The new implicit task assigned to each thread becomes tied, meaning that if it is suspended no other thread can resume the execution of it. If a thread in the workgroup encounter a new parallel region and nested parallelism is enabled it creates a new workgroup and becomes the master thread of it.

There is an implicit barrier at the end of a parallel region and only the master thread continues after it with the task region it suspended when it encountered the parallel construct.

The critical construct

Protecting shared memory is done with critical regions, which bind to the enclosed block. A critical region can have an optional name associated with it, and all critical regions without a name have the same unspecified name. When a thread encounters a critical region it waits until no other thread in the program, not just in the same workgroup, is executing a critical region with the same name until it enters.

The flush construct

Since the threads are allowed to have a temporary view of the shared memory, by storing variables in local storage, it need to synchronize the memory so it can see changes made by other threads or store it own changes. The flush construct make the encountering thread synchronize its temporary view of the shared memory with the shared memory. If the thread has changed a variable that is flushed, it will write it to the shared memory. Otherwise, it will discard the temporary view of the variable, which ensures that the next time it is used it will be read from the shared memory. An optional list can specify which variables to flush and if no list is present, everything is flushed. If a pointer is in the list, only the pointer itself is flushed and not the memory it points to. To flush memory only accessible by pointers, the flush construct without a list has to be used. Only the temporary view of the encountering threads memory is affected by a flush. A flush with no list occurs every time a thread enters or leaves a parallel or critical region.

The for construct

To split up the iterations in a *for*-loop between multiple threads the *for* construct is used. The *for* construct restricts the structure of the *for*-loop to the form described in [16]. The reason that it restricts the structure of the *for*-loop is so that the iteration count can be computed before the execution of the loop so the iterations can be divided between the threads according to the scheduling method. The construct can be com-

bined with the parallel construct to create a parallel region containing only a *for*-loop. The scheduling method is defined with a *schedule(method,chunk_size)* clause where *chunk_size* is optional. The different methods are:

- Static. When *chunk_size* is specified, the iterations are divided into chunks of that size, which are assigned to the threads in a round-robin fashion. If *chunk_size* is left out the iterations are divided into chunks of approximately equal size and at most one chunk is assigned to each thread.
- Dynamic. The threads request a chunk of iterations, execute them, and then request another until no chunks remain. Each chunk has the size *chunk_size*, if specified, except the last which may contain less iteration. If *chunk_size* is left out, each chunk will contain one iteration.
- Guided. Like dynamic each thread request a chunk of iterations, executes them, and then request another until no chunks remain. Unlike the dynamic method, the chunks differ in size. Each chunk is proportional to the number of unassigned iterations divided by the number of threads and they are decreasing over time to *chunk_size*. When the *chunk_size* is unspecified, the default value is 1.

5.2.3 Example

```
#pragma omp parallel for schedule(static)
for(int i=0; i<n; ++i) {
        for(int j=0; j<n; ++j) {
                int temp = 0;
                for(int k=0; k<n; ++k) {
                        temp += matrix1[i][k] * matrix2[k][j];
                }
                matrix3[i][j] = temp;
        }
}
```

5.2.4 Overhead

Creating a parallel region cost some overhead, both to start up the threads and assigning work to them. In addition, when a parallel region has been executed the memory is synchronized which also takes some time. Therefore, it is important that there is enough work in the parallel region so that the performance gain exceeds the overhead. The overhead increases with the number of threads so there is no guarantee that increasing the number of threads will increase the performance if the workload is small in the parallel region. Figure 5.1 shows the performance for a different number of threads for the matrix multiplication example from Section 5.2.3.

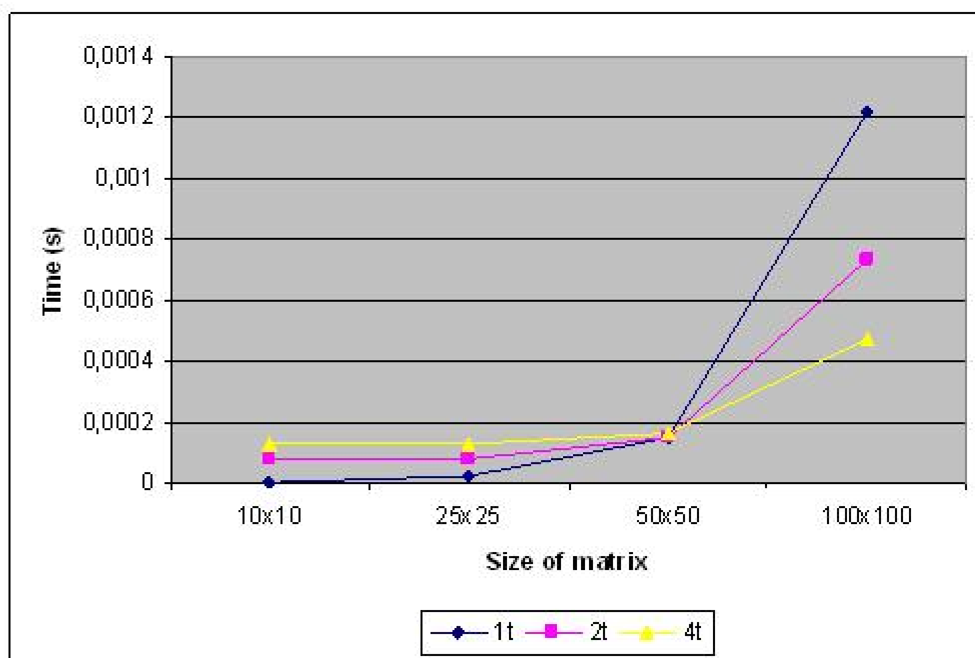


Figure 5.1: Example from Section 5.2.3 run with different matrix sizes and threads.

The different kind of scheduling methods for work distribution in a for-loop also comes with different overhead. The static method has the least amount of overhead but for it to be best performance wise there must be an even workload in the for-loop. Also if each iteration in the loop has a lot of work it can be a big performance increase if the threads request work each time they have finished, i.e. dynamic or guided scheduling. It should also be noted that if a thread is interrupted, e.g. by the operating system, and static scheduling is used the performance can be severely degraded.

