Time Domain Based Image Generation for Synthetic Aperture Radar on Field Programmable Gate Arrays

Von der Fakultät für Elektrotechnik und Informatik
der Gottfried Wilhelm Leibniz Universität Hannover
zur Erlangung des akademischen Grades
Doktor-Ingenieur
(abgekürzt: Dr.-Ing.)
genehmigte Dissertation

von Herrn
Fabian Cholewa M.Sc.

geboren am 25. Februar 1981
in Winsen an der Luhe / Deutschland

2019

1. Referent: Univ.-Prof. Dr.-Ing. Holger Blume
2. Referent: Univ.-Prof. Dr.-Ing. Jörn Ostermann

Tag der Promotion: 20.06.2018

Danksagung

Die vorliegende Dissertation ist im Rahmen meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Institut für Mikroelektronische Systeme (IMS) der Gottfried Wilhelm Leibniz Universität entstanden.

An erster Stelle möchte ich mich bei Herrn Prof. Dr.-Ing Holger Blume für die langjähre fachliche Unterstützung und die Übernahme des 1. Referats bedanken. Herr Blume hat in seiner Funktion als Leiter des Instituts außerdem einen wesentlichen Einfluss auf die positiven Ergebnisse dieser Arbeit. Des Weiteren möchte ich mich bei Herrn Prof. Dr.-Ing Jörn Ostermann für die Übernahme des Koreferats und die freundliche Zusammenarbeit im Bereich SAR bedanken. Mein weiterer Dank richtet sich an Herrn Prof. Dr.-Ing Peter Pirsch, für die jahrelange fachliche Unterstützung im Bereich SAR, sowie für die Übernahme des Prüfungsvorsitzes.

Ich möchte mich außerdem bei meinen Kollegen für die gute Atmosphäre am Institut, sowie für die diversen fachlich hilfreichen Diskussionen bedanken. Für die erfolgreiche Zusammenarbeit im Projekt, die Unterstützung und den Teamgeist möchte ich mich besonders bei Matthis Wielage, Christian Fahnemann und Martin Pfitzner bedanken.

Meinen Freunden danke ich für die Motivation und das Gefühl mit den täglichen Sorgen nicht alleine zu sein. Der größte Dank gilt meiner Familie, die mir immer das Gefühl der vollen Unterstützung geben und sich immer für mich eingesetzt haben. Ohne diese Unterstützung während meiner Laufbahn, wäre ich niemals bis zu diesem Punkt gekommen. Dafür möchte ich meiner Mutter Bärbel Stürzbecher, meinem Vater Erik Cholewa, meinem Stiefvater Emil Stürzbecher, meiner Schwester Melanja Tucker, sowie meinen Großeltern Christine und Rolf Meyer danken.

Ich widme diese Arbeit meinem Vater Erik Cholewa, der diesen Tag leider nicht mehr erleben durfte.

Abstract

Aerial images are important in different scenarios including surface cartography, surveillance, disaster control, height map generation, etc. Synthetic Aperture Radar (SAR) is one way to generate these images even through clouds and in the absence of daylight. For a wide and easy usage of this technology, SAR systems should be small, mounted to Unmanned Aerial Vehicles (UAVs) and process images in real-time. Since UAVs are small and lightweight, more robust (but also more complex) time-domain algorithms are required for good image quality in case of heavy turbulence. Typically the SAR data set size does not allow for ground transmission and processing, while the UAV size does not allow for huge systems and high power consumption to process the data. A small and energy-efficient signal processing system is therefore required.

To fill the gap between existing systems that are capable of either high-speed processing or low power consumption, the focus of this thesis is the analysis, design, and implementation of such a system. A survey shows that most architectures either have to high power budgets or too few processing capabilities to match real-time requirements for time-domain-based processing. Therefore, a Field Programmable Gate Array (FPGA) based system is designed, as it allows for high performance and low-power consumption. The Global Backprojection (GBP) is implemented, as it is the standard time-domain-based algorithm which allows for highest image quality at arbitrary trajectories at the complexity of $\mathcal{O}(N^3)$. To satisfy real-time requirements under all circumstances, the accelerated Fast Factorized Backprojection (FFBP) algorithm with a complexity of $\mathcal{O}(N^2 logN)$ is implemented as well, to allow for a trade-off between image quality and processing time. Additionally, algorithm and design are enhanced to correct the failing assumptions for Frequency Modulated Continuous Wave (FMCW) Radio Detection And Ranging (Radar) data at high velocities. Such sensors offer high-resolution data at considerably low transmit power which is especially interesting for UAVs.

A full analysis of all algorithms is carried out, to design a highly utilized architecture for maximum throughput. The process covers the analysis of mathematical steps and approximations for hardware speedup, the analysis of code dependencies for instruction parallelism and the analysis of streaming capabilities, including memory access and caching strategies, as well as parallelization considerations and pipeline analysis. Each architecture is described in all details with its surrounding control structure. As proof of concepts, the architectures are mapped on a Virtex 6 FPGA and results on resource utilization, runtime and image quality are presented and discussed. A special framework allows to scale and port the design to other FPGAs easily and to enable for maximum resource utilization and speedup.

The result is streaming architectures that are capable of massive parallelization with a minimum in system stalls. It is shown that real-time processing on FPGAs with strict power budgets in time-domain is possible with the GBP (mid-sized images) and the FFBP (any image size with a trade-off in quality), allowing for a UAV scenario.

Keywords: Synthetic Aperture Radar, SAR , Time-domain, Backprojection, FPGA

Kurzfassung

Luftbilder sind ein wichtiger Bestandteil in der Kartographierung, der Überwachung, des Katastrophenmanagements, der Generierung von Höhenmodellen usw. Die Technologie des Radars mit synthetischer Apertur, oder Synthetic Aperture Radar (SAR), eröffnet die Möglichkeit diese Bilder auch durch Wolken und in Dunkelheit zu erzeugen. Um diese Technologie in der Breite verfügbar zu machen, ist es nötig SAR Systeme zu entwickeln, die in Drohnen (Unmanned Aerial Vehicles (UAVs)) eingesetzt werden können und in der Lage sind Bilder schritthaltend zu generieren. jedoch machen die geringe Größe und das damit einhergehende geringe Gewicht von Drohnen, diese anfälliger für atmosphärische Einflüsse. Deshalb werden bei schwereren Turbulenzen robustere (aber auch komplexere) zeitbereichsbasierte Verfahren eingesetzt, um eine ausreichende Bildqualität gewährleisten zu können. SAR Systeme liefern in der Regel hohe Datenraten die, aufgrund der geringen Geschwindigkeiten der Funkverbindung, nicht schritthaltend zu einer Bodenstation übertragen werden können, um die Daten dort schritthaltend zu verarbeiten. Aufgrund der rauschartigen Charakteristik können die Daten auch nicht komprimiert werden. Da Drohnen üblicherweise nur über ein beschränktes Platz- und Energiebudget verfügen, müssen die eingesetzten SAR Systeme innerhalb dieser Spezifikationen liegen und dennoch in der Lage sein die Daten schritthaltend zu verarbeiten.

Um die Lücke zwischen bestehenden System zu schließen, die entweder nicht zu einer schritthaltenden Verarbeitung hochratiger Sensordaten in der Lage sind oder energieeffizienten arbeiten, liegt der Fokus dieser Arbeit auf der Analyse, dem Entwurf und der Implementierung eines Systems, das beide Eigenschaften vereint. Eine Voruntersuchung ergibt, dass die meisten nicht dedizierten Architekturen diese beiden Anforderungen nicht gleichzeitig erfüllen können. Deshalb wird eine dedizierte Field Programmable Gate Array (FPGA) Architektur entwickelt, die Energieeffizienz mit hoher Rechenleistung kombiniert. Das Standardverfahren im Zeitbereich ist die globale Rückprojektion (Global Backprojection (GBP)) mit einer Komplexität von $\mathcal{O}(N^3)$. Dieses wird implementiert, da es beliebige Trajektorien bei maximaler Bildqualität ermöglicht. Um die schritthaltende Verarbeitung bei größeren Datensätzen zu ermöglichen, wird außerdem das beschleunigte Rückprojektionsverfahren (Fast Factorized Backprojection (FFBP)) implementiert, das bei einer Komplexität von $\mathcal{O}(N^2 log N)$ einen Abtausch zwischen Bildqualität und Berechnungszeit ermöglicht. Außerdem wird die Architektur erweitert, um grundlegende Annahmen zu korrigieren die versagen, wenn Frequency Modulated Continuous Wave (FMCW) Radardaten verarbeitet werden, die bei hohen Geschwindigkeiten aufgenommen wurden. FMCW Sensoren sind speziell für UAVs interessant, da sie hohe Auflösungen bei vergleichsweise geringer Sendeleistung ermöglichen.

Für einen effizienten Entwurf, der vorhandene Hardware Ressourcen maximal nutzt, werden die Verfahren einer vollständigen Analyse unterzogen. Dies beinhaltet die Analyse des mathematischen Modells, sowie die Analyse möglicher Approximationsverfahren zur beschleunigten Ausführung auf dedizierten Architekturen.

Außerdem die Analyse von Instruktionsabhängigkeiten zur Maximierung der Parallelisierung, die Analyse der kontinuierlichen Datenwortverarbeitung in Bezug auf den Hauptspeicher- und Zwischenspeicherzugriff, sowie die Analyse zum Pipelining der Verfahren. Alle Architekturen werden inklusive der Kontrollstrukturen detailliert beschrieben. Zum Nachweis der Machbarkeit werden die Architekturen auf einen Virtex 6 FPGA übertragen und die benötigten Ressourcen, die Laufzeiten sowie die resultierende Bildqualität präsentiert und bewertet. Ein spezielles Framework erlaubt die einfache Portierung auf weitere FPGAs, sowie die ressourcenabhängige Skalierung, um die jeweils maximale Beschleunigung zu erreichen.

Das Resultat ist eine Architektur, die eine nahezu kontinuierliche Datenwortverarbeitung bei massiver Parallelisierung und gleichzeitig minimaler Unterbrechung der Pipeline ermöglicht. Es wird gezeigt, dass eine schritthaltende Verarbeitung bei gleichzeitiger strikter Begrenzung des Energiebudgets, im Zeitbereich mit dem GBP (bis zu mittleren Bildgrößen) sowie mit dem FFBP (für beliebige Bildgrößen bei gleichzeitigem Abtausch gegen Qualität), möglich ist. Der Einsatz auf UAVs ist damit sinnvoll möglich.

Schlagworte: Radar mit synthetischer Apertur, SAR, Zeitbereich, Rückprojektion, FPGA

Contents

List of terms and abbreviations **XI**

List of symbols **XIII**

1 Introduction **1**
- 1.1 A brief history of Radar and SAR . . . 3
- 1.2 Challenges and resulting motivation . . . 6
- 1.3 Research objectives . . . 9
- 1.4 Structure of this work . . . 10

2 SAR image processing **11**
- 2.1 SAR signal model . . . 11
- 2.2 Azimuth compression in frequency-domain . . . 16
- 2.3 Azimuth compression in time-domain . . . 17
  - 2.3.1 Global backprojection algorithm (GBP) . . . 17
  - 2.3.2 Fast factorized backprojection algorithm (FFBP) . . . 21
  - 2.3.3 FMCW backprojection algorithm (start-stop-approximation) . 25

3 Related work **29**
- 3.1 Non dedicated architectures . . . 29
- 3.2 Dedicated architectures . . . 31
- 3.3 Evaluation and discussion . . . 32

4 Conceptual hardware design **33**
- 4.1 Field Programmable Gate Arrays (FPGAs) . . . 33
- 4.2 Principles for dedicated hardware implementation . . . 36
- 4.3 Analysis of the mathematical SAR model . . . 39
- 4.4 Analysis of processing dependencies . . . 40
  - 4.4.1 GBP dependencies . . . 44
  - 4.4.2 FFBP dependencies . . . 47
  - 4.4.3 Conclusion of dependency analysis . . . 53
- 4.5 Analysis of streaming capabilities . . . 54
  - 4.5.1 Memory access patterns (loop ordering) . . . 54
    - 4.5.1.1 General considerations for RAM access . . . 55
    - 4.5.1.2 Considerations for GBP RAM access optimization . 57
    - 4.5.1.3 Considerations for GBP RAM access reduction . . . 60
    - 4.5.1.4 Considerations for FFBP RAM access optimization 65
    - 4.5.1.5 Considerations for FFBP RAM access reduction . . 68

4.5.2 Inter loop streaming (read write balancing) 69
4.5.2.1 GBP inter loop streaming 70
4.5.2.2 FFBP inter loop streaming 75
4.5.2.3 Inter loop streaming on module level 78
4.5.3 Intra loop streaming (pipelining) 80
4.5.3.1 GBP intra loop streaming 83
4.5.3.2 FFBP intra loop streaming 85

5 Hardware implementation 89
5.1 Basic dedicated signal processing function units 89
5.1.1 Range distance approximation 90
5.1.2 Square root and trigonometric functions (CORDIC) 92
5.1.3 Interpolation 97
5.1.3.1 Evaluation of interpolation methods 97
5.1.3.2 Implementation of interpolation 101
5.1.4 Complex multiplication 104
5.2 Dedicated GBP implementation 106
5.2.1 GBP PE 107
5.2.2 GBP control structure 111
5.3 Dedicated FMCW implementation 112
5.4 Dedicated FFBP implementation 114
5.4.1 Analysis of factorization impact on image quality 116
5.4.2 Mapper 121
5.4.3 Factorizer 126
5.4.4 Full FFBP system 131
5.5 Integration of dedicated modules 132

6 Results and evaluation 135
6.1 Resource utilization 135
6.1.1 GBP resource utilization 135
6.1.2 FMCW resource utilization 139
6.1.3 FFBP resource utilization 141
6.2 Performance results 145
6.2.1 GBP performance results 145
6.2.2 FMCW performance results 148
6.2.3 FFBP performance results 148
6.3 Image results 151
6.3.1 GBP image results 151
6.3.2 FMCW image results 152
6.3.3 FFBP image results 153
6.4 Evaluation and discussion of results 154

7 Summary 159

List of terms and abbreviations

ALU	Arithmetic Logic Unit
ARU	Arithmetic Unit
ASIC	Application Specific Integrated Circuit
BRAM	Block Random Access Memory
CAT	Computer Aided Tomography
CBP	Convolution Backprojection
CE	Cordic Element
CLB	Configurable Logic Block
CORDIC	Coordinate Rotation Digital Computer
COTS	Commercial Of The Shelf
CUDA	Compute Unified Device Architecture
DDR	Double Data Rate
DSP	Digital Signal Processor
FBP	Filtered Backprojection
FFBP	Fast Factorized Backprojection
FFT	Fast Fourier Transform
FIFO	First In – First Out
FIR	Finite Impulse Response
FMCW	Frequency Modulated Continuous Wave
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
FU	Function Unit
GBP	Global Backprojection
GPP	General Purpose Processor
GPU	Graphic Processing Unit
HDL	Hardware Description Language
ISLR	Integrated Sidelobe Ratio

LFM	Linear Frequency Modulated
LUT	Look Up Table
MAC	Multiply-Accumulate
MSE	Mean Squared Error
NoC	Network on Chip
PE	Processing Element
PLD	Programmable Logic Device
PPP	Pulse to Pixel Projection
PSLR	Peak Sidelobe Ratio
PSNR	Peak Signal to Noise Ratio
Radar	Radio Detection And Ranging
RAM	Random Access Memory
RAR	Real Aperture Radar
RDA	Range Doppler Algorithm
RISC	Reduced Instruction Set Computer
RMS	Root Mean Square
ROM	Read Only Memory
SAR	Synthetic Aperture Radar
SDRAM	Synchronous Dynamic Random Access Memory
SIMD	Single Instruction Multiple Data
SoC	System on Chip
Sonar	Sound Navigation And Ranging
SRAM	Static Random Access Memory
TDC	Time Domain Convolution
TDP	Thermal Design Power
UAV	Unmanned Aerial Vehicle
UEMU	Unified EMUlation Framework
UWB	Ultra WideBand
WKA	Wavenumber Domain Algorithm

