Real-time airborne signal processing for Synthetic Aperture Radar

by

Chris Wouters

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Monday March 8, 2021 at 14:00.

Student number:	4959876	
Thesis committee:	Dr.ir. Stephan Wong,	TU Delft, supervisor
	Dr. Przemysław Pawełczak,	TU Delft
	Ir. Matern Otten,	TNO
	Ir. Danilo Tromp,	TNO

An electronic version of this thesis is available at http://repository.tudelft.nl/.

TU Delft

Contents

1 Introduction

TNO regularly develops radar systems, with different RF frontends and digital signal processing solutions. The processing for radar often reuses similar algorithms, even if the type of radar and the applications are different. However, for every new analog RF frontend that is developed, the entire digital board, doing data acquisition and online real-time processing of the signal, has to be redesigned almost every time. The goal of this research is to produce knowledge on what computing platforms are suited for different kinds of algorithms in use, given the system constraints, and optimize the design process of future systems by making recommendations for required hardware and implementation parameters.

In order to make sensible recommendations, one radar system will be analyzed in detail, and based on the findings, conclusions can be drawn. This system is a 32-channel frequency-modulated continuous-wave (FMCW) Synthetic Aperture Radar (SAR) mounted on a drone, developed by TNO for the Office of Naval Research (ONR). It is equipped with an Intel Cyclone V FPGA, and currently being developed to be used for wire detection in fields. Other applications, that will be considered, are (ground) moving target detection (GMTI), and the M-RaISR SAR modes. The ONR system currently only does data acquisition and storage onboard, and all other processing steps are currently done offline, in MATLAB and Python.

1.1. Problem statement

With the MATLAB code for the ONR system, it takes tens of minutes to generate an image for a single angle. Eventually, the area will be 20 times as large, and there will be 10–20 more aspect angles. Multiplying these numbers gives an extra workload of 200–400 times, which gives a total processing time of days, using the poorly optimized MATLAB code. Since this system will be used in the field, to quickly scan an area for objects, this is way too long. Ideally, the latency should be brought back to seconds, to enable real-time operation.

In this thesis, real-time SAR processing on small unmanned aerial vehicles (UAV) will be considered. These types of aircraft give rise to the following two limitations:

- The typical available wireless link has a speed on the order of 5 Mbps, limiting the possibilities of only doing data acquisition on the drone, and transmitting it to the ground, where real-time processing is done. Data compression can be attempted.
- Computing resources are limited by power and area usage, and the preference of a solution without an operating system running on a CPU.

In addition to aiding the hardware engineer in developing digital processing solutions for future airborne SAR systems, the communication between the radar engineer and the hardware engineer can be improved. When designing an algorithm for a new SAR system, the radar engineer typically assumes practically unlimited numerical precision (using double-precision floating points), and leaves it to the hardware engineer to adapt the algorithm for implementation on an FPGA. Therefore, if a feedback loop between hardware and radar engineer can be created, the development process can be sped up significantly.

1.2. Research questions

Based on the introduction and problem statement above, the main research question can be formulated:

How can real-time performance be achieved for specific algorithms found in many synthetic aperture radar systems, within the limitations of airborne applications?

The main question can be answered by studying a number of sub-questions, that are listed below:

1. What are the relevant system parameters that affect the computational requirements for the radar processing algorithms that are regularly used in different types of systems?
2. Of those algorithms, how can they be implemented for reuse in different systems? What parameters should be varied?
3. What is the impact of implementation in hardware on numerical precision?
4. What are the hardware requirements (e.g. type of FPGA), given the system parameters and required processing?
5. What development tools are needed?
6. What power (order of magnitude) do specific algorithms use on a chosen platform?
7. What is the reusability of the implementations for offline processing if real-time is not a requirement?

The answers to such questions lead to a better understanding of what type of processing architectures are suitable for compact radar processing solutions.

1.3. Focus and methodology

Two digital signal processing algorithms will be considered. For each of them, literature study will be performed, to investigate their inner workings. Consequently, the current MATLAB code of the ONR system is to be benchmarked and expanded with instrumentation to store the intermediate results. Additionally, the code will be adapted to test the impact of fixed-point arithmetic on numerical precision.

Tools will be developed to aid with the development of the hardware implementations, and to assess the performance in terms of latency, throughput, area and numerical precision. The hardware implementations themselves need to be parametrized, to make them adaptable to any given radar system. Ideally, the results will show a selection of radar systems and the type of FPGA that is large enough to fit all the processing. If the algorithms do not fit on any FPGA on the market, it can be investigated if it will still be useful to do data reduction, and perform the real-time processing on the ground.

1.4. Thesis outline

The thesis is organized as follows:

Chapter 2
This chapter explains the background knowledge required for understanding SAR, and introduces the system context and parameters of the radar systems that will be studied. Consequently, the theory of the signal processing algorithms *range compression* and *backprojection* is explained.

Chapter 3
In order to be able to make estimations about future airborne SAR systems, the implementation of range compression for one particular system has to be made. This process is explained in this chapter. After that, the methods of evaluating the numerical precision, throughput, latency and area are introduced.

Chapter 4
Similar to Chapter 3, this chapter is all about the implementation of backprojection. A motivation is given for the implementation platform (language) and design requirements. Consequently, it is explained how the implementation works, and lastly, the methods of evaluating the numerical precision are introduced, also giving answers to how the communication between the radar engineer and hardware engineer can be improved.

Chapters 5 and 6
The following two chapters are about the results, recommendations and conclusions for the two implemented algorithms (range compression and backprojection). Predictions from Chapters 3 and 4 are verified, and overviews are given of the required resources for a range of studied SAR systems. These results also enable predictions for future SAR systems, using extrapolation.

Chapter 7
A summary of the entire thesis is given, the research questions are answered, and future work is recommended.

2

Background

In this chapter, an explanation of the relevant theory and system context will be given. In particular, the following things will be discussed:

- Brief explanation of Synthetic Aperture Radar (Section 2.1)
- An overview of signal processing chain, algorithms and parameter ranges that will be considered. (Section 2.2)
- A note on beamforming in SAR (Section 2.3)
- Theory of range compression (Section 2.4)
- Theory of backprojection (Section 2.5)
- Conclusions (Section 2.6)

2.1. Synthetic Aperture Radar

Synthetic Aperture Radar (SAR) is a type of radar system, which scans a target while in motion with respect to that target. The biggest advantage over a stationary radar is increased resolution[32][15]. The resolution of a radar system depends on the size of the antennas *(aperture)*. For a SAR system, the received echos are accumulated over the travelled distance into a *synthetic* aperture, high resolutions can be achieved with a small physical aperture[19].

Figure 2.1 shows the geometry of an airborne SAR. In the figure, the X and Y directions are called the azimuth and range, respectively. The image is created by processing all information of the received echos during the *integration time*, during which a certain distance is travelled along the azimuth axis. The image of the ground essentially respresents a 2D reflectivity map of the terrain[22]. The azimuth and range resolutions

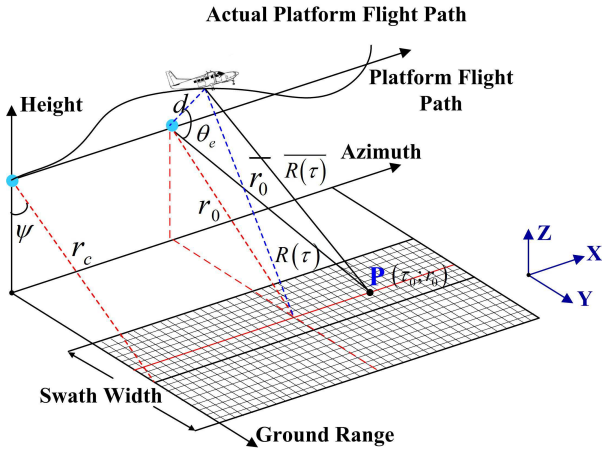


Figure 2.1: FMCW SAR imaging geometry. Source: [2]

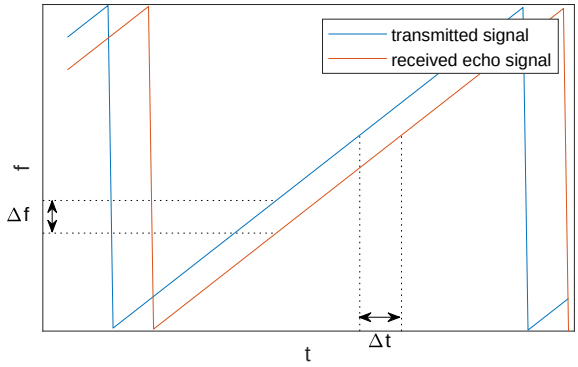


Figure 2.2: The FMCW radar works by sweeping the frequency of the transmitted signal. The phase delay Δt of the received echo signal manifests itself as a beating frequency Δf when it is demodulated using the transmitted signal. This is why the first step in the processing chain (range compression) consists of transforming the signal to the frequency domain (FFT). See also Fig. 2.3

are independent, since they depend on different parameters. Whereas the range resolution is inversely proportional to the bandwidth of the transmitted waves (B) [22], the azimuth resolution depends on the size of the aperture (and therefore the distance flown) [20].

SAR can be run in multiple different *modes*, by changing the pattern at which the target is illuminated[22]. Every mode has specific advantages in terms of resolution or swath width. In this work, modes that will be considered are stripmap and spotlight, as well as (ground) moving target indication (GMTI). The details of the inner workings of these modes are not relevant for this research, since the implementation of the algorithms is identical.

2.1.1. FMCW

In this project, we will consider Frequency-Modulated Continuous-Wave (FMCW) radar as opposed to a Pulse-Doppler radar. It has the following advantages:

- Outputting short high-power pulses is less power efficient than continuously transmitting power[2].
- For a pulse radar, the pulse is sampled and digitized directly. This requires a sampling frequency of twice the bandwidth (2 GHz in the case of this system). For FMCW, the demodulated signal is sampled, requiring a much lower sampling frequency.
- The range resolution can be much higher with the same bandwidth and Pulse Repetition Frequency (PRF) [34].

With a Pulse-Doppler radar, the delay between the sent pulse and the received echo is measured, and from this, the range of the object can be calculated (distance away from the radar) [29]:

$$R = \frac{c \cdot \Delta t}{2} \tag{2.1}$$

where c is the speed of light and Δt is the delay between two pulses.

In a FMCW radar, there are no pulses, but a continuously sweeping wave, as can be seen in Fig. 2.2. The received signal is already mixed in hardware with the transmitted signal before it arrives at the ADC. The output is called the *intermediate frequency* (IF), which is the sum of all beating frequencies. This beating frequency is shown as Δf in Fig. 2.2. In the digital domain, the signal is stored as separate *sweeps*, that will be processed and combined into an image.

2.1.2. Why SAR?

As mentioned in Section 2.1, SAR has significant advantages over stationary radar, especially on a moving platform like airborne or spaceborne radar. However traditionally, in order to make real-time SAR processing practical, assumptions had to be made to speed up the calculations, which compromise image quality. For example, those assumptions could involve restricting the ground positions or antenna positions along the flight path to regular grids[32]. Some of the algorithms that use those approximations are hybrid correlation [26][30] and omega-K [5].

Since about the year 2009, processing platforms became small and efficient enough, so that it became possible for some applications to perform real-time SAR processing without any compromises on accuracy

[32]. Instead of doing assumptions on antenna or ground positions, the exact GPS coordinate could be used for each recorded sweep, and the exact desired coordinate on the ground for each image pixel. A visualization of this can be seen in Fig. 2.1, where the actual platform flight path can be very different than a straight line, especially on smaller aircraft. To accomplish generating an image without doing those assumptions, for every pixel, the partial reflectivity of each sweep has to be calculated, and accumulated. The complexity of generating a image is $\mathcal{O}(N^3)$ for an $N \times N$ SAR image [9], where every "operation" in turn involves many multiplications, some square roots and a complex exponent[31]. Concluding, SAR image processing can be considered very computationally intensive.

2.1.3. Processing platforms: motivation for using FPGAs

Since the generation of a SAR image is an “embarassingly” parallel – the algorithms are linear and the pixels are all independent from each other[24] – it is attractive to use processing platforms that provide paralellism, like GPUs, FPGAs or Cell processors[32]. For this work, FPGAs were chosen, for multiple reasons:

- Previous work on real-time SAR on GPUs has already been done at TNO [28].
- GPUs require an operating system to run, whereas FPGAs can run standalone. This gives FPGAs an advantage in applications where weight and power are of limited supply, like airborne and spaceborne applications.
- FPGAs were found to be significantly more power efficient for fast-factorized backprojection [33], which is an algorithm used in SAR image processing, similar to global brute-force backprojection, which will be considered in this thesis.
- Modern work involving another SAR mode (VideoSAR) shows promising new technology using FPGAs: [40]. This application is similar to ONR in the number of aspect angles and the integration times (for explanation of those terms, see the following sections).

2.2. SAR systems and parameters

We now have a rough understanding how SAR works in general. The next step is to find a way to evaluate the real-time processing workload. In order to do this, we first introduce two signal processing algorithms that will be considered, and then take a look at the global system parameters. Then, in Section 2.2.1, the ONR SAR system is introduced for which an implementation was made. How the findings from this will be used to draw conclusions about other systems, is explained in Section 2.2.2. Therefore, this section will answer research question 1.

Algorithms

Range compression

This involves converting the time-domain information of the ADC samples into the frequency domain. It is implemented using a Fast Fourier Transform (FFT). The frequency domain corresponds to range (distance) as seen in all direction from the radar antennas. The theory of range compression is explained in Section 2.4, and Chapter 3 discusses how to approach implementing it in hardware.

Backprojection

This algorithm takes all the sweeps from the output of the range compression step, and integrates them into a final image. For the theory of backprojection, refer Section 2.5. How it was implemented is shown in Chapter 4.

Parameters

The computational complexity of the radar signal processing algorithms depend on the system parameters that are listed below. Not all parameters are independent: some of them can be expressed in terms of the others. However, explaining how these parameters are determined is outside the scope of this thesis, since it would require much more understanding of SAR design theory. We assume that the values are given by the radar engineer.

Sample rate

The rate at which the ADC samples the antennas.

Bit depth

Bits per sample of input data, which can be the raw SAR data, or the range compressed sweeps (as input for backprojection).

Decimation factor
The downsampling factor before processing.

Integration time
The total time in seconds over which data is integrated into one final image. This determines the **azimuth resolution**.

Pixels per second
pixels per second = swath width × flying speed/(resolution)^2

Upsampling factor
After range compression, the sweeps can be upsampled. This increases the frequency resolution. This can either be done separately, or embedded in the range compression (by using a larger FFT and zero-padding the input – this is done in the ONR system).

Sweep time or PRF
The total time per sweep, or the number of sweeps per second (Pulse Repetition Frequency). This directly influences the workload of the range compression with respect to the backprojection. If the sweeps are shorter, the range compression gets easier, since the complexity of an FFT is $\mathcal{O}(N \log N)$ using big O notation. On the other hand, with shorter sweeps, more of them have to be integrated to form a final image using backprojection, which has complexity $\mathcal{O}(MXY)$, for an $X \times Y$ SAR image with M sweeps [33]. So in theory, decreasing the sweep length will make the complexity of the FFT proportionally less than the complexity of the backprojection will increase.

Beamformed channels
The number of channels for which the entire processing chain has to be repeated. In this thesis, beamforming will not be implemented, but its purpose and working will be discussed in Section 2.3.

From these parameters, we can derive the number of samples per sweep:

$$\text{samples per sweep} = \frac{\text{sample rate} \times \text{sweep time}}{\text{decimation factor}} \times \textit{(upsampling factor)} \tag{2.2}$$

Multiplication with the upsampling factor is only applicable when the upsampling is done inside the FFT, as is the case with the ONR radar. For more on upsampling, see [14].

2.2.1. ONR SAR system on drone

The block diagram of the signal processing chain for the ONR SAR system is shown in Fig. 2.3. The system contains 32 antennas, but typically, around 8 are used simultaneously in order to get the desired field-of-view[25]. This means that only a section of the sphere around the drone can be illuminated at once. The first processing step is range compression. In this thesis, the optional pre-beamforming step is not considered. The output of the range compression step is a set of sweeps, that are needed in the backprojection step in order to generate the final 2D image. The ONR SAR system is a multi-aspect SAR. This means that the backprojection step is repeated multiple times, to generate 10 to 20 images of the same terrain as seen from different angles[25].

In addition to the sweeps, the backprojection step also requires as its input the real-time coordinates of the radar, in addition to the sweeps, and the output is the final image, that can be sent to the ground or shown on the screen in real-time.

Values for the parameters listed in Section 2.2 for the ONR system are shown in Table 2.1. If N_{FFT} is calculated using Eq. (2.2), the value is higher than expected, because only a fixed number of 2048 samples is taken during every sweep. The sweep time (1/PRF) is slower than the time to take 2048 samples at approximately 2 MHz. Therefore there is some "dead time" in every sweep.

2.2.2. Expansion to other systems

Most other SAR systems also require range compression and backprojection, albeit with different parameters. Based on the findings from exploring the ONR system, conclusions will be drawn for other applications that are related. TNO has determined parameters as in Table 2.1 for some other SAR and (G)MTI modes. For the MTI modes, we will only focus GMTI on the ONR system. Other (G)MTI modes are reserved for future work. The parameter values will be shown in Chapter 3 and Chapter 4, for range compression and backprojection, respectively.

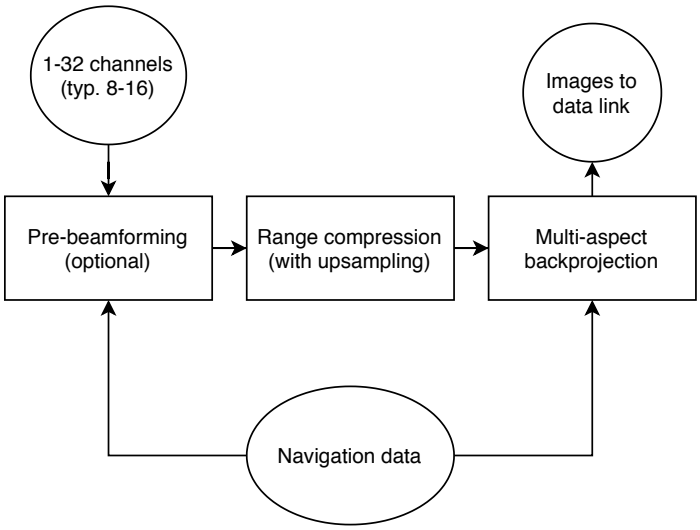


Figure 2.3: Block diagram of the real-time signal processing algorithms of the ONR drone radar

Table 2.1: System parameters of ONR.

Sample rate	1 953 125 Hz
Bit depth	16
Decimation factor	1
Integration time	4 s
Pixels/s	500 000
Upsampling factor	8
PRF	814 Hz
Channels	8 – 16
N_{FFT}	16384

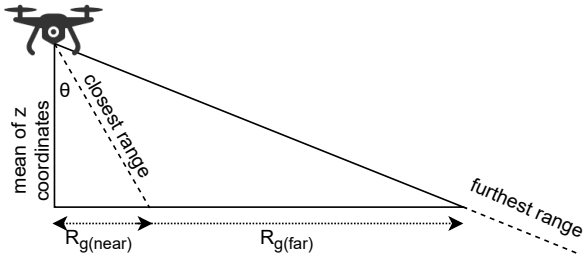


Figure 2.4: This is a two-dimensional section of the illumination sphere around the radar. To decrease data storage requirements and possibly processing power, the parts of the FFT spectrum that correspond to the dotted lines can be discarded. The input parameters $R_{g(near)}$ and $R_{g(far)}$ are used to calculate this range, and $R_{g(near)}$ is related to the smallest viewing angle $\theta > 20°$ that provides useful data.

2.3. Beamforming

As mentioned when discussing the parameters in Section 2.2, we will not look into the implementation of beamforming in hardware, but since beamforming is a very useful concept in SAR, and may be used in the ONR system (as shown in Fig. 2.3), its purpose is briefly described here.

With SAR, there is a tradeoff between the signal-to-noise ratio (SNR), and the resolution. Ideally, we want high SNR and small resolution (in meters). High SNR is achieved by using a large antenna. However, the resolution can never be smaller than the antenna size, so larger antennas cause larger pixels. To get the best of both worlds, beamforming can be used to combine the signals of multiple smaller antennas into one, so that a larger antenna is simulated, with the benefit of high SNR. The resulting signal then keeps its advantage of small resolution. This does not come for free, however: beamforming adds a processing overhead.