5.2.5 Behaviour of an OpenMP program

There is no way to specify how the threads are assigned to the logical processors, which lead to that the behavior of OpenMP programs is not completely deterministic. How the threads are assigned to the logical processors can have a big impact on the shared cache. For example if there are two dual core CPUs and there are four threads in the workgroup, the work consist of two different tasks that are split into two subtasks. If the two threads that do the subtask from the same task are assigned to different CPUs they cannot take advantage of the shared cache together, so two different runs of the same program can differ a lot in performance. It has been announced that the ability to assign the threads is being developed for OpenMP 3.2[17].

Running the exact same task on the same thread multiple times can result in different running times since the core that the thread is run on is not exclusive to the OpenMP application, e.g. the operating system can interrupt the application to run garbage collection. Therefore, in an application with multiple threads the threads cannot be assumed identical in speed so it is important that an application does not depend on that false assumption.

6

Optimizing and parallelizing the FFB

Over 98% of the work is spent in the merging part of the code so it would seem reasonable to parallelize only that part. However in the first iteration when there are only a few subapertures to merge, the workload is so small (given that the number of range bins isn't extremely high) that the overhead for creating the parallel region neglects the performance gain that should occur when adding more threads , see Section 5.2.4.Therefore has also the data collection been parallelized so that there are a workgroup that fetch a subaperture from the input and store them in the ping-pong buffers. Whenever there are enough subapertures to do a merge, one of the threads create a new workgroup to do the merge.

6.1 Data collection

Reading the subaperture from the input has to be done in a critical region so two threads does not try to read the same subaperture. As mentioned in section Section 5.2.5, the threads can be interrupted at any time and since reading a subaperture from the input is very fast, even the smallest interrupt can make the subapertures to be added in the buffer in the wrong order if the subapertures are added at the next free location. To avoid that the subapertures are added in the wrong order, each one is assigned a number when it is read that decides where it is added in the buffer. For the result to be correct the right consecutive supapertures has to be merged with each other, e.g. if base two is used the first two are merged and the next two and so on. Because the subapertures is not necessarily added to the buffer in the right order a control structure has to be used to keep track on when and where a merge can occur. If merges are allowed to occur in the wrong order, with respect to the time the subapertures were collected, the disorder ascend to the next iteration so there have to be a control structure for all the iterations. Figure 6.1 shows an example of how the subapertures are added and merged in the buffers.

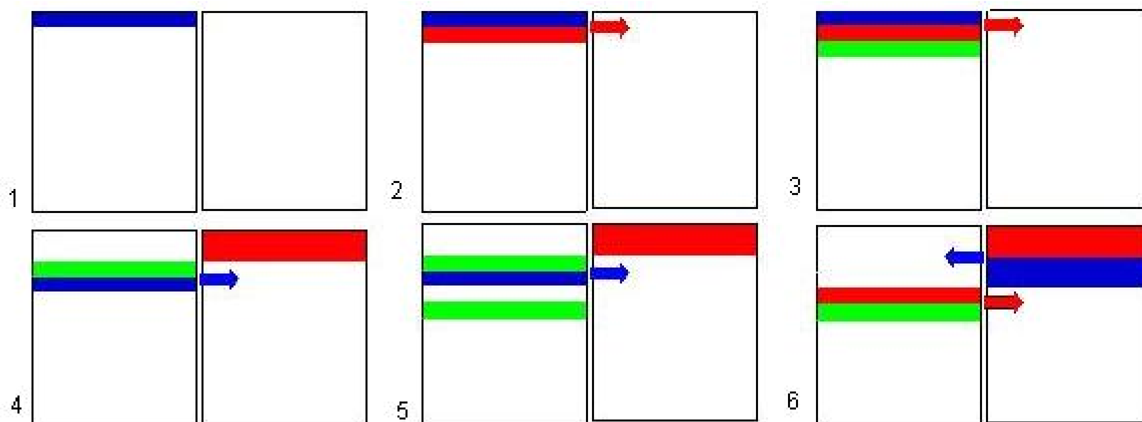


Figure 6.1: The ping pong buffer – subapertures is added to the pingbuffer in the order it was received. In picture 5, a subaperture has not yet been added to the buffer but the data structure handles this and leaves an empty slot that is filled in picture 6.

For each iteration the buffer can be divided into blocks where each block contains the same number of subapertures as the merging base is. The control structure then have to keep track of the number of blocks for each iteration, where in the buffer they are located and how many subapertures they contain. When a block is full, the thread that added the last subaperture performs the merge on the block. The control structure keeps track on the same number of blocks, as there are threads so all the threads can merge at the same time. Since the control, structure is shared between all the threads all updates, i.e. each time a subaperture is added and each time a merge occurs, have to be done in critical regions. Each time a merge is done the control structure replace that block with the next one.

6.2 Merging

When a thread is going to merge together subapertures, it creates a new workgroup with a number of threads, including itself. The work is divided over the number of rows that is going to be created, see Figure 6.2. The number of rows that is assigned to each thread is not divided immediately over all threads. The work is scheduled using the OpenMP schedule dynamic, with chunk size 1, among the threads. The choice of dynamic is because of the unbalanced amount of calculations done in different iterations. The difference comes from the optimization to skip additions with zero when the indices are out of range and the fact that a thread can be interrupted to take care of some other task. If we use bigger chunk size the other threads could be idle, waiting for one thread.

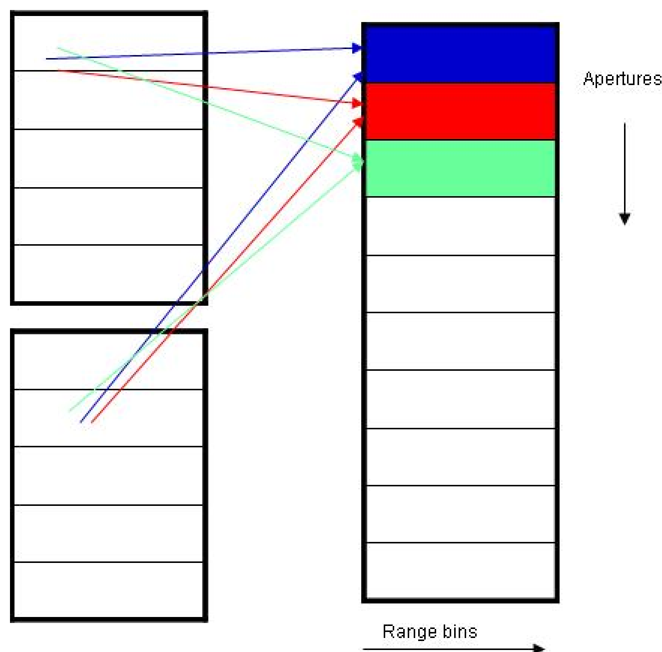


Figure 6.2: Different threads create the resulting rows, collecting contributing elements from the same subapertures, independent of other each other.