List of symbols

Notation	Description	Dimension
Θ_{az}	aperture angle in azimuth direction	$\circ$
Θ_{rg}	aperture angle in range direction	$\circ$
l_a	antenna lenght	m
t_z	aperture position vector z component	m
t_y	aperture position vector y component	m
t_x	aperture position vector x component	m
N_{az}	apertures in azimuth	pure
r_a	azimuth resolution	m
b_w	beam width	m
b_l	burst length of memory transfer	pure
B_{cf}	bus clock frequency	Hz
$calc_{GBP}$	calculated runtime of the GBP	s
IY_{subC}	center position of subimage on ground in y	m
IX_{subC}	center position of subimage on ground in x	m
I_{subC}	center position of subimage on ground	m
δy	distance between pixel in y axis	m
δx	distance between pixel in x axis	m
r_{range}	covered distance in range line	m
i_{rg}	discrete sample position in range	pure
i_{az}	discrete aperture position in azimuth	pure
i_y	discrete y coordinate for pixel	pure
i_x	discrete x coordinate for pixel	pure
r_z	distance in z between pixel and antenna	m
r_y	distance in y between pixel and antenna	m
r_x	distance in x between pixel and antenna	m
r_{max}	distance antenna to end of samples	m
rd	distance between sample and aperture	m
r_{min}	distance antenna to first sample	m
δrg	distance between range samples	m
δr	distance between pixel and antenna	m
τ_p	duration of pulse	s
IY_{subR}	half width of subimage on ground in y	m

Notation	**Description**	**Dimension**
Z	height map array for ground area	m
z	height of coordinate in z axis	m
IP	image pixel position array on ground	m
I	image plane array	pure
y	image pixel in y direction	pure
x	image pixel in x direction	pure
r_{cell}	index for range cell	pure
i_r	index of sample in range line	pure
S_{comp}	interpolated phase corrected aperture	pure
S_{int}	interpolated aperture	pure
hw_{GBP}	measured runtime of the GBP in hardware	s
T'	new aperture position array	m
p_{fint}	number of parallel interpolations FUs	p
p_{fact}	number of parallel factorizer PEs	pure
f_{sub}	number of subimages in total	pure
N_y	number of pixel in y	pure
N_x	number of pixel in x	pure
f_{sub_y}	number of subimages in y	pure
f_{sub_x}	number of subimages in x	pure
f_{apt}	number of factorized apertures	pure
T	original aperture position array	m
p_{FFBP}	parallelization degree of the FFBP PE	pure
p_{GBP}	parallelization degree of the GBP PE	pure
p	parallelization degree of processing elements	pure
pc_c	phase correction constant	pure
ϕ_{corr}	phase correction angle	$\circ$
PE_{pcf}	processing element pipeline frequency	Hz
PE_{cf}	processing element clock frequency	Hz
r_{comp}	range compensation factor	complex
r_r	range resolution	m
N_{rg}	samples in aperture	pure
S'	set of new apertures	pure
S	set of apertures	pure
B	signal bandwidth	Hz
rg	single range sample position	pure

Notation	**Description**	**Dimension**
d_s	slant range to ground	m
c_0	speed of light	$\frac{m}{s}$
δr_s	squared distance between pixel and antenna	m
r_{zs}	squared distance in z between pixel and antenna	m
r_{ys}	squared distance in y between pixel and antenna	m
r_{xs}	squared distance in x between pixel and antenna	m
IY_{subR}	startposition of subimage on ground in y	m
PE_{llt}	time to load a full range line	s
PE_{pt}	time of processing of one PE	s
τ	time of travel	s
d_{nc}	vector between new aperture and new center	m
d_{oc}	vector between old aperture and new center	m
d_{ns}	vector between new aperture and sample	m
d_{cn}	vector between center and new aperture	m
d_{on}	vector between old and new aperture	m
λ	wavelength of the radar	m
D_{ww}	width of data word	bit
B_{ww}	width of memory bus word	bit
IY_{subD}	width of subimage on ground in y	m
IY_{absD}	width of image on ground in y	m
IX_{absD}	width of image on ground in x	m

1 Introduction

The principle of visual perception is a powerful and important mechanism of sensing. Many daily routines, whether complex or simple, rely strongly on this principle. The power lies within the capability of the eyes, to constantly assimilate a vast amount of surrounding information, coupled with the brain that processes and interprets this data within a very short time. Thus, many technical systems forward information only or mainly in a visual manner. The principle of Radio Detection And Ranging (Radar) constitutes no exception to that. Radar sensors send electromagnetic waves to illuminate the surrounding area, the reflected echoes carry wavelength characteristic information back to the sensor. This information is visualized and can be interpreted with a certain knowledge of the system. The most common use case for Radar technology is the range detection of objects and determination of their velocity and direction. This is used for the coordination of maritime and air traffic. The big advantage of this technology becomes obvious in four features:

1. The range of the human eye is extended
2. As an active system radar is independent of daylight
3. Information that is not visible for the human eye becomes accessible
4. Specific electromagnetic waves can penetrate certain materials and atmospheres

Besides the very basic and spread application of maritime and air traffic control, the principle of Radar can be enhanced in many different ways to gather and represent supplementary information. One of these enhanced principles is the imaging technique Synthetic Aperture Radar (SAR). Radar systems usually present information in a greatly reduced complexity, objects are reduced to a dot or their Radar cross-section. Since the process of gathering SAR information is different from Radar, certain information is lost, while other information is gained. In contrast to Radar, SAR systems are capable of presenting information in a way, that is more likely to be interpreted directly by the human eye. When mounted on an airborne platform, the gathered data can be processed to become comparable to electro-optical ground images. Depending on the used SAR principle, this technology is useful in many scenarios such as:

1. Topographic mapping of planetary surfaces or ocean floors [1, 2]
2. Monitoring the development and condition of crops [3]
3. Disaster control (e.g. landslide identification [4])
4. Military or civil surveillance [5]

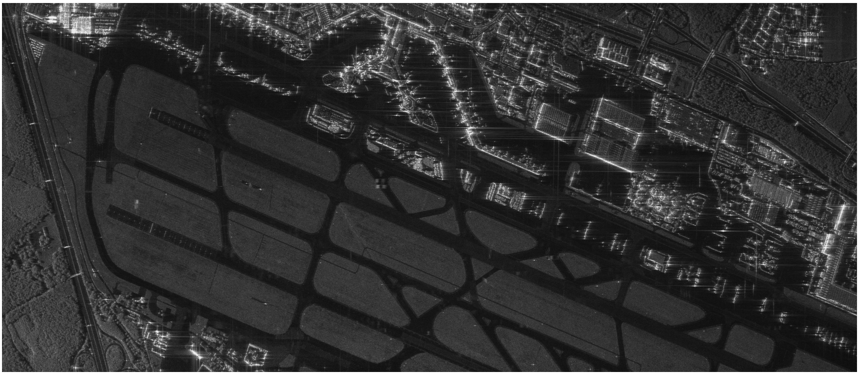


Figure 1.1: SAR image of Frankfurt airport provided by Intermap Technologies [6]

An example for SAR generated images is given in Fig. 1.1 provided by Intermap Technologies Canada. The image is part of a mapping campaign surrounding Frankfurt airport (Germany).

On a first glimpse, this image resembles a standard planar representation of a top-down grayscale aerial photo. But a detailed examination of Fig. 1.1 reveals fundamental differences to electro-optical top-down images. Some structures are emphasized while others are blurred, buildings seem to be observed from an angle and not top-down, and although streets and runways are present, no cars or planes seem to be present in the image. The differences to electro-optical images result from the SAR principle and how data is recorded and processed.

First, electromagnetic microwaves can be reflected almost completely, pass through or be absorbed depending on the material. This can be an advantage, as it creates the mentioned highlighting or fading of structures. Second, the area was scanned in $8.8\ km$ height with a slant range of $9.2\ km$. This results in an angle of roughly 45°, creating a slightly tilted representation of higher objects. Still, it would not be perceived as a 45° angle. With SAR, even sharper angles are possible whereas the perception is not affected to the same extent. This allows for extreme slant range scans. In contrast to optical systems, the achievable spatial resolution of a SAR system does not change with the slant range, making it a perfect candidate for satellite missions. Third, SAR images base on data recorded over time, while an optical image is recorded in an instant. This results in smearing and dislocation of moving objects in a SAR image (e.g. white artifacts in the runway area). This effect is a disadvantage if objects need to be identified, which is important in surveillance but not in mapping means. In some cases, this can become an advantage, as objects can be indicated as being in regular motion or static.

Ground imaging is the classic SAR scenario, but many adaptations exist which allow for the retrieval of different scientific data, making SAR an important tool for different applications. Still, this technology comes with certain disadvantages which exist in recording and fast and accurate processing of SAR data. The presented thesis tackles especially the problems of faster and more accurate processing especially in harsh environments.

1.1 A brief history of Radar and SAR

The first Radar system was developed in the year 1904. The German engineer Christian Hülsmeyer developed a device called Telemobilskop, which allowed to detect ships in a distance of up to 3000 m with the help of electromagnetic waves [7]. In 1922 the Italian physicist Guglielmo Marconi developed a device to detect metallic objects [8]. This principle was adopted in the same year by two American engineers, A.h.Taylor and L.C. Young, who build the first Continuous Wave (CW) Radar [9]. The second world war boosted the research and development of different radar technologies significantly, resulting in increasingly powerful constructions. In 1939 a Radardevice was used onboard an airborne platform for the first time [9]. This fast development shows that Radar was applicable in the early stages with available technologies.

Compared to Radar, SAR demands for more complex calculations. The theory of wavefront reconstruction was developed by the Hungarian engineer and Nobel Prize winner Dennis Gabor in 1948 [10]. In the year of 1951, the American mathematician Carl A. Wiley recognized, that this theory can be combined with the airborne Radar technology to generate images of an observed ground area. The principle of SAR was born, but the technology to perform the actual math for such complex signal processing was not available at that time [11]. Nevertheless, the first simple analog experimental SAR system was operating in 1952 at the University of Illinois [12]. The first SAR images of the coast of Key West (Florida) were created in 1953 [13] with this system. The patent was registered only later by Wiley in 1954 [14] and published in 1965.

More precise SAR focusing systems were developed in the optical field by the American engineers Loius J. Cutrona, Weston E. Vivian and Emmett N. Leith [15]. They started the development of optical (analog) SAR processors using the principle of Fresnel approximation. This was the standard way to generate SAR images until the development of digital SAR signal processors. The first concepts and simulations for digital SAR processing were conducted during the 1970s by the American engineers William M. Brown [16], John C. Kirk [17], W.J. van de Lindt [18] and C. Wu [19]. With the introduction of digital processing in SAR focussing, many new problems raised about how to record and process the digital data [20]. An operating digital SAR processor was first designed for the SEASAT satellite in 1978 [21]. From this day, digital SAR focusing became relevant.

In general SAR data processing can be separated in the frequency-domain-based and time-domain-based algorithms. Since the mathematical model of wavefront reconstruction was too complex for the available processing capacities in the late seventies, a less demanding algorithm had to be developed for SEASAT. This was done by MacDonald Dettwiler (MDA) and the Jet Propulsion Lab (JPL) [20] and got to be known as Range Doppler Algorithm (RDA) [22]. Therefore, the RDA was the first frequency-domain-based operating digital SAR processor. Approximations were used to simplify the mathematical model of SAR processing, which allows for faster computation with less accuracy for some cases. Many developments in the decades after the SEASAT project increased the potential of SAR, including antennas, digital components, and algorithms. Leading to applications in many different fields besides ground imaging, like topography, terrain discrimination, forestry, urban growth, oceanography, and many others.

With exponential growing processing capacities in the following decades, other frequency-domain-based algorithms like the Wavenumber Domain Algorithm (WKA) [23] and the Chirp Scaling Algorithm (CSA) [24] were developed, which increased the quality of SAR images. The WKA was the first algorithm to fully embrace the principle of wavefront reconstruction. All of the named algorithms are still popular today. Especially the RDA is widely spread as it is fast and simple and still offers good image quality. But all frequency-domain algorithms are limited in certain aspects because approximations or assumptions are made to simplify the calculation. The most important limitations are the maximum bandwidth of the Radar and the shape and extent of deviation from a linear path.

Processing in the time-domain is free from the here mentioned limitations but also more complex. The basic principle of time-domain-based algorithms is known as backprojection and is based on the Radon transformation postulated by the Austrian mathematician Johann Radon in the year 1917 [25]. It is used to transform a plane into a set of lines (projections), where each line (projection) contains the density information of the entire plane from one defined angle. Radon himself already postulated the inversion formula that is used to construct the plane information from the lines (projections). To retrieve the line information, an object or area needs to be circled by several aligned sources and several aligned sinks, facing each other to collect the density information. This is called a parallel beam geometry.

The base of the time-domain algorithms, which are also used in SAR, emerged first from the field of astronomy with the formulation of the projection slice theorem in 1956 by the Australian physicist Ronald Bracewell [26]. Thereby, Bracewell formulated the bridge between the Radon transform and the Fourier Transform by expressing the relation between the projection data and their two dimensional Fourier transform. The theorem was revisited in the early sixties for the Computer Aided Tomography (CAT) principle by different scientists, the American physician William H. Olfendorf in 1961 [27], the American physician David E. Kuhl in 1996 [28] and the South African physicist Allan McLeod Cormack in 1963 [29], who formulated the basic adaption. But a clear image could not be retrieved with the given settings.

Cormack continued on the field and was able to improve the method, but it was up to Bracewell and Riddel in 1967 [30], who enhanced the theorem with appropriate filters to remove ambiguities. The method is know today as Filtered Backprojection (FBP), Convolution Backprojection (CBP) or Time Domain Convolution (TDC). The American mathematician Larry Shepp and the electrical engineer Benjamin F. Logan implemented this method for CAT the first time in 1974 and improved it further, creating clear images of a section of a human head. The principle was adapted in parallel by Lakshminarayanan [31] and Budinger et. al [32] in 1975, to a fan beam geometry with only one source covering the entire object.

When compared, a specific type of a SAR scenario (spotlight mode) and fan-beam CAT scenarios are sorts of similar. In both cases, an area is circled and scanned from all angles by a certain type of electromagnetic waves. Considering the similarities, it is surprising that the class of time-domain algorithms was recognized for SAR much later. Compared to frequency-domain-based algorithms, time-domain-based algorithms include more detailed geometrical information during the image forming process. This results in better image quality but also a higher degree of complexity and therefore more demanding processing capacities. The early use of time-domain-based algorithms in CAT might origin from the fact that CAT systems create smaller images, while SAR systems usually create large images. Processing capabilities were simply not given at that time. When compared in more detail, CAT and SAR scenarios are different for many reasons. CAT is a circular scenario with an exact, predefined path for the multiple transmitters and receivers which are located in opposite positions, facing each other (bistatic), and are active simultaneously. Most SAR scenarios only use one transmitter and receiver simultaneously which are placed directly next to each other (monostatic), or sender and receiver are the same antenna alternating between transmission and reception. On top of that, the path is not predefined and could be either circular or linear with no exact position information. In addition to these differences, another frequency band, which can also be much wider in bandwidth, is used for SAR. But for the mentioned reasons, time-domain-based solutions could not be applied directly to SAR.

In 1983 the American engineer David C. Munson proved, that FBP could be adapted to process SAR data of a spotlight scenario [33]. A more common case for SAR is the so-called stripmap geometry where an area is scanned while flying along a linear path. The adaption to a SAR stripmap geometry was covered later by many authors independently. The American and Swedish mathematicians John A. Fawcett [34] and Hans Hellsten et. al [35] proved, independent from each other, a valid method to invert the Radon transformation for a SAR setup in 1985 and 1987 by using Fourier Hankel transformations. Both rely on straight, instead of circular, paths which can be used for stripmap scenarios. In the approach of the Swedish mathematician Lars Erik Andersson [36] in 1988 it is described how the algorithm is adapted to handle Ultra WideBand (UWB) signals, which is a key feature for SAR data processing.