2.4. Range compression

In Section 2.1.1, the concept of sweeps was introduced, and the fact that a sweep contains the beating frequencies of the reflections of all the objects in range. To extract the individual beating frequencies and their phases, this signal has to be transformed to the frequency domain by means of an FFT. This is called *range compression*, and it is the first algorithm from Fig. 2.3, that will be discussed.

In this section, we will first derive the *range* (distance) as seen from the radar, that every FFT bin corresponds to. Based on this knowledge, we will arrive at a possible optimization of the range compression algorithm by discarding part of the output and thus saving processing time and memory (in Section 3.4.3).

2.4.1. Determining the range of the observed objects

After range compression, we have Δf for every object, as shown in Fig. 2.2. The slope of the frequency sweep $\frac{df}{dt}$ is a system parameter called the *sweep rate* (α). It is simply the radar bandwidth (B) multiplied by the Pulse Repetition Frequency (PRF):

$$R = \frac{c \cdot \Delta f}{2\alpha} = \frac{c \cdot \Delta f}{2B \cdot (PRF)} \tag{2.3}$$

Since the width of an FFT frequency bin is determined as follows:

$$\Delta f = \frac{f_s}{N_{FFT}} \tag{2.4}$$

it is now possible to calculate the range (distance) that every FFT bin corresponds to:

$$R_{bin} = \frac{c \cdot \frac{f_s}{N_{FFT}}}{2B \cdot (PRF)} \tag{2.5}$$

2.4.2. Finding the maximum range

The drone on which the radar system is mounted, is equipped with an IMU to provide navigational data. Amongst others, it provides three-dimensional coordinates. In this section, we will use the z-coordinate to find a way in which possibly processing time and data storage can be saved. This is done by only storing the part of the FFT spectrum that we are interested in. The total range of the radar is larger than necessary, so we

Table 2.2: System parameters of the ONR drone radar and the available dataset.

Input parameters	
PRF	814 Hz
Radar bandwidth (B)	1 GHz
N_{FFT}	16384
$R_{g(far)}$	75 m [1]
$R_{g(near)}$	10 m
θ	20°
Calculated parameters	
Range per FFT bin (R_{bin})	2.195 cm
z_{drone}	31.061 m
Relevant FFT bins	1485 to 3698

[1] The system supports larger values of $R_{g(far)}$, but for the sake of this analysis, this is a typical value.

can discard the part of the spectrum that extends beyond the ground level. Additionally, we can discard the part of the spectrum that is nearest the radar. To calculate the number of FFT bins that we can discard, we use the following input parameters of the system: the far ground range $R_{g(far)}$ of the radar image segment and the near ground range $R_{g(near)}$. Ranges that are further resp. nearer than this distance can be discarded, as illustrated in Fig. 2.4. Another way to calculate it is to use the looking angle of the radar: from around 20 degrees with respect to the z-axis, the data becomes relevant. We can calculate the relevant FFT bins, based on the range per FFT bin from Eq. (2.5):

$$\#_{FFTbin} = \left\lceil \frac{\sqrt{{z_{drone}}^2 + {R_g}^2}}{R_{bin}} \right\rceil \tag{2.6}$$

where z_{drone} is the mean of all z coordinates of the flight path (i.e. the height of the drone above the ground) and R_g is the far or near ground range.

If the looking angle θ is used, $R_{g(near)}$ can be calculated as follows:

$$R_{g(near)} = z_{drone} \tan(\theta) \tag{2.7}$$

In Table 2.2, the real-world parameters can be found, specifically for the ONR drone radar, and the dataset that was used for this project.

2.5. Backprojection

The second algorithm from Fig. 2.3, that will be discussed, is *backprojection*. As mentioned in Section 2.1.2, there are many ways to implement backprojection, making various assumptions to decrease computational load. Here, however, we will focus on brute-force time-domain backprojection [32]. This algorithm incrementally calculates radar reflectivity for each desired location on the ground (pixel).

For every pixel, we iterate over all the range compressed sweeps. For every sweep, the partial contribution of the reflected power is determined by finding the correct range (FFT bin) for that ground location, given the radar antenna's exact position. All the partial contributions of the sweeps are accumulated. This is done for every channel (antenna).

2.5.1. Algorithm

The algorithm will now be studied in more detail. First of all, we define the image as a set of world locations $\vec{w} \in W$. Since the image is two-dimensional, we call the individual elements pixels and denote them with $p(\vec{w})$. A single range compressed sweep with complex samples is defined as $s[r]$, indexed by range r. If we assume that the drone or airplane travels a negligible distance during the capture time of a single sweep (called the *stop-and-go* assumption), the observation of the sweep is done by the antenna at three-dimensional position $\vec{a}$. Since multiple sweeps have to be observed before an image can be constructed, we use the index k

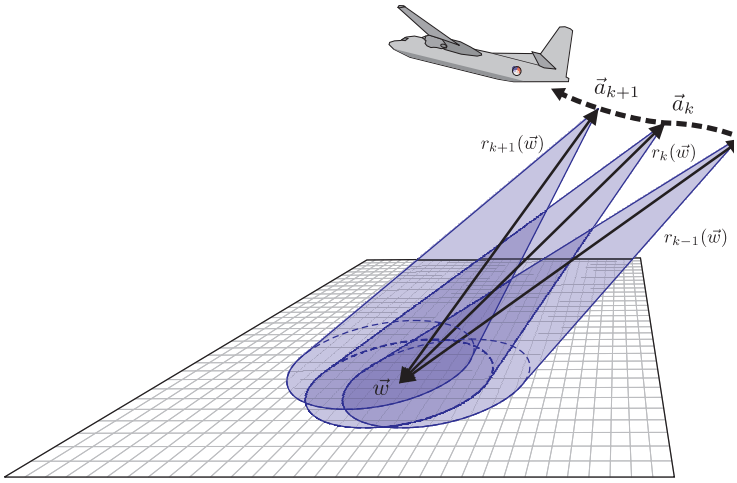


Figure 2.5: Illustration of the geometry of the brute-force backprojection algorithm. After the plane has captured enough data to produce an image, range compression is performed on all sweeps individually. Consequently, backprojection can be done for every sweep k by calculating the ranges $r_k(\vec{w})$ between the world positions $\vec{w} \in W$ and the antenna positions $\vec{a}_k$. The sweep data is then projected back to the ground along the estimated phase shift, calculated from the range. Source: [32]

to refer to the sweep index, giving rise to sweep $s_k[r]$ and corresponding antenna positions $\vec{a}_k$. We have now defined the variables as shown in Fig. 2.5.

Assuming only one channel, the range r (the Euclidean distance between the antenna and the world position) can be calculated as follows:

$$r_k(\vec{w}) = |\vec{w} - \vec{a}_k| \tag{2.8}$$

In the ONR system, the actual antenna position $\vec{a}_k$ first has to be derived from the rotation matrix $\mathbf{R}$ that is provided by the IMU and the fixed antenna offset coordinates $\vec{o}$:

$$\vec{a}_k = \mathbf{R}\vec{o} \tag{2.9}$$

In order to calculate the partial contribution of the sweep to the pixel, we need the complex phasor $\Theta_k(\vec{w})$, which is a complex exponential of the phase shift $\phi_k(\vec{w})$ that the distance $r_k(\vec{w})$ is equivalent to:

$$\phi_k(\vec{w}) = 2\pi \frac{r_k(\vec{w})}{\lambda} \tag{2.10}$$

$$\Theta_k(\vec{w}) = e^{2j\phi_k(\vec{w})} \tag{2.11}$$

where λ is the wavelength of the transmitted wave.

We now get to the exciting part, and the part that gives this algorithm its name: we are going to shift ("project") back the radar observations (sweeps) along the phase $\phi(\vec{w})$, by multiplying it with the complex conjugate of the phasors, which yields us the image, comprising of pixels $p(\vec{w})$:

$$\forall \vec{w} \in W \quad p(\vec{w}) = \sum_{k \in K_{\vec{w}}} s_k[r_k(\vec{w})] \tag{2.12}$$

Since ONR and most of the other modes considered have multiple channels, all these calculations have to be repeated for the set of channels C:

$$p(\vec{w}) = \sum_{c \in C} p_c(\vec{w}) \tag{2.13}$$

2.5.2. Making it real-time

The goal is to make the backprojection implementation real-time. This means, that all the data, that the radar captures, must be processed within the time needed to capture that data. As explained in Section 2.1,

the total integration time T determines how much time is needed to capture enough data for the desired azimuth resolution δ_a in meters per pixel. It can be calculated as follows [22]:

$$T = \frac{\lambda r_0}{2v\delta_a} \tag{2.14}$$

where λ is the wavelength, r_0 is the slant range as shown in Fig. 2.1, and v is the radar velocity relative to the target.

Alternatively, we can calculate the number of pixels per second that are required for real-time operation, assuming that the range and azimuth resolution δ are identical:

$$\text{px/s} = \frac{d_{swath}v}{\delta^2} \tag{2.15}$$

where d_{swath} is the swath with as shown in Fig. 2.1.

The actual number of pixels per second, that the implementation achieves, can then be compared to this value.

2.6. Conclusions

In this chapter, the relevant theory and system context was given. SAR was introduced as a radar technique superior to conventional stationary radar in terms of resolution. However, the computational requirements are large. The two most important signal processing algorithms for SAR were introduced: range compression and backprojection. Those will be implemented for a range of different applications. The implementation platform will be an FPGA, as we saw that the algorithms are easily parallellizable, and FPGAs are significantly more power efficient than GPUs and Cell processors, which is important in airborne applications.

The system that will be studied in most detail, is the ONR system. A block diagram of the processing chain was given. When the implementations work for ONR, they will be tested for a range of other applications.

To summarize, SAR works by moving the radar relative to the object that is to be imaged. Sweeps of raw data are collected. The larger the distance flown, the higher the azimuth resolution, and the higher the radar bandwidth, the higher the range resolution.

Range compression is implemented as simply an FFT. The raw SAR data is a set of beating frequencies corresponding to the ranges (distances) of the reflections. In the frequency domain, we can use this information to build an image, where the reflections – from peaks in the FFT spectrum – are shown as bright spots in the corresponding locations. The integration of all the range compressed sweeps into a final image is done using backprojection.

Backprojection works by calculating the range (and corresponding phase shift) between each ground location to the radar, for every sweep, and using that to shift ("project") back the raw SAR data along the phase. All these observations are then accumulated for every sweep, and every channel, and the final image is yielded.

3

Range compression in hardware

In this chapter, the first step of the processing chain will be explored: range compression. The goal of this thesis is to make generalized recommendations for different radar systems. However, in order to accomplish that, one system has to be studied in detail first. Therefore, in this chapter, we will study the implementation of range compression for the ONR system (Section 2.2.1).

The organization of this chapter is shown below:

- Section 3.1: introduction;
- Section 3.2: the parameters of the IP core;
- Section 3.3: numerical precision: bit width of calculations;
- Section 3.4: throughput: sweeps must be processed at the Pulse Repetition Frequency (PRF);
- Section 3.5: a note on latency;
- Section 3.6: selecting the optimal FFT core configuration;
- Section 3.7: conclusions.

3.1. Introduction

In Section 2.1.3, it was explained that FPGAs are the best hardware platform for airborne SAR image generation. Instead of implementing the FFT function from scratch, a prewritten implementation will be used. This choice was made in order to be able to spend more time on the main research question (Section 1.2). The development time can then be minimized, since the point of this research is not to find the best implementation for any specific radar system.

For the prewritten implementation, the Intel FFT IP core [12] is the most obvious choice, since we are using an Intel Cyclone FPGA. An implementation from Opencores was also considered [8]. This seemed favourable because the code is open source, but the project is not actively maintained, and according to the readme has not been tested on real hardware yet. Therefore, the Intel FFT IP core was chosen.

3.1.1. Motivation for performing full onboard real-time processing

As stated in the problem statement in Section 1.1, the following processing methods can be identified:

1. Full onboard real-time processing
2. Partial onboard processing or data reduction
3. Real-time processing on ground
4. Non-real-time processing on ground

Option 4 is not a good option, because images should be available in real-time, as the radar is active. Therefore, at first sight the easiest method seems option 3, because on the ground there are less strict limitations on power, processing power and weight. However, all raw data has to be transferred to the ground in real-time, which requires a fast wireless link. The speed of the link is calculated as follows:

$$\text{link speed} = \text{channels} \times \text{sample rate} \times \text{bit depth} \tag{3.1}$$

Table 3.1: System parameters for the SAR/GMTI applications that are considered for range compression

application	PRF	Samples per sweep	N_{FFT}	# channels
ONR SAR	814	2048	16384	8
ONR GMTI	3333	300	512	8–32
other GMTI	2000	1024	1024	8–32
M-RaISR SAR strip	500	30000	32768	1
M-RaISR SAR spot	400	75000	65536 or 131072	4

Substituting the numbers from Table 2.1 gives 250 Mbps. This rate is totally infeasible for a low-power medium-distance wireless link – in fact, those links typically have a speed of 5 Mbps [24]. Therefore, option 3 is not possible to implement. Option 1 is then left as most desirable, since the design is simpler than option 2, because there is no separate processing system on the ground, and the wireless link only has to transfer back the images. However, if onboard data reduction is not sufficient, option 2 is the only option.

3.1.2. Radar systems

Range compression will be implemented for a number of SAR and GMTI applications. The specific parameters of these systems are shown in Table 3.1. For more information on what these mean, refer to Section 2.2.

3.1.3. Onboard FPGA

The SAR system board that is developed by TNO for ONR, contains an FPGA. Originally, this board was only intended to do data acquisition and storage. However, for this project, this FPGA will be used as a reference to see what is possible with it. The board is of the following type: Enclustra MA-MA3-C6-7I-D10. It contains an Intel Cyclone V type 5CSXFC6C6U23I7N. This FPGA has 557 RAM blocks of each 10240 bits, 84 DSP blocks that can do two 18×19 signed complex multiplications or one 27×27 multiplication [13].

3.2. Intel FFT IP core parameters

As mentioned in Section 3.1, the FFT function of the range compression will be implemented by using pre-existing IP from Intel. The Intel FFT IP core is configurable during design time with a number of parameters. Table 3.2 lists the relevant parameters. In this system, the length is fixed at 16384, as explained in Section 2.2. How the bitwidth will be determined, will be explained in Section 3.3 and the results are in Section 5.1. The data representation is coupled to the *data flow* configuration – all configurations except Variable Streaming demand the use of block floating points. For an overview of block floating points, refer to Section 3.3.4.

The data flow is the parameter that gives the most control over the latency, together with the number of output engines. The IP core estimates its latency immediately when setting the parameters, but to get the area usage, synthesis is necessary. Two types of latency are given, that are called *calculation latency* and *throughput latency*. Their meaning is not obvious from the name, and it is not explained in [12]. An additional complication is that the Burst configuration does not estimate its latency correctly when using one output multiplier. Therefore, simulations had to be run to find this out.

3.2.1. Twiddle factors

The twiddle factors are constants, that are multiplied with the data. The value of the constants are independent of the data – they are complex exponents, and we will not go into detail of the mathematics of FFTs. In hardware, the twiddle factors can be stored in ROM. The bitwidth can be set for the data and the twiddle factors separately. An experiment will be done to reduce the twiddle factor width, without reducing the data width. Since reducing the bitwidth introduces a constant error to the FFT output, it is expected that the FFT The results of this are in Section 5.1.1.

3.2.2. Types of latency

While simulating, the following relevant types of latency can be measured, in order to discover what calculation and throughput latency mean:

1. Time between first input sample and first output sample

Table 3.2: Most relevant Intel FFT IP core parameters

Transform length	2^3 to 2^{18}
Data flow	memory usage / latency tradeoff
Burst	slowest
Buffered Burst	
Streaming	fastest, together with Variable Streaming
Variable Streaming	enables changing the transform length during runtime
Data representation	
Fixed point	simplest and uses least area; only for Variable Streaming
Single floating point	only for Variable Streaming
Block floating point	Advantages of both; uses a single exponent per block. Is the only option for Buffered Burst and Streaming.
Bitwidth	8 to 32
Data	For fixed point, input and output width are specified separately
Twiddle factors	
Number of multipliers in output engine	1 (only for Burst) or 4
Number of output engines	1, 2 or 4

Table 3.3: The latencies that the Intel FFT core reports for a 16384-length Buffered Burst core.

	latency (ns)	latency (cycles)
calculation latency	122880	18432
throughput latency	82620	12393

2. Time between first input sample and last output sample
3. Time between first input sample and first input sample of next sweep

We are only interested in type 3, since we want to determine if the FFT core is fast enough to keep up with the PRF. From the latencies reported for the (Variable) Streaming modes, it can be seen that type 1 is not reported by the Intel Parameter Editor. That is because the throughput and calculation latencies are the same – both are equal to the FFT length, which means that there is an output sample for every input sample. For (Buffered) Burst, there is a difference.

3.2.3. Matching the latency prediction with measurements

In order to determine which latency is the relevant one, we need to learn about a few input and output signals of the FFT core. Not all of them are relevant, so only a subset will be discussed. First of all, `sink_` means that the signal is an input, and `source_` means that it is an output.

`*_sop`
: Start of packet. A pulse of this signal means that the first sample of the sweep is pushed in or out. We want to know what is the frequency at which the FFT core accepts input sweeps, so we will measure the time between two pulses of `sink_sop`.

`*_eop`
: End of packet. This signal is pulsed at the last sample of the sweep.

`sink_ready`
: When this is high, it means that the core is ready to receive input samples.

A 16384-length Buffered Burst core was simulated using a hand-written testbench that inputs four sweeps and captures the FFT output. The latencies that this core reports in the Intel Parameter Editor are shown in Table 3.3. In Fig. 3.1, the simulation results are shown. In Table 3.4, measurement results of time differences between signal pulses are shown. The colored cells are matching up (approximately), and from this it can be concluded that the calculation latency is the relevant metric that we are after. The throughput latency does not seem to be relevant at all, because it shows the latency between the input and output of seemingly the

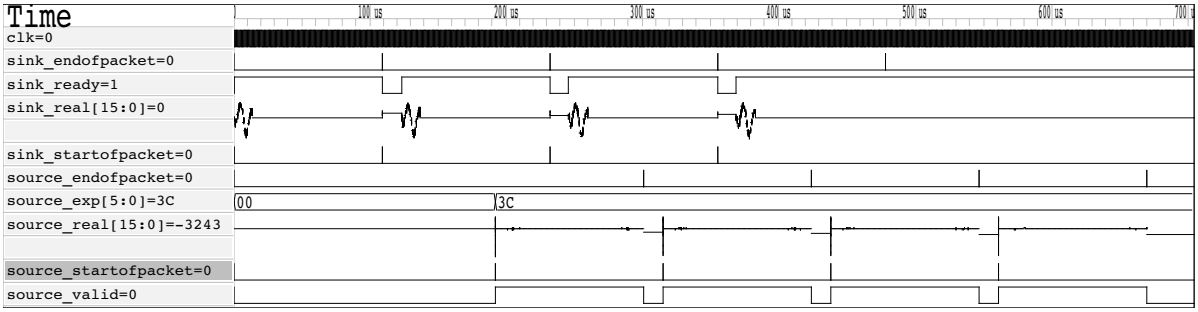


Figure 3.1: Simulation of four sweeps of a 16384-length Buffered Burst FFT core. From the simulation, conclusions can be drawn about the latencies that the Intel Parameter Editor reports.

Table 3.4: Measured time differences between different points in the simulation of the 16384-length Buffered Burst core from Table 3.3.

measured from – to	latency (ns)	latency (cycles)
`sink_sop` sweep 1 – `source_sop` sweep 1	191887	28783
`sink_sop` sweep 1 – `sink_sop` sweep 2	122948	18442
`sink_sop` sweep 2 – `source_sop` sweep 1	82672	12401

wrong order of sweeps. The reported latency is not correct, however, if the number of output multipliers is 1. This was found by accident while running the simulations.

3.3. Numerical precision

The original implementation in MATLAB uses double-precision floating point numbers. When designing a real-time processing solution in hardware, floating point arithmetic is usually not practical, because of the much larger latency and area usage compared to fixed point (integer) arithmetic. Therefore, in this section, we will explore the tradeoffs of numerical precision using the FFT IP core.

Firstly, Sections 3.3.1 to 3.3.3 are about how the numerical precision is evaluated. Section 3.3.4 explains how block floating point numbers work, which is the number representation that is used in (Buffered) Burst and Streaming modes of the Intel FFT core. Floating point and fixed point are options for Variable Streaming, and there are some notes on that in Section 3.3.5. Finally, Section 3.3.6 shows how to lower memory requirements with approximately a factor four using block-adaptive quantization.