A thread requests a row to create, do all calculations for that row and when it is done asks for another one. For every row, the thread iterates over the range bins in the resulting row and collects the contributing element from one subaperture, given by the geometric calculations and interpolations that are done for every element. This is done for every subaperture that contributes to the resulting row.

6.3 Vectorization

The calculations carried out in the merge are also vectorized. Since the geometrical calculations are approximated within the range interval δr_{limit}, with $(\Delta r, \Delta\theta)$ fixed for each step, the calculations within the interval are vectorized. Without vectorization the same calculations for every (r,θ) would be executed without utilizing the Streaming SIMD Extensions (SSE).

The compiler can vectorize code itself but then it has to be written in a way that the compiler easily can see it. Instead of doing calculations for each range bin one at a time in a for-loop iterating over δr_{limit}, we pre-calculate the values that are going to be used inside the same interval and store them in arrays with the same size as the interval. Then a for-loop can be written, iterating over δr_{limit}, for each calculation. The compiler then notices that we are executing the same instruction on multiple data, which makes it possible to use the vector registers, and vectorize the code.

The test-machine TM1, see Chapter 7, has 128-bit vector registers, which can hold four floating-point numbers. Since δr_{limit} is not always divisible by four then dummy values could be added to the arrays to achieve this. This is not considered in this project but will preferably be taken in consideration when δr_{limit} is fully decided.

Another aspect that lowered the performance of the vectorization was that not all of the rewritten code could be vectorized. Most of the calculations was easily vectorized but e.g. when pre-calculating the $(\Delta r, \Delta\theta)$ with multiplication and vector instructions there were precision errors (compared to when addition is used) so the vector instructions were not used for that.

The optimization to skip additions with zero when the indices are out of range also affects the vectorization. Since some calculations inside a δr_{limit} interval are removed, the remaining calculations are done without vectorziation.

7

Test environment

Test cases

Three different test cases were set up that correspond to three different types of SAR systems. The amount of ground mapped varies between the cases, which result in different demand on performance needed to create the image in real time. A description of the test cases is given in Table 7.1.

	Small	**Medium**	**Large**
Flight time	74.66 s	149.33 s	512 s
Number of merging stages	7	8	8
Memory size	2x240.3 MB	2x7.8 GB	2x99.3 GB

Table 7.1

In order to put a perspective of the differences in computational burden between the test cases and to get an indication in how the computations is distributed over the iterations the number of adds is counted. We define one add as the operation done in a merge when the contribution from a contributing subaperture is added to the resulting subaperture for a single element in the data matrix. The number of adds for the test cases are approximately $1.5 * 10^8$ for the small, $4.9 * 10^9$ for the medium and $7.1 * 10^{10}$ for the large case. The work distribution over the iterations and the portion of the computations that require that the last pulse is collected can be seen in Table 7.2.

	Small	**Medium**	**Large**
Iteration 1	24.1%	21.2%	30%
Iteration 2	13.8%	12.1%	10%
Iteration 3	13.8%	12.1%	10%
Iteration 4	13.8%	12.1%	10%
Iteration 5	13.8%	12.1%	10%
Iteration 6	13.8%	12.1%	10%
Iteration 7	6.9%	12.1%	10%
Iteration 8	-	6.1%	10%
After last pulse	16.1%	14.1%	13.3%

Table 7.2

For all test cases synthetic input data were used, see Figure 7.1. The resulting image after backprojecting that data is seen in Figure 7.2.

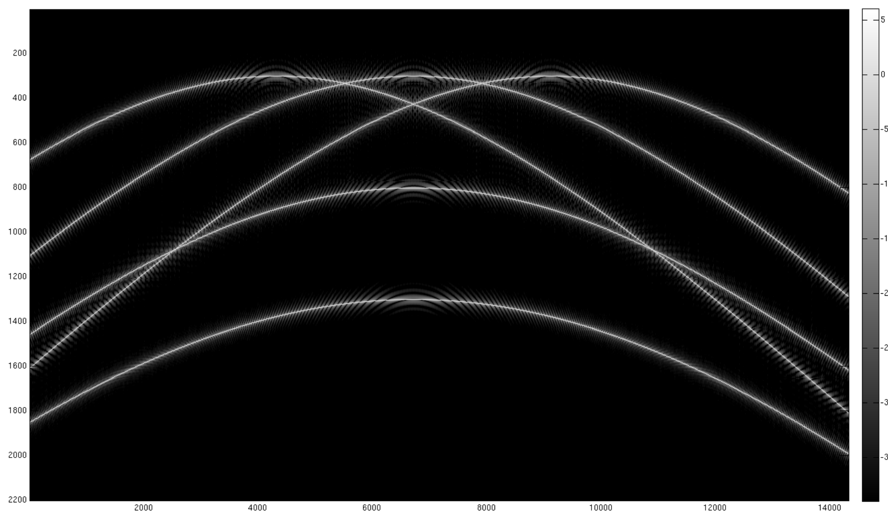


Figure 7.1: The synthetic raw data used.