When both methods are combined, a general algorithm can be derived that can process UWB SAR data in a stripmap geometry. This algorithm is often referred to as Global Backprojection (GBP) or TDC algorithm and has become a golden reference for the processing of SAR data, as it provides the best image quality. But it has high demands in processing capacities at the same time. Because of this, it is not often used in systems that need to deliver fast image results.

To counter long processing times, faster versions of the GBP algorithm were developed over the past years. Two different classes beside the basic GBP algorithm class exist. First, the two-stage algorithms which reside in a complexity class between the GBP and frequency-domain algorithms, e.g. the Fast Backprojection algorithm (FBP) developed by Ali. F. Yegulap in 1999 [37] or the Local Backprojeciton (LBP) algorithm developed by Swedish mathematicians Olle Seger et. al in 1998 [38]. Second the multiple stage algorithms that can actually reach the same complexity of the frequency-domain algorithms, e.g. the Quadtree Backprojection algorithm (QTBP) developed by John McCorkle et. al in 1996 [39], the Links Fast Backprojection algorithm (LFBP) developed by the Swedish mathematicians Stefan Nilsson et.al in 1998 [40], the Fast Hierarchical Backprojection algorithm (FHBP) developed by the American engineer Samit Basu et. all in 2000 [41] and the Fast Factorized Backprojection algorithm (FFBP) developed by the Swedish mathematician Lars Ulander et. al in 2000 [42]. Frankly, the modified versions reduce the computations for the GBP step, by combining adjacent areas of SAR data to reduce the processing complexity. As there is no free lunch, speeding up the processing results in a decrease in image quality under certain circumstances.

Depending on the size and quality of SAR data, the demands in image quality, processing capacities and the given SAR scenario, any of the time or frequency-domain algorithm can be used. The ongoing fast development of computer technology increased the capabilities of SAR systems continuously, which created an interest in the technology in military and civil organizations, making it a key technology in remote sensing.

1.2 Challenges and resulting motivation

Inverse image reconstruction from multiple separated echos is of interest in fields of application like SAR, CAT and Sound Navigation And Ranging (Sonar). Different wavelengths, modes, and antenna settings allow for a variety of information to be obtained from the observed area. SAR can be used in near field scenarios, but the most common use cases are a far-field scenario like spaceborne and airborne SAR. For airborne scenarios, Unmanned Aerial Vehicles (UAVs) gain more importance due to the high reduction of costs and risk. This allows for wider civil usage of SAR in the fields of disaster prediction and control, crop and forest monitoring, archaeology, cartography, and others.

But implementing SAR on UAVs calls for strict restrictions regarding system dimensions and energy consumption of the system processing SAR data. Furthermore small and lightweight airborne platforms react more sensitive to atmospheric conditions then bigger and heavier planes. Therefore, the imaging algorithm needs to be robust enough to compensate for the effects of motion in the platform trajectory. Nevertheless, because of the mentioned advantages, this work is focused on a UAV based SAR scenario.

Depending on the application case and the sensor, the amount of raw data can be in the range of mega samples per second (MS/s). As image reconstruction algorithms come with a high degree of complexity, this results in large processing times (growing exponentially with the amount of raw data) and high energy consumption. For air- and spaceborne platforms these aspects are critical for operation. For effective missions, the image should be available shortly after the data was sampled (real-time mission control). Storing the data on-board for ground processing is therefore not a preferred option. Two option exist to handle this (runtime/energy) challenge. The first and trivial option is a real-time transmission of the raw data to the ground, so that image reconstruction can be performed on high-performance computers. But based on the sensor scan rate, the amount of raw data per second cannot be transmitted via standard downlinks. This calls for broadband solutions or the compression of the raw data before transmission. But SAR raw data resembles a noise structure, which is why compressing results in few percentages of data reduction. Broadband downlinks are not always possible either, common limitations are the given terrain (no line of sight) or the maximum energy consumption (for transmission) of the given platform (e.g. satellites). The sophisticated and second option is on board image reconstruction. Standard image compression codecs such as H264 can then be used to reach high data reduction factors on the generated images, allowing for standard downlinks to be used for real-time transmission. This is advantageous in two aspects. First, only a fraction of the raw data is stored onboard in small processing buffers, which reduces storage hardware to a minimum. Second, real-time mission control is possible if the delays between echo reception onboard and image reception on the ground can be greatly reduced. This is possible when the delays for processing, compression and image transmission are small enough. The delays for image transmission and compression are considerably small, as highly efficient hardware solutions exist for image compression, leaving only a fraction of the original data for fast image transmission. Only the delay for processing is challenging as image generation algorithms demand high-performance computing. This automatically results in bigger systems dimensions and high energy consumption. To reduce the requirements the simplest image generation algorithm should be picked.

When only runtime complexity is considered, frequency-domain-based algorithms are the better choice for processing SAR data, as they are faster at least one order of magnitudes when compared to time-domain algorithms (backprojection). The downside of these algorithms lies within the approximations used for image generation. Approximations simplify the mathematical model in a way that only simple trajectories (linear movements) of the platform result in good image quality. But airborne platforms underlie strong deviations, which can result in a drastically reduced image quality. These errors can be compensated to a certain degree, but a full recovery is not possible. As a rule of thumb, it can be stated, that a deviation within the amount of a quarter wavelength of the carrier frequency from a linear path in any axis, does not affect the quality of the image. For SAR systems the carrier frequencies are partitioned in wavebands from $0.2\ GHz$ to $300\ GHz$, resulting in wavelengths from $1.5\ m$ to $1\ mm$. Applying the quarter wavelength rule, already the upper boundary of $1.5\ m$ would limit the allowed motion deviation to $37.5\ cm$. For any type of airborne platform, this limitation would be violated by the typical flight path deviations. Time-domain algorithms, on the other hand, are more complex, but do not rely on approximations wherefore any trajectory is allowed and any deviation can be compensated. This is of special interest for small airborne platforms (UAVs) as they are more affected by weather conditions. This work is addressing the question under which conditions this problem can be solved, which hardware could be used for such a demanding task, and how this might limit the achieved result.

This creates a challenging scenario: A small airborne platform with reduced space, and restrictions on weight and energy consumption, has to be capable of high-performance computing for time-domain-based SAR processing. For UAVs all aspects need to be reduced to the lowest possible values to use a maximum of resources for the actual SAR sensor system. Common General Purpose Processor (GPP) architectures are not suited for this task. The power of GPPs lies within the broad variety of tasks they can handle. Due to this, the processing performance for tasks is limited on average. While multicore or manycore architectures, containing many GPP cores could partly close this gap, constraints on system size and energy consumption prohibit this solution, as already one powerful GPP core can consume the energy of $100\ W$ and above at peak performance. Another option are Graphic Processing Units (GPUs) which are tailored for image processing. The power of GPUs lies within the massive parallel processing but makes these platforms also very energy-demanding which can result in $200\ W$ power dissipation and above. Additionally to the high energy demands, GPUs need a host system which increases the system size. Digital Signal Processors (DSPs) which are processors that are tailored for signal processing tasks consume less energy when compared to standard GPPss or GPUss, but the performance for massive parallel tasks is reduced.

The optimal architecture that combines all criteria are Application Specific Integrated Circuits (ASICs). For maximum performance, the circuit can be designed to only implement the signal processing operations required for the specific task. Omitting unnecessary operations automatically reduces the architecture to the lowest possible system dimensions and energy consumption. The parallelism of the implemented operations can be used to increase performance. The big downside of an ASIC is the fact, that it cannot be updated to algorithmic adaptations and that it exceeds development time and cost by far, when compared to other architectures.

A compromise solution is Field Programmable Gate Array (FPGA) technology, which allows the same level of adaption and parallelism but does not match the same small system dimensions, low energy consumption, and fast operating frequency as ASICs. However, system dimension and energy consumption will still reach lower levels when compared to other high-performance platforms. The relatively low operating frequencies of FPGAs can be compensated by the massive parallelism. Development time is however comparatively high, as it requires a full hardware description. In general, the time factor needs to be considered for system design but becomes less relevant as system constraints get harsher. In contrast to ASICs, FPGAs can be adapted to algorithmic changes, although not as fast as any other mentioned architecture. When leaving development time aside, considering all given constraints and the pros and cons of each technology, FPGA technology is the best compromise. The resulting motivation is to investigate how and within which limits, onboard processing of SAR data in the time-domain is possible with FPGA based systems.

1.3 Research objectives

As a key technology for surveillance, disaster control and cartography, SAR image generation is a continuous research topic. Different approaches in time-domain were already examined on different non-dedicated architectures [43, 44, 45, 46] and dedicated architectures [47, 48, 49]. Nevertheless, it becomes apparent that a gap in the field of fast time-domain-based processing on FPGAs exits.

Therefore, as a first step, one major objective is the identification of fast time-domain-based processing algorithms suitable for FPGAs implementation. A preliminary study with a high-level software implementation is necessary prior to any implementation in hardware. The study serves as a reference for quality evaluation and to examine algorithm settings, while the objective is to identify the best trade-off between achieved quality and complexity. The second objective is a low-level accuracy analysis. In contrast to pure software implementations, dedicated designs demand a deeper understanding of the underlying hardware structure for optimal results. This starts with the number representation in fixed-point format rather than floating-point format as used in software implementations. Fixed point format demands detailed analysis to understand how the reduced number format might affect the accuracy. The third objective is an efficient implementation. FPGA designs cannot compete with other architectures in terms of operating frequency, wherefore the objective is to choose algorithms, which are either suited for massive parallelization or divisible in small steps to implement a fine-grained pipelined streaming architecture. In addition to the algorithm structure, all steps can be performed with different accuracy. It is the objective, to identify approximations that fit the needs, are efficient to implement and are easy to pipeline or used in parallel. The fourth objective is scalability and portability. As the resources of FPGAs on the market constantly increase, the designed architecture should be divided in generic modules that can be scaled in size, accuracy or amount to max out the resources of a wide variety of FPGAs. For universal utilization, the architecture should only depend on common hardware periphery like Double Data Rate (DDR) memory. To map the architecture on different FPGAs, the signal processing modules should be wrapped in a module that translates between the board-specific periphery and the internal communication interface. This would allow for module internal changes without any effect on the external periphery and vice versa. The fifth objective is the proof of the entire design in the form of a fully working, closed-loop real-time laboratory setup. As every setting of SAR sensors and every parameter in backprojection algorithms can change the outcome of the image result, a laboratory demonstrator is necessary to validate the universal application of the architecture. The whole setting should include a SAR sensor, signal sampling, and signal processing in real-time. Where real-time means, data needs to be processed within the same duration of sampling time. To increase the effectiveness of real-time mission control, the latency of processing should be as small as possible.

1.4 Structure of this work

This thesis is structured as follows. Chapter 2 will explain the fundamental theories of the SAR principle and signal model. Based on this, the idea behind backprojection is described and the used algorithms will be explained in more detail. To understand the relevance of the presented work, Chapter 3 will give an overview of the state of the art SAR processing with backprojection on different architectures. To understand how an efficient implementation of time-domain-based processing on dedicated architectures can be achieved, Chapter 4 will explain the fundamental ideas for fast implementation. All alternative concepts will be discussed, considering the fact, that an on-board integration of time-domain-based processing on dedicated hardware is the main goal. After this, in Chapter 5 the basic signal processing blocks for fast implementations and the backprojection architectures are explained in detail, to understand which concrete measures were adopted for implementation. In Chapter 6 the concepts will be evaluated regarding resource utilization and runtime. Image quality is evaluated on different data sets which vary in their SAR scenario setting. Also, further options for fast dedicated implementations are discussed. A summary of the whole work is given in Chapter 7.

2 SAR image processing

Radio Detection And Ranging (Radar) is a key technology for a variety of different applications in many different fields. Synthetic Aperture Radar (SAR) is one of these fields, where Radar technology is used to generate digital 2D images of an observed ground area. SAR systems usually base on pulsed Radar systems in a far-field scenario which involves an airborne or space-borne platform. Due to the latest development in SAR sensor technology, apart from far-field scenarios, also near field scenarios can be covered. This is possible by the use of Frequency Modulated Continuous Wave (FMCW) Radar sensors, which offer finer resolutions but do not cover bigger distances. The basic SAR signal model and the generation of images from the raw data will be explained in the following sections. The processing steps for image generation, the acquired data and the applied algorithms will be outlined.

2.1 SAR signal model

All SAR system base on the principles of Real Aperture Radar (RAR) systems [50]. Enhancements of the technology, which mostly based on signal processing, allowed to overcome the natural limitations of RAR. The limitations can be understood by looking closer into the process of how RAR systems generate images. To retrieve ground information, a RAR system sends multiple narrow beams of energy perpendicularly to the flight path (azimuth direction) of the airborne or space-borne platform to the ground. The echos are collected over time and are used to generate the ground image. The geometry is depicted in Fig. 2.1.

Based on the beam width and the distance to the ground a rather small or rather big area, called the swath, is illuminated by the beam. The distance between any point within the swath and the RAR system is called slant range. This swath has a length in range direction (perpendicular to the flight path in azimuth direction) which results from the slant range d_s and the aperture angle in range direction Θ_{rg} of the used antenna. Within this swath, echos from further ranges return at a proportionately larger time then echos from closer ranges. Based on these time values, the echos can be separated into equidistant cells, which are combined to a range cell vector (range line N_{az}). The relative intensity of the echos in the vector is used to generate a single image line of a narrow swath of terrain. The size of the cells defines the resolution of the image line in range direction. The resolution in azimuth direction is defined by the width of the swath, which depends on d_s and the aperture angle Θ_{az} in the azimuth direction. Due to the continuous forward movement of the platform, the next pulse is transmitted at a slightly different position in azimuth.

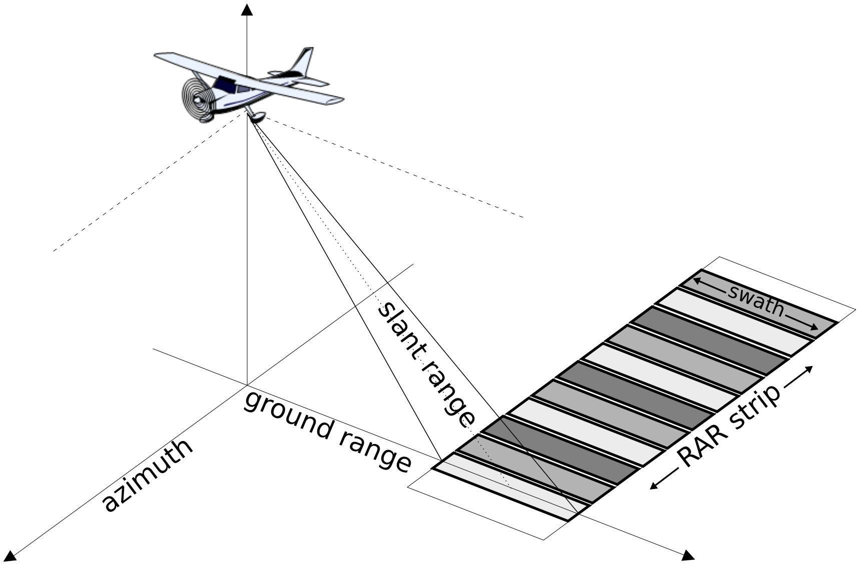


Figure 2.1: Geometry for stripmap RAR on airborne platform

Therefore, a slightly different strip of terrain will be illuminated. By forming a sequential strip (RAR strip) of terrain swaths side by side in azimuth, a two-dimensional data array that forms the RAR image is created.