3.3.1. Evaluation of precision

In order to assess the numerical precision, two different metrics will be used:

1. mean relative error of the output of the hardware FFT with respect to the full-precision MATLAB FFT;
2. noise floor of the backprojected image.

Item 1 is not relevant by itself for the final image, but it can be used to determine the amount of input scaling (Section 3.3.2). Item 2 is explained in Section 3.3.3.

An estimation can be made of the required bitwidth of the FFT. There are 2048 samples per sweep, which are all added by the FFT. We are using a 16384 FFT for oversampling, but the extra samples are all 0. The theoretical bit increase is therefore 11. However, the SNR increase of a signal that includes noise is only half, as the noise power itself increases by N instead of N^2. So the expected effective bit increase is 5-6. On top of 11 bits effective input that is 16-17 bits.

3.3.2. Input scaling

The FFT core is the most accurate, when the input samples are of the largest possible amplitude within the amount of bits available. Most sweeps do not make use of the full 16-bit width of the samples. Therefore, to increase accuracy, the raw data has to be scaled up, before feeding it to the FFT. There are two possible approaches:

1. Normalization by finding the largest sample in the sweep;

2. Increasing the average power, while setting overflowed samples to the maximum value – wrapping around introduces a larger error.

The second approach may improve the accuracy of the FFT, however, it will introduce different errors in the backprojected image that are due to the clipping. We choose the first approach because it is simpler, and adds only a very small overhead.

After scaling up and running the FFT, the output should be scaled down by the same factor. The scaling can be done by bitshifting, so it does not add any processing penalty. The downside is that a less optimal scaling is performed, because we can only scale with powers of two.

The number of bits N_s that the input should be shifted with can be calculated as follows:

$$N_s = L - \lceil \log_2 \max \boldsymbol{x} \rceil - 1 \tag{3.2}$$

where L is the bitwidth of the FFT function and $\boldsymbol{x}$ is the input vector. The subtraction with 1 is needed because signed integers are used.

To verify this theory and to quantify the errors that are introduced if improper input scaling is done, bit-accurate simulations with real-world input data were run. The results are discussed in Section 5.1.1.

3.3.3. Quantization increases the noise floor of the backprojected image

In this section, a hypothesis will be made about how a lower-accuracy FFT will impact the final backprojected image. A simplified view on using less bits for the data path of the FFT is simply that the output of the full-accuracy FFT is quantized. This should introduce quantization noise in the final image. This quantization noise should increase the noise floor of the image, which could be measured in the darkest part of the image – the part where there are no objects visible. When compared to the image that is made using the full-accuracy FFT, the average power of the darkest part should increase if it is lower than the noise floor. From physics, we know that the power of a field quantity is proportional to its amplitude squared. Therefore we can average all the pixels in the darkest part of the image, and calculate the square. The increase in decibels with respect to the original image can then be calculated. The expectation is that the image is not degraded if the lowest levels are not increased. The result of this will be shown in Section 5.1.2.

3.3.4. Block floating point

Floating point numbers can express a large dynamic range with many fewer bits than fixed point numbers. However, floating point units cost a lot of area and are slow. A hybrid between the two representations is *block floating point* (BFP). With floating point, every number has an individual exponent. With BFP, this exponent is the same for an entire block of data, that has roughly the same range. This applies well to raw SAR data. Therefore, only fixed-point arithmetic is needed, while still having the advantage of large dynamic range in the whole algorithm [6].

The Intel FFT core uses BFP for the (Buffered) Burst and Streaming configurations [12]. After every stage in the algorithm, the data is scaled down by bit shifting. The total number of right shifts is accumulated, and output as the signal `source_exp`. This can be seen in Fig. 3.1. The output therefore has to be shifted left by `source_exp` bits to get the correct magnitude.

3.3.5. Fixed point (for FFT lengths > 65536)

The Variable Streaming configuration supports floating point number representations until an FFT length of 65536. Above that, only fixed-point is supported. As mentioned in Table 3.2, the input and output bitwidth should be specified separately. Since every stage grows the data with maximum 2 bits [12], and there are $\log_2 N$ stages, the maximum needed output width is:

$$\text{input width} + \log_2(N) + 1 \tag{3.3}$$

3.3.6. Block-adaptive quantization

Block-adaptive quantization (BAQ) is a way to encode the raw SAR data in a way that the dynamic range is wider than if only truncation is used [16] [7]. Recent work has introduced multichannel BAC (MC-BAC), that can reduce the bitwidth from 16 to 4 bits per sample [21]. If this technique is used, the data can be compressed with a factor 4, potentially enabling real-time processing on the ground instead of the onboard FPGA. Substituting the parameters from Table 2.1 into Eq. (3.1), the data rate of the digitized raw SAR data

can be calculated:

$$8\,\text{channels} \cdot 1953125\,\text{Hz} \cdot 16\,\text{bit}/4 = 62.5\,\text{Mbps} \tag{3.4}$$

Since the wireless downlink from the drone to the ground is only around 4 Mbps, a reduction of an order of magnitude or larger is needed, to make real-time processing on the ground possible (see option 2 from Section 3.1.1).

3.4. Throughput

This section will deal with the throughput, i.e. the number of processed sweeps per second. Firstly, Section 3.4.1 will introduce parallel processing. Subsequently, Section 3.4.2 will show how the time budget per sweep is calculated, and Section 3.4.3 will show a way to save a significant amount of memory and processing time by discarding unused data.

3.4.1. Processing in parallel with multiple FFT cores

If the system contains multiple channels that have to be processed per sweep, they can be processed sequentially, using one faster FFT core, or in parallel, using slower (and smaller) cores. Partly parallel is also an option, if the number of channels is 4 or more. If one FFT core is not fast enough to process all the channels sequentially, instantiating multiple cores is the only option.

It is expected that running multiple slower cores in parallel will use more area than using fewer faster cores. The reason for this is that there is more overhead in input and output signals, interconnect, and memory. The optimal configuration will be selected using a simple algorithm, that will be explained in Section 3.6.

3.4.2. Processing time budget

The radar systems from Table 3.1, that we are considering, have a fixed PRF. Therefore, there is a fixed amount of available time for processing each sweep. This time is calculated in cycles as follows. First, we calculate the total number of cycles available per sweep C_{tot}:

$$C_{tot} = \frac{f_{clk}}{\text{PRF}} \tag{3.5}$$

The number of available cycles per core C_{core} depends on how many channels each core has to process:

$$C_{core} = \left\lfloor \frac{C_{tot}}{\left\lceil \frac{N_{\text{channels}}}{N_{\text{cores}}} \right\rceil} \right\rfloor \tag{3.6}$$

From now on, we will assume a clock frequency of 150 MHz. This seems a reasonable number, since the Intel FFT IP core manual lists f_{max} to be 132 to 240 MHz on the Cyclone V [12, p. 9-10]. To maintain real-time processing, the backprojection step also has to be finished within the available time. However, since backprojection can run in parallel on the previous sweep, the FFT can use all available time. This will increase latency by an additional sweep, but this is not significant, as will be explained in Section 3.5.

3.4.3. Saving cycles in Burst mode

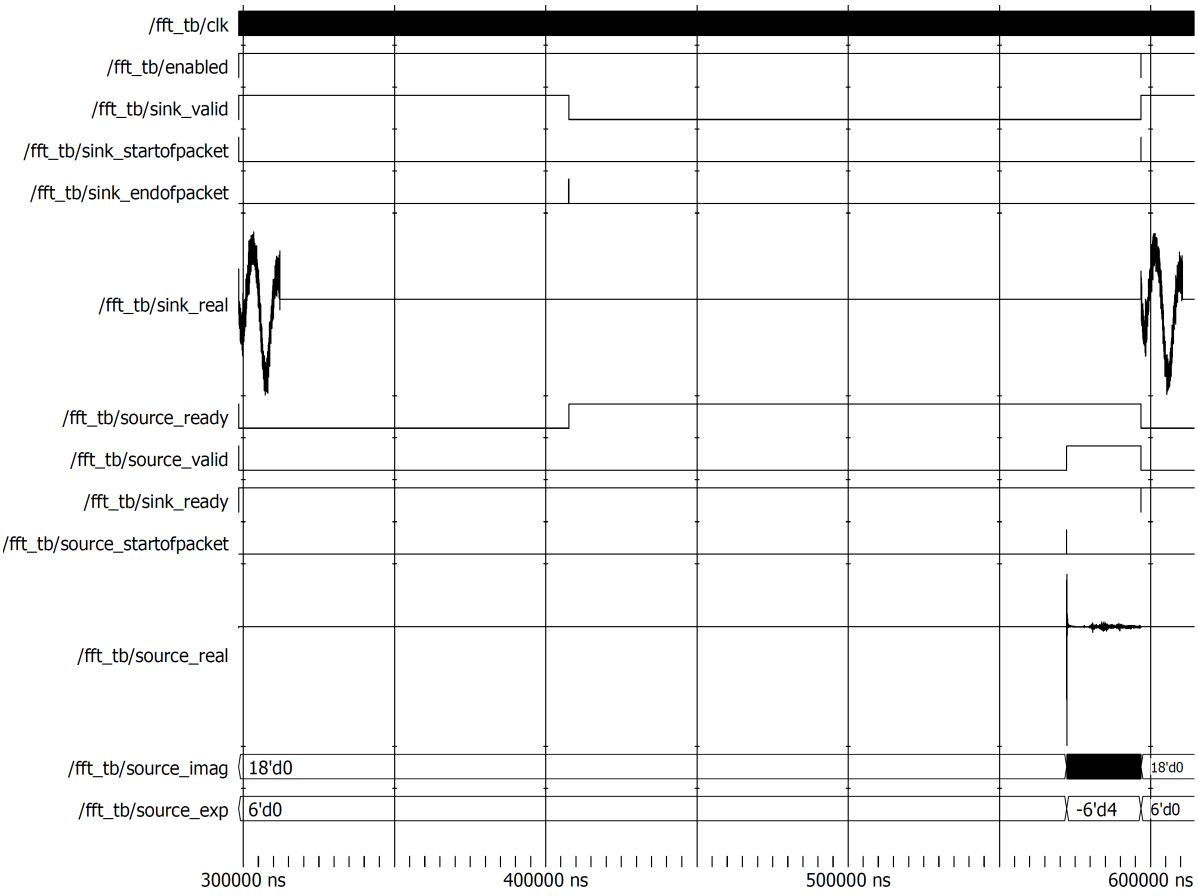
In Section 2.4.2, the range of relevant FFT bins were calculated, which for the ONR drone radar is bin 1485 to 3698 (Table 2.2). Because all other data can be discarded, it is possible to save RAM resources in the FPGA. Additionally, for some FFT configurations, processing time can be saved as well. With the Burst configuration, all output samples are pushed out, before input samples of the next sweep are accepted. Therefore, the FFT core can be reset after the first 3698 samples have been pushed out. Figure 3.2 shows this method in practice.

If the FFT were implemented manually, even more optimizations would be possible, by not only discarding part of the output data, but also skipping part of the computation, making use of the FFT structure (decimation in time or frequency).

3.5. Latency

For a real-time processing system, the latency can be equally important as the throughput. However, since the integration time of the ONR drone radar is 4 seconds, a backprojected image is only expected every 4 seconds (assuming no overlap). We can define that the total latency of the signal processing chain should be

Figure 3.2: Simulation of the FFT core in the Burst configuration. The core is reset after receiving 3690 bins from the output to save processing cycles (the `enabled` signal goes low just before 600000 ns).



under an order of magnitude below the integration time, to ensure a smooth user experience. That would mean that there can be a latency of the equivalent of over 300 sweeps, based on the PRF of 814 Hz. This is quite a large amount, so we conclude that latency is not a critical parameter to optimize.

3.6. Finding the optimal configuration

An SQLite database will be made with FFT configurations. The parameters of Table 3.2 will be varied, and for each application in Table 3.1, the optimal configuration will be found. The optimal configuration is the one with smallest area, while still being fast enough to finish before the next sweep starts. It can be assumed in general, that there is an inverse relationship between the area and the latency of the FFT core (this will also be verified in Section 5.2.4). The latency of a single core determines the throughput of the entire system – if we define throughput as sweeps per second. The selection will be done using an algorithm, as described in Section 3.6.2. However, first, we will look at the concept of area in an FPGA (Section 3.6.1).

3.6.1. How to determine FPGA area usage

For an FPGA, the circuit area cannot be straightforwardly defined, like for an ASIC. The actual area of the blocks inside the FPGA is not so relevant, since we are not designing the actual chip, but only using the already designed chip. Therefore, we have to make a useful analysis of the used blocks. There are there main types of blocks:

- Adaptive logic modules (ALMs) with look-up tables (LUTs)
- RAM blocks
- DSP blocks

Section 5.2 will show that the number of ALMs and DSP blocks are mostly proportional to each other, whereas the RAM blocks change independently, so they will be separately regarded when determining the area usage.

3.6.2. Algorithm

The synthesis results of many FFT configurations, where the parameters from Table 3.2 are varied, are stored in an SQLite database. For every application in Table 3.1, a query is done on the SQLite database to select the configurations whose latency fits within the processing time budget for the current application. The results are sorted, in a way that the highest-latency configurations are at the top. This is done because the expectation is that higher latency means less area. If there is a configuration with lower RAM usage and lower latency, however, that one is selected. Lower DSP or ALM usage is given less weight, because those resources are available in abundance (Section 5.2). A configuration with less DSP and ALM usage will only be selected, if this does not result in more RAM usage.

A description of this algorithm is also given in Fig. 3.3.

3.7. Conclusions

In this chapter, the implementation of the range compression algorithm was discussed. The goal is real-time processing, but since the data is generated on an airborne platform, a choice has to be made between processing on the ground or onboard. Despite the possibility of compressing the data with a factor 4 using block-adaptive quantization, the wireless data link is not fast enough for realtime data transmission. Therefore, full onboard processing was chosen as the preferred method.

Range compression is implemented as an FFT. We can choose to implement the FFT from scratch, or use a ready-made parametrizable implementation. Keeping the main research question in mind, the goal of this thesis is not to invent a new implementation of an FFT. Rather, we want to produce knowledge about how SAR signal processing on FPGAs behave for different types of radar systems. Therefore, it was chosen to use a prewritten implementation.

The following points are important to remember from this chapter:

- The FFT IP core reports two types of latency. *Calculation latency* is the time needed for processing of one sweep, except when the number of output multipliers is 1. Then simulation is needed to find the actual latency. Optimizing the latency of a single sweep is equivalent to optimizing the throughput of the system, because more sweeps can be processed per second.
- Multiple FFT cores processing in parallel will be considered if one core is too slow. Using an algorithm, the lowest-area configuration will be found.

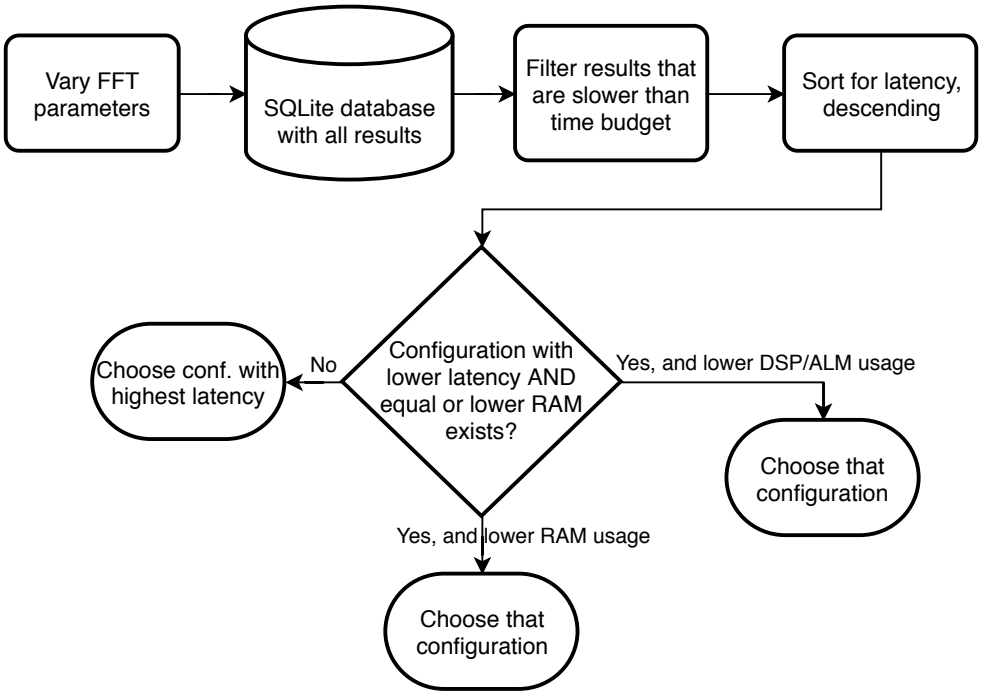


Figure 3.3: Block diagram of the algorithm that selects the most optimal FFT core configuration, using the least area.

- The calculations can be sped up by discarding a part of the FFT spectrum without losing any information.
- The latency of the calculation of the FFT of all the sweeps is not a critical parameter for optimization, as opposed to throughput.
- The input samples have to be scaled up to make full use of full output bit width. If the dynamic range is not used fully, the relative error will be higher.
- The required FFT width is expected to be 16-17 bits.
- The numerical precision of the FFT will be verified by generating a backprojected image, and verifying whether the noise floor of the image increased. Quantization noise should stay below the original noise floor in order to not add extra noise.

4

Backprojection in hardware

After implementing the first algorithm from Fig. 2.3 in Chapter 3 (range compression), this chapter is about the implementation of backprojection. For the theory of how backprojection works, Section 2.5 can be referenced. The implementation will initially be made for the ONR system (Section 2.2.1), but since the goal of this thesis is to make generalized recommendations for different radar systems, the implementation will be made parametrized and tested for a few others, with possibility for easy expansion for arbitrary future FMCW SAR systems.

The organization is as follows:

- Section 4.2: an elaboration on how to determine the computational requirements of the radar systems that will be considered;
- **??**: implementation platform and design goals that the implementation must adhere to;
- Section 4.3: theory of some HLS aspects
- Section 4.4: implementation details
- Section 4.5: how the numerical precision will be analyzed.

This chapter focuses on the higher-level design choices. If more practical detail is desired of how the implementation and testing framework is built, and how the results can be reproduced, Appendix B can be referenced.

4.1. Platform

For range compression, it was decided to use pre-written libraries for the actual implementation (Section 3.1). However, since backprojection is an algorithm that is specific for SAR, it is not present in any standard library and we will need to study previous work done on backprojection implementations on FPGAs. The work that was studied includes [4][18][27][40]. They will be cited again where appropriate, when the details are discussed.

Since the goal of this thesis is to produce knowledge for TNO for the real-time airborne SAR applications, an own implementation will be written. The implementation will be optimized for the ONR system, but parametrized so it can be adjusted to work for other systems as well. In this section, a motivation will be given for which implementation platform was chosen.

For hardware implementations, many platforms and languages are available. Apart from VHDL and Verilog, there are multiple flavours of High Level Synthesis (HLS), and newer technologies like CλaSH [1]. With HLS, hardware is inferred from imperative languages originally designed for CPUs, and CλaSH is a compiler for the functional language Haskell, that enables designing cycle-accurate RTL using functional abstractions and Haskell's type system.

The advantage of HLS is that the design process is much simpler for the hardware engineer, since many aspects like scheduling and binding are done automatically by the tooling. However, it is more difficult to get an optimal result. With CλaSH, the code is much clearer to read and less error prone to design than VHDL / Verilog, and testing is much easier, but the exact architecture still has to be designed by the engineer [10], making it more complex than HLS. Table 4.1 shows an overview of the considered languages, and their qualities and flaws.

Table 4.1: Overview of some platforms that could be used for the implementation of backprojection, and their advantages and disadvantages.

	VHDL / Verilog	Vivado HLS	CλaSH
Design complexity	high	low	medium
Quality of results	high	medium	high
Testing	hard	easy	easy
TNO experience	yes	no	no

All things considered, Vivado HLS was chosen as the design platform. For this research, it is not crucial to obtain an optimal result, since the objective is to get an orientation for the hardware requirements for the different SAR modes. Additionally, TNO had no prior experience with Vivado HLS, giving more opportunities for learning about this technology. A Xilinx ZC702 design board with a Zynq-7000 SoC was provided for prototyping.