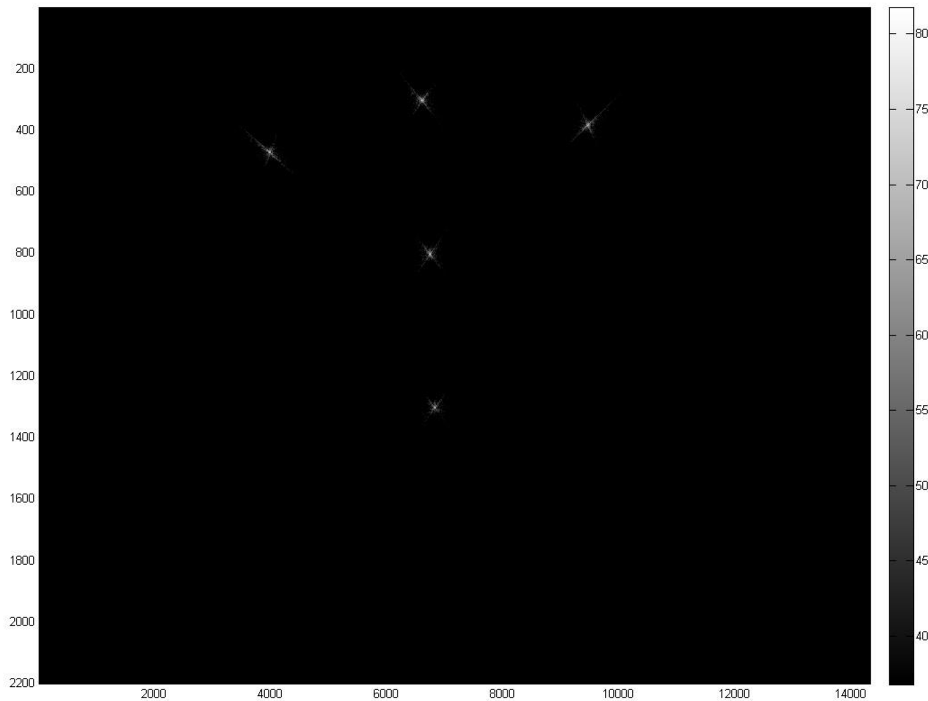


Figure 7.2: The resulting image from the raw data in Figure 7.1.

Test-machine 1

One machine used for benchmarking was a server running SUSE Linux Enterprise Server 11.2 (64-bit). It has two Intel Xeon X5675 hexa-core processors running at 3.06 GHz and 96 GB DDR3 RAM. The processors have a 3-level cache where each core has a 32 kB instruction and a 32 kB data L1 cache, 256 kB of L2 cache. The six cores also share a 12 MB L3 cache. Each core has a 128 bit vector register.

Test-machine 2

Another machine used for benchmarking was a desktop also running SUSE Linux Enterprise Server 11.2 (64-bit). It has one Intel E8400 dual core processor running at 3 GHz and 3.25 GB RAM. The processor has a 2-level cache where each core has a 32 kB instruction and a 32 kB data L1 cache and the two cores share a 6 MB L2 cache.

Compiler

The project was compiled on all machines with gcc 4.6.2, which has support for OpenMP 3.0. The flags used were -O3 to optimize for speed and -fopenmp to enable OpenMP.

8

Result

The two smaller test cases were benchmarked in two modes, one with delay to simulate a real-time system and one without delay, and with two different code versions, one that take advantage of the vector registers when merging (OMP+Vec) and one that does not (OMP). The medium test case was only benchmarked on test-machine 1 (TM1) since test-machine 2 (TM2) could not allocate the buffers. The largest test case could not be run on any of the test-machines because they were unable to allocate the buffers.

The most effective thread setup depends on the number of threads that shall be used. When only a few threads are available they are all used for the data collecting part so each thread does a merge by itself. The reason for this is the work distribution, see Table 7.2, and the fact that the first iterations contain relatively few computations so the overhead, see Section 5.2.4, makes the gain of adding more threads there very small. When more threads are available the critical regions for the data collection makes it inefficient to use them in that part because of the waiting time. Therefore, it is better to use more than one thread in the merges when many threads are available. For test-machine 1 the threshold for this transit is at six threads.

The execution times for the different systems and versions are shown and they are later discussed in Chapter 10.

8.1 Small test case with delay

The time it takes to create an image, with different number of threads, for the small system in real-time is shown in Figure 8.1. The figure shows the two test machines running two versions of the code.

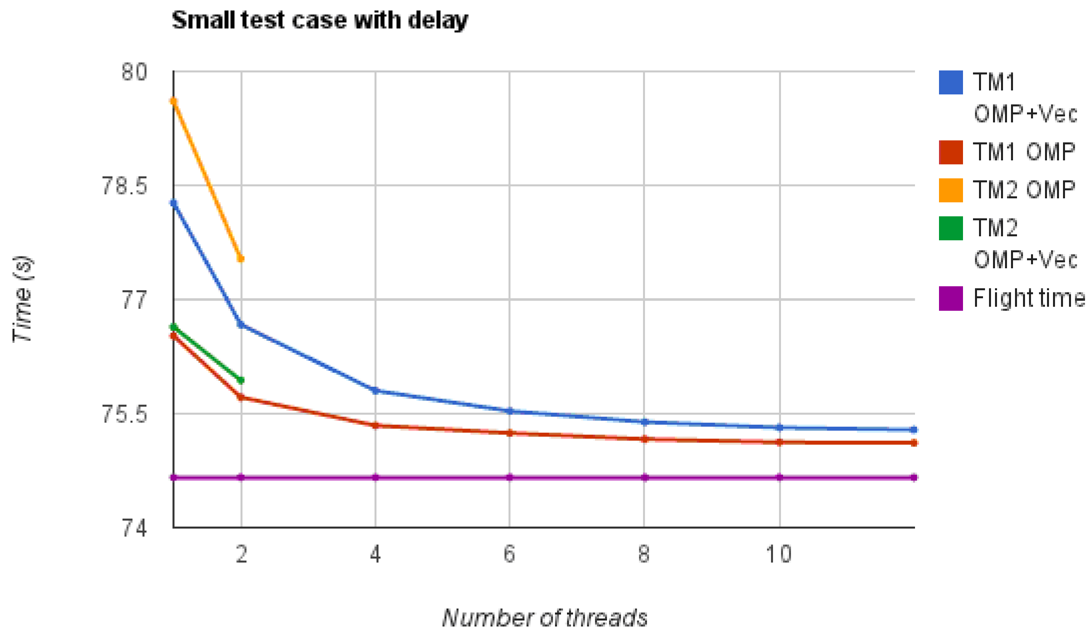


Figure 8.1

8.2 Small test case without delay

Figure 8.2 shows the time it takes to create an image, when all data is available from start. Both for one core with the different versions compared to the original code and with different number of threads run on the test machines.