Due to the principle of how RAR systems work, the resolution of the image is heavily limited. To be able to distinguish between two objects in range direction, the objects must be separated by a minimum distance in slant range d_s in order to locate them in two different range cells. The distance d_s to an object is defined by the travel time τ of the pulse to an object. Since the pulse travels to the object and back to the sensor with the speed of light c_0, the actual distance results to Eq. (2.1) [20].

$$d_s = \frac{c_0 \cdot \tau}{2} \tag{2.1}$$

This distance d_s must be greater than the radar pulse length τ_p. Otherwise the back scattered pulse of the second object would overlap with the back scattered pulse of the first object and would locate both echos in the same range cell. Range resolution r_r is therby defined to equation Eq. (2.2) [20].

$$r_r = \frac{c_0 \cdot \tau_p}{2} \tag{2.2}$$

To focus in range direction, in order to improve the resolution in range, the length of the radar pulses τ_p should be as short as possible. But shortening the pulse is problematic for three reasons.

First, the higher the frequency of a Radar system, the more it is affected by weather conditions like rain or clouds.

Second, the pulse must transmit enough energy, so that the backscattered echo energy is still strong enough to enable for the detection of the reflected signals. To keep the energy level constant, shortening the pulse demands for an increase of pulse energy (amplitude). This is problematic, since the design of systems that transmit very short, but high energy pulses are difficult and uneconomical.

The third reason is the direct dependency of range and azimuth resolution. To counter the range resolution problem, the first enhancement to RAR systems is introduced with the technique of range compression. Range compression is one of the two major steps in focusing Radar data and thereby an important step for moving from RAR to SAR systems. Range compression uses so-called chirped pulses (chirps) which are used in most Radar systems. Chirps are characterized by a Linear Frequency Modulated (LFM) pulses instead of short pulses with a constant frequency. Another type of pulses are so-called FMCW pulses which are covered later. Chirps can be much longer than a not modulated pulse and thus allows the pulse energy to be transmitted with lower peak power. When the echoed chirp is filtered with a matched filter, a narrow pulse is the result which contains condensed pulse energy. This improves range resolution and signal to noise ratio. The matched filtering can be implemented in many ways. One efficient method is the use of Fast Fourier Transforms (FFTs) and is done in the frequency-domain. Such operations have certain advantages in terms of processing speed, which is one reason for the existence of frequency-domain-based SAR processing algorithms. The matched filter corresponds to the convolution of the echo with a replica of the original pulse. This focuses the pulse to a much shorter length. Overlapping pulses can now be distinguished by referencing time to frequency within the signal bandwidth B. The range resolution is then defined by Eq. (2.3) [20].

$$r_r = \frac{c_0}{2 \cdot B} \tag{2.3}$$

The range compression finishes the focusing of the Radar data in range direction. As already mentioned, shortening the pulse length is problematic also because of the direct dependency of the azimuth resolution from the range resolution. The azimuth resolution defines the ability to separate two objects in azimuth direction. This resolution is defined by the aperture angle Θ_{az} of the antenna. All objects that are illuminated by the antenna beam while having the same distance to the antenna, will backscatter energy at the same time. These echoes are received at the same time by the sensor, wherefore the sensor cannot distinguish between the two objects, although the might lie on the opposite edges of the beam.

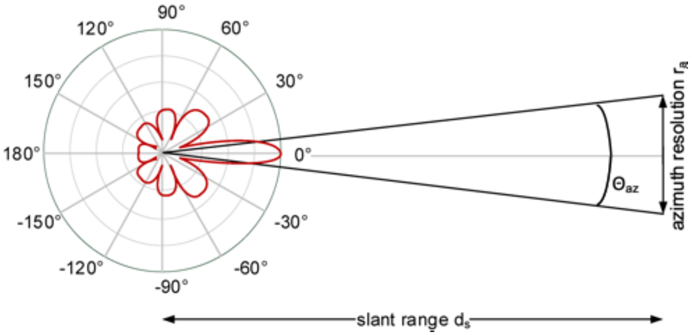


Figure 2.2: Antenna diagram with aperture

Therefore, the beam needs to be as narrow as possible in order to illuminate as few objects in azimuth at the same time. Figure 2.2 shows an antenna diagram with the corresponding aperture angle Θ_{az} [20]. The angle is defined by the width of the main lobe of the antenna, whereby the main lobe is defined as the area where loss of echo energy does not drop under $3\ dB$ (50%). Since the aperture angle Θ_{az} defines the beamwidth and is a fixed value for a given antenna, the diagram shows that the azimuth resolution r_a (so also the range resolution) depends on the slant range distance d_s to the object. The beamwidth b_w depends on the physical length l_a of the RAR antenna in azimuth direction and the wavelength λ of the Radar and results to Eq. (2.4) in meter. [20].

$$b_w = \frac{\lambda}{l_a} \tag{2.4}$$

This shows that to narrow the beamwidth, either the wavelength (pulse width) needs to be reduced or the size of the real antenna must be increased. Due to the already discussed limitations, the wavelength can only be shortened to a certain limit. The same applies to the antenna length l_a. Since the resolution in azimuth results to Eq. (2.5) [20], only the slant range distance can be reduced.

$$r_a = d_s \cdot \tan b_w \cdot 2 \tag{2.5}$$

Also, the slant range can only be reduced in certain limits on airborne and spaceborne platforms and would not be sufficient to reduce azimuth resolution as required.

To solve this problem, the data must be focused in azimuth. This is the second step after range compression and completes the processing of SAR data. As already indicated by the name, a long real aperture is synthesized by stitching several small real apertures together, forming a virtual antenna.

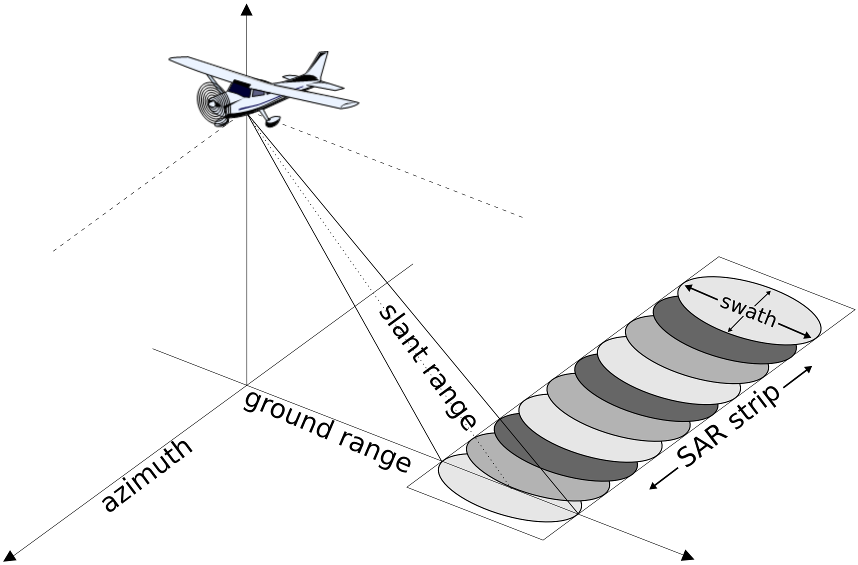


Figure 2.3: Geometry for stripmap SAR on airborne platform

To form the synthetic aperture, the linear movement of the platform along a linear track in azimuth is used. This is called stripmap mode in SAR. In contrast to RAR systems, the antenna of such systems is small. Thereby the aperture and the resulting swath size on the ground is wide. Therefore, an object on the ground will be illuminated by many different pulses which overlap each other. This also means that an object is illuminated from many different angles during the flight (antenna synthesis). The principle is shown in Fig. 2.3.

While in RAR systems, an image is formed rather simple due to the single image line scans, SAR systems rely on a rather complex signal processing to focus the image. This is also called azimuth compression. The different techniques to perform azimuth compression are discussed in the following sections and is the main focus of this work. The basic principle of azimuth compression relies on the fact that every pulse that covered the object, contains a small amount of echo energy from this object. The echo energy is located in the range line N_{az}, the position within the range line depends on the distance between object and sensor. Due to the wide aperture, this distance information results in a blurred position, represented by a circular arc around the antenna position. Due to the movement of the platform, the distance between sensor and object will change continuously, which shifts the position of the echo energy in every following range line N_{az}. The multiple samples of blurred information create an overdetermined equation system, allowing to trace back an object location.

The backtracing is done by simple summation of energy. An object at a certain position will create an energy pulse in a specific range position, according to the distance to the antenna. This position will shift in every line due to the movement of the antenna. This energy can be projected as a circular arc around each antenna position. By doing this, all arcs will overlap at the correct position to focus the object. This belongs to the mathematical group of inverse problems since the image is traced back from a set of indirect observations. According to [20] the azimuth resolution for SAR systems results to Eq. (2.6), so to half of the antenna length l_a.

$$r_{as} = \frac{l_a}{2} \tag{2.6}$$

This shows the superiority of SAR systems. First, to increase the number of samples per object the beam width should be widened, which is contrary to RAR system. Second, the resolution does not depend on slant range d_s anymore, since the length of the virtual antenna can be used to compensate for the change in slant range. This enables for high ground resolutions which are not possible with RAR systems. The disadvantage of SAR system is the high demand for processing power to perform the signal processing to focus the image. In addition to the stripmap mode, other modes exist where the antenna is always steered to focus on one spot during forward-motion on a straight path (spot mode). Another form is circular SAR, where an area is circled and continuously scanned. Both modes allow for even higher resolutions. The processing is divided into frequency and time-domain-based algorithms, both having pros and cons which are covered in the following sections.

2.2 Azimuth compression in frequency-domain

While range compression is similar for all domains, azimuth compression is divided into frequency and time-domain-based algorithms. The most prominent algorithms in the frequency-domain are the Range Doppler Algorithm (RDA) and the Wavenumber Domain Algorithm (WKA). The undeniable advantage of frequency-domain-based processing is the relatively small complexity when compared to time-domain-based processing. Both the RDA and the WKA reside within the complexity class of $\mathcal{O}(N^2 logN)$ for images with the size of $N \times N$ pixels. Where N represents the number of collected echo lines N_{az}. Data is transformed into the frequency-domain via a FFT for processing. The class of algorithms is named after that. In the frequency-domain, a simplification is possible as the convolution theorem [51] states that a convolution in the time-domain equals a pointwise multiplication in the frequency-domain. This allows substituting all matched filter convolutions in the algorithms with multiplications. While processing speeds up, the simplification does not allow for a direct interpretation of time-dependent parameters. This can create ambiguities and reduced image quality under certain conditions (motion variations from a linear track). Motion compensation algorithms are capable of correcting the deviations to a certain degree.

Nevertheless, the RDA and WKA are still state of the art algorithms and are used in most SAR systems for processing. But one has to bear in mind, that SAR systems usually were mounted to bigger and more stable platforms in the past, wherefore motion was not a severe problem. With the availability of smaller SAR frontends which are capable of high-resolution scans, smaller platforms are also used. Especially small Unmanned Aerial Vehicles (UAVs) are affected by atmospheric turbulences, because of the lightweight system design, comparatively lower speed, and altitude. For this reason, the assumption of an ideal flight path fails in most practical airborne scenarios, which makes the precise correction of motion errors mandatory.

2.3 Azimuth compression in time-domain

Frequency-domain-based compression assumes a rather straight flight path to work properly. While the assumption of an ideal straight flight path (without bigger deviations) can be valid for bigger platforms, such assumptions are usually invalid for smaller lightweight platforms. The big advantage of time-domain-based azimuth compression over frequency-domain-based azimuth compression is the inclusion of all available trajectory information, which allows compensating any deviation during processing. Additionally, the principle provides the best possible image results for SAR data. The Global Backprojection (GBP) is the basic algorithm for time-domain-based azimuth compression and is therefore explained in detail. But it imposes the highest degree of complexity for processing. To reduce complexity, several different methods exist, from which the Fast Factorized Backprojection (FFBP) shows the most potential for complexity reduction, flexibility, and preservation of image quality. Therefore, this method is examined as well. All of these methods were developed under the principle of very short Radar pulses, the FMCW technology with longer frequency pulse ramps might violate this assumption for processing. In order to compensate for longer pulse ramps, the principle of FMCW Radar is explained.

2.3.1 Global backprojection algorithm (GBP)

The backprojection principle is based on the Radon transform [25], which can be used for indirect image formation. It is commonly used in the medical field [30] as data sets and image dimensions are smaller, which allows for decent runtimes despite the high complexity of the mathematical problem. This problem resides in the complexity class of $\mathcal{O}(N^3)$ [52], where N represents the number of collected echo lines and processed image pixels in x and y dimensions. Therefore, computation time is challenging when high-resolution images are required in real-time due to exponentially growing processing times. Different adaptations reformulated the problem and adapted it to SAR scenarios [33, 34, 35, 36]. The GBP follows the basic physical and geometric principles of SAR which is depicted in Fig. 2.4.

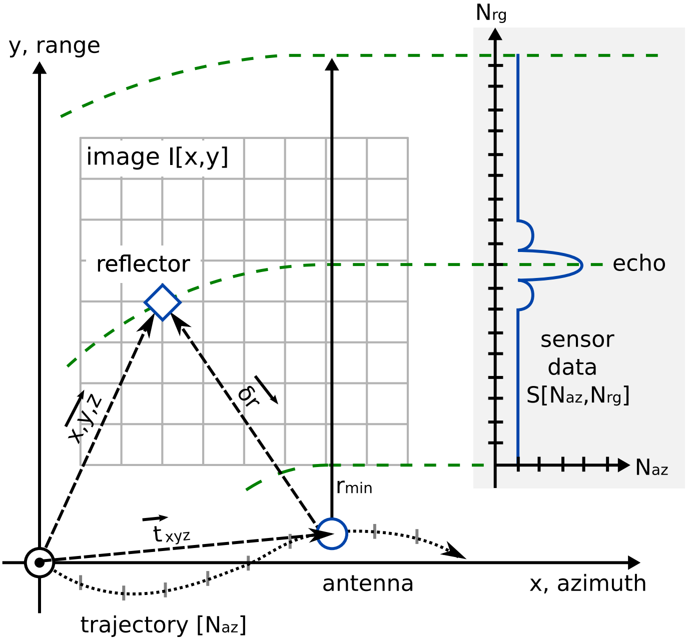


Figure 2.4: SAR geometry with trajectory vector and overlapping image plane [53]

SAR data is acquired by moving a wide aperture sensor on a path in azimuth while looking sidewards during pulse transmission. During acquisition, all objects (which are illuminated by the sensor) induce an echo signal in the raw sensor data S of each pulse. By traveling N_{az} positions, N_{az} views (range lines) of an observed object from different angles and distances are collected. System-specific range compression, according to Section 2.1, is used for each of these range lines, to form the equidistant range samples N_{rg} or range cells. These range samples form a range line that is part of the sensor data set S. The index of every range sample i_r represents the time when this sample was acquired. The described process is a direct observation result of the ground, viewed from the moving platform.

To form an image with backprojection, this traced back and transferred onto the image plane. This can also be understood as a projection of the energy back on the ground, which explains the name of the algorithm. The time indices i_r of each range sample in a range line, can be used to allocate the specific range sample in each range line that holds the echo signal of one specific object on the ground. This requires the position of the object on a 2D pixel raster to form an image plane.