4.2. Parameters and requirements

In Section 2.2, the system parameters were introduced that are relevant to the evaluation of the computational complexity of the signal processing algorithms. In this section, an overview will be given of the parameters specifically important for backprojection, and the requirements that the design has to adhere to.

4.2.1. Parameters

For each SAR mode, the parameters listed below were determined. For a reminder on the general geometry and workings of SAR, refer to Section 2.1.

- Integration time (Section 2.5.2)
- **Dimensions** of image:
  - **Azimuth direction** = flying speed × flying time/resolution
    Instead of generating the entire image for all the data collected at once, for real-time operation it is more practical to generate sub-images at a regular frequency. This can be done during each integration time, for instance. If smaller or larger images are desired, this can be adjusted without having an effect on the computation complexity.
  - **Range direction** = swath width/resolution
- **pixels/s** = swath width × flying speed/(resolution)^2 × aspect angles
  As explained in Section 2.5.2, this is the fundamental parameter that determines if the implementation is actually real-time.
- which **FFT bins** to store in memory
  Section 2.4.1 explains this topic.
- **sweeps/image** = integration time × PRF
- number of channels
- The minimum **clock speed**, can be calculated, making a number of assumptions (e.g. one image pixel is produced every period, and the contribution of channels is done in parallel).

4.2.2. Design requirements

The final implementation needs to adhere to the following requirements:

- The image should be finished within the set time, as explained in Section 2.5.2.
- It has to fit on a Zynq-7000 FPGA.
- The code should be parametrizable for other SAR systems. It will be tested for the parameters of the systems that are shown in Table 4.2 (ONR, MRaISR Strip, MRaISR Spot, MRaISR Spot short range).

In Table 4.2, the number of operations per second required for real-time image generation can be found. This number was calculated from the theoretical complexity as mentioned in Section 2.2: $\mathcal{O}(MXY)$, where M is the number of sweeps, and X and Y are the dimensions of the image. We also have to multiply this with the number of channels. These numbers do not give a lot of information by themselves, but in Section 6.2, they will be compared to the hardware resource usage on the FPGA in order to verify the correlation between this number and the hardware resource usage.

Table 4.2: System parameters of the SAR applications that are considered for backprojection.

	ONR	M-RaISR strip	M-RaISR spot short	M-RaISR spot
Integration time	1 s	4 s	9 s	9 s
Resolution	10 cm	1 m	15 cm	30 cm
Px (azi. × range) [1]	50 × 750	120 × 5000	1800 × 2000	900 × 1667
Pixels / s	375 000	150 000	400 000	166 667
FFT bins	3611	17009	6966	7459
sweeps/image	814	2000	3600	3600
Channels	8 – 16	1	4	4
operations / s	2.4×10^9 [2] 3.7×10^9 [3] 4.9×10^9 [4]	3×10^8	5.7×10^9	2.4×10^9
min. clock speed [5]	305 MHz [6]	300 MHz [6]	1440 MHz [6]	600 MHz [6]

[1] Assuming an image is generated for every integration time. If images spanning a different area are desired – with identical resolution –, this can be changed without having an effect on the computational complexity.
[2] 8 channels
[3] 12 channels
[4] 16 channels
[5] Assuming all sweeps are processed sequentially, and all channels in parallel.
[6] Since the highest clock frequency of the ZC702 development board is 250 MHz, it will be necessary to generate multiple pixels of the image in parallel.

Additionally, Table 4.2 shows the minimum required clock frequency of the FPGA, assuming parallel processing of all channels, but sequential processing of every sweep and also image pixel. Since the maximum clock speed of the ZC702 development board is 250 MHz, multiple image-generating cores will need to be instantiated in parallel to retain real-time operation.

4.3. HLS theory

Before explaining the details of the implementation of backprojection in HLS in Section 4.4, some aspects of HLS will be explained for the unfamiliar reader. For further reading, refer to the Vivado Design Suite User Guide [37] and the Vivado HLS Optimization Methodology Guide [36].

4.3.1. HLS stream

Tasks inside FPGAs perform best when random-access reads to large memories are avoided, because memories have a limited number of input and output ports. Therefore, if it can be guaranteed that access to large arrays is sequential, the memory can be replaced by a FIFO (e.g. a shift register), which reduces resources and improves performance. One way to guarantee that access is sequential, is to use the **HLS streaming** interface, which allows only reading every array value once and in sequence.

4.3.2. Loop unrolling

When writing a loop for a program that is to be run on a single-threaded CPU, the iterations are always performed sequentially. On an FPGA, a choice can be made if the iterations should be executed in time, or in space. A hybrid of the two is also possible, parallelizing only part of the iterations. **Loop unrolling** directives can be added to the HLS code to instruct the compiler to instantiate the hardware inside the loop multiple times. In some cases, the loop is automatically unrolled.

4.3.3. Pipelining

Pipelining is a method to significantly speed up the hardware function, with only a small area penalty. Figure 4.1 shows the basics of how it works. Without pipelining, all operations in the iteration of the loop are executed sequentially. Every operation is only active once per iteration – but is taking up valuable resources on the FPGA. When the intermediate results of the operations are stored in registers, and the new inputs are fed in when the processing of the old input is done, the hardware can be active a larger percentage of cycles,

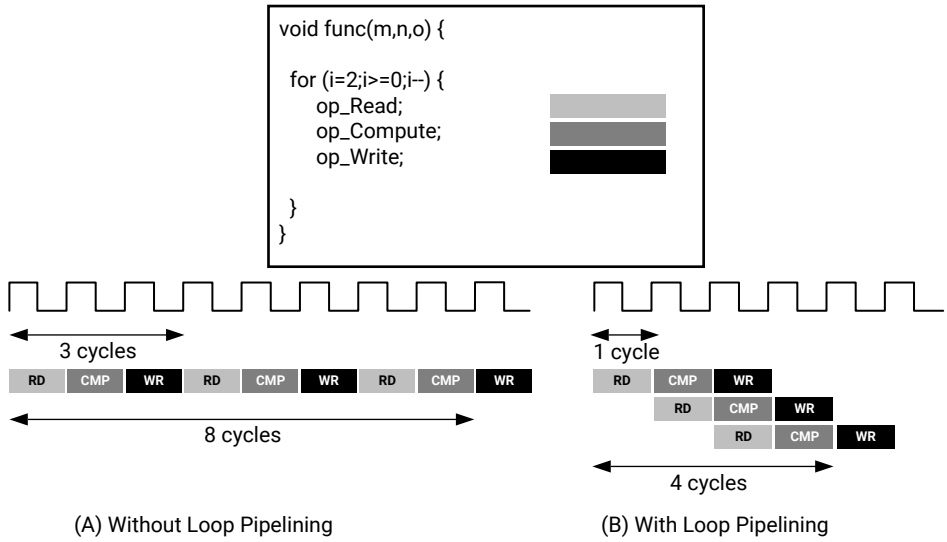


Figure 4.1: Optimizing for throughput using pipelining. Source: [36]

therefore improving the throughput. The number of cycles that it takes for the hardware to be ready for the next input is called the *initiation interval* (II). In Fig. 4.1, this is 1 cycle.

4.3.4. Task-level parallelism: dataflow

By default, Vivado HLS sequentially schedules tasks (functions) that depend on each other's data. When parallel execution of multiple tasks is desired, the **dataflow** pragma can be used. Vivado will then attempt to schedule the functions in parallel as much as possible, starting the each task as soon as all the data it depends on is available. How this works, is shown in Fig. 4.2. It can be seen, that without dataflow, the end result of one iteration is available after 8 cycles, and new input can be fed in after 8 cycles as well. With dataflow, this is reduced to 5 respectively 3 cycles.

With dataflow optimization, extra memories have to be employed, similar to pipelining. The intermediate results of the different tasks can then be stored, and used by the dependent tasks in the next iteration. Vivado HLS will create this hardware structure automatically if the code follows some guidelines:

- Follow single producer-consumer model
  Values or arrays can only be written once, by one task and read once, by another task.
- Tasks cannot be bypassed or conditionally executed
- No feedback between tasks
- Single exit condition from loops

The dataflow hardware structure is shown in Fig. 4.3. Between each task, memories are inferred. This memory can be implemented as a pingpong buffer or a FIFO. Pingpong buffers are used when random access is desired, and FIFOs are sufficient when the data can be streamed sequentially. How pingpong buffers work, is shown in Fig. 4.4. The producing task writes into one memory, while the consuming task reads from the other memory concurrently. The next iteration, the memories are swapped. This causes one iteration extra latency, but the gains in throughput are large, if the number of iterations is high enough.

4.4. Implementation

In Section 2.5, the backprojection algorithm was explained. This section describes how an FPGA implementation was made from the original mathematics.

4.4.1. Approach

At first sight, two possible implementation approaches can be identified:

1. pixel-by-pixel: store all sweeps, and calculate each pixel once, by looping over all sweeps for every pixel of the final image;

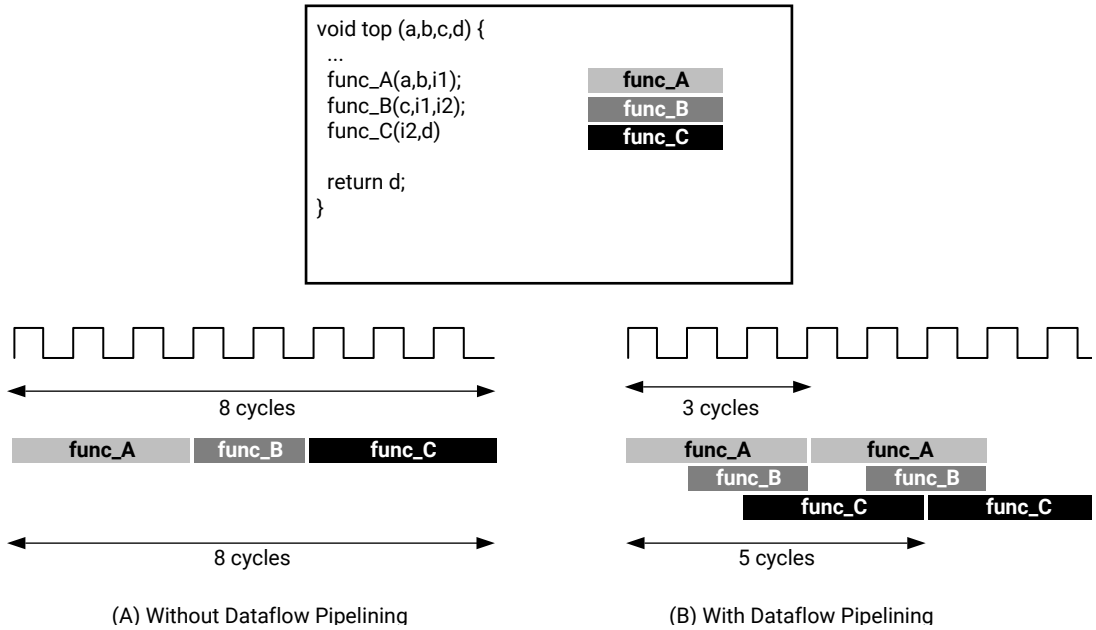


Figure 4.2: Optimizing for throughput using task-level paralellism (DATAFLOW pragma). Source: [36]

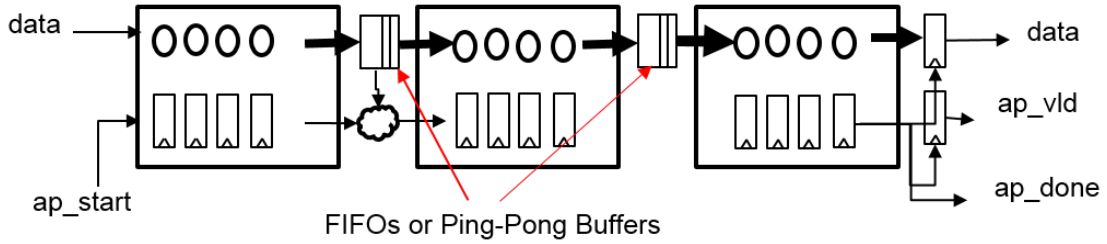


Figure 4.3: Hardware structure created when using dataflow optimization. Source: [37]

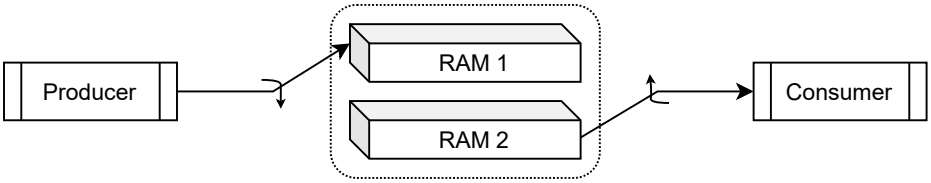


Figure 4.4: Pingpong buffer

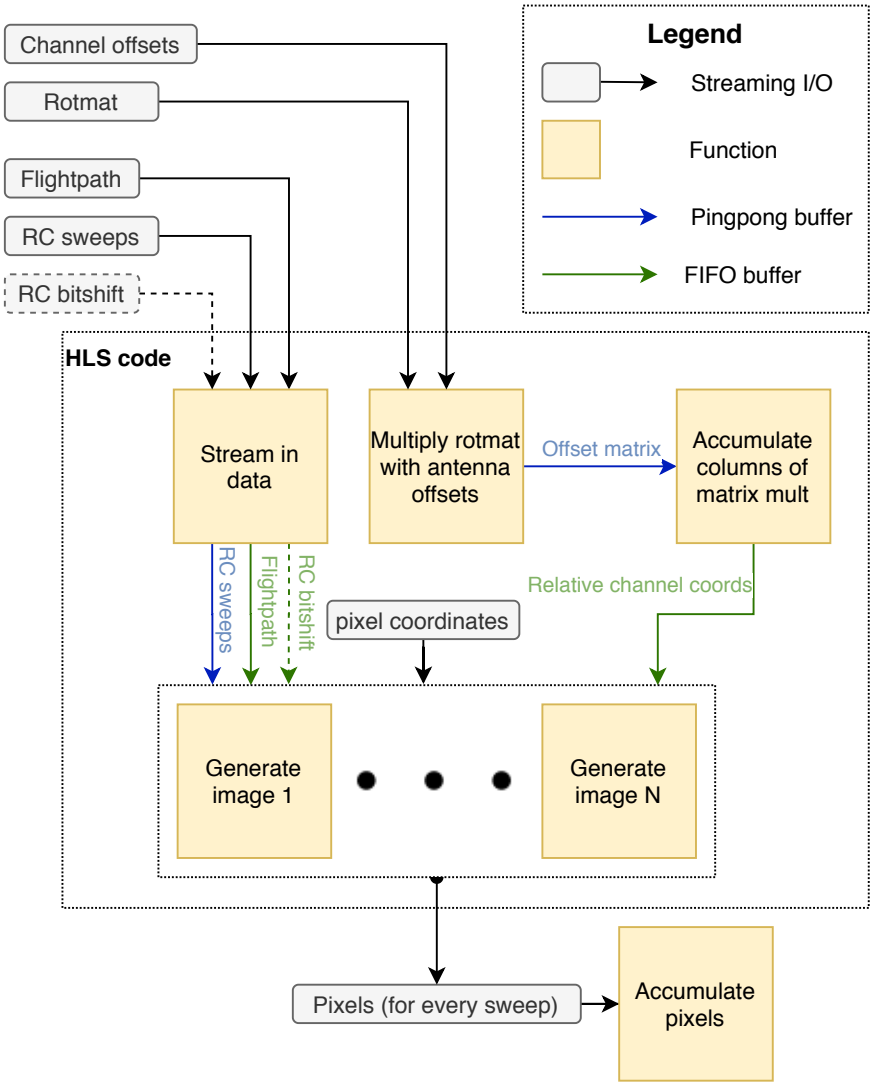


Figure 4.5: Block diagram of the HLS implementation of backprojection. For explanation on the entries in the legend, refer to Section 4.3 or [36] and [37].

2. sweep-by-sweep: update all pixels of the final image after a new sweep is available.

The two approaches are equivalent in the number of required calculations, but the second approach requires less memory, since only one sweep and the final image need to be stored. The final image should in theory always be smaller than the input data, because the intention is that it is a more efficient representation of the information. Assuming that the dynamic range of the output is used optimally, it cannot be larger than the input, because if it were, redundant information is added. Therefore the second approach is chosen for implementation.

4.4.2. Block diagram

A block diagram of the entire implementation is given in Fig. 4.5. It will be explained below.

I/O and accumulation of pixels

The inputs and outputs are streaming, so access is sequential. For the arrays that need random access (the rotation matrix, channel offsets and the range compressed sweeps), localized caches are created. A fundamental problem for backprojection inside FPGAs is that the pixels of the image are updated for every sweep. This means, that the image would need to be stored in RAM, and a lot of I/O access is required. However, inside the FPGA, not enough block RAMs are available. Previous work has solved this by building the image line-by-line [4], therefore requiring less RAM at a time. Other work has used two parallel external SRAMs [18].

In this work, it was chosen to postpone the accumulation of the sub-images, and let the CPU of the Zynq-7000 SoC do it offline, effectively making it recommended future work.

Dataflow optimization
The functions as shown in Fig. 4.5 are inside a dataflow optimization (Section 4.3.4). In order to adhere to the code guidelines, a variable can only have a single producer and a single consumer. Since a matrix multiplication requires reading and writing to the same memory location, it had to be separated in two different functions: one for the multiplication, and one for the accumulation of the columns. Below, the intermediate memories between the functions are listed:

Range compressed sweeps
From the mathematical description of the algorithm in Section 2.5.1, it can be seen that access is random, and therefore a pingpong buffer is needed.

Flightpath
The sweeps and coordinates are processed in order, so a FIFO suffices.

Bitshift of RC sweeps
A bitshift correction is required for every sweep (in order), therefore a FIFO.

Offset matrix
Random access: therefore pingpong buffer.

Relative channel coordinates
Because the channels are processed in order, a FIFO buffer does the job.

Parametrization
The following parameters can be changed at compile-time:

- dimensions of image (azimuth and range)
- number of sweeps required for one image
- number of relevant FFT bins per sweep
- number of channels
- number of parallel image generation instances
  Following the conclusion of Table 4.2, multiple instances are required for real-time operation.
- Whether or not to use half floats or lookup tables for trigonometry or the square root (see Sections 4.4.3 and 4.4.4)

Further optimizations
In order to process one pixel per clock cycle, the image generation function needs to be fully pipelined with an initiation interval of 1. Since all channels are processed in parallel, the flightpath, sweeps and channel coordinates need to be partitioned into separate FIFOs / block RAMs.

4.4.3. Lookup tables for trigonometry and square root

In addition of using the trigonometry and square root functions from the HLS math library provided by Vivado, lookup table versions of those functions were built, that can be turned off and on with compile-time flags. Intermediate values are interpolated linearly. For the trigonometry, the constant $k = 2\pi/\lambda$ (the factor to multiply with range to get the phase, Section 2.5.1) was recalculated as if π were half the size of the lookup table (which is 1024 in this case):

$$k_{1024} = \frac{k}{2\pi} \cdot 1024 \tag{4.1}$$

The lookups can then be done with zero performance penalty.

4.4.4. Range compressed data format

The Intel FFT IP core, introduced in Chapter 3, can output the range compressed sweeps both in floating point or integer format. A compile-time flag was added to support both formats as input to the backprojection core. Internally, the intermediate values of the image pixels are stored as floating points, so the range compressed sweeps need to be converted to floats anyway. Since 16 bits of information will be shown to be enough (Sections 5.1.1 and 6.1), half-floats are chosen as the data type for the sweeps.

Listing 4.1: HLS code snippet that performs the floating point division by two by subtracting from the exponent.

```
// convert int16 to floating point
union convert {
    float fl;
    uint32_t in;
};
Complex<convert> shifted;
shifted.real.fl = rc->real;
shifted.imag.fl = rc->imag;

// efficient way of dividing by a power of two (subtracting from the exponent)
// in order to scale the RC data back
if (shifted.real.in != 0) {
    shifted.real.in = (shifted.real.in & 0b10000000011111111111111111111111)
        | ((shifted.real.in >> 23) - rc_shift.real << 23);
}
if (shifted.imag.in != 0) {
    shifted.imag.in = (shifted.imag.in & 0b10000000011111111111111111111111)
        | ((shifted.imag.in >> 23) - rc_shift.imag << 23);
}

complex<float> rc2(shifted.real.fl, shifted.imag.fl);
```

Scaling

In Section 3.3.2, a method was introduced for making the most use out of the dynamic range of integers, by scaling up the values in each sweep using bitshifting. If the values are simply scaled back while they are still in integer format, the extra precision is lost again. Therefore, we must first convert to floating point, and then scale back. However, floats cannot be scaled by simply bitshifting as we can do with integers. The most efficient method is adding and subtracting from the exponent. Listing 4.1 shows how this is done practically in the HLS code.