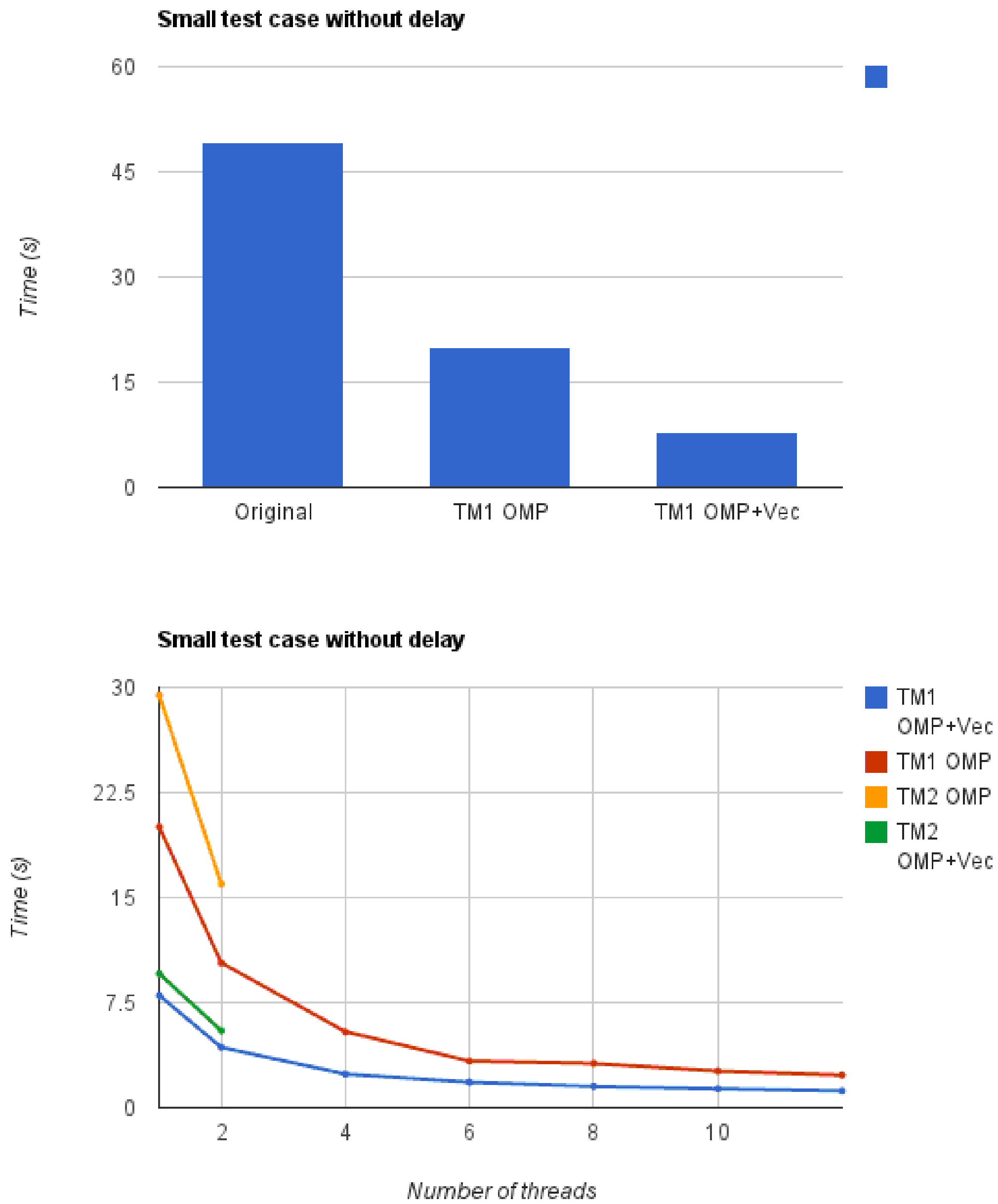


Figure 8.2

8.3 Medium test case with delay

The time it takes to create an image, with different number of threads, for the medium system in real-time is shown in Figure 8.3. The figure shows the two test machines running two versions of the code. A graph with four threads and more is included for better visibility.