The position $\overrightarrow{xyz}$ is composed with the help of the components calculated according to Eq. (2.7) - Eq. (2.9).

$$x = i_x \cdot \delta x \tag{2.7}$$

Where i_x is the discrete coordinate of the current pixel in x in the image plane and δx is the increment in distance between two pixel. It is assumed, that the origin of the platform trajectory (which define the antenna positions) and the origin of the image plane are identical.

$$y = i_y \cdot \delta y \tag{2.8}$$

Where i_y is the discrete coordinate of the current pixel in y in the image plane I and δy is the increment in distance between two pixel.

$$z = Z[x, y] \tag{2.9}$$

Where z is the height value in m for every ground coordinate. The value is read from an array that holds a height map of the scanned aerial. The z component is required for correct calculation but not for the 2D image raster. Based on the antenna position $\overrightarrow{t_{xyz}}$ and the position of the object on the raster $\overrightarrow{xyz}$, the distance between the object and the antenna can be calculated following Eq. (2.10).

$$\delta r = |\overrightarrow{t_{xyz}} - \overrightarrow{xyz}| \tag{2.10}$$

Due to the movement of the platform, the distance δr between an object and the antenna changes for every range line N_{az}. This gives an individual index or time value r_{cell} for the object in each range line and results to Eq. (2.11)

$$r_{cell} = \frac{\delta r - r_{min}}{\delta rg} \tag{2.11}$$

Where δrg is the distance between each range sample, and r_{min} is the distance from the antenna to the first range sample. Both factors are defined by the SAR system. As r_{cell} will not be a discrete value, interpolation is required, to match r_{cell} with the equidistant pattern of indices i_r of a range line. Since the complex valued range samples not only carry information about the reflected energy, but also about the phase signal, a phase correction is required as well. The correction requires the angle ϕ_{corr} which results to Eq. (2.12).

$$\phi_{corr} = \delta r \cdot \frac{4\pi}{\lambda} \tag{2.12}$$

This finishes the mapping of one range sample of one pulse N_{az} to one pixel in the image plane I. This is also called a Pulse to Pixel Projection (PPP). As image reconstruction takes place in the time-domain, which directly corresponds to the spatial domain, no approximations are required while an exact solution is calculated.

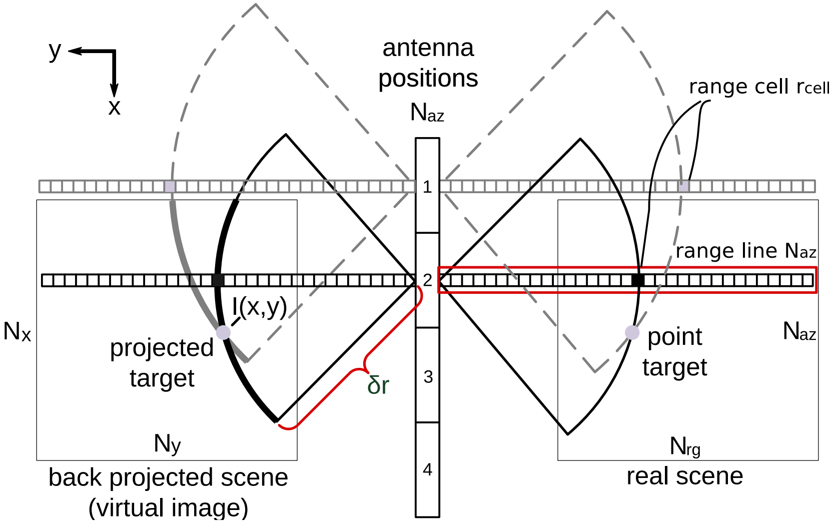


Figure 2.5: The process of projecting sample energy back to image plane I [54]

To process the entire image plane I, each pixel is defined as an object and must be processed. Iterating over all N_{az} pulses (range lines), the distance δr between each antenna position $\overrightarrow{t_{xyz}}$ on the trajectory vector and the pixel position $\overrightarrow{xyz}$ is calculated. Each range line N_{az} at all positions n in the sensor data arry S, contributes one value to the pixel position $I[x,y]$. This is done by the interpolation of S corresponding to the distances δr. This results to S_{int}, which is phase corrected based on ϕ_{corr} and coherently accumulated to form the final pixel value according to Eq. (2.13). This finishes the projection of one pixel to the image plane and requires N_{az} PPPs.

$$I(x,y) = \sum_{n=1}^{N_{az}} S_{int}(n, r_{cell}) \cdot e^{i \cdot \phi_{corr}} \tag{2.13}$$

The resulting magnitude of an output pixel finally matches the reflectivity at that particular position Ix,y. This is done for all $x \times y$ pixel in the image plane Iaccording to Eq. (2.14) and yields a total number of $\mathcal{O}(N^3)$ complex valued projections.

$$I = \sum_{n=0}^{N_{az-1}} \sum_{x=1}^{N_x} \sum_{y=1}^{N_y} S_{int}(n, r_{cell}) \cdot e^{i \cdot \phi_{corr}} \tag{2.14}$$

This is also described as pseudo-code in Algorithm 4.1. The principle of the circular projection of echo energy across the image plane I is also called smearing and is depicted in more detail in Fig. 2.5.

The numbers 1-4 correspond to the N_{az} antenna positions where the radar sensor transmits/receives the sent/reflected pulses. The range compression is applied to form the range lines with N_{rg} range samples. During data acquisition, all point targets in the real scene, which are covered by an aperture, induce an echo in the raw sensor data array S of the pulse. The point target echo is allocated to a range cell r_{cell} (as indicated by the black and grey cells in the real scene) based on the time passed between sending and receiving. This cell position changes for each antenna position, as indicated by position 1 (grey line) and 2 (black line). The reflection value of each range cell is backprojected (smeared) on the virtual image I, based on the distance δr between a pixel position and the antenna position. Since the calculated distance usually does not match with the equidistant discrete cell of the range line, the values have to be interpolated (S_{int}) and phase-corrected. But only the distance δr and not the direction of the reflection is known, therefore, the energy is projected on circular lines. This adds the energy to each pixel that is covered by the circular line. If this is done for every antenna position in flight direction (azimuth), accumulation ensures that the pixel is focused at the correct geometrical position. All lines, corresponding to the point target, overlay at a certain point (projected target) and therefore focuses the pixel. The resolution of the virtual image plane I is arbitrary, as the grid positions can be chosen freely. In contrast to frequency-domain-based algorithms, the GBP can handle arbitrary trajectories without additional motion-compensation, as data is processed directly in the time-domain, so no ambiguities are created.

2.3.2 Fast factorized backprojection algorithm (FFBP)

The problem complexity $\mathcal{O}(N^3)$ of the GBP limits the image size and raw data size for real-time application. Different modifications were developed to reduce the complexity to $\mathcal{O}(N^2\sqrt{N})$ [37] and $\mathcal{O}(N^2 logN)$ [42]. The modifications serve as a good trade-off between problem complexity, robustness, and quality. The modification presented by Ulander [42] is called FFBP and shows the most potential in terms of performance result and for the adaptation to dedicated hardware architecture.

The FFBP is a time-domain-based algorithm which uses the GBP algorithm for core calculations. To reduce the overall complexity, the impact of one of the three factors that lead to the high GBP complexity is reduced. Such reductions are possible for GBP processing, when the image dimension in x or y direction or the number of incoming apertures per calculated image are reduced. But a reduction of image resolution is not optional as image resolution should always match, or at least be close to the maximum possible quality. Otherwise, the SAR system should be reduced in complexity. This leaves the aperture set as the only candidate for complexity reduction. Well designed SAR systems sample data close to the Nyquist rate [55]. This means that every reduction of the aperture set inevitably leads to undersampling. While the GBP can run with a reduced aperture set, the undersampling will cause aliasing effects which leads to ghost images.

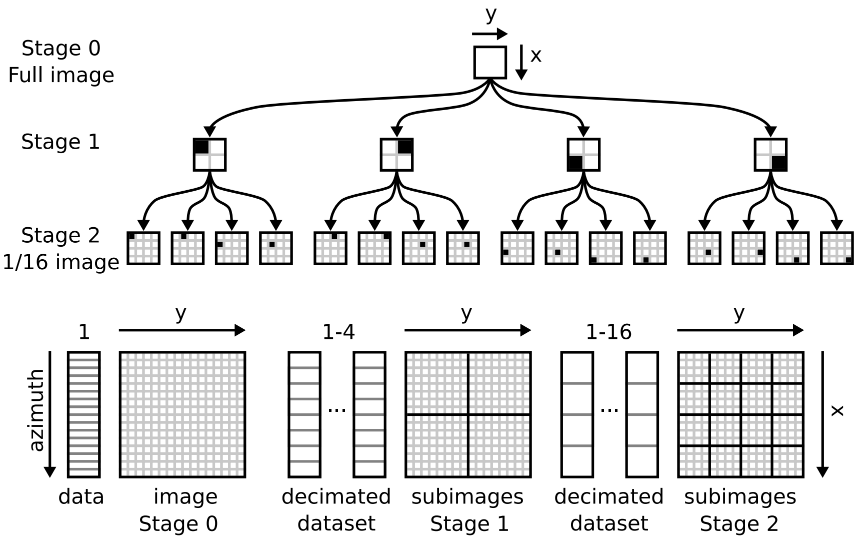


Figure 2.6: Factorization of image I and apertures N_{az} in two FFBP stages s

The amount of ghost images, which manifest as a mitigated version of the original image, correlates with the rate of undersampling and decrease image quality accordingly. Nevertheless, the approach of the FFBP is to reduce the aperture set, but instead of skipping apertures a subset of adjacent apertures is merged (factorized) to counteract the effect of undersampling. The idea is to hold the information of many apertures in one new merged (factorized) aperture. A merged set of adjacent apertures is called a subaperture, while the size of a subaperture is defined by f_{apt}. A set of subapertures is called a decimated dataset.

As there is no free lunch, also the factorization introduces image errors in the form of information loss. To counter the error (or loss of information), the process is repeated for different sections of the image. These sections are referred to as subimages in the following. The number of subimages is controlled by f_{sub_x} in x direction and f_{sub_y} in y direction of the image. Each aperture of the decimated dataset is rectified to the geometrical center of its subimage, to be merged to a new subaperture for this specific subimage. This results in multiple decimated datasets of unique information for each subimage. The process can be repeated in successive stages s, while an increasing number of subimages is generated with progressively fewer subapertures in each stage. A decimated dataset is created by factorizing the apertures of the previous stage. The error becomes smaller the more subimages are created per stage. A sketch of a symmetric process of factorization with two stages is depicted in Fig. 2.6 as an example.

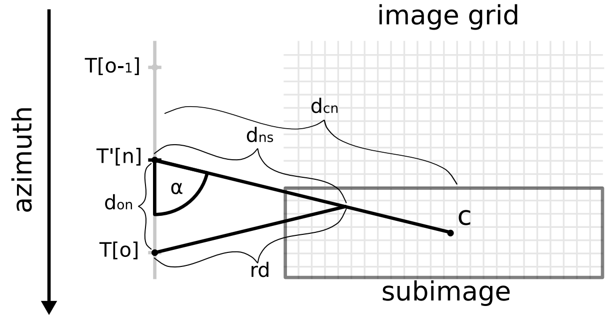


Figure 2.7: Geometry for factorization of subapertures and subimages

In each of the two stages s, 4 apertures are merged to a subaperture ($f_{apt} = 4$). At the same time, the image is split in x and y direction by a factor of 2 ($f_{sub_x} = f_{sub_y} = 2$) in each stage. This example is called symmetric, because the total factor of aperture factorization ($4 \times 4 = 16$) equals the total number of subimages ($2 \times 2 \times 2 \times 2 = 16$). When splitting an image into subimages, the resolution of each subimage is reduced by the factor of $f_{sub_x} \times f_{sub_y}$. Assuming an aperture count of 16 and an image resolution of 16×16, each of the 16 subimages would have a resolution of 4×4. The GBP would required $16 \times 16 \times 16 = 4096$ PPP for this example. The FFBP creates 16 subimages with 2 subapertures, while each subimage is processed by the GBP, and would therefore require $16 \times (2 \times 4 \times 4) = 512$ PPP. This shows the possible potential of the FFBP and how complexity reduction is achieved. Although parts of the information is lost within this process (wherefore the effect of undersampling cannot be contained completely), quality can be kept to a certain degree while complexity is reduced. A reduction in complexity is reached, when the effort for factorization and the processing of the reduced aperture set with the GBP, is smaller than the effort for full GBP processing. With this trade-off for image quality, the FFBP reduces the processing effort and tends to reach a complexity of $O(N^2 logN)$.

The whole geometry of the FFBP scenario is formulated in polar coordinates and is depicted in Fig. 2.7. For aperture factorization, the midpoint of the decimated dataset must be know. It will become the new position of the subaperture and is based on the set of original aperture positions $T[N_{az}]$. The position of the new subaperture will be referred to as $T'[n]$, it becomes a member of the set of subaperture positions $T'[N_{az}/f_{apt}]$. The actual required mathematical operations for the FFBP can be formulated as followed. $T'[n]$ results to Eq. (2.15). The indices n, s and k always start at 0. While f_{apt_s} is the factorization factor for stage s.

$$T'[n] = \frac{1}{f_{apt_s}} \cdot \sum_{k=0}^{f_{apt_s}-1} T[f_{apt_s} \cdot n + k] \tag{2.15}$$

The vector d_{on} between the subaperture position $T'[n]$ and a position $T[o]$ of the original aperture o results to Eq. (2.16).

$$d_{on} = T'[n] - T[o] \tag{2.16}$$

The vector d_{cn} between the subaperture position $T'[n]$ and the subimage center c results to Eq. (2.17).

$$d_{cn} = c_0 - T'[n] \tag{2.17}$$

Knowing d_{on} and d_{cn} allows to calculate the cosine for angle $\alpha = d_{on} \measuredangle d_{cn}$ between the two vectors following Eq. (2.18).

$$\cos\alpha = \frac{d_{on} \cdot d_{cn}}{||d_{on}|| \cdot ||d_{cn}||} \tag{2.18}$$

To form a range sample of a subaperture, one range sample value of each of the f_{apt} apertures, within the decimated dataset, is required for accumulation. To find the corresponding range sample position in an old (original) aperture o, rectification is required. This process requires information about the range sample in the subaperture first. Each range sample has an index i_r which corresponds to a distance d_{ns} from the subaperture position $T'[n]$ to the range sample position s. This distance results to Eq. (2.19).

$$d_{ns} = i_r \cdot \delta rg + r_{min} \tag{2.19}$$

Where δrg is the distance between each range sample, and r_{min} is the distance from the antenna to the first range sample. These values originate from the GBP geometry. d_{ns} overlays with d_{cn} up to the range sample position. Thus the cosine of α spans between d_{ns} and d_{on}. Having enough knowledge about the geometrical setup, the law of cosine is used to calculate the range distance rd between the range sample i_r and the aperture position $T[o]$ according to Eq. (2.20).

$$rd = \sqrt{||d_{on}||^2 + d_{ns}^2 - 2 \cdot ||d_{on}|| \cdot d_{ns} \cdot \cos\alpha} \tag{2.20}$$

The distance rd spans a circular arc around the aperture position $T[o]$, and therefore crosses the sample in aperture o that corresponds to sample i_r in subaperture n.