4.4.5. HLS C++ to MATLAB MEX

The radar engineer generally develops his algorithms in MATLAB. The hardware engineer, however, needs to know the numerical ranges an required precisions for every intermediate value in the calculations. Additionally, there needs to be an easy way to verify the quality of the HLS implementation, by the radar engineer. To solve the first problem, the original MATLAB code can be instrumented using the MATLAB fixed-point toolbox. The second problem was solved by writing a MEX wrapper around the HLS code, to compile it for running in MATLAB. The results that the HLS implementation produces can then be directly analyzed by the radar engineer. The solutions are shown schematically in Fig. 4.6. In this way, there can exist a feedback loop between the radar engineer and the hardware engineer, and therefore answers research question 7 (and partly 5).

4.5. Numerical precision

We now have an understanding of the structure of the implementation. This section elaborates on the numerical precision requirements. Predictions are made based on theory, and simulations were done.

4.5.1. Dynamic range of image

We will start by determining the expected required bitwidth of the output pixels of the image. As a rule of thumb, the theoretical SNR of a digitized signal is 6 dB per bit [17, p. 279]. The backprojected image that was generated using the test dataset provided by TNO is shown in Fig. 5.1a. It can be seen that the dynamic range is about 100 dB. Taking a wider dynamic range of 144 dB, and using the rule of thumb, 24 bits are expected to be required for losslessly storing the backprojected image.

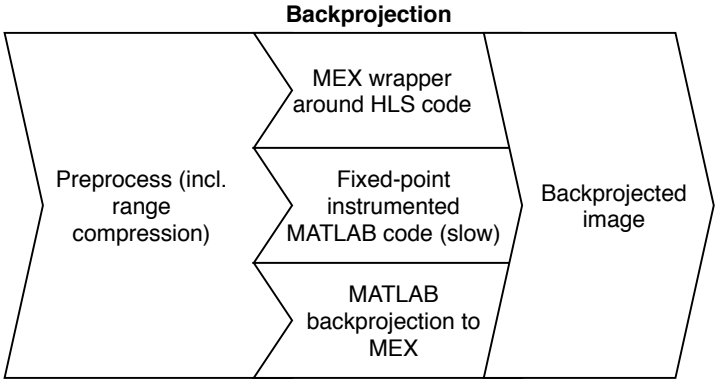


Figure 4.6: A framework was developed for MATLAB to help the radar engineer think about numerical precision for the hardware engineer, by providing three different ways to execute the backprojection algorithm.

Name	Class	Signedness	WL	FL	Percent of Current Range	Always Whole Number	Sim Min	Sim Max
R_rx	embedded.fi	Unsigned	24	17	67	No	32.104644775390625	84.90106201171875
Rxyz2	embedded.fi	Signed	24	23	100	No	-0.12584364414215088	0.9985411167144775

Figure 4.7: Determining the range of all variables in the code using the MATLAB fixed-point toolbox.

4.5.2. Fixed-point simulation in MATLAB

MATLAB's fixed-point designer was used to determine the ranges of each variable in the code. A screenshot is shown in Fig. 4.7. From the ranges, the number of required integer bits can be estimated. The number of fractional bits of every variable is more difficult to determine, since it depends on the required image quality and the exact calculations being done and their significance in the final result. So in order to determine that, full simulations have to be done, generating the images in practice.

4.5.3. Dynamic range of range compressed sweeps

In Section 3.3.1, it was determined that the expected that the range compressed data has a dynamic range of approximately 16 bits. If the data is input using integers, the same method as described in Section 3.3.2 can be used to lose as little precision as possible due to rounding. In short, the maximum value of the sweep is found, and per sweep, all data points are scaled up with a factor of a power of two, so that the largest value is as large as possible without overflowing. The values are subsequently truncated and converted to integers. Section 4.4.4 already showed how to efficiently scale down the data again.

4.5.4. Image quality metrics

After determining the integer bit width of the fixed point values as described in Section 4.5.2, the number of required fractional bits has to be determined. We are not optimizing for the best possible implementation for a specific application, but the goal of the research is to get general ideas about resource usage. Previous work was studied that goes in depth about floating point word width [27], classifying image quality as a single number. The metrics peak signal-to-noise ratio (PSNR) and structural similarity (SSIM) are used [41].

However, these numbers cannot be used to verify the correctness of the implementation, since a single number does not show where in the image errors occur, and how large those errors are. Therefore, a different approach was chosen. The image is compared with the original – full-precision – image by normalizing both and then subtracting from each other, as shown in the MATLAB code snippet below (Listing 4.2). Normalization is necessary because the images may have a constant-factor offset. This does not have an impact on the validity of the result, because the average power is not a meaningful quantity.

Section 6.1 shows the error image produced by this code.

Listing 4.2: MATLAB code for drawing a relative error image, comparing the full-precision image with the one produced by the HLS code.

```
mean1 = mean(abs(img1(:)))
mean2 = mean(abs(img2(:)))
normalized1 = img1/mean1;
normalized2 = img2/mean2;
imagesc(20 * log10(abs(normalized1 - normalized2) + 1));
```

4.6. Performance comparison with previous work

The implementation will be compared with previous work. Generally, it is hard to make a comparison between various implementations, because all implementations were tested with different parameters and datasets, and ran on different FPGAs. In [3], the algorithm identical to the one studied in this paper – velocity-independent global backprojection for FMCW SAR – was implemented. Table III in this paper contains the runtimes of the various tests that were done, and it includes the image size in pixels, and the number of sweeps per image. In this table, they are shown as $N_x \times N_y$ and N_{az} respectively. This is all the information we need to calculate the algorithm complexity, for which we have to simply multiply those numbers together – $\mathcal{O}(N^3)$, refer to Section 2.1.2.

In order to compare our results with the results of [3], we can divide all runtimes by the complexity. This normalizes the numbers and makes them more comparable. The hardware, on which the circuit is run, is is still different, but a general comparison can definitely be made. Using the same method, the speedups of the two hardware implementations can additionally be compared with any of the two MATLAB software implementations.

The results of this are shown in Section 6.2.2.

4.7. Conclusions

We discussed the implementation of backprojection in this chapter. When weighing the design complexity, quality of results, difficulty of testing the code, and whether or not TNO had prior experience with the technology, Vivado HLS was chosen as the implementation platform. Section 4.1 and in particular Table 4.1 show the considerations that were made.

Before starting the implementation, the following was design requirements were identified:

- For real-time processing, all data, that is captured during the integration time, has to be processed into a final image in time. Multiple smaller images can also be produced in the same time if desired.
- The code should be parametrizable for other SAR systems. In this way, we can fulfil the the research objective of estimating the performance requirements of future SAR systems that use backprojection.

A fully pipelined implementation was made with an initiation interval of 1. For every sweep, one or more image pixels are produced per clock cycle – depending on the amount of image generation cores that were instantiated. After implementation, the code was tested for a number of already existing SAR systems (shown in Table 4.2).

The following parameters can be changed at compile-time in the HLS code:

- dimensions of image (azimuth and range)
- number of sweeps required for one image
- number of relevant FFT bins per sweep
- number of channels
- number of parallel image generation instances
- whether to use integers or half floats for the input sweeps
- choose between lookup tables or floating point calculation for trigonometry or the square root (see Sections 4.4.3 and 4.4.4)

The bitwidths of the variables in the HLS code can be given by the radar engineer, by simulating the algorithm with fixed-point arithmetic in MATLAB. In order to validate the resulting image quality, an error image is drawn, as described in Section 4.5.4.

5

Range compression results

In this chapter, the experimental results of the range compression step will be shown. The hypotheses from Section 3.7 will be verified, and optimal FFT configurations will be chosen for each of the applications from Table 3.1. Firstly, the numerical precision and required bitwidth will be evaluated in Section 5.1. The area-latency tradeoff will be discussed in Section 5.2. Therefore, this chapter answers research questions 3 and 4.

5.1. Accuracy and errors

The numerical precision will be verified in two steps: firstly, the quantized FFT output is compared to the full-precision FFT (Section 5.1.1). Secondly, the darkest part of the backprojected image made by the quantized FFT is compared to the full-precision image, in order to see whether the noise floor increased (Section 5.1.2).

In short, this section provides answers to research question 3.

5.1.1. FFT

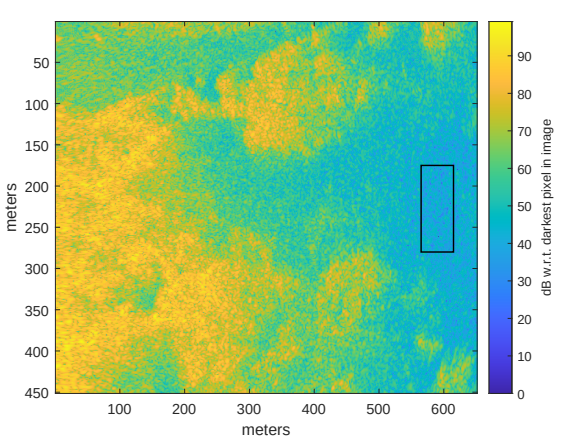
A dataset was obtained of 24414 sweeps of each 2048 samples, that were recorded during a single flight path. To determine the required bit width of the FFT core, the data width and twiddle factor width were varied, and the mean errors with respect to MATLAB's builtin double-precision `fft` function were calculated. Data widths lower than 16 bits are not possible, since the ADC samples of the IF data are 16 bits, and we can never be certain that there will be no sweeps that actually use the full range of the integer. As said in Section 3.3.2, not all 16 bits are always used – therefore the input needs to be scaled with a bitshift. To determine the impact of scaling on the average error, the amount of bitshifts was varied. The results are shown in Table 5.1. The blue cells indicate the bitshift as calculated by Eq. (3.2). It can be seen that they mostly correspond to the lowest error (except in the 20/12 configuration). When the input is shifted too much, it overflows and the error jumps up to above 100 %. Therefore, care has to be taken to never shift more than the input datatype can hold.

The first and simplest conclusion that we can draw from Table 5.1 is that varying the width of the twiddle factors does not matter much in accuracy. It will be shown in Section 5.2.2 that it also does not matter much in area savings. Going from the 20/20 configuration to 20/16 has no significant effect on the error. In some cases, the error even decreases! This is due to the random selection of input sweeps. Why it has no effect, can be seen when the twiddle factor width is decreased to 12 bits. Since the twiddle factors are constants, the errors they introduce are also constant and unrelated to the input data size. Looking at the 20/12 configuration in Table 5.1, it can be seen that the error matches the 20/20 configuration when the input is shifted fewer than 4 bits. Before that, the error that is introduced by the input not being scaled up enough dominates that of the error introduced by the twiddle factors. Beyond that point, the twiddle factor error starts dominating, and the error does not decrease anymore – it stays constant at around 3 %.

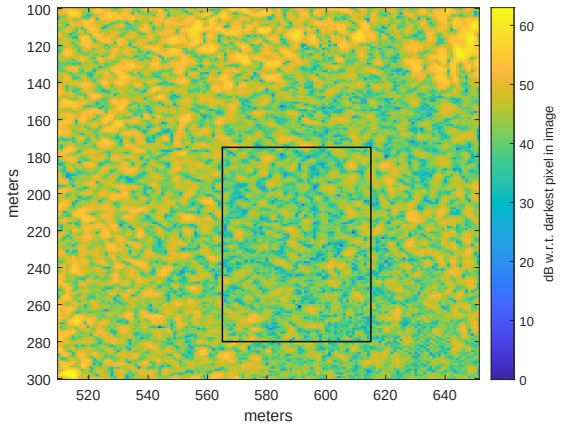
The second conclusion is that the error depends a lot on the amount of bits shifted. In this particular dataset, all data fits within 11 bits, even though the samples are 16-bit signed integers. If one of the sweeps has samples that are close to 2^{15}, the samples should not be shifted, but if all the others fit in 11 bits, the error will be around 70 % if no shifting is performed, which is obviously unacceptable. It means that it is indeed necessary to determine the largest sample in a sweep, before processing it, as explained in Section 3.3.2.

Table 5.1: Relative errors for FFT simulations with different data and twiddle factor widths. Input data are 25 sweeps, randomly selected from a set of 24414 sweeps of real SAR data during a single flight path. Error is with respect to MATLAB's builtin `fft` function. The input vectors are scaled up as described in Section 3.3.2, with the number of bits shown in the first column. The blue cells indicate the bitshift that is calculated by Eq. (3.2).

input	FFT width: **data/twiddle**				
shift	**16/16**	**18/18**	**20/20**	**20/16**	**20/12**
0	72 %	68 %	67 %	68 %	68 %
1	35 %	32 %	32 %	32 %	33 %
2	21 %	18 %	17 %	17 %	17 %
3	12 %	8.2 %	7.4 %	7.7 %	7.7 %
4	7.6 %	5.1 %	4.2 %	4.0 %	5.1 %
5	>100 %	2.6 %	2.0 %	1.9 %	3.5 %
6	>100 %	2.0 %	1.1 %	1.2 %	3.4 %
7	>100 %	>100 %	0.84 %	0.76 %	3.1 %
8	>100 %	>100 %	0.62 %	0.59 %	3.2 %
9	>100 %	>100 %	>100 %	>100 %	>100 %



(a) The entire image, with the full brightness range.



(b) The same image as Fig. 5.1a, but zoomed in (notice the different color scale).

Figure 5.1: A single backprojected image, with the darkest area highlighted.

From this table alone, it is hard to draw a conclusion about which configuration is best: that mostly depends on the extra errors introduced in the backprojected image. At first glance it looks like only 20/20 is accurate enough, since errors above 1 % seem large. However, we will see that the image suffers surprisingly little when the data from the 16/16 configuration is used as an input for the backprojection algorithm. As mentioned, the data width can never be lower than 16 bits. The twiddle factors could be reduced in precision, however.

5.1.2. Backprojected image

In Section 3.3.3, a hypothesis was made, that introducing larger quantization errors in the preprocessing step – range compression –, will manifest itself as an increased noise floor in the darkest part of the image. An image, made with the available dataset, is shown in Fig. 5.1. The darkest part of the image was manually found and highlighted with a black rectangle.

In Table 5.2, the average energy increase for the darkest part of the image is shown for multiple FFT widths. It can be seen that the energy increase is completely insignificant: 0.028 dB is a very low negligible number. However, it is likely that the difference is much more visible in a darker image, because the energy in the dark part of this image is still quite high and possibly higher than the noisefloor. Additionally, there is never a negative difference – meaning that the energy of the dark area of the quantized image is lower than of the full precision image. Therefore, the hypothesis can be confirmed: quantization errors in the FFT do increase the noise floor, however small. The pixel with the largest error may not be a relevant metric, but it is surprising to

Table 5.2: Increase of the energy of the dark area of the backprojected image, as calculated with data from FFT simulations with different data widths. There are a few single pixels that have a very large error (see also Fig. 5.2).

FFT config.	dark area energy increase	max error in image
16/16	0.0281 dB	24.7 dB
18/18	0.0025 dB	9.88 dB
20/20	0.0013 dB	10.2 dB

see that there are single pixels with a very large error. This effect is further investigated in Appendix A.

To compare the images in another way, histograms were drawn showing the frequency of a specific error in dB. These histograms are shown in Fig. 5.2. An image that perfectly corresponds with the original image would have a histogram with a single peak at 0 dB. The 18-bit image has only a few tens of pixels with errors larger than 1 dB, and the 20-bit image is doing even better. And again, also the 16-bit image is performing well, with the probability of a pixel having an error larger than 1 dB being smaller than 0.0067. What the exact requirements are will depend on the wire detection algorithm, and how sensitive it is to single pixels having a large error. Researching this will be part of recommended future work. Errors in single pixels are unlikely to have a large impact on the wire detection, however, because the wires are very long compared to only one pixel.

5.2. Area-latency tradeoff

This section will provide answers to research question 4, by exploring which FFT configurations are feasible for the different SAR and GMTI applications (Section 5.2.1). Additionally, some small observations and possible optimizations are shown in Sections 5.2.2 and 5.2.3. Finally, area-latency graphs will be used to draw some conclusions specifically about the ONR drone radar (Section 5.2.4).

5.2.1. Optimal configurations for SAR and GMTI modes

A Python script was written, that selects the optimal FFT configuration for the respective SAR or GMTI application (Section 3.6.2). Table 5.3 shows the results. The optimization from Section 3.4.3 has been performed on the ONR radar results, but not on the others, because the ranges are not available (Section 2.4.2).

5.2.2. Twiddle factors

In area usage, using 16 bits for the twiddle factors in comparison with 20 saves 13 RAM blocks; that is 4 % of the total of 305 RAM blocks in use. Additionally, some logic is saved, but since RAM is the resource in highest demand, we can disregard this. So if the FPGA is almost full, and saving a few RAM blocks is important, the twiddle factor width can safely be reduced.

In Section 3.2.1 it was said that the twiddle factors are constants and could therefore be stored in ROM. From these results we can see that this is not happening. The twiddle factors are stored in RAM, just like the data.

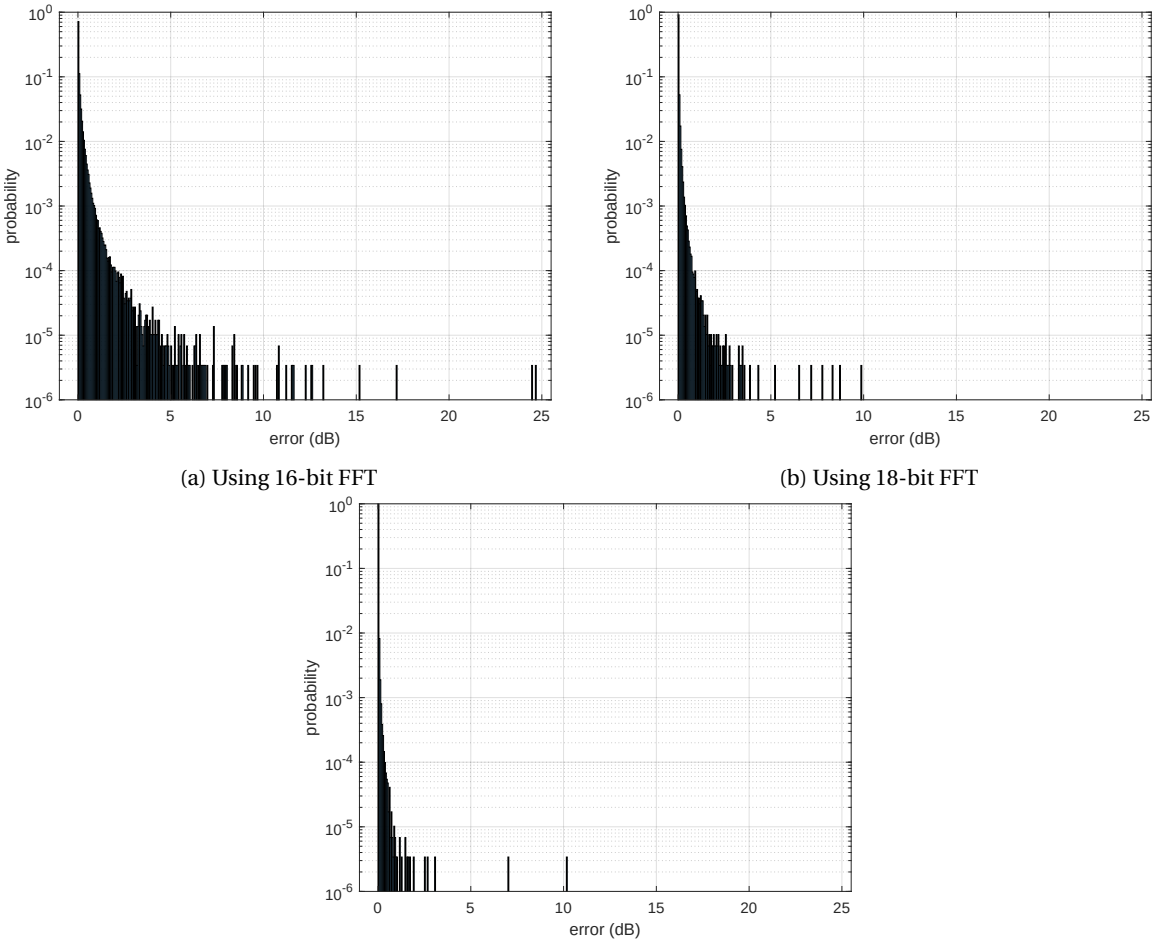
5.2.3. Multipliers and DSP blocks

As mentioned in Section 3.1.3, the multipliers in the Cyclone V DSP blocks can perform one 27×27 multiplication or two 18×19 multiplications. Therefore, for bitwidths larger than 18, more DSP blocks are used.