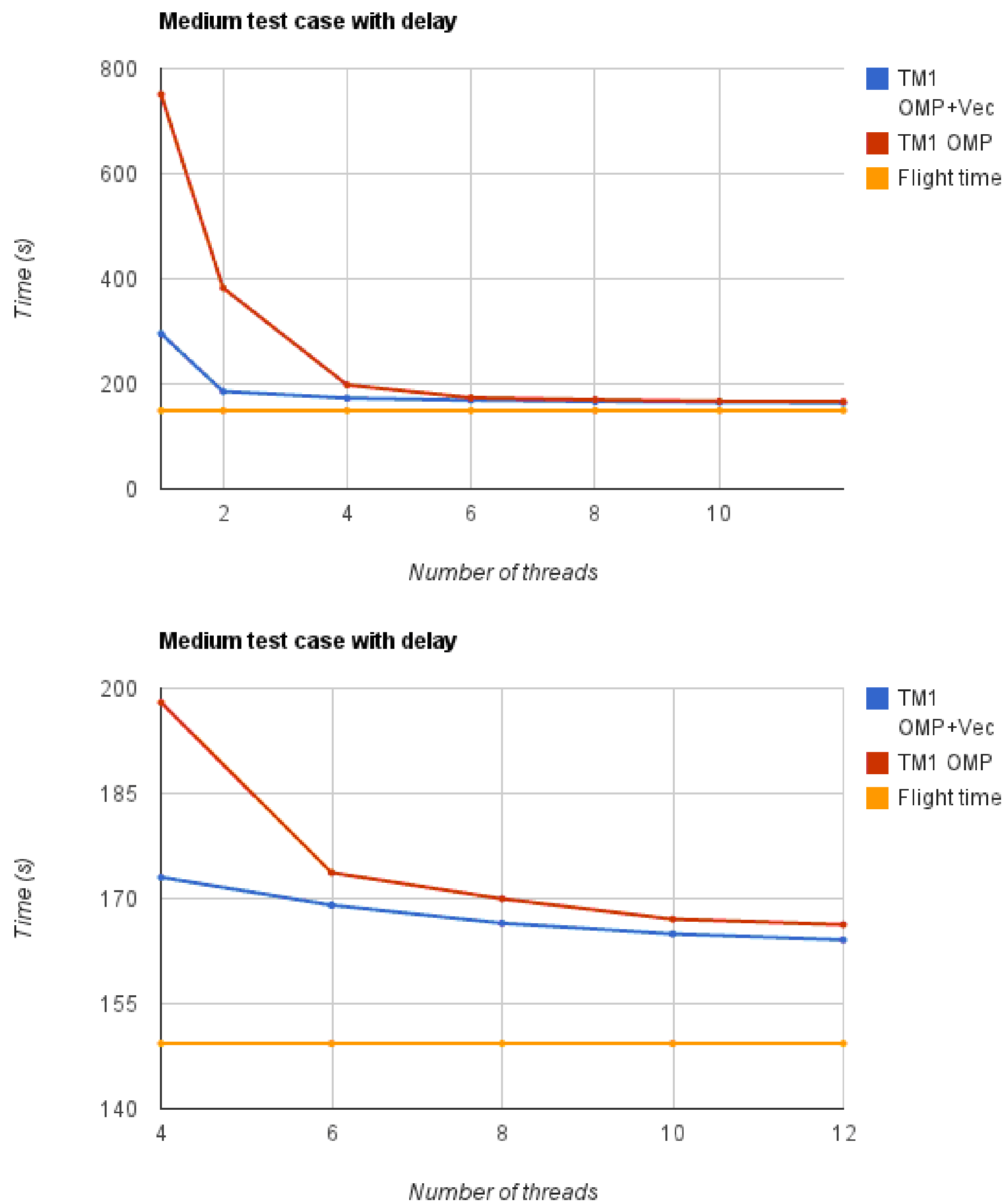


Figure 8.3

8.4 Medium test case without delay

Figure 8.4 shows the time it takes to create an image, when all data is available from start. Both for one core with the different versions compared to the original code and with different number of threads run on the test machines.

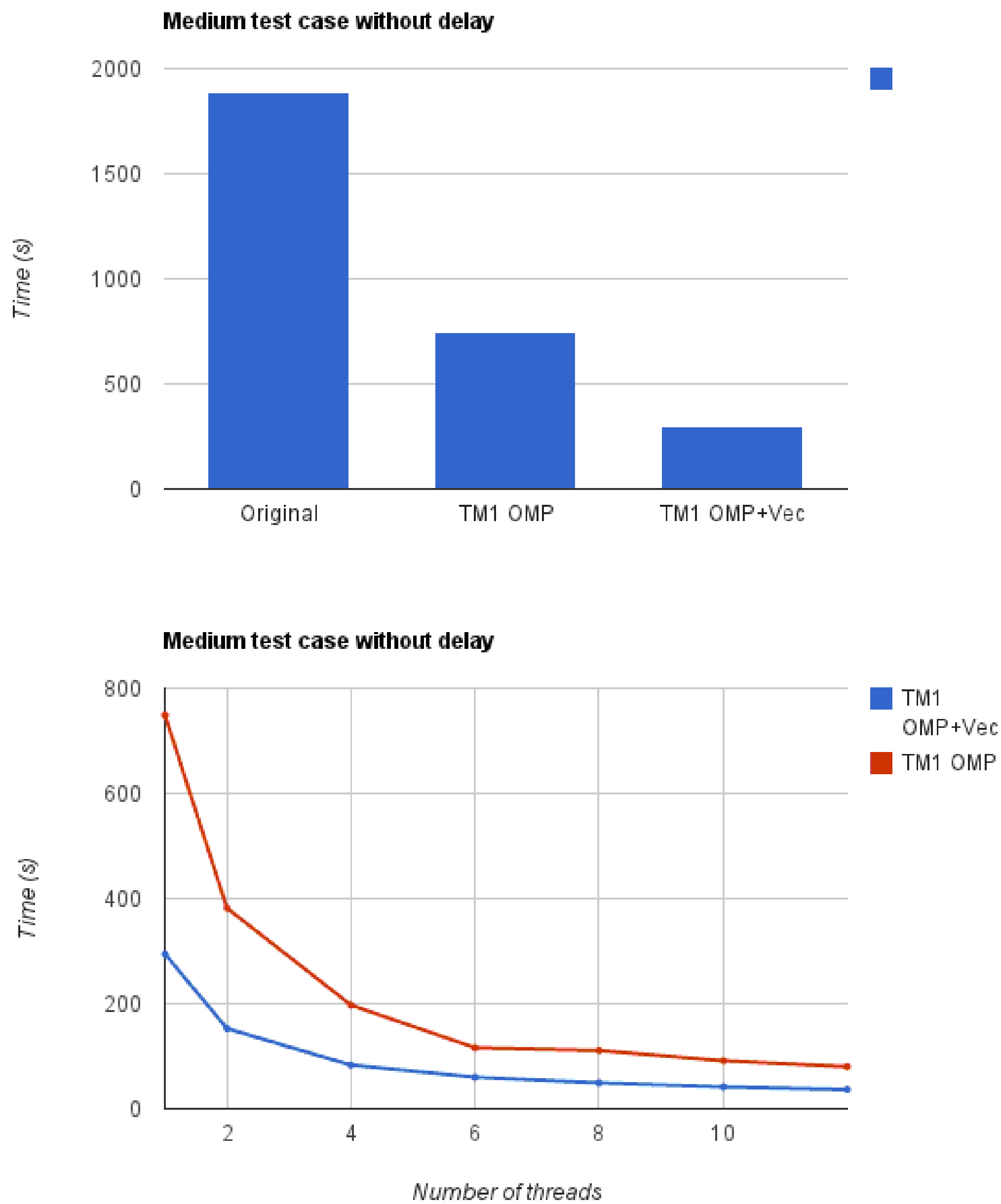


Figure 8.4

9

Performance analysis

To evaluate the performance of an application it is often not enough to look at the execution time, but also at the CPU utilization and the cache misses. The analyses in this section have been done with test machine 1, described in Chapter 7, on the small test case without delay. The performance was evaluated using Vtune Amplifier XE 2011.

9.1 FLOPS

A common way to measure performance for an application is to count Floating Point Operations Per Second (FLOPS). Counting FLOPS can give a good indication on how the application run on different machines because it can be assumed that the same ratio between measured FLOPS and peek FLOPS for an application is roughly the same for all machines. For the assumption to hold, the memory architecture has to be similar or at least offer similar performance. FLOPS can be computed by calculating the total number of floating point operations in an application and divide with the running time. Calculating the total number of floating point operations in an application can be really time consuming so another way is to use the hardware counter that are available on most processors. Hardware counters are registers that can be used to count different events, such as floating point operations or cache misses.

The theoretical peak performance of test-machine 1 is 147 GFLOPS [18]. This value will be compared with the benchmarked FLOPS for the FFB implementation to give an estimation of how well the CPUs are utilized. To estimate the FLOPS of the code the number of floating point operations is divided with the elapsed time. This gives, according to Vtune, the following table.

	Original	**TM1 OMP**	**TM1 OMP+Vec**
GFlops	10.39	33.19	39.75

Table 9.1

The Flops was measured for one core and then adjusted for twelve cores (multiplied with 12). Note that the value for the original code is only theoretical since it can only be run on one core. The vectorized version utilizes 27% of the theoretical maximum.