To retrieve the distance index r_{cell} of this sample in o, Eq. (2.19) can be rearranged to Eq. (2.21)

$$r_{cell} = \frac{rd - r_{min}}{\delta rg} \tag{2.21}$$

Like in the GBP, the calculated range sample index will not be a discrete value, thus interpolation and phase correction is required for compensation. Interpolation is based on distance index r_{cell}. To obtain the angle ϕ_{corr} for phase correction, d_{ns} is subtracted from rd, to get the difference in length. The result is multiplied with a phase correction constant $pc_c = 4\pi/\lambda$. The rectification of one range sample to the beam between $T'[n]$ and the center c of a subimage then results to Eq. (2.22).

$$S_{int}(T[f_{apt_s} \cdot n + k], r_{cell}) \cdot e^{j \cdot \phi_{corr}} \tag{2.22}$$

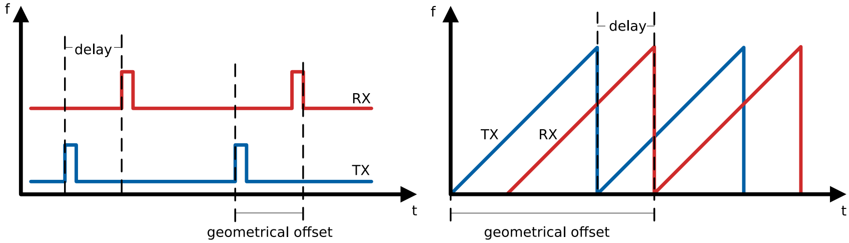
Where S is the array of all apertures. The full merge of one sample of a subaperture results to Eq. (2.23).

$$S'[i_{apt}, i_r] = \sum_{k=0}^{f_{apt_s}-1} S_{int}(T[f_{apt_s} \cdot n + k], r_{cell}) \cdot e^{j \cdot \phi_{corr}} \tag{2.23}$$

Where S' is the array of all subapertures. This process is performed for each sample of the new subaperture. The error that is introduced during this process, or the loss of information, can be explained by the aperture enlargement through aperture factorization. The longer the virtual subaperture becomes, the more the subaperture angel is narrowed. Thus, a subaperture does not illuminate the same image area as the aperture before. By this, smaller subimage areas are formed during the process. This process is called factorization. The number of iterations, of subimages in each iteration, and the number of combined apertures for every subimage can be chosen independently. Therefore, sets of parameters are discussed in Section 5.4.1, where a comparison of quality and runtime is given. Pseudo-code for the FFBP is listed in Section 4.4.2.

2.3.3 FMCW backprojection algorithm (start-stop-approximation)

Common SAR algorithms were formulated for pulsed Radar systems. The short duration of such pulses allowed for the so-called star-stop-assumption or start-stop-approximation. This assumption implies, that the position for emitting a pulse is identical with the position for echo reception.



(a) Geometrical offset for pulsed Radar (b) Geometrical offset for FMCW Radar

Figure 2.8: Comparison of geometrical offset for pulsed and FMCW Radar [54]

This assumption derives from the propagation speed of electromagnetic waves (light speed) in combination with very short pulse durations and low platform velocity. To gain better resolutions, especially in mid and near field scenarios, FMCW Radar sensors were developed lately. FMCW sensors emit a continuous signal ramp, which is modulated in frequency for the duration of the ramp. The long duration allows to emit a Ultra WideBand (UWB) signal, which increases range resolution according to Eq. (2.3). While for pulsed Radar a transmission takes a few microseconds, FMCW ramps last a couple of milliseconds. This means, that the distance that is covered between transmission of a ramp and reception of the entire echo is significantly longer for FMCW sensors. Therefore, in case of longer frequency ramps in combination with a fast-moving platform, the change in angle is so fast, that the start-stop-assumption is corrupted. The difference is shown in Fig. 2.8. It is obvious that FMCW differs from pulsed Radar in a way, that the geometrical offset between pulse transmission and echo reception is much greater in case of FMCW. Table 2.1 compares the geometrical distance for different velocities of pulsed and FMCW Radar. While a velocity of $1\ m/s$ results in an insignificant offset for both systems, already $10\ m/s$ generate a huge difference for FMCW Radar, which results in a loss of focusing quality. This effect increases/decreases with the ramp length and reveals why the start-stop-approximation has to be abolished for FMCW Radar.

Ribalta [56] presented a modified GBP that can correct these effect of defocussing, as it considers the movement of the platform during the transmission of a ramp. In [57] this correction was successfully examined with real sensor data. The modification of the GBP is graphically explained in Fig. 2.9. Each cell holds a reflection value that corresponds to a fixed signal runtime, which is defined by the sampling frequency of the reflected signal. Based on the signal runtime the GBP assigns a distance/position to every range cell. In the case of pulsed Radar, the signal runtime corresponds with the assumed distance/position due to the very short traveled distance during the very short pulse.

Table 2.1: Flight distance at different pulse-/ramp-length

flight velocity	signal length	moved flight distance
10 m/s	pulse : 1 μs FMCW : 10 ms	10 μm 10 cm
100 m/s	pulse : 1 μs FMCW : 10 ms	100 μm 1 m
1000 m/s	pulse : 1 μs FMCW : 10 ms	1000 mm 10 m

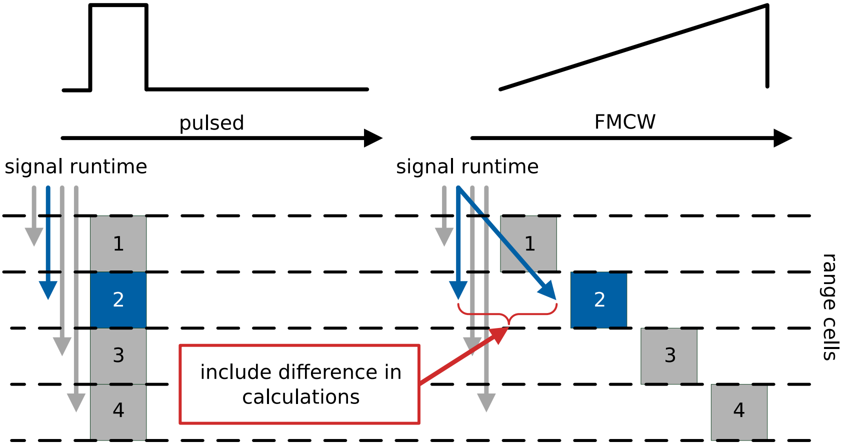


Figure 2.9: Correction of azimuth position for every range cell of a FMCW ramp [54]

For FMCW Radar, the signal runtime results in a mismatch of assumed position and actual position. This leads to a defocussing during the back projection of each ramp. The modification takes the time difference in the form of a delta in x direction into account. The quotient of this value and δr is calculated and multiplied with a constant, that includes several system parameters like ramp-length, velocity, wavelength, cutoff frequencies, etc. [56]. The sample distance (range) is corrected for each range cell individually to focus the correct range cell for an image pixel. Basically, every single range cell is adapted in azimuth position, as depicted in Fig. 2.9. With this correction added in the GBP, arbitrary ramp lengths and velocities are possible, while regular focusing quality is achieved.

3 Related work

Time-domain-based Synthetic Aperture Radar (SAR) data processing is a well-known principle due to the given advantages. However, the high complexity standard time-domain-based algorithms hampered the implementation for fast, compact and energy-efficient platforms. The ongoing technological improvements during the past years allowed to reconsider the obstacle of complexity. Besides dedicated Field Programmable Gate Array (FPGA) architectures, other types of non dedicated architectures such as Digital Signal Processors (DSPs), Graphic Processing Units (GPUs), multicore and manycore systems are available for implementation. The chosen scenario of real-time processing on board of Unmanned Aerial Vehicles (UAVs), imposes the requirements of high processing performance, standalone ability, compact design and low power dissipation to any utilized architecture. Although real-time processing of SAR data is not broadly covered in the scientific community, several publications that cover or partly cover this topic exist. Since the Global Backprojection (GBP) is an established algorithm, in particular in the medical field, the majority of these publications cover only the GBP. Due to the short time of existence compared to the GBP, the Fast Factorized Backprojection (FFBP) is much less discussed for real-time implementations. This is true for dedicated and non-dedicated architectures. Additionally, it is hard to compare the FFBP performance, since the possible permutations of configuration parameters span a wide design space. The performance of GBP implementations can be compared relatively simple by Pulse to Pixel Projections (PPPs) per time, as this value is not altered by configurations. Therefore, the given comparison of platforms base on GBP performance. To some extent this allows for the extrapolation of the FFBP performance as the GBP usually consumes the major part in FFBP processing time. The most potent candidate (that was researched) of each architecture is picked to represent the technology group in the following section. Parts of the presented related work were already discussed in [53, 58, 59, 60].

3.1 Non dedicated architectures

Due to the high flexibility and fast implementation, software solutions on General Purpose Processors (GPPs) are widely spread. A generalized software implementation usually lacks in performance and power efficiency. The software for GPPs can be transferred directly or after modifications to multi- and manycore systems. This allows to sustain flexibility and combine it with parallel processing. Multi- and manycore systems are partly a consequence of the limitations in clock frequency.

While packing density of transistors continues to increase, the simultaneous increase in clock frequency is slowing down due to the disproportion of speed and leakage current. Therefore, the available resources are distributed into multiple cores of the same type (homogeneous) or different types (heterogeneous) on the same processor die. This has also lead to the manycore class, which is not clearly separated from the multicore class, but usually consist of significantly more cores and is optimized to handle code executed in parallel. The cores in heterogeneous systems can be of any nature and can therefore also include cores of dedicated hardware accelerators, such as smaller GPU cores, DSPs or other types besides a standard GPP architecture. An example of a manycore architecture is presented by Park et al. in [43]. It considers the implementation of a GBP for a circular SAR scenario on a Xeon Phi processing cluster. The cluster consists of one Intel Xeon processor E5-2670 [61] with 8 cores at $3.30\ GHz$ and two Intel Xeon Phi 5110 coprocessors [62] with 60 cores at $1.05\ GHz$ each. Based on the Thermal Design Power (TDP) of the datasheets, the implementation consumes up to $605\ W$ and can process 35 Giga PPP per second. A multicore architecture is presented by Wielage et al. in [59]. The used platform is an ODROID-XU4 [63] multicore plaform with a heterogeneous Samsung Exynos 5 Octa 5422 System on Chip (SoC) [64], which consists of one Cortex-A15 [65] with 4 cores at $2.2\ GHz$, one Cortex-A7 [66] with 4 cores at $1.4\ GHz$ and one Mali-T628 GPU which is not used for processing. The implementation features a FFBP implementation, which also consists a GBP core, which is used for performance comparison. Based on the TDP of the datasheets, the implementation consumes up to $11\ W$ and can process 23 Mega PPP per second.

As the name already indicates, a DSP is a special type of processor, designed to handle typical operations, required during signal processing. Since signal processing often includes Single Instruction Multiple Data (SIMD) like processing, DSPs feature fast interfaces for fast data input and output. The incoming data can then be processed by efficiently implemented operations. A common operation required during signal processing tasks is a combination of a two operand multiplication, followed by accumulation with another operand, a so-called Multiply-Accumulate (MAC) operation. A MAC is, among other things, required for filter and Fast Fourier Transform (FFT) implementation. The GBP and FFBP algorithms are no exception to that. Although DSPs are usually used for less demanding tasks than backprojection they should be considered as a possible candidate since they are designed for highly efficient processing. An example for a DSP architecture for backprojection processing is presented by Zhao et al. [44]. Since DSPs are usually not very powerful in terms of massive parallel processing, a cluster was used for parallel execution. The used DSP core is a Texas Instruments TMS320C6678 [67], which consists of 8 TMS320C66x DSP [68] cores, each running with up to $1.25\ GHz$. Based on the TDP of the datasheet, the implementation consumes up to $10\ W$ and can process 9 Mega PPP per second.

In contrast to DSPs, GPUs are a comparatively young architecture type which underrun a very fast development. To speed up the processing of graphic imaging tasks, which have a strong SIMD character, GPUs contain hundreds or thousands of small cores for massively parallel processing. Since the typical environment are not constrained in energy consumption, GPUs tend to consume over $100\ W$ and more. Therefore, standard GPUs do not fulfill the low power constraints for the given scenario. A new embedded type of GPUs, specially developed for mobile processing, is a possible candidate for comparison. Nevertheless, a regular GPU architecture is compared as well. In 2011 Chapman et al. [45] presented an optimized GBP SAR implementation for a Tesla C2050 GPU, which consists of 14 streaming multiprocessors, each holding 32 streaming processors, which sums up to a total of 448 cores running at $575\ MHz$. Based on the TDP of the datasheet, the implementation consumes up to $238\ W$ and can process 6.2 Giga PPP per second. The more interesting candidate is the embedded GPU implementation that was presented by Wielage et al. in [60] on a nVidia Jetson TX2 GPU [69]. It is equipped with 1 Cortex-A57 [70] processor with 4 cores at $2\ GHz$, 1 nVidia Denver processor with 2 cores at $2GHz$, and 256 cores running with $1.3\ GHz$ max. The presented architecture also features a FFBP implementation, but only GBP results are used for comparison. Based on measurements, the implementation consumes up to $9\ W$ and can process 543 Mega PPP per second.

3.2 Dedicated architectures

Due to the real-time constraint in a mobile environment, FPGAs are chosen as target technology because of their power efficiency, compact size, and high potential of performance. The FPGA technology is explained in more detail in Section 4.1. Compared to non-dedicated architectures, the effort for a FPGA implementation is considerably high. FPGA designs are additionally interesting, as they offer the option to implement the architecture as an Application Specific Integrated Circuit (ASIC). This requires additional effort, in terms of time and money, but can increase performance by orders of magnitude, while enabling for ultra low power architectures. Although the principle of backprojection is also used in the medical field, for example in Computer Aided Tomography (CAT), FPGA implementations are very rare, most probably because of the high effort required for the design.

Schleuniger et. al. presented a GBP design for SAR processing in 2013 [47]. The used FPGA was a Xilinx Virtex 7. A multicore system with 64 Tinuso processors [46] was established. The whole Network on Chip (NoC) design consumed 60% of the hardware resources on the used FPGA. Since soft processor cores were implemented, the design is programmable and is therefore flexible for changes. The whole design consumed $10\ W$ and is capable of processing 46 Mega PPPs per second.

In 2010, Chapman et al. [48] presented theoretical design considerations for different Altera FPGAs, in order to process SAR data with the GBP algorithm. The theoretical design covered different topics, such as the central processing pipeline for back projection calculations and a cache architecture to minimize external memory access latency. The design on a simulated Altera Stratix IV EP4S100G4 FPGA consumes up to $15\ W$ based on values in the datasheet while processing 2.9 Giga PPPs per second. But it has to be considered that this is a theoretical design, wherefore stalls of the design, because of memory bus limitations, are not included.

In 2015 Pritsker [49], presented an architecture for SAR data processing with the GBP algorithm on an Altera Arria V FPGA. The architecture implements a pipelined systolic array structure and uses parallelism over image pixels and Radio Detection And Ranging (Radar) pulses. The whole design consumed $27\ W$ and is capable of processing 2.1 Giga PPPs per second.

Implementations on FPGAs addressing the FFBP algorithm are practically not existing. Hast et al. [71] presented an architecture which implements a Power-PC processor on a FPGA, but no comparable performance results were given, on top of that, it is a combined software/hardware structure.

3.3 Evaluation and discussion

Although the Intel Xeon manycore implementation shows high performance results, an architecture with a consumption of up to $605\ W$ is not applicable to the focused UAV scenario. With $11\ W$ in power dissipation, the heterogeneous Exynos multicore implementation provides values that are applicable to small UAV like platforms, but with only 23 Mega PPP per second, the processing speed is far from real-time for bigger images. Like the multicore implementation, the DSP consumes relatively little power with only $10\ W$, but with only 9 Mega PPP per second, it is too insufficient for real-time implementation. Similar to the manycore architecture, the standard GPU is highly suitable to process GBP SAR data, but the downside is the high power consumption of up to $238\ W$, which is not feasible for a mobile setup. The embedded GPU has a much lower power dissipation and can also provide a high amount of PPP per second. When compared to non-dedicated architectures, the dedicated architectures show significantly more processing performance, when normalized on the average power. The only exception to this is the embedded GPU, wherefore this is the only notable candidate besides the dedicated architectures. All presented FPGA implementations stay within a power consumption range that is feasible for mobile setups.

The presented implementations do not intend to be exhaustive in terms of finding the most potent candidates of a specific architecture. Nevertheless, it shows the basic trend for each architecture and why the implementation on FPGAs must be considered for real-time implementation of backprojection SAR, under the constraint of low power dissipation.