For floating point (Variable Streaming), DSP resource optimization can be turned on. This will use less DSPs and reduce f_{max}. As can be seen in Table 5.3, this is only an option for one application that was considered. This option was not turned on, because it may make the latency estimation invalid if the f_{max} drops below 150 MHz. Additionally, saving DSPs is not a high priority at this point, because for every configuration, enough DSPs were available.

5.2.4. Pareto plot

In Section 5.2.4, two area-latency plots are shown. For the ONR system – with FFT length 16384 – all configurations were tested. The number of RAM blocks is shown in a different graph than the mean of the ALMs and DSP blocks, because observations showed that the number of ALMs and DSP blocks are mostly proportional to each other, whereas the RAM blocks change independently. The following observations can be made:



(a) Using 16-bit FFT

(b) Using 18-bit FFT

(c) Using 20-bit FFT

Figure 5.2: Histogram of the pixel errors in decibel for the entire backprojected image using different size FFTs.

Table 5.3: SAR/GMTI applications and the corresponding optimal FFT implementations. Intel FPGA configurations from [11]. For the ONR system (length 16384), the optimization from Section 3.4.3 has been performed for Burst mode, by subtracting the latency of the FFT core with 16384 − 3698 = 12686.

Length	PRF	# ch	# cores [1]	avail. cycles [2]	config	# eng [3]	# mult [4]	latency [5]	RAM	DSP	ALMs
Cyclone 5CSXC2									**140**	**36**	**25k**
512	3333	8	1	5626	Burst	2	1	2430	10	4	998
512	3333	12	1	3750	Burst	2	1	2430	10	4	998
512	3333	16	1	2813	Burst	2	1	2430	10	4	998
512	3333	20	1	2250	Burst	1	4	1626	10	6	1311
512	3333	24	1	1875	Burst	1	4	1626	10	6	1311
512	3333	28	1	1607	Burst	2	4	1370	17	12	2150
512	3333	32	1	1406	Burst	2	4	1370	17	12	2150
1024	2000	8	1	9375	Burst	1	1	8315	6	2	613
1024	2000	12	1	6250	Burst	1	4	3162	12	6	1429
1024	2000	16	1	4688	Burst	1	4	3162	12	6	1429
1024	2000	20	1	3750	Burst	1	4	3162	12	6	1429
1024	2000	24	1	3125	Burst	2	4	2650	17	12	2295
1024	2000	28	1	2679	Burst	2	4	2650	17	12	2295
1024	2000	32	1	2344	Buffered Burst	1	4	1291	20	6	1421
Cyclone 5CSXC4									**270**	**84**	**40k**
16384	814	8	1	23034	Buffered Burst	2	4	18432	212	12	2355
32768	500	1	1	300000	Burst	2	1	196775	225	4	1130
16384	814	12	2	30712	Burst	4	4	26354	216	48	8632
Cyclone 5CSXC6	In ONR radar system								**557**	**112**	**110k**
16384	814	16	2	23034	Buffered Burst	2	4	18432	424	24	4710
Cyclone 5CGXC9									**1220**	**342**	**301k**
65536	400	4	1	93750	Buffered Burst	2	4	73728	963	12	2434
e.g. **Agilex**	Does not fit on Cyclone V										
131072	400	4	2	187500	Variable Streaming, fixed point	1	4	131072	1556	86	14380

[1] Number of instantiated FFT cores in parallel.
[2] Time budget per sweep in clock cycles.
[3] Number of output stages (engines).
[4] Number of multipliers per output stage.
[5] Number of clock cycles needed to process one sweep.

- RAM is a scarcer resource than DSPs/ALMs.
- Faster configurations do not always use more area, and vice versa.
- Sometimes there is a tradeoff between using more DSPs/ALMs or more RAM. As an example, stepping up from the B,1,4 configuration to B,2,4 decreases the latency and the RAM usage. However, it costs more DSPs and ALMs.

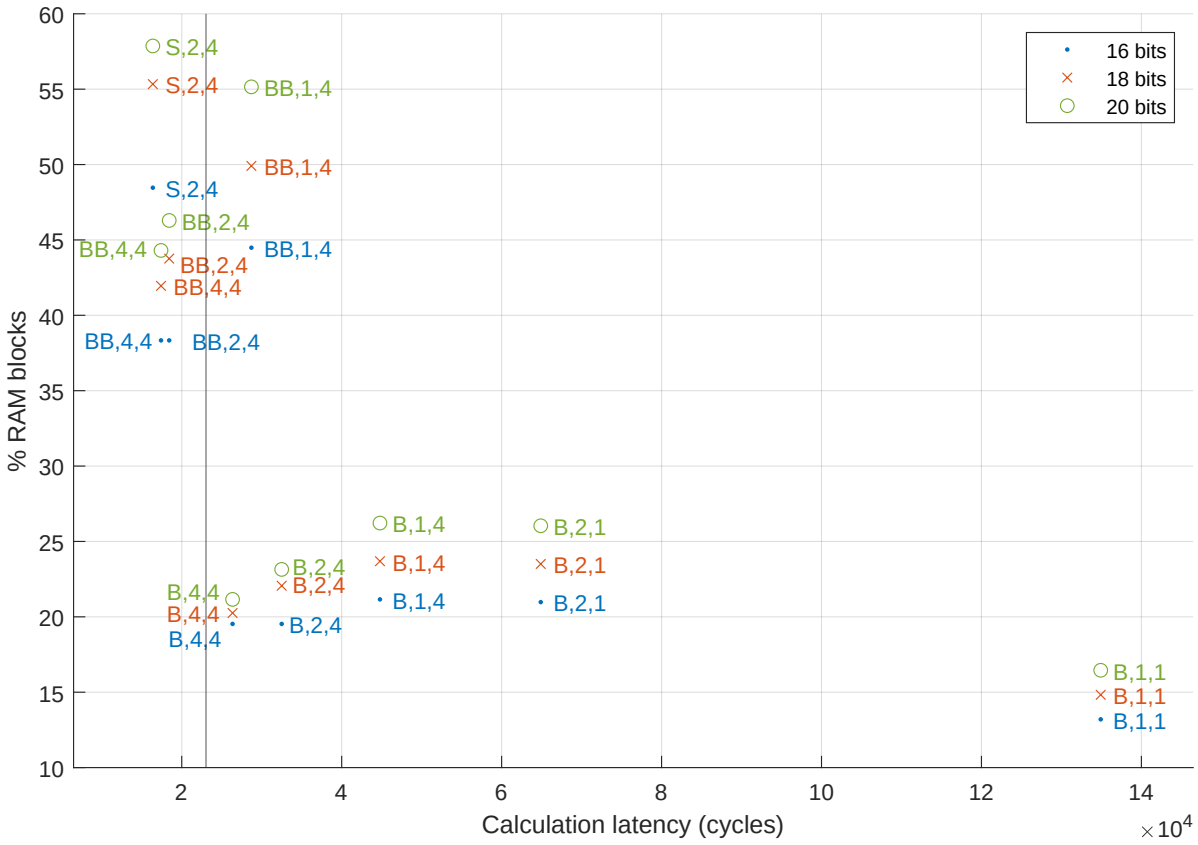
5.3. Conclusions

This chapter showed the results of the investigations into the range compression algorithm, following the methods that were described in Chapter 3. In that chapter, a prediction was made that the FFT would require approximately 16-17 bits of output range. In this chapter, that prediction was confirmed, by verifying that the noise floor of the backprojected image did not increase significantly. In order to optimally use the dynamic range, however, it is important to scale up every sweep individually before feeding it into the FFT. Combined, this provides answers to research question 3.

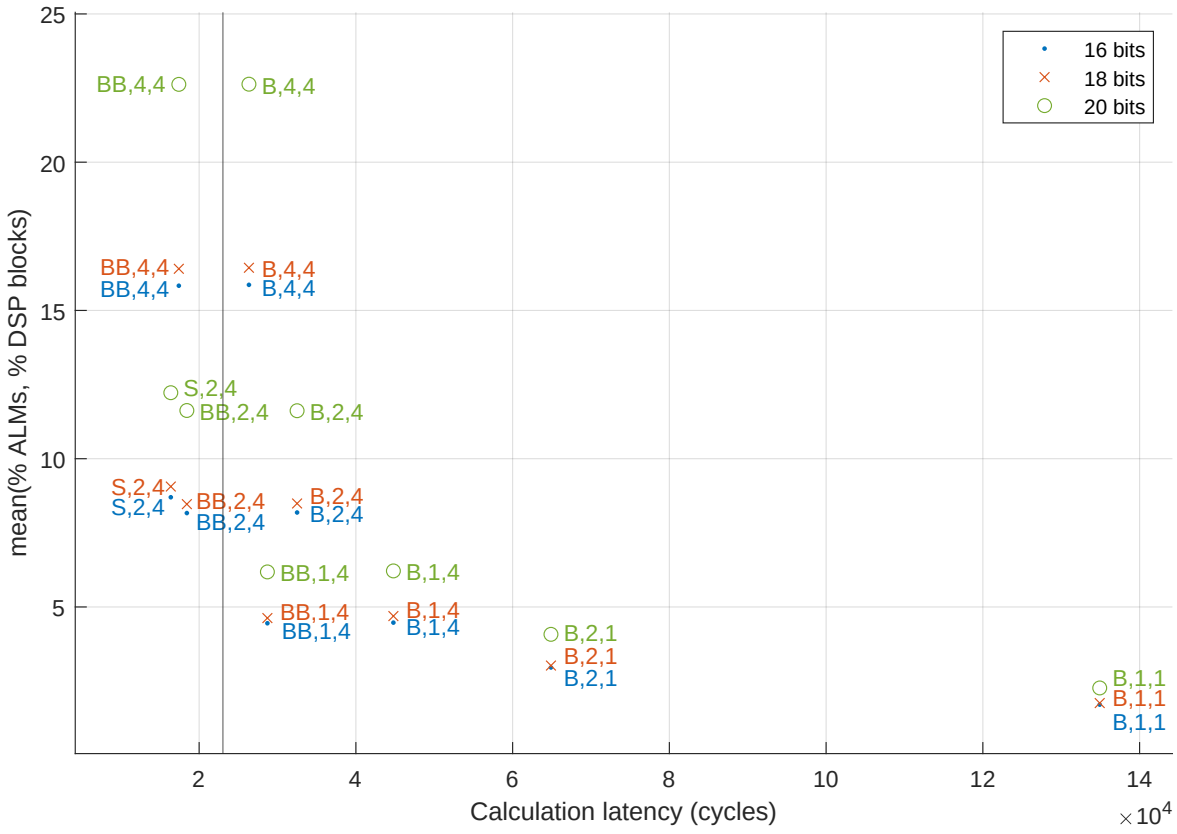
First of all, in this chapter, the hypotheses from Section 3.7 were confirmed. The required FFT bitwidth was indeed 16 bits, and the noise floor of the backprojected image did not increase significantly (Section 5.1.2). To answer research question 4, refer to Section 5.2 and in particular Table 5.3.

Some other interesting observations were made:

- If all RAM of the FPGA is utilized, a few resources can be freed up by decreasing the twiddle factor. This introduces a constant error, which is negligible if it stays below the other errors in the system (Table 5.1).
- Even if the error from the FFT is larger than 7 %, accurate images are still generated.



(a) Area is shown as percentage of RAM blocks in use.



(b) Area is shown as the mean of the percentages of ALMs and DSP blocks in use.

Figure 5.3: Area-latency graph showing the Pareto points of a 16384-point FFT. The area usage is shown as a percentage of the total available resources, so it is easier to see the usage in context of the ONR system. For absolute resources and generalisation for other systems, refer to Table 5.3. The grey line shows the time budget of the ONR system, which means that all configurations on the right of this line are too slow.

6

Backprojection results

This chapter is to Chapter 4 (backprojection in hardware), what Chapter 5 (range compression results) is to Chapter 3 (range compression in hardware). The results of the implementation of backprojection will be analyzed, and conclusions will be drawn about the feasibility and hardware requirements for real-time processing of the different radar systems that are tested (as introduced in Table 4.2). And similar to Chapter 5, we will analyze the numerical accuracy (Section 6.1) and area and latency (Section 6.2). Additionally, the power requirements are shown (Section 6.3) and there is a brief section on the attempts at running the implementation on real hardware (Section 6.4).

6.1. Accuracy and errors

Section 4.5.2 introduced the concept of determining the minimum integer bit width using the MATLAB fixed-point toolbox. However, there is no simple way to determine the optimal fractional bit width. Because the output image has a dynamic range of 24 bits (Section 4.5.1), as a starting point, all intermediate variables were also given 24 bits in total. This can be seen in the HLS code of the image generation core in Listing C.2 (Appendix C). Because the goal of this research is not to fully optimize a single application, but to draw general conclusions about the requirements for various radar systems, the precision was kept at these 24 bits.

We will now verify the correctness of the HLS implementation, and the errors introduced by the fixed-point calculations. An error image was produced as described in Section 4.5.4. The images from the HLS core are compared to the original, full-precision MATLAB-produced image. Both were normalized and then subtracted from each other, after which the amplitude is taken and it is converted to decibels. Normalization is necessary because the images may have a constant factor offset, which vanishes if the power is shown in decibels.

The error image is shown in Fig. 6.1. Errors under 1 dB can be considered small, and it can be seen that the entire image satisfies this constraint. No implementation errors in the form of unexpected anomalies can be seen. We only observe noise and a few parabolas. What those parabolas originate from, is explained in Appendix A (this explains the phenomenon from Fig. A.1). In short, errors in the FFT bins (input to the backprojection core) can cause these artefacts.

6.2. Area and latency

Just like in Section 5.2, in this section, we provide an answer to research question 4, by analyzing the area usage and latency results of the backprojection implementation as parametrized for the systems under investigation. Firstly, an overview is given in a table, showing the performance and area requirements of the implementations. Secondly, we compare the performance of this implementation to the original MATLAB implementation, and refer to previous work to put the performance in context. Consequently, it is verified whether or not the elements that were intended to be parallelized, are indeed executed in parallel. And lastly, a comparison is made between the computational load of the system (in operations per second) and the resulting area usage on the FPGA.

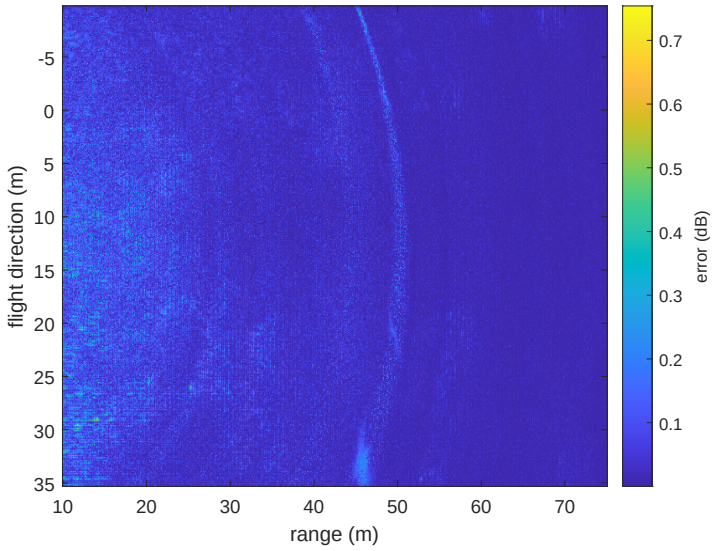


Figure 6.1: Difference between images produced by HLS code and full-precision MATLAB.

Table 6.1: SAR applications from Table 4.2 and the corresponding backprojection configurations. Xilinx Zynq-7000 FPGAs from [38].

System	req. px/s	meas. px/s	Int. time	latency	parallel [1]	BRAM (18kb)	DSP	Flipflops	LUT
XC7Z014S (Artix-7)						**214**	**170**	**81 200**	**40 600**
M-RaISR Strip	150 000	249 896	4 s	2.40 s	2	156	66	9 864	9 073
XC7Z030 (Kintex-7)						**530**	**400**	**157200**	**78600**
M-RaISR Spot	166 667	208 246	9 s	7.20 s	3	360	386	56 841	50 265
ONR 8 ch [2]	375 000	398 322	1 s	0.941 s	2	354	386	105 085	80 682
XC7Z035 (Kintex-7)						**1000**	**900**	**343 800**	**171 900**
ONR 12 ch [2]	375 000	265 957	1 s	1.14 s [3]	2	530	578	156 681	120 214
M-RaISR Spot short range	400 000	416 618	9 s	8.64 s	6	720	770	112 793	99 896
ONR 16 ch [2]	375 000	199 468	1 s	1.88 s [3]	2	706	770	586 454 [4]	158 917

[1] Number of image generation cores.
[2] 10 aspect angles calculated in series.
[3] Pipeline initiation interval is equal to $N_{channels} \times N_{FFT}$. So the bottleneck is streaming in the range compressed samples, not calculating the actual image. This is easily solved by adding parallel I/O ports.
[4] Not proportional with ONR 8 ch and ONR 12 ch. Expected approx. 200 000 flipflops. The synthesis report shows that a lot of extra 32-bit registers were inferred. Investigating this will be future work when this mode will actually be implemented in the real world.

Table 6.2: Comparison of the backprojection implementation this thesis, with another implementation of the same algorithm by other authors [3].

	This thesis	Paper [1]	Factor [6]
Software runtime [2]	6.28×10^{-7}	1.27×10^{-7}	4.95
Hardware runtime [2] [3]	7.42×10^{-10}	1.29×10^{-9}	0.57
Speedup w.r.t. thesis [4]	846	486	1.74
Speedup w.r.t. paper [5]	171	98	1.74

[1] The data from Table III from [3] was normalized and averaged.
[2] In seconds per pixel per sweep.
[3] Averaged values from Table 6.1.
[4] MATLAB runtime of this thesis divided by respective FPGA runtime.
[5] MATLAB runtime of paper divided by respective FPGA runtime.
[6] The value of this thesis divided by the value of the paper.

6.2.1. Overview

The overview of the performance and area usage is given in Table 6.1. The latency, that is shown, is the time taken to produce a single image with all data that is gathered during the integration time. For more information on this, refer to Sections 2.2 and 2.5.2. The rows printed bold show the FPGA model number that fits the application (data from [38]). The column named "parallel" shows how many image generation cores are instantiated (refer to Section 4.4.2).

6.2.2. Performance w.r.t. previous work

In Section 4.6, a method was introduced to be able to compare our results to the results from other backprojection implementations. Table III of [3] was taken as an example, because this paper contains an implementation of the identical backprojection algorithm on an FPGA. Table 6.2 shows the results.

From this table, it can be seen clearly that the implementation from this thesis performs better than the one from [3]. In absolute terms, the execution time per pixel per sweep is 0.57 times that of the paper, and the speedup with respect to the MATLAB code is 1.74 times better. We will now look at some factors that may have influenced the results.

First of all, is from 2016, so the speedup may be partly due to improved tooling. However, we have used HLS, which is generally less efficient than writing the RTL by hand. The hardware that was used can also make a difference. In the paper, a Virtex-6 was used, which is of an older generation than the generation 7 FPGAs that were used in this thesis. However, all we needed for the largest-area system, was a Kintex-7, which is a lower-end FPGA than the Virtex. All in all it is hard to determine who had the advantage here.

To this argument, it can also be added that this implementation was not even designed to have the best possible performance for a single system, but rather to get performance data on a range of current systems, and make it easy to adapt for future systems. The fact that it competes well, or even performs better, than the compared previous work, could be called a great success.