As a comparison to this result an FFT-implementation, fftw [19], were benchmarked, since FFT is a fast algorithm that has been refined over the years and thus should serve as indication on practical peak performance. The FFT obtained a peak performance of 104.4 GFLOPS, which gives a utilization of 71% compared to the theoretical maximum.

9.2 Cache utilization

Another metric that is a good indicator on how well the application is performing is to measure cache misses. Because if a cache miss occur there will be stalls to fetch the needed data, either from a higher level cache or from the RAM. Of course, cache misses depend heavily on the application so there is not a threshold that tells if the application is optimal. If an application uses very few instructions per memory unit, the miss rate will be high by default.

Table 9.2 shows the percent of cycles due to long latency data access, for the L2 and L3 cache, for the different versions of the code.

	Original	**TM1 OMP**	**TM1 OMP+Vec**
L2	0.36%	1.83%	3.55%
L3	0.82%	2.01%	4.35%

Table 9.2

As can be seen in Table 9.2 the percent of cycles is increased in the optimized versions. The number of actual cache misses is the same in the TM1 Original and TM1 OMP but less in the TM1 OMP+Vec version. There is a big decrease in clock cycles used, for every version, which leads to increasing percentage of cycles due to long latency data access. In Table 9.3, the events used to calculate this for the last level cache (L3) is shown.

	Original	**TM1 OMP**	**TM1 OMP+Vec**
MEM_LOAD_RETIRED.LLC_MISS	7 840 000	7 840 000	6 840 000
CPU_CLK_UNHALTED.THREAD	172 052 000 000	69 962 000 000	28 280 000 000

Table 9.3

From this, it can be seen that with the increased speed-up, in every version, that the cache misses takes up a larger part of the total runtime. The pure cache miss ratios (miss/hit) are given in Table 9.4.

	Original	**TM1 OMP**	**TM1 OMP+Vec**
L2	0.06	0.1	0.09
L3	0.44	0.24	0.25

Table 9.4

Table 9.4 shows that there is a big difference in cache miss ratio for the L3 cache. In the original code there is 0.44 misses for every hit in the L3 cache. In the optimized code, the ratio decreased to 0.24 and 0.25, which is almost half the misses compared to the original code. The L3 cache is better utilized in the optimized versions, while the L2 cache hits is almost the same as the original code. The hardware counters were not able to measure the cache misses/hits for the L1 cache, which would have given a more thorough analysis.

9.3 Cycles per instruction

Cycles per instruction (CPI) are a measure of how many clock cycles that is carried out when an instruction is executed. It is dependent on the application and the platform. The CPI is fairly constant in all the versions, as can be seen in Table 9.5, since both the number of instructions and the number of clock cycles has been decreased with the same ratio.

	Original	**TM1 OMP**	**TM1 OMP+Vec**
CPI	0.52	0.57	0.50

Table 9.5

10

Discussion

The scope of this work was to optimize and parallelize the algorithm to meet real-time requirements for the system. This has been done and evaluated in respect to performance, portability, scalability and engineering efficiency.

The algorithm has been optimized and parallelized in steps where two versions have been evaluated. The main difference between the two is that one utilizes the compilers ability to vectorize code.

10.1 Performance

When we look at performance in respect to real-time, it is important to notice that the algorithm is idle when waiting for data to be received. This means that when we continuously produce images, along a flight-track, in stripmap mode we have time to catch up the data collected, for the next image, during the final calculations and image output.

10.1.1 Small system

If we consider the small system in real-time, see Figure 8.1, we have evaluated the performance on both test-machines. Both machines, with vectorized code on one core, process the data in the rate it is received, and are spending less than two seconds, on processing, after the last pulse is received. However, the core is not idle enough during the integration time, 74.66 seconds, which means that it is not able to catch up the data processing from the next integration length. Both code versions are able to achieve real-time performance if two cores are used.

When we look at the case when not considering real-time, i.e. the case when the data already is collected and processed on the ground, we can see a huge difference in performance with the optimizations, parallelization and vectorization. The original code take 49.33 seconds to run and the vectorized version takes 8.02 running on one core. This gives a reduction in running time of 6.15 times with just optimizations and vectorization.

When using all cores on test-machine 1, which gives a running time of 1.20 seconds, we have a reduction of 41.1 times.

10.1.2 Medium system

The medium system is much bigger than the small system and is not as realizable as the small one. In the real-time case we have an integration time of 149.33 seconds and in comparison to the small system and with two cores, we are not able to process data in the rate we receive it. We need at least four cores to be able to process the data in the same rate as we receive it. Still with four cores, there are 23.67 seconds of data processing after the last pulse is received. Even with twelve cores, there is 15.17 seconds of data processing after the last pulse. However when multiple images are created in stripmap mode we are able to catch up the data processing from the next integration length already with four cores, which means that we are able to achieve real-time performance also for the medium system.

As with the small system, we can see a huge difference in performance between the code versions when considering the case without delay. The original time for processing the medium system is 1882 seconds, with just optimizations we reach a time of 750 seconds and when we are also using vector instructions, we reach a time of 295.62 seconds, which is a speed-up of 6.36 times the original when using one core. If we also use the parallelization with twelve cores, we achieve a time of 36.32 seconds, which is 51.39 times faster than the original.

10.2 Portability

The portability have not been thoroughly investigated, we have only tested the algorithm on two test-machines that is similar with the same compiler. Using another compiler that support a different version of OpenMP or does other optimizations could change the performance.

When examining the portability between the two test-machines, which has a similar clock frequency, the time difference is probably determined by the differences in the cache. But, we have not been able to benchmark with Vtune on test-machine 2 and cannot be certain. The ratio between the processor frequencies is though just 1.023 while the time differs with ratio of 1.193. The L1 cache has the same size on both machines but test-machine 1 has a three level cache while test-machine 2 only has a two level cache.

10.3 Scalability

To measure the scalability we need to take Amdahl's- and Gustafson's law in consideration. In the final vectorized version, we measured that 95.8% of the code can be parallelized. According to Amdahl's law, this gives us a possible speed-up of 8.2 times with 12 threads. On the other hand, Gustafson's law gives us a possible speed-up of 11.5 times given a data set big enough. The real speed-up received by pure parallelization is though 6.66 times in the small test case and 8.04 times in the medium test case. The reason for not achieving a speed-up closer to Amdahl's law is believed to be the overhead of creating a parallel region each time a merge is done, especially for the first iteration that contain a small workload.