4 Conceptual hardware design

As already presented in Chapter 3, the processing of Synthetic Aperture Radar (SAR) data is not limited to a specific hardware architecture. In the case of less demanding constraints on performance and energy consumption, a non-dedicated General Purpose Processor (GPP) architecture is the best choice, as the development effort is reduced to a minimum, while full algorithmic flexibility is provided. In the case of more demanding constraints, for any of the mentioned aspects, the optimal architecture will depend on the characteristic of the constraints. As Unmanned Aerial Vehicles (UAVs) are the chosen target platform, constraints for low-energy consumption, small system dimensions and fast processing times are required to allow for efficient operation. Additionally, the system is dedicated to one specific task only, whereby the system with the highest customization potential is attractive. For the given reasons Field Programmable Gate Arrays (FPGAs) are the best option as they combine all the mentioned aspects. Considering the fact, that development time is a nonrecurring factor, the benefits of dedicated hardware implementation is worth the effort.

In this chapter the entire hardware design process and the considerations regarding the presented design decisions are discussed. This includes the explanation of general principles for efficient dedicated hardware implementations. All decisions are driven by the requirement to combine and fulfill all given constraints at the best possible rate. As processing time is the most critical aspect, this work is focused on the concept for efficient and exhaustive resource utilization of any given FPGA, to maximize data throughput rate at a minimum of latency.

4.1 Field Programmable Gate Arrays (FPGAs)

FPGAs belong to the group of re-configurable architectures. To be more specific, to the class of Programmable Logic Devices (PLDs). Compared to other members in the class of re-configurable architectures, they are the most flexible and allow for very fine grained adjustment, to tailor the hardware to the required signal processing function. This is possible through the direct implementation of logic functions in memory [72]. Basically a FPGA is a big mesh of Look Up Tables (LUTs), which are freely programmable. Depending on the FPGA type LUTs of different size are used. In general a k-input-LUT is compound of 2^k Static Random Access Memory (SRAM) cells, which can be addressed with a $2^k : 1$ multiplexer. While one k-LUT can realize any logic function with k inputs, many interconnected LUTs can implement logic functions of variable size. Additionally, LUTs can also be used to implement memory blocks. A smaller set of LUTs, together with flip-flops, a carry logic and different type of multiplexers form a slice. The connection of slices allows for a variety of functions to implement. In Xilinx FPGAs such block are called Configurable Logic Blocks (CLBs). The CLBs are connected to the routing fabric with configuration boxes (CB). The routing fabric interconnects all CLBs, Block Random Access Memorys (BRAMs) and Digital Signal Processors (DSPs) in a matrix across the entire FPGA. These interconnections are configurable via numerous switching matrices across the island style FPGA, whereby complex functions can be implemented.

Although this leaves a high degree of flexibility, a considerable overhead is created through this architecture. In fact, the interconnection network accounts for the bigger part of the consumed area in a FPGA. The basic structure of an FPGA is depicted in Fig. 4.1. To benefit from the very compact size of fixed interconnected logic, additional DSP blocks and BRAMs are interwoven with the CLB matrix. The DSPs perform multiplication in two's complement at variable bit-width up to a certain boundary. Multiplications with multiplicands wider then a DSP block need to be partitioned and distributed to numerous DSPs. A so-called BRAM primitive can be used as a small memory or to cascade multiple BRAMs to create bigger memory blocks. The input and output ports of a BRAM are usually highly configurable and therefore, they provide everything from a high memory width and low memory depths (words with one bit in depths) up to low memory width and high memory depths (maximum combined width of all ports of one interface type). Combined, this can reach up to input and output ports with several hundred bits in width. All mentioned elements build the core structure of a FPGA. Dependent on the FPGA type other blocks like smaller Reduced Instruction Set Computer (RISC) processors or bigger blocks of memory might be added to the structure. IO (input/output) blocks are distributed around the FPGA, to allow for massive input and output interactions with the outside world. How an FPGA is arranged, how LUTs, BRAMs, DSPs and all other resource elements are structured, and how many elements of each resource type are available, depends highly on the manufacturer and the specific FPGA family. Dependent on the signal processing task, the proper FPGA must be picked, in order to get the right ratio between LUTs, BRAMs and DSPs. Table 4.1, list a selection of Xilinx high end Virtex FPGAs as examples. Due to the wide variety, of how for examples a LUT is implemented, these numbers are not always 100 percent comparable. To be at least partly comparable, only one family of FPGAs from one manufacturer is listed.

Each of the listed FPGAs in Table 4.1 represent one Xilinx Virtex family. They were picked for better comparison. In terms of resources they are positioned at the lower end of the upper third of the FPGAs within the family. Between the FPGAs of one family big differences in the amount and distribution of resources exist. The lower end and upper end of the entire range have a more extreme configuration. Between the families, the gain in resources per mm^2 is significant. This is due to new or improved techniques and denser packaging by smaller fabrication processes. While LUTs increased with a factor of 9, BRAMs increased much more with a factor

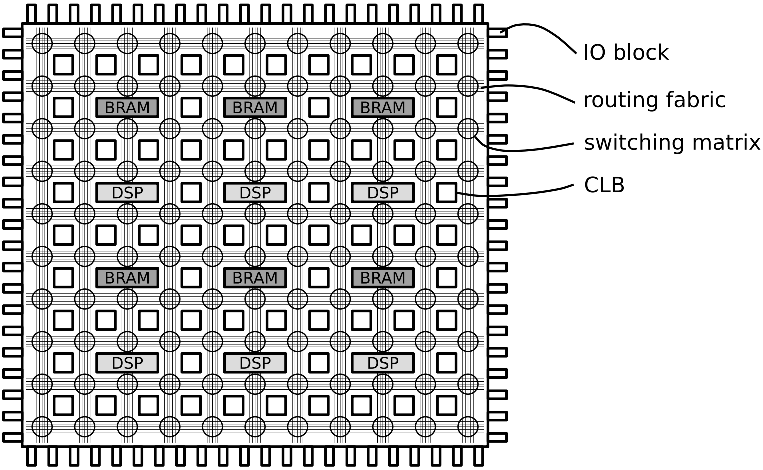


Figure 4.1: Basic structure of a generic island-style FPGA structure

of 32 and DSPs even increased by a factor of 54 across the picked FPGAs. This intense gain in resources allows for the implementation of complex signal processing tasks making FPGAs more attractive. Beside the plain amount of resources, the mapping of a signal processing task on the resources is equally important. Based on the mapping, resource utilization varies in efficiency and design clock varies in frequency. Two aspects influence the efficiency of the mapping process. One is the designer, as mapping can be supported by a sophisticated and well balanced architecture. The second is the used synthesis tool that maps a circuit, described in a Hardware Description Language (HDL), on the actual resources. The synthesis involves many steps, the first step is an automated global optimization of the logic

Table 4.1: Resource comparison of different Xilinx high end Virtex FPGAs

FPGA		year	process	DSP		LUT		BRAM	
family	type		$[nm]$	type	#	type	#	size $[Kb]$	#
Virtex5	XC5VLX110T	2006	65	25x18	64	6-LUT	69120	36	148
Virtex6	XC6VLX240T	2009	40	25x18	768	6-LUT	150720	36	416
Virtex7	XC7VX690T	2011	28	25x18	3600	6-LUT	433200	36	1470
Virtex US	XCVU125	2014	20	25x18	1200	6-LUT	716160	36	2520
Virtex US+	VU5P	2014	16	25x18	3474	6-LUT	601000	36	4668

functions and the mapping of these functions to FPGA specific libraries. Afterwards, the libraries are placed on the actual resource elements. Finally the routing of all elements is performed. All steps represent an optimization problem, which can also end in a local optimum. Therefore, each synthesis can turn out with different results.

Compared to implementations on other architectures, like GPPs or Graphic Processing Units (GPUs), the design for an FPGA is by far more time consuming. This is due to the high degree of freedom in the design process and that basic functions must be implemented manually. A faster development process for other architectures comes with the drawback of lower throughput rates. This can only be compensated through massive parallelization, which in turn creates a higher power dissipation. Nevertheless, recent architectures like embedded GPUs might be able to close this gap in the future [60]. The main argument for using FPGAs is the good trade-off between power dissipation and throughput, making it the perfect platform for UAV based SAR processing in real-time. Especially the regular structure of time-domain-based backprojection SAR algorithms match perfectly with the regular structure of island-style FPGAs. Furthermore FPGAs allow exploiting the possible parallelism in the algorithms. It has already been shown in [73], that frequency-domain-based SAR processing is possible in real-time on compact systems that do not exceed the power dissipation of $15\ W$. In case, a system with ultra-low power dissipation and even higher processing power is required, an Application Specific Integrated Circuit (ASIC) is the only option. Here, the simulated and emulated FPGA is of dual-use, as it is a necessary step towards an ASIC design, and as it already covers the digital design verification part during development.

4.2 Principles for dedicated hardware implementation

Architectures others then FPGAs, always base on unchangeable underlying hardware blocks or Processing Elements (PEs). For example, Compute Unified Device Architecture (CUDA) cores are used in GPUs, special hardware blocks in DSPs, or the Arithmetic Logic Unit (ALU) in a GPP. To max out the efficiency on any of these platforms for any given task, the task must be analyzed in detail, to understand the nature of the basic PEs. To optimize the mapping on a specific architecture, detailed knowledge of the architectural structure is indispensable. Besides the knowledge on the given PE dependencies, knowledge about connection limits and side effects is required.

To explain better which kind of problems arise during the processing of such algorithms, pseudo-code for the Global Backprojection (GBP) is listed in Algorithm 4.1. The pseudo-code shows the flexibility of the algorithm. The constrain parameters (i_{az},i_x,i_y) for the loops in line 1, 2 and 3 can be interchanged with each other (alongside with some minor changes in code order). If done correctly, any parameter order will result in the same image, since this only changes how data is accessed and in which sequence the pixel, of the final image, is generated. According to the order,

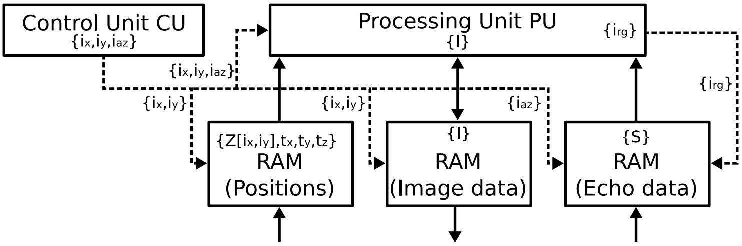


Figure 4.2: Block diagram of a generalized GBP processing platform

specific data access rates might be slow (outer loop), median (intermediate loop) or fast (inner loop), but every pixel can be processed independently from each other.

Independent of the used hardware for processing, the scheme can be simplified to highlight the critical aspects. This scheme is depicted in Fig. 4.2.

Three elements need to be taken care of for GBP processing. Loop control is handled in a control unit CU. Parameters for the actual raw data section i_{az} and the current image position i_x,i_y are required by the actual processing unit (PU) and the Random Access Memory (RAM) section. The GBP is data greedy, wherefore the data flow must be analyzed in depth to reduce (respectively optimize) the transfers between the RAM and the PU. Therefore, the following sections will cover this topic in detail. Another aspect is the actual processing inside the PU. The dependencies in the code are a natural limitation for processing speed. A pure sequential algorithm cannot be enhanced by parallel processing for example. Therefore, dependency analysis is required, to check on possible candidates for optimization. Besides this, the costly core functions need to be checked in advance, to analyze possible substitutions (approximations) for easier and faster processing. Therefore, the important steps are:

1. Analysis of the algorithm's mathematical model
2. Analysis of dependencies
3. Analysis of streaming capabilities

All this is not obsolete by the use of dedicated hardware platforms (FPGAs), as they also need to interface with the memory, as long as raw data cannot be stored completely inside of the FPGAs internal memory. But between all architectures, dedicated hardware offers the biggest degree of freedom in all design aspects. This means an algorithmic implementation does not have to be adapted to the underlying hardware architecture. But the hardware architecture can be adapted to a given algorithm. In the case of careful design, this will result in the most efficient implementation in terms of size and energy consumption.

Algorithm 4.1: GBP algorithm

```
input  : $pc_c$ = phase correction constant $(4\pi/\lambda)$
         $N_{az}$ = number of incoming sensor echos in azimuth
         $N_{rg}$ = number of incoming range samples per range line
         $\delta rg$ = distance between each range cell
         $r_{min}$ = minimum distance covered by sensor echo
         $r_{max}$ = maximum distance covered by sensor echo
         $t_x[N_{az}]$ = trajectory vector of the platform in $x$
         $t_y[N_{az}]$ = trajectory vector of the platform in $y$
         $t_z[N_{az}]$ = trajectory vector of the platform in $z$
         $S[N_{rg},N_{az}]$ = sensor data array
         $N_x$ = number of image pixel in $x$
         $N_y$ = number of image pixel in $y$
         $\delta x$ = distance between image pixel in $x$
         $\delta y$ = distance between image pixel in $y$
         $Z[N_x,N_x]$ = height of each image pixel on ground
         $I[N_y,N_y]$ = initial image array on ground (set to 0)
output : $I[N_x,N_y]$ = processed image
for ($i_{az}$ = 0 to $N_{az}$-1) do          /* iterate over echos in azimuth */
    for ($i_x$ = 0 to $N_x$-1) do            /* iterate over image pixel in $x$ */
        for ($i_y$ = 0 to $N_y$-1) do        /* iterate over image pixel in $y$ */
            $x = i_x \cdot \delta x$
            $y = i_y \cdot \delta y$
            $r_x = x - t_x[i_{az}]$
            $r_y = y - t_y[i_{az}]$
            $r_z = Z[i_x,i_y]$ - $t_z[i_{az}]$
            $r_{xs} = r_x \cdot r_x$
            $r_{ys} = r_y \cdot r_y$
            $r_{zs} = r_z \cdot r_z$
            $\delta r_s = r_{xs} + r_{ys} + r_{zs}$
            $\delta r = \sqrt{\delta r_s}$
            if ($r_{min} \leq \delta r \leq r_{max}$) then      /* if pixel in sensor range */
                $r_{range} = \delta r - r_{min}$
                $r_{cell} = r_{range} / \delta rg$
                $\phi_{corr} = pc_c \cdot \delta r$
                $r_{comp} = e^{j \cdot \phi_{corr}}$          /* then follows interpolation */
                $S_{int}[i_{az},r_{cell}] = int\{[i_{az},r_{cell}],S[i_{az},i_{rg}]\}$
                $S_{comp} = S_{int}[i_{az},r_{cell}] \cdot r_{comp}$
                $I[i_x,i_y] = I[i_x,i_y] + S_{comp}$
            end if
        end for
    end for
end for
```

While fewer restrictions need to be considered, the high degree of freedom implies a much bigger design space. Depending on the structure of the given algorithm, a full custom dedicated implementation, or softcore processors with dedicated accelerators might be the better choice. But the more flexibility is needed within the algorithm and the more dependencies during processing might occur, the more likely it would be to pick other architectures then FPGAs, as this is not a strong feature of FPGAs. For fast FPGAs architectures, the algorithms should allow for vast internal parallelism and data streaming. As described in Section 4.1, FPGAs contain LUTs, DSP slices and BRAMs. Every FPGA type comes with different ratios of these resources and it is up to the designer, to either chose an FPGA with the best fitting resource ratio or to adapt the design to a given FPGA, to max out the resources. To implement full dedicated hardware architectures, including the smaller dedicated accelerators PEs for time-domain SAR algorithms, the following sections will briefly explain the above-mentioned steps.

4.3 Analysis of the mathematical SAR model

Both time-domain-based algorithms and the underlying mathematical steps have been described in Chapter 2. Every mathematical operation is formed by the four basic arithmetical operations of addition, subtraction, multiplication, and division. Therefore, every architecture features hardware blocks to perform these basic operations. But more complex operations are not necessarily available, which might result in sub-optimal implementations. The description in Chapter 2 shows that processing SAR data in frequency and time-domain share the same mathematical operations:

1. Interpolation
2. Complex multiplication
3. Square root calculation
4. Sine and cosine calculation

A dedicated hardware block for all operations would increase performance drastically but would also consume more resources. To reduce the number of required resources, an analysis of every operation, in terms of required accuracy and/or less resource-demanding approximations, can be very useful. Depending on the type of the chosen FPGA, the entire design should be converted from floating point to fixed-point number format. While this conversion results in more efficient resource utilization, a loss in accuracy and dynamic range occurs. Therefore, in addition to the above-mentioned steps, it might be necessary to analyze area-efficient fixed-point approximations. Section 5.1 will provide a description of such approximations.