6.2.3. Verification of paralellization

We can determine if the latency of the implementation is as intended, meaning that all with the knowledge that all sweeps and pixels are processed sequentially. Equation (6.1) shows how the theoretical expected latency can be calculated:

$$\text{latency} = \frac{\text{pixels} \times \text{sweeps} \times \text{aspect angles}}{\text{parallel} \times \text{clock freq.}} \tag{6.1}$$

If we substitute the numbers from Tables 4.2 and 6.1, we see that for most applications, the measured latency matches the ideal latency perfectly. This means that the per clock cycle, one pixel in one sweep is calculated, as intended. The results are shown in Table 6.3. It can be seen that it does not hold for the ONR system. The reason for this, is that the bottleneck is not the calculation, but the streaming of the range compressed data. This was concluded because the pipeline initiation interval of the image generation core is equal to $N_{channels} \times N_{FFT}$, meaning that it performs as intended. For ONR 12 and 16 channels, extra input ports can be added to stream in the data in parallel and ensuring real-time operation of the core.

Table 6.3: Expected (ideal) theoretical latency versus measured latency during calculation of a single image (for ONR: 10 aspect angles).

System	expected latency	measured latency
M-RaISR Strip	2.40 s	2.40 s
M-RaISR Spot	7.20 s	7.20 s
ONR 8 ch	0.611 s	0.941 s
ONR 12 ch	0.611 s	1.14 s
M-RaISR Spot short range	8.64 s	8.64 s
ONR 16 ch	0.611 s	1.88 s

6.2.4. Comparison

Section 4.2.2 introduced the design requirements that the implementation has to adhere to. Besides the number of pixels per second that have to be calculated, an estimation was made of the total number of operations per second. In this section, we will verify that the hardware resource usage on the FPGA is proportional to this number. If that is true, an estimation can be made for future systems, how many resources they will require, by extrapolating this data. In Fig. 6.2, the results from Table 6.1 are shown in this manner. It is clear that there is a correlation between the computational load and the area usage, which is linear. However, the system with the most computational load (*M-RaISR Spot short range*) does not follow the trend – it uses less area than expected.

6.3. Power

After synthesis, Vivado reports an estimation of the power usage on the FPGA. For the backprojection core, this is 2.7 W. A medium-sized quadcopter drone uses around 150 W for its motors, in order to fly. Therefore, the FPGA power is negligible. ONR even uses a octocopter.

To make this point stronger, the Xilinx Power Estimator (XPE) [35] was used to estimate the theoretical maximum power that a Kintex-7 XC7K325T can draw. This FPGA is similar to the one inside a Zynq-7000 XC7Z035 SoC. In the XPE, all logic in the entire FPGA was set to switch every clock cycle, at 250 MHz. At this extreme scenario, only 20 W is estimated to be used, as can be seen in Fig. 6.3.

6.4. Testing on real hardware

An effort was made to run the HLS core on real hardware: a ZC702 evaluation board. In the end, this was not successful because more time was spent on getting the results through simulations, and focusing on the research questions.

To get a HLS design to run on a Zynq device, tutorials from Xilinx's UG871 [39] can be followed. We will now discuss a few issues that were encountered.

In the Vivado project, we need the Zynq processing system block to connect to the HLS IP block using AXI ports. Three types of ports are available: general-purpose (GP), high-performance (HP), and accelerator coherency port (ACP). The GP port is designed for switching functions off and on, and similar low-bandwidth purposes. The HP and ACP ports both have high performance, but the ACP port has the advantage of keeping the CPU cache coherent. However, if high performance is desired for multiple competing tasks, the HP port performs better [23]. Cache coherency then has to be ensured manually.

After finishing the Vivado project, and generating the bitstream, it has to be imported into the Vitis IDE. Here, C code can be written, that runs on the CPU of the Zynq SoC. This code performs the tasks of starting and stopping the hardware core, and moving data between the DDR memory and the FPGA using direct memory access (DMA). This part was unsuccessful, and eventually abandoned in favour of running simulations.

6.5. Conclusions

In this chapter, it was shown that the fixed-point HLS code executes correctly as intended. The produced image has very small errors compared to the original full-precision image, and the code is working as intended, also when the parameters are changed. One exception is the ONR system for 12 channels and more:

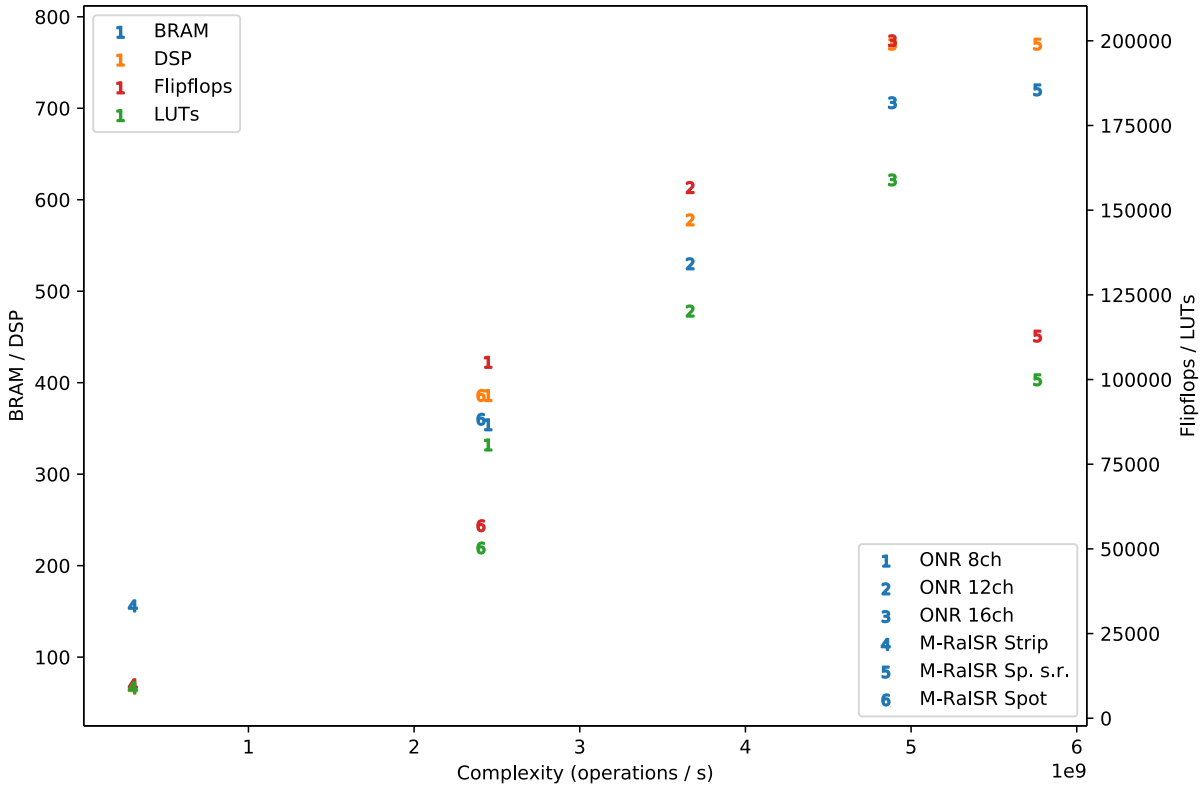


Figure 6.2: A plot where the computational load in operations per second is compared to the area usage. BRAMs and DSPs are shown on a different axis than flipflops and LUTs, because they fit in a similar order of magnitude. We expect a the area usage to be proportional to the load, and this is roughly true, except for *M-RaISR Spot short range*, which has a lower area usage than expected. For ONR 16 channels, the amount of flipflops were set to 200 000 manually (the expected amount approximately), because there was a problem during synthesis and it inferred many extra unexpected 32-bit registers.

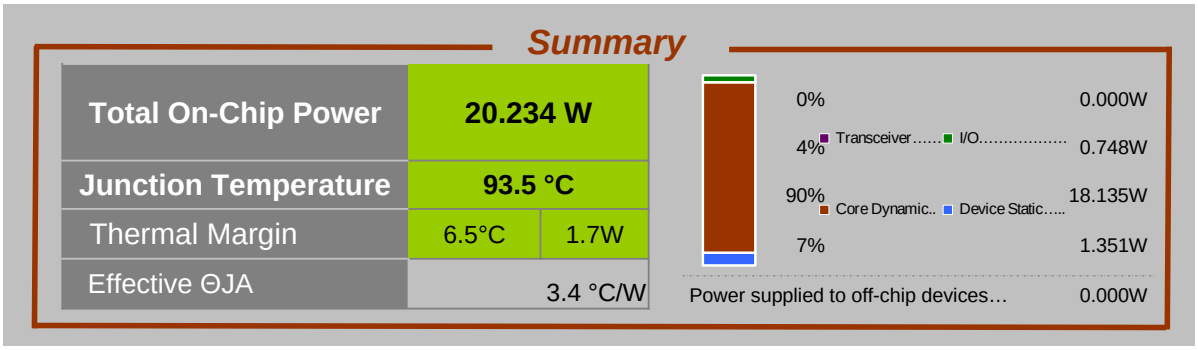


Figure 6.3: Power estimation of a Kintex-7 XC7K325T at 250 MHz using the Xilinx Power Estimator. All logic on the FPGA switches every clock cycle.

it cannot operate in real-time, because the bottleneck is not the image generation calculations, but the actual streaming in of the range compressed sweep data. This can be fixed by adding extra input ports that operate in parallel. It is likely that for future systems, similar unforseen issues occur, that have to be fixed in the code, in order to optimize for that specific system. But all in all, the code is still useful to draw conclusions whether or not the system is expected to be feasible on a certain FPGA family.

The performance of the implementation was compared with previous work, and we found that this implementation performs 1.74 times better. The speedups with respect to the original MATLAB implementations range from 171 to 846, whereas for the reference implementation this is only 98 to 486. The circumstances and hardware used were found to be similar.

In order to test the linearity of the relationship between the calculation load in operations per second, and the hardware usage, these two quantities were plotted against each other. The linearity was confirmed for most systems, except the most demanding one – it uses less area than expected. This result makes it easy to estimate resource usage of future systems by extrapolating the data by using the trend of the four systems that seem to form a linear relationship. If future systems with higher computational load use less area than this trend estimates, this can still be used as a conservative (safe) estimate.

Another point to remember from this chapter is that the power usage of the FPGA is negligible to the quadcopter motors. The power usage of the backprojection core for ONR 8 channels was estimated to be 2.7 W, which is about an order of magnitude smaller than the motor power. Even when all FPGA logic is set to switch every clock cycle, causing it to use the maximum power possible, it only uses 20 W, which is more significant, but still not a high power compared to the motors.

7

Conclusions

7.1. Summary

In this research work, the two most common Synthetic Aperture Radar (SAR) signal processing algorithms were implemented and tested for a variety of SAR systems, with the goal of aiding in future designs of processing systems for airborne SAR. SAR is a radar technique superior to conventional stationary radar in terms of resolution. It works by physically moving the radar antennas with respect to the object that is to be imaged, and building an image using accumulated data from a longer time period. The larger the distance flown, the higher the azimuth resolution, and the higher the radar bandwidth, the higher the range resolution.

The implementations were done for FPGAs specifically, because the algorithms are easily parallellizable, and FPGAs are significantly more power efficient than GPUs and Cell processors, which is important in airborne applications. Initially, the algorithms were implemented for an airborne SAR system that was already developed for the Office of Naval Research (ONR). Changing the identified system parameters makes the implementations suitable for a wide range of other applications.

Since the wireless downlink was determined to be too slow to enable real-time processing on the ground, two options were left: onboard data reduction – therefore decoding and processing the data on the ground –, and onboard real-time processing. An algorithm (block-adaptive quantization) was shown to be compressing the data insufficiently, and for both range compression and backprojection, it was shown that for all tested systems, the implementations fit on midrange FPGAs.

Since range compression is implemented as simply an FFT, and designing and implementing a custom FFT core is outside the scope of this research, pre-existing IP from Intel was used. For backprojection, an implementation was made using Vivado HLS.

The numerical precision requirements are verified by generating a backprojected image, and verifying whether the noise floor of the image increased. Quantization noise should stay below the original noise floor in order to not add extra noise. The predictions, that were made, were found to be correct; namely, that the FFT would require approximately 16-17 bits of output range and the backprojected image about 24 bits. It is however important to scale every sweep individually, in order to make optimal use of the available dynamic range available. The bitwidths of the variables in the HLS code can be determined by the radar engineer, by simulating the algorithm with fixed-point arithmetic in MATLAB.

The performance of the backprojection implementation was compared with previous work, and this implementation was found to perform 1.74 times better. The speedups with respect to the original MATLAB implementations range from 171 to 846, whereas for the reference implementation this is only 98 to 486. The circumstances and hardware used were relatively similar.

The produced image has very small errors compared to the original full-precision image. However, it is likely that for future systems, unforseen issues will occur. For that specific system, the code will have to be hand-optimized. But the original goals of the implementations were met: the code is still useful to draw conclusions about the feasibility of the radar processing chain for the desired FPGA family.

The final point, that was made, is that the power usage of the FPGA is at one or two orders of magnitude below the quadcopter motors. This makes power usage of low concern.

7.2. Main contributions

In the introduction of this thesis (Chapter 1), the goals, problems and research questions were introduced. In short, the problem is that TNO has to redesign the digital frontend of new SAR systems, every time when new developments are made on the analog side. It is generally hard to predict the computational requirements of such a new system. Therefore, the goal of this research was to produce knowledge about how to achieve real-time processing for a variety of different SAR systems. Focus is put on two very common algorithms used in SAR processing: range compression and backprojection.

The research questions will now be revisited, and answered, starting with the main research question:

- *How can real-time performance be achieved for specific algorithms found in many synthetic aperture radar systems, within the limitations of airborne applications?*

  For airborne applications, FPGAs are the platform of choice. Real-time performance can be achieved by redesigning all the hardware from scratch, every time a new analog frontend is developed, but this thesis provided reusable parametrized implementations for two of the most common signal processing algorithms used in SAR.

1. *What are the relevant system parameters that affect the computational requirements for the radar processing algorithms that are regularly used in different types of systems?*

   The system parameters were introduced in Section 2.2. For range compression, the tested parameter values were listed in Table 3.1. For backprojection, the same was done in Table 4.2.

2. *Of those algorithms, how can they be implemented for reuse in different systems? What parameters should be varied?*

   For range compression, there exist prewritten solutions, of which one (the Intel FFT IP core) was tested in detail. Its latency-area tradeoff can be configured using its parameters (Section 3.2), and a parameter sweep was done (see question 4).

   Backprojection was implemented from scratch in Vivado HLS, which was determined to be the best platform for this goal. Development and testing is easy, and optimal results are not strictly required in this stage. In principle, the code should work or be easily adaptable for any future SAR application.

3. *What is the impact of implementation in hardware on numerical precision?*

   It was shown that it is possible to make the algorithms run in real-time on midrange FPGAs, without introducing an error in the final image, that is higher than the already-present noise floor.

4. *What are the hardware requirements (e.g. type of FPGA), given the system parameters and required processing?*

   For range compression and backprojection, tables were produced that clearly show the required FPGA for all the tested systems (Tables 5.3 and 6.1, respectively). Additionally, for backprojection, the linearity of hardware usage with respect to the computational complexity was verified, making it possible to predict hardware usage for future systems by extrapolation.

5. *What development tools are needed?*

   MATLAB fixed-point toolbox

   For easily determining the bitwidths of the fixed-point variables as described in Section 4.5.2.

   Various custom scripts

   For finding the optimal FFT configuration and verifying the numerical precision of the image.

   Intel Quartus

   For development and synthesis of the range compression code.

   Vivado (HLS)

   For development and synthesis of the backprojection code.

   MATLAB MEX compiler

   For compiling the HLS code for testing in MATLAB.

   Modelsim / Questasim

   For simulating the FFT core.

6. *What power (order of magnitude) do specific algorithms use on a chosen platform?*
   Backprojection for ONR was estimated to use 2.7 W. The exact amount was found to be irrelevant, compared to the power usage of the quadcopter motors.
7. *What is the reusability of the implementations for offline processing if real-time is not a requirement?*
   The HLS code can be run offline on a CPU as well as synthesized for running on an FPGA. However, this is mostly useful for testing and verification purposes, and not for production settings. The implementation is not optimized for running on CPUs at all, and therefore runs much slower than possible. So we can conclude that this question was not really answered; however, for testing and verification purposes, this is still useful to TNO, since the design process can be optimized in this way, and the communication between radar engineer and hardware engineer improved.

7.3. Future work

Two main types of future work will be recommended. One possibility is continuing on the set path by adding new algorithms, and unifying them into a more complete package. Before that, it can be considered to first improve on the current shortcomings. Another possibility is putting focus on a different type of system: space-based radar instead of airborne applications.

Firstly, let's list the shortcomings of the current work.

- In the backprojected image, single pixels can have very large errors (Appendix A). However, it was also estimated that it is not a big issue for wire detection (Section 5.1.2)
- The image pixels are currently not accumulated. Instead, this task is deferred to the CPU to do it offline (Section 4.4.2).
- For other systems than ONR, the algorithms were only parametrized and synthesized. No verification of the functionality and numerical precision were done, because there was only test data available for ONR. Therefore, doing these tests can give new insights in how to make a better generalized implementation.
- Table 6.1 showed an anomaly in the number of required flipflops for ONR 16 channels, due to the unexpected inference of a large amount of 32-bit registers. This effect can be investigated.
- Table 6.1 also showed that the input bandwith of the backprojection core is too slow for certain systems. The code will need to be improved, with the possibility of adding parallel input ports.
- Figure 6.2 showed that most systems have a linear relationship between the computational load and the area usage, except *M-RaISR Spot short range*. This can be investigated by adding more systems, and checking the linearity, or studying why these particular parameters cause lower area usage than expected.

7.3.1. Unification and expansion

The ultimate goal of this project is a premade set of hardware blocks for most of the regularly-used radar signal processing algorithms, that can be used on FPGAs as well as on CPUs or in MATLAB. Additionally, further optimizations can be made for specific applications, yielding a modular and reusable, yet efficient and optimized package.

As a first step to achieve this, range compression can be joined with backprojection, connecting them together and running them both simultaneously. In this work, they were seen as separate projects – also because the ONR system uses Intel FPGAs, and the actual goal in TNO is to implement everything for Xilinx FPGAs.

Suggestions for other algorithms to add to the package are:

- Beamforming
- (G)MTI: (Ground) moving target indication
- Wire detection

The last suggestion, is investigating CλaSH (or other HDLs) as a possible language to develop further algorithms in, as introduced in Section 4.1. This enables complete platform independence, making it possible to run the code on any FPGA, or even CPUs with reasonable efficiency.

7.3.2. Space-based radar

Apart from expanding forward, from the current path, a side path can be taken, and developing a similar solution as presented in this work for space-based radar. This type of radar has different power and weight

requirements than airborne radar. Whereas we determined that FPGA power usage is not an issue for airborne applications (Section 6.3), in space, this could be a large issue. The same holds for weight, which is of somewhat smaller importance in airborne radar.

Space-based radar has the focus on the following algorithms, that will need to be especially optimized:

- FFT (range and doppler)
- 2/3-element beamforming

A

Large pixel errors in backprojected image

In Fig. A.1, the same image as in Fig. 5.1 is shown, but here the pixels with the largest errors are highlighted. It can be seen, that the pixels with the largest relative error are mostly in the darkest area. This is not unexpected, because small absolute errors have a larger impact there.

Most of largest absolute errors are in the brightest area, where larger deviations are expected because the energy is high (note the dB scale of the image). However, something interesting can be seen: a part of the black crosses are on a circle in the middle of the image. Since the drone with the radar is flying to the left of the image, that means that those crosses are at an equal distance to the drone. Since every FFT bin corresponds to a distance, if one of them has a large error, this could result in similar error patterns. This hypothesis was tested, by manually setting one of the FFT bins to 0, and generating a backprojected image. When subtracting this image from the original, it can be seen that indeed, a similar arc is generated (Fig. A.2).

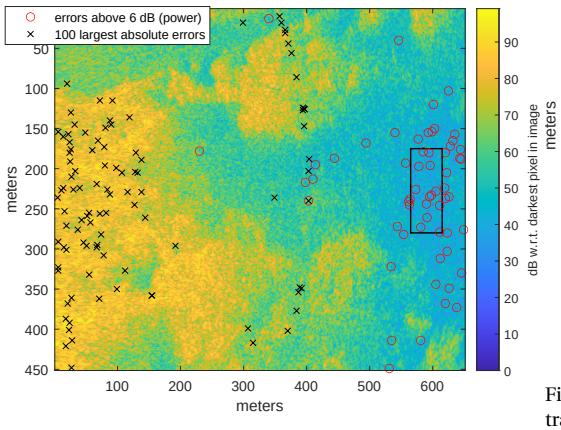


Figure A.1: A single backprojected image. The largest errors are marked and darkest area is highlighted as a black square.