10.4 Engineering efficiency

When evaluating the engineering efficiency we take many factors in consideration, the time to learn parallelizing techniques, how easy it is to implement, how much code needs to be rewritten etc.

In this thesis, we used OpenMP that only focuses on parallelizing over the CPU-cores. OpenMP is quite easy to use since, if the code is embarrassingly parallel, you only need to add pragma directives to the code and the compiler split the work over the different cores. However, if there are many dependencies in the code and workload is uneven the programmer needs to rewrite and reorganize the code.

At first, the parallelization of the merging part in the original code was straightforward, we just added the pragma directive at the outer loop since the data used for merging every row in the resulting aperture is independent. However, when the code was optimized and vectorized a static distribution of the work was not a good option. Some optimizations reduced the number of calculations for some iterations and the distributed work got uneven. Still the modification of the OpenMP directives was quite straightforward. Some analysis of the overhead when not using a static distribution was done and the choice was to use dynamic scheduling with chunk size 1.

When parallelizing the data collection we got more complications due to data dependencies and data races. When optimizing and speeding up the data collection, the data was put in the wrong order, which gave a faulty resulting image. This was though corrected with a different data structure that knew where to put the data. The data collection also required critical sections, which locks all sections of the code handling the same variables. One challenge with these sections was to minimize them to get rid off as many stalls at possible.

Another challenge was to evaluate and choose good work-groups depending on the amount of threads that was available. The fork-join method was used where a number of threads took care of the data collection and when a merge was possible, created a new work-group and distributed the work. This choice is dependent on the amount of work that has to be done when processing the data and how many threads that is needed to take care of the received data. When optimizing the merging more and more we needed fewer threads to do the actual merge.

Besides OpenMP, we also rewrote the code to make the compiler vectorize as many of the calculations as possible. Even if every resulting row can easily be parallelized because they are not dependent of each other, there are many data dependencies when creating the row. Much effort was put on splitting up and arranging the calculations in the most efficient way in order to make the best use of the vector instructions.

11
Conclusion and further work

The FFB implementation has been optimized and parallelized so it can create an image in real-time for a small SAR system without using all the computing power (if we have more than one core) until the last pulse has been collected. This means we could reduce the approximations and get better image quality.

The time from the last pulse is collected to the image is created is quite high for the medium test case. There are however enough idle resources during the pulse collection to catch up, using at least four cores (3.06 GHz), when a series of images created in stripmap mode. Therefore, we meet the real-time demands also for a medium sized SAR system.

Using OpenMP, we were able to carry out the parallelization with high engineering efficiency, since it was easy to add parallelism and focus could be put on removing dependencies and optimizing the code.

There are however more improvements that can speed up the implementation even more. The part of the running time, that was stalled due to cache misses, increased in the vectorized and optimized version, compared to the original version. So optimizing the memory accesses would be something to look into in the future. A big motivation is to improve the memory accesses when reading non-linear paths in memory because a non-linear flight path will also cause distortion in the memory paths. Therefore, unlike the case if a linear flight path where memory paths only are distorted in the later iterations, non-linear read paths can arise for all iterations. To avoid reading in non-linear paths in the memory, it could be possible to pre-compute the indices for the contributing subapertures in a merge, before they are accessed in memory, and shuffle them to utilize the cache better.

The merging bases were not considered in the scope of this work but that could possibly result in another improvement if a better set up were found that reduced to overhead for creating parallel region while still maintaining the desired image quality.

Another possible direction to go would be to do all or part of the calculations on GPUs, which has been studied to some extent in [20].

Bibliography

[1] M. Skolnik, *Introduction to Radar Systems.* McGraw-Hill Book Co., second ed., 1980.

[2] A. Olofsson, *Real time Signal Processing for Airborne Low-Frequency Ultra Wideband SAR (In Swedish).* Chalmers., first ed., 2003.

[3] This image is used with courtesy of Annelie Wyholt.

[4] M. Budge, "Radar waveforms & signal processing," Spring 2011. `http://www.ece.uah.edu/courses/material/EE710-Merv/SARPart1_11.pdf` (24 May 2012).

[5] L. Ulander, H. Hellsten, and G. Stenström, "Synthetic-aperture radar processing using fast factorized back-projection," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 39, no. 3, pp. 760–776, 2003.

[6] L.-E. Andersson, "On the determination of a function from spherical averages," *SIAM Journal on Mathematical Analysis*, vol. 19, no. 1, pp. 214–232, 1988.

[7] L. Ulander, P. Frölind, and T. Martin, "Processing and calibration of ultra-wideband sar data from carabas-ii," *Proceedings of CEOS SAR Workshop*, vol. 450, pp. 273–278, 1999.

[8] This image is used with courtesy of Anders Åhlander.

[9] A. Åhlander, H. Hellsten, K. Lind, J. Lindgren, and B. Svensson, "Architectural challenges in memory-intensive, real-time image forming," *Parallel Processing, 2007. ICPP 2007. International Conference*, p. 35, 2007.

[10] G. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 30, no. 8, 1965.

[11] G. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," *AFIPS spring joint computer conference*, pp. 483–485, 1967.

[12] "Amdahls law." `http://upload.wikimedia.org/wikipedia/commons/e/ea/AmdahlsLaw.svg` (24 May 2012).

[13] J. Gustafson, “Reevaluating amdahl’s law,” *Communications of the ACM*, vol. 31, no. 5, 1967.

[14] “Gustafsons law.” `http://upload.wikimedia.org/wikipedia/commons/d/d7/Gustafson.png` (24 May 2012).

[15] `http://www.openmp.org/` (24 May 2012).

[16] O. A. R. Board, “Openmp application program interface,” May 2008. `http://www.openmp.org/mp-documents/spec30.pdf` (24 May 2012).

[17] “Openmp is being improved for accelerators, multicore and embedded systems,” March 2012. `http://openmp.org/wp/2012/03/openmp-is-being-improved-for-accelerators-multicore-and-embedded-systems/` (24 May 2012).

[18] “Server performance summary.” `http://www.dclink.com.ua/lib/userfiles/Server_Performance_Summary.pdf` (24 May 2012).

[19] “Fftw.” `http://www.fftw.org/` (24 May 2012).

[20] M. Blom and P. Follo, “Vhf sar image formation implemented on a gpu,” *Geoscience and Remote Sensing Symposium, 2005. IGARSS ’05. Proceedings. 2005 IEEE International*, vol. 5, pp. 3352–3356, 2005.