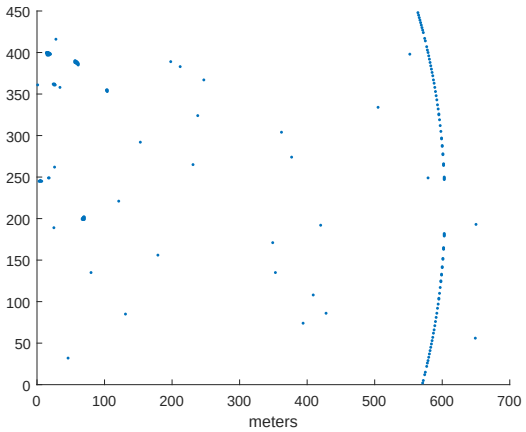


Figure A.2: The image made full-precision FFT was subtracted from a version where 1 FFT bin was set to 0. In total the image is made up from 768 sweeps with length 16384. The dots in this figure are pixels where the absolute value of the difference is 2. It can be seen that this single point of error, that is introduced, gives a similar arc in the image as in Fig. A.1.

B

Backprojection development setup

In Chapter 4, the design and implementation of backprojection in HLS was discussed. This appendix will go into practical detail of how the framework was built, and how the results from Chapter 6 can be reproduced.

Initially, the radar signal processing algorithms are designed and implemented in MATLAB by the radar engineers. By the hardware engineer, the resulting algorithm then has to be reimplemented on an FPGA. To accomplish this, a lot of extra communication between the radar engineer and the hardware engineer is required, and the hardware engineer has to understand some details about the workings of the algorithm itself in order to optimize the numerical precision. To aid in this process, the framework shown in Fig. B.1 was developed.

In the MATLAB domain, the radar engineer initially develops the algorithm using `for`-loops. Consequently, the code can be manually vectorized (converting the loops in to matrix operations), instrumented with fixed point arithmetic, or compiled to a MEX file for faster execution. Another way to arrive at a MEX file, is to use the wrapper around the HLS code and the `compile_hls_mex.m` script to compile the HLS C++ code to a MATLAB MEX.

The script that is instrumented with fixed-point code is used in order to aid the hardware developer with the bitwidths of the intermediate variables. For more on this, refer to Section 4.5.2.

The testbench of the HLS code reads and writes binary data files, where the raw complex data samples are stored in sequence. Two scripts are provided that convert to and from the MATLAB format, in order to input the data, and extract the final image for examination in MATLAB.

Finally, after it is optimized and functionally correct, the HLS code can be compiled to VHDL to make it ready for synthesis on real hardware.

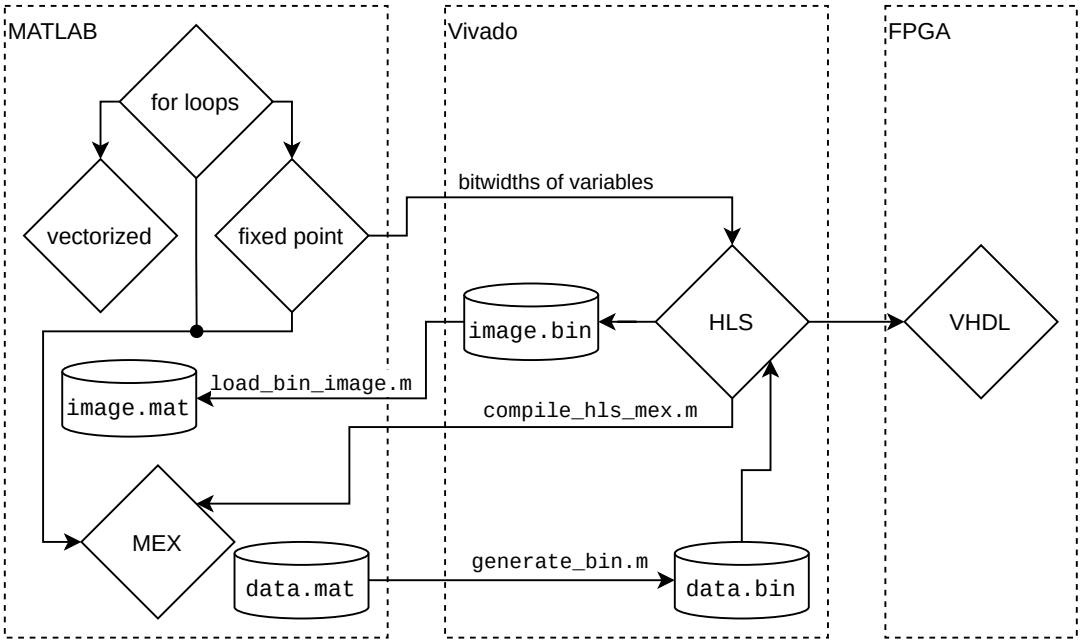


Figure B.1: Overview of the three different domains where backprojection can be executed, and the conversions between those domains.

C

Backprojection HLS code listings

The block diagram from Fig. 4.5 shows the structure of the HLS toplevel function in Listing C.1.

Additionally, in Listing C.2, the image generation function is shown. In this code, the number of integer and fraction bits per intermediate variable can be seen, as mentioned in Section 6.1.

Listing C.1: Toplevel function

```
1  void backprojection(hls::stream<RC_interface> &rc_stream,
2  #ifndef RC_HALF_FLOAT
3          hls::stream<RC_shift> &rc_shift_stream,
4  #endif
5          hls::stream<Flightpath_axis> &fp_stream,
6          hls::stream<Rotmat_axis> &rotmat_stream,
7          hls::stream<BP_grid> (&bp_gridX_stream)[PARALLEL_INSTANCES],
8          hls::stream<BP_grid> (&bp_gridY_stream)[PARALLEL_INSTANCES],
9          hls::stream<Px_interface> (&image_stream)[PARALLEL_INSTANCES])
10 {
11     loop_sweep1: for (int sweep_i = 0; sweep_i < NSWEEPS; ++sweep_i) {
12 #pragma HLS dataflow
13         RC_ram rc_ram[PARALLEL_INSTANCES];
14         Flightpath_ram fp_ram[PARALLEL_INSTANCES];
15 #pragma HLS array_partition variable=rc_ram complete dim=1
16 #pragma HLS array_partition variable=fp_ram complete dim=0
17         Xyzrel xyzrel_mul[NCHAN][3][3];
18         Complex<RC_shift> rc_shift;
19
20         stream_in(rc_ram, fp_ram, rc_stream, fp_stream
21 #ifndef RC_HALF_FLOAT
22                 , rc_shift_stream, rc_shift
23 #endif
24                 );
25         xyzrel_mult(xyzrel_mul, rotmat_stream);
26
27         Xyzrel_ram xyzrel;
28 // all elements of xyzrel have to be read every clock cycle in generate_image
29 #pragma HLS array_partition variable=xyzrel complete dim=0
30
31         xyzrel_accumulate(xyzrel_mul, xyzrel);
32
33         for (int i = 0; i < PARALLEL_INSTANCES; ++i) {
34 #pragma HLS unroll
35             generate_image(i, rc_ram[i],
36 #ifndef RC_HALF_FLOAT
37                 rc_shift,
```

```
#endif
                fp_ram[i], xyzrel, bp_gridX_stream[i],
                bp_gridY_stream[i], image_stream[i]);
        }
    }
}
```

Listing C.2: Image generation core

```
void generate_image(int instance, const RC_ram &rc_ram,
#ifndef RC_HALF_FLOAT
        Complex<RC_shift> &rc_shift,
#endif
        const Flightpath_ram &fp_ram,
        const Xyzrel_ram &xyzrel,
        hls::stream<BP_grid> &bp_gridX_stream,
        hls::stream<BP_grid> &bp_gridY_stream,
        hls::stream<Px_interface> &image_stream)
{
    loop_img: for (int img_i = instance * PIXELS_PER_INSTANCE;
            img_i < (instance+1) * PIXELS_PER_INSTANCE; ++img_i) {
#pragma HLS pipeline II=1

        BP_grid bp_gridX, bp_gridY;
        bp_gridX_stream >> bp_gridX;
        bp_gridY_stream >> bp_gridY;

        complex<float> pixel(0, 0);

        ap_ufixed<24, 7> R_tx;
        {
            ap_fixed<24, 6> a = fp_ram[0] + xyzrel[0][0] - bp_gridX;
            ap_fixed<24, 8> b = fp_ram[1] + xyzrel[0][1] - bp_gridY;
            ap_ufixed<24, 5> c = fp_ram[2] + xyzrel[0][2];
#ifdef LOOKUP_SQRT
            R_tx = lookup_sqrt(ap_ufixed<24, 13>(a * a + b * b + c * c));
#else
            R_tx = MATH::sqrtf(a * a + b * b + c * c);
#endif
        }

        loop_ch2: for (int ch = 0; ch < NCHAN; ++ch) {
#pragma HLS unroll

            ap_ufixed<24, 7> R_tr;
            {
                // range from image pixel to receiver antenna
                ap_fixed<24, 6> a = fp_ram[0] + xyzrel[ch][0] - bp_gridX;
                ap_fixed<24, 8> b = fp_ram[1] + xyzrel[ch][1] - bp_gridY;
                ap_ufixed<24, 5> c = fp_ram[2] + xyzrel[ch][2];

                ap_ufixed<24, 7> R_rx;
#ifdef LOOKUP_SQRT
                R_rx = lookup_sqrt(ap_ufixed<24, 13>(a * a + b * b + c * c));
#else
                R_rx = MATH::sqrtf(a * a + b * b + c * c);
#endif
                // the division is automatically optimized to a right shift
                R_tr = (R_tx + R_rx) / 2;
            }
#ifdef LOOKUP_TRIG
```

```
53              ap_ufixed<10, 10, AP_RND> minus_phase = minus_k_1024 * R_tr;
54              complex<float> fact(
55                  lookup_cos(minus_phase),
56                  lookup_sin(minus_phase)
57              );
58  #else
59              float minus_phase = minus_k * R_tr;
60              complex<float> fact(
61                  MATH::cosf(minus_phase),
62                  MATH::sinf(minus_phase)
63              );
64  #endif
65
66              // ceil(log2(2995+1)) = 12 (uint so don't add sign bit)
67              ap_uint<12> index_r = ap_ufixed<13, 13, AP_RND>(R_tr * Rsample_inv)
68                  - ri_offset;
69
70              if (index_r < NBINS) {
71                  auto *rc = &(rc_ram[index_r][ch]);
72  #ifdef RC_HALF_FLOAT
73                  complex<float> rc2(rc->real, rc->imag);
74  #else
75                  // convert int16 to floating point
76                  union convert {
77                      float fl;
78                      uint32_t in;
79                  };
80                  Complex<convert> shifted;
81                  shifted.real.fl = rc->real;
82                  shifted.imag.fl = rc->imag;
83
84                  // efficient way of dividing by a power of two
85                  // (subtracting from the exponent)
86                  // in order to scale the RC data back
87                  if (shifted.real.in != 0) {
88                      shifted.real.in =
89                          (shifted.real.in & 0b10000000011111111111111111111111)
90                          | ((shifted.real.in >> 23) - rc_shift.real << 23);
91                  }
92                  if (shifted.imag.in != 0) {
93                      shifted.imag.in =
94                          (shifted.imag.in & 0b10000000011111111111111111111111)
95                          | ((shifted.imag.in >> 23) - rc_shift.imag << 23);
96                  }
97
98                  complex<float> rc2(shifted.real.fl, shifted.imag.fl);
99  #endif
100                 pixel += fact * rc2;
101             }
102         }
103
104         image_stream << pixel;
105
106     }
107 }
```

Bibliography

[1] C.P.R. Baaij. *Digital circuits in CλaSH: functional specifications and type-directed synthesis*. PhD thesis, University of Twente, January 2015. doi: 10.3990/1.9789036538039.

[2] Hui Bi, Guoan Bi, Lu Wang, and Xianpeng Wang. Airborne FMCW SAR sparse imaging: Initial results. February 2019. doi: 10.1109/ICDSP.2018.8631544.

[3] F. Cholewa, M. Wielage, P. Pirsch, and H. Blume. An FPGA architecture for velocity independent back-projection in FMCW-based SAR systems. In *2016 IEEE International Symposium on Signal Processing and Information Technology (ISSPIT)*, pages 252–257, 2016. doi: 10.1109/ISSPIT.2016.7886044.

[4] F. Cholewa, M. Wielage, P. Pirsch, and H. Blume. Synthetic aperture radar with fast factorized back-projection: A scalable, platform independent architecture for exhaustive FPGA resource utilization. In *International Conference on Radar Systems (Radar 2017)*, pages 1–6, 2017. doi: 10.1049/cp.2017.0494.

[5] I. G. Cumming, Y. L. Neo, and F. H. Wong. Interpretations of the omega-K algorithm and comparisons with other algorithms. In *IGARSS 2003. 2003 IEEE International Geoscience and Remote Sensing Symposium. Proceedings (IEEE Cat. No.03CH37477)*, volume 3, pages 1455–1458, 2003. doi: 10.1109/IGARSS.2003.1294142.

[6] David Elam and Cesar Iovescu. A block floating point implementation for an N-point FFT on the TMS320C55x DSP. *TMS320C5000 Software Applications*, SPRA948, Sep 2003. URL https://www.ti.com/lit/an/spra948/spra948.pdf.

[7] Giorgio Franceschetti and Riccardo Lanari. *Synthetic aperture radar processing*, page 66. CRC press, 1999.

[8] Dan Gisselquist. Double clocked FFT core, 2018. URL https://opencores.org/projects/dblclockfft.

[9] LeRoy A. Gorham and Linda J. Moore. SAR image formation toolbox for MATLAB. In Edmund G. Zelnio and Frederick D. Garber, editors, *Algorithms for Synthetic Aperture Radar Imagery XVII*, volume 7699, pages 46 – 58. International Society for Optics and Photonics, SPIE, 2010. doi: 10.1117/12.855375.

[10] M. Hofstra. Comparing hardware description languages. University of Twente, 2012.

[11] Intel. Devices: 28 nm device portfolio; Cyclone V FPGA features. URL https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/pt/cyclone-v-product-table.pdf.

[12] Intel. FFT IP core: User guide, November 2017. URL https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug_fft.pdf.

[13] Intel. *Cyclone V Device Handbook*, volume 1: Device Interfaces and Integration. October 2019. URL https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/cyclone-v/cv_5v2.pdf.

[14] J. J. L. Kant. Development of a FMCW SAR pre-processing interface. Internship report, De Haagse Hogeschool, May 2019.

[15] Martin Kirscht and Carsten Rinke. 3D reconstruction of buildings and vegetation from synthetic aperture radar (SAR) images. *MWA*, 1998.

[16] Ronald Kwok and William T. K. Johnson. Block adaptive quantization of Magellan SAR data. *IEEE Transactions on Geoscience and Remote Sensing*, 27:375–383, Jul 1989. doi: 10.1109/36.29557.

[17] B.P. Lathi and Z. Ding. *Modern Digital and Analog Communication Systems*. Oxf Ser Elec Series. Oxford University Press, 2010. ISBN 9780195384932.

[18] Miriam Leeser, Srdjan Coric, Eric Miller, Haiqian Yu, and Marc Trepanier. Parallel-beam backprojection: An FPGA implementation optimized for medical imaging. *Journal of VLSI signal processing systems for signal, image and video technology*, 39, 2005. doi: 10.1007/s11265-005-4846-5.

[19] Jiaguo Lu. *Design Technology of Synthetic Aperture Radar*. Wiley, 2019. ISBN 9781119564546.

[20] Maged Marghany. Chapter 8 - principle theories of synthetic aperture radar. In Maged Marghany, editor, *Synthetic Aperture Radar Imaging Mechanism for Oil Spills*, pages 127 – 150. Gulf Professional Publishing, 2020. ISBN 978-0-12-818111-9. doi: https://doi.org/10.1016/B978-0-12-818111-9.00008-2.

[21] M. Martone, M. Villano, M. Younis, and G. Krieger. Efficient onboard quantization for multichannel SAR systems. *IEEE Geoscience and Remote Sensing Letters*, 16(12):1859–1863, Dec 2019. ISSN 1558-0571. doi: 10.1109/LGRS.2019.2913214.

[22] A. Moreira, P. Prats-Iraola, M. Younis, G. Krieger, I. Hajnsek, and K. P. Papathanassiou. A tutorial on synthetic aperture radar. *IEEE Geoscience and Remote Sensing Magazine*, 1(1):6–43, 2013. doi: 10.1109/MGRS.2013.2248301.

[23] Rikin J. Nayak and Jaiminkumar B. Chavda. Comparison of accelerator coherency port (ACP) and high performance port (HP) for data transfer in DDR memory using Xilinx ZYNQ SoC. In Suresh Chandra Satapathy and Amit Joshi, editors, *Information and Communication Technology for Intelligent Systems (ICTIS 2017) - Volume 1*, pages 94–102, Cham, 2018. Springer International Publishing. ISBN 978-3-319-63673-3.

[24] M. Otten, W. Vlothuizen, H. Spreeuw, and A. Varbanescu. Real-time processing of multi-channel SAR data with GPUs. In *2016 European Radar Conference (EuRAD)*, pages 65–68, 2016.

[25] M. Otten, N. Maas, R. Bolt, M. Caro-Cuenca, and H. Medenblik. Circular micro-SAR for mini-UAV. In *2018 15th European Radar Conference (EuRAD)*, pages 321–324, 2018. doi: 10.23919/EuRAD.2018.8546633.

[26] M. P. G. Otten, J. S. Groot, and H. C. Wouters. Development of a generic SAR processor in The Netherlands. In *Proceedings of IGARSS '94 - 1994 IEEE International Geoscience and Remote Sensing Symposium*, volume 2, pages 903–905 vol.2, 1994. doi: 10.1109/IGARSS.1994.399295.

[27] J. J. Pimentel, A. Stillmaker, B. Bohnenstiehl, and B. M. Baas. Area efficient backprojection computation with reduced floating-point word width for SAR image formation. In *2015 49th Asilomar Conference on Signals, Systems and Computers*, pages 732–726, 2015. doi: 10.1109/ACSSC.2015.7421230.

[28] G. Pinitas. Towards real-time SAR. MSc thesis, TU Delft, July 2014.

[29] Merrill I. Skolnik. Pulse radar. Encyclopædia Britannica, March 2019.

[30] T. Truong, I. Reed, R. Lipes, A. Rubin, and S. Butman. Digital SAR processing using a fast polynomial transform. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 32(2):419–425, 1984. doi: 10.1109/TASSP.1984.1164307.

[31] L. M. H. Ulander, H. Hellsten, and G. Stenstrom. Synthetic-aperture radar processing using fast factorized back-projection. *IEEE Transactions on Aerospace and Electronic Systems*, 39(3):760–776, 2003. doi: 10.1109/TAES.2003.1238734.

[32] Wouter Vlothuizen and Maarten Ditzel. Real-time brute force SAR processing. *IEEE Radar Conference, Pasadena, CA, USA*, May 2009.

[33] M. Wielage, F. Cholewa, C. Fahnemann, P. Pirsch, and H. Blume. High performance and low power architectures: GPU vs. FPGA for fast factorized backprojection. In *2017 Fifth International Symposium on Computing and Networking (CANDAR)*, pages 351–357, 2017. doi: 10.1109/CANDAR.2017.101.

[34] Christian Wolff. Frequency-modulated continuous-wave radar (FMCW radar). radartutorial.eu, 2018. URL https://www.radartutorial.eu/02.basics/pubs/FMCW-Radar.en.pdf.

[35] Xilinx. Power Estimator (XPE) 2019.1.2. URL https://www.xilinx.com/products/technology/power/xpe.html.

[36] Xilinx. UG1270: Vivado HLS optimization methodology guide, Apr 2018. v2018.1.

[37] Xilinx. UG902: Vivado design suite user guide, Jan 2019. v2019.2.

[38] Xilinx. Zynq-7000 SoC product selection guide, 2019. URL https://www.xilinx.com/support/documentation/selection-guides/zynq-7000-product-selection-guide.pdf.

[39] Xilinx. UG871: Vivado design suite tutorial, August 2020. v2020.1.

[40] Y. Zhang, D. Zhu, X. Mao, X. Yu, J. Zhang, and Y. Li. Multirotors video synthetic aperture radar: System development and signal processing. *IEEE Aerospace and Electronic Systems Magazine*, 35(12):32–43, 2020. doi: 10.1109/MAES.2020.3000318.

[41] Zhou Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4):600–612, 2004. doi: 10.1109/TIP.2003.819861.