4.4 Analysis of processing dependencies

To understand the potential and limits of a given hardware architecture, it is not only important to determine the pure size of possible processing elements. Bottlenecks can generate anywhere, either due to the design of the architecture or due to conditions that generate from components which interface with the architecture. A classic bottleneck is the memory, which could be too small, too slow or too inflexible. This can call stalls for a subset or even all processing elements, reducing the average throughput rate of the whole architecture. Some bottlenecks can be overcome with different ways of implementations, a rearrangement of the processing order or a rearrangement of how data is stored. Only under optimal conditions, a hardware architecture can reach the full possible processing capabilities. A beneficial condition is the lack of dependencies, as any dependency can create conflicts which usually results in stalls.

Systems which perform only one command at a time sequentially can handle any dependency without conflict. Systems which work in parallel suffer from performance drops under such conditions due to data dependencies which result in pipeline hazards. Especially, FPGAs gain performance from massive parallel processing. Therefore, it is of special interest to analyze an algorithm, to find and resolve arising dependencies [74]. Standard compilers try to optimize the code to reduce such conflicts automatically with a dependence analysis with sufficient quality. For dedicated applications, this analysis needs to be done manually and on a deeper level, as operations can be implemented in different ways and order. According to the Bernstein condition [75], dependencies exist when process A cannot execute in parallel to B, because A effects values used by B or the other way around. Nor can A and B execute in parallel if any subsequent process C uses data effected by A and B, i.e. whose value might depend on the order of execution. To analyze such data dependencies is a standard technique to check on possible parallel data processing, to avoid pipeline hazards and to react correctly on hardware jumps. The dependencies split up into:

1. Data dependency
    a) Flow dependency (read after write)
    b) Anti dependency (write after read)
    c) Output dependency (write after write)
    d) Input dependency (read after read)
2. Control dependency
3. Loop dependency

Data dependencies relate to a change of the resulting value of one or more instructions, while control dependencies relate to the execution of one or more instruction based on the result of another instruction. A flow dependency (read after write) occurs, when an instruction depends on the result of a previous instruction (as shown in Algorithm 4.2). $INST_3$ depends on the update of value $INST_2$. The same relation exists between $INST_2$ and $INST_1$. All instructions are in a transitive relation, wherefore $INST_3$ also depends on $INST_1$.

Algorithm 4.2: Flow dependency

```
1 $INST_1 = x$
2 $INST_2 = INST_1$
3 $INST_3 = INST_2$
```

An anti dependency (write after read) occurs when a subsequent instruction writes a value, that was read in an earlier instruction, see Algorithm 4.3. $INST_2$ anti depends on the $INST_1$ instruction in line 3 as it is updated later. Therefore, the ordering of these instructions cannot be changed, as it would change the final value of $INST_2$. This is also the reason why the two instructions cannot be executed in parallel as race conditions could occur and might change the ordering of the instructions.

Algorithm 4.3: Anti dependency / output dependency

```
1 $INST_1 = x$
2 $INST_2 = INST_1 + y$
3 $INST_1 = z$
```

This is also called a name dependency as changing variable names could remove the dependency as shown in Algorithm 4.4. A copy of $INST_1$ is declared with the name of $INST_{2a}$. By renaming the anti dependency between line 2 and line 3 is removed and two instructions could be processed in parallel. But clearly, a new dependency was introduced between $INST_{2a}$ and $INST_1$ which forms a classical flow dependency.

Algorithm 4.4: Anti dependency rename

```
1 $INST_1 = x$
2 $INST_{2a} = INST_1$
3 $INST_2 = INST_{2a} + y$
4 $INST_1 = z$
```

An output dependency (write after write) occurs when instruction ordering will change the final output value, as it would be the case for Algorithm 4.3. An output dependency exists between the two instructions of line 1 and line 3, wherefore these instructions cannot be executed in parallel. This is again called a name dependency. Like in the example above, renaming can be used to remove the dependency as shown in Algorithm 4.5.

Algorithm 4.5: Output dependency rename

```
$INST_{1a} = x$
$INST_2 = INST_{1a} + y$
$INST_1 = z$
```

Another type of dependency is the input dependency. It can only occur during parallel processing when two instructions try to access the same resource at the same time as shown in Algorithm 4.6. A reordering of instructions is possible as this will not cause different results. This will only change how long one process has to wait for accessing this resource.

Algorithm 4.6: Input dependency

```
$INST_1 = x + 1$
$INST_2 = x + 2$
```

While data dependencies relate to the influence of instructions on values, control dependencies relate to the influence of a value on the execution of another instruction:

Algorithm 4.7: Control dependency

```
1 if ($INST_1 == INST_2$) then
2     $INST_1 = INST_1 + INST_2$
3 end if
4 $INST_2 = INST_1 + INST_2$
```

In Algorithm 4.7 the instruction in line 2 has a control dependency on the preceding instruction in line 1, as the outcome of $INST_1$ determines the execution. In terms of execution, the instruction in line 3 is independent of the outcome of line 1, still, the value of $INST_2$ might be influenced by the execution of line 2. In general, it can be stated, that any condition will generate control dependencies if the true/false body includes variables which are used in other parts of the algorithm. These dependencies must be analyzed thoroughly based on the control flow graph of an algorithm, as such dependencies might be forwarded through many decisions and variables, wherefore a clear identification might become a complex task. As already mentioned, in conventional sequential processing of an algorithm, control dependencies will not cause problems. But the implication of a control dependency for parallel processing can either hinder parallelism or, when executed without prior analysis, it can cause hazards in the form of wrong results.

A more complex dependency is formed when one of the above-mentioned conditions exits between two instructions of different loop iterations. This could either happen inside a loop segment, between two different loops or inside nested loops. For sequential code, loop dependencies will not occur as loop segments or different loops will not be processed in parallel. But since loops usually consume a large part of processing time, parallelism is vital. Therefore, an analysis is important, but also more complex than line based dependencies in sequential code since time-lines are not directly visible. Examples are given in Algorithms 4.8 to 4.10, where loops create dependencies over time. Dependencies would occur already if iterations would be equally split on two processors p, so that $p_1 \in (I_1 = 1, ..., I_1 = \frac{MAX}{2})$ and $p_2 \in (I_1 = \frac{MAX}{2} + 1, ..., I_1 = MAX)$.

Algorithm 4.8: Flow loop dependency

```
for $(i = 0; i < MAX; i++)$ do
    $A[i+1] = A[i]$
end for
```

In Algorithm 4.8, the value of $A[i]$ which was written in the prior iteration, is read and stored in $A[i+1]$, creating MAX flow dependencies between iteration i and $i+1$. After complete processing, every element up to position MAX has the same value as the element at position $i = 0$. Basically, the value in element $A[i = 0]$ is copied to the next element in every subsequently iteration. The loop runs could not be split on p_n processors, as the parallel processing would split the array A in p_n strings of different values.

Algorithm 4.9: Anti loop dependency

```
for $(i = 0; i < MAX; i++)$ do
    $A[i-1] = A[i]$
end for
```

In Algorithm 4.9, the value of $A[i]$, which was written in the prior iteration, is read and stored in $A[i-1]$, creating MAX anti dependencies between iteration i and $i-1$. After complete processing, every element up to position MAX is shifted by one position to the left or lower. The loop runs could not be split on p_n processors, as this would create p_{n-1} inconsistencies in the string at the transitions from $A[i] \in p_m$ to $A[i] \in p_{m+1}$.

In Algorithm 4.10, the value of $A[i]$ and $A[i+1]$ is written in one iteration, creating MAX output dependencies between iteration i and $i+1$ as the two following iterations would write to the same location. After full sequential processing, every element up to position MAX will contain the value of x. As with anti dependencies, the loop runs cannot be split up on p_n processors as parallel processing inconsistencies would occur within the string at the transitions. As already mentioned, input dependencies only occur during parallel processing when the same resources need to be accessed at

Algorithm 4.10: Output/input loop dependency

```
for ($i = 0; i < MAX; i++$) do
    $A[i] = x$
    $A[i+1] = y$
end for
```

the same time. In case Algorithm 4.10 would be split on p_n processors, it can happen that up to p_n read attempts to x or y will occur. It is up to the system periphery how and in which order these attempts are handled. In a worst-case scenario, this will lead to sequential processing, in a best-case scenario all accesses are valid at the same time. However, the system will handle such situations, it should not lead to inconsistencies when compared to the pure sequential run on one processor.

4.4.1 GBP dependencies

The above-described dependencies need to be identified in the algorithm, to either resolve or at least alleviate them as much as possible. While the basic principle of the GBP algorithm is described in Section 2.3.1, Algorithm 4.1 formulates the algorithm in an architecture-independent pseudo-code for analysis.

In Algorithm 4.1 the order is adapted to inline processing of slowly incoming sensor data. As soon as one complete echo is received, by the sensor (one i_{az} azimuth position), the energy of this echo is projected fast in the entire image I in the following loops. This means every pixel of the image receives incremental updates with every processed echo, so every outer loop run. In case the order is changed, this would either change across which direction of the image the energy is projected fast, or how fast sensor data is accessed. If, for example the i_{az} parameter is switched to the inner loop with the i_y parameter, data is projected fast across the x direction instead of the y direction. Also, the image is generated differently, as sensor data is now accessed in the inner loop. Instead of incremental updates for a pixel, one inner loop run results in a completely processed image pixel which will be not accessed for the entire remaining run of the algorithm. But this also requires every sensor echo to be received before the inner loop is run for the first time. Which means data cannot be processed while sensor data is incoming. Again, the final result and how often data is read or written to perform an instruction respectively, will not be changed by reordering, but the image is generated in a different sequence.

As already mentioned, every pixel can be processed independent, which could be useful for a full parallelization. In the case of Algorithm 4.1, this would mean to process every iteration in a single processing element. It seems promising, that the algorithm does not contain output loop dependencies and that every instruction in the inner loop accesses resources, which are only affected by the current loop run. But although it is not directly visible, flow loop dependencies exist. They are introduced by the outer loop as the two dimensional array $I[i_x,i_y]$ is accessed $N_{az} \cdot N_x \cdot N_y$ times. A full parallelization of all loops would, therefore, result in N_{az} parallel write accesses to every dimension of I. The problem of parallel write accesses can be resolved by increasing the array to three dimensions $I[i_x,i_y,i_{az}]$ as shown in Algorithm 4.11 (under the assumption of an unlimited memory bandwidth).

Algorithm 4.11: Flow loop dependency resolved

```
for ($i_{az}$ = 0 to $N_{az}$-1) do
    for ($i_x$ = 0 to $N_x$-1) do
        for ($i_y$ = 0 to $N_y$-1) do
            $I[i_x,i_y,i_{az}+1] = I[i_x,i_y,i_{az}] + S_{comp}$
        end for
    end for
end for
```

Still, the flow dependency would continue to exist, as the final value of a pixel has to be updated with every new processed azimuth echo, which could never be done in parallel. Even if this would be possible, one has to bear in mind that, based on the size of the dimensions, in reality, this could also result in a drastic increase of required memory. Therefore, N_{az} flow dependencies between iteration i_{az} and $i_{az}+1$ exist. Based on this assumption, the entire process could be distributed on $N_x \cdot N_y$ processors at maximum. But as soon as loop runs are processed in parallel, input loop dependencies will occur, either instantly or at a certain degree of parallelization, depending on the architecture. Only ideal architectures with unlimited resources could handle the vast amount of read and write accesses to the same memory location and the necessary bandwidth for operations in parallel. The rearrangement of the loops can change the characteristics or the number of dependencies. However, different measures like local memories, in forms of caches, for example, can reduce or resolve such conflicts faster.

Besides the parallelization across iterations and different loops, parallelization of the instructions within on iteration could be possible as well. As already mentioned before, the possible dependencies are basically structured identical. The dependencies are given in Table 4.2, where the notation $\{1,2\} \overset{x}{\mapsto} \{3,4,5\}$ means, that line 1 and 2 depend on line $3,4$ and 5 where $x \in \{f,a,o,i,fl,c\}$ and represents a flow, anti, output, input, loop-flow or control dependency. Based on Table 4.2, different subsets of instructions can be performed in parallel without creating any hazards. The subsets depend on each other and must be processed sequentially, which means a delay must be introduced until all instructions in the subset are completely processed. Depending on the instruction type in the subset, this can require multiple cycles. It is assumed, that the architecture can solve input dependencies without additional delays by the additional use of resources. Therefore, subsets are circumscribed by

Table 4.2: Dependencies of the GBP algorithm (Algorithm 4.1)

Flow dependency:	$\{6\} \overset{f}{\mapsto} \{4\}$; $\{7\} \overset{f}{\mapsto} \{5\}$; $\{8\} \overset{f}{\mapsto} \{4,5\}$
	$\{9\} \overset{f}{\mapsto} \{6\}$; $\{10\} \overset{f}{\mapsto} \{7\}$; $\{11\} \overset{f}{\mapsto} \{8\}$
	$\{12\} \overset{f}{\mapsto} \{9,10,11\}$
	$\{13\} \overset{f}{\mapsto} \{12\}$
	$\{14,15,17\} \overset{f}{\mapsto} \{13\}$; $\{14\} \overset{f}{\mapsto} \{13\}$
	$\{15,17\} \overset{f}{\mapsto} \{14\}$
	$\{16\} \overset{f}{\mapsto} \{15\}$; $\{18\} \overset{f}{\mapsto} \{17\}$
	$\{19\} \overset{f}{\mapsto} \{18\}$
	$\{20\} \overset{f}{\mapsto} \{19\}$
	$\{21\} \overset{f}{\mapsto} \{20\}$
Loop flow dependency:	$\{21\} \overset{fl}{\longmapsto} \{21\}$
Anti dependency:	$\{21\} \overset{a}{\mapsto} \{21\}$
Input dependency:	$\{7,8\} \overset{i}{\mapsto} \{6\}$; $\{6,8\} \overset{i}{\mapsto} \{7\}$; $\{6,7\} \overset{i}{\mapsto} \{8\}$
	$\{14\} \overset{i}{\mapsto} \{15\}$; $\{15\} \overset{i}{\mapsto} \{14\}$
Control dependency:	$\{15,16,17,18,19,20,21\} \overset{c}{\mapsto} \{14\}$

flow dependencies mainly. The first subset is formed by line 4 and 5 and can be processed in parallel, as they do not depend on any inner loop instruction. The second subset formed by line 6,7 and 8 as they all depend on line 4 and/or 5 but not on each other. The third subset is consists of line 9,10 and 11. Line 12 and line 13 form a subset each. Line 14, 15 and 17 all depend on line 13 and could be processed in parallel as a closed subset if line 15,16,17,18,19,20 and 21 would not be depending on line 14 in form of a control dependency. Therefore line 14 is the only instruction directly depending on line 13. While line 15 and 17 form a subset depending on line 14. Line 16 and 18 depend on the subset of line 15 and 17. All remaining lines 19,20 and 21 are single subsets which all depend on their preceding instruction. Instruction 21 could also be reformulated in a $a = b + c$; followed by $b = a$ statement. As this is a write after read behavior, this forms an anti dependency. Based on this, the whole inner loop can be split into eleven consecutive processing stages (flow dependencies), which would then define the maximum gain through parallelization. This is just a theoretical consideration, as it is not clear how an architecture could perform different instructions. While standard operations like addition, subtraction and multiplication are easy to define in terms of delays, other more costly instructions can be implemented in different ways. This is especially true for the instructions of line 13, 18 and 19 as they represent a square root, the transformation from real to complex via Euler's formula, an interpolation and a complex multiplication. These instructions need to be analyzed in more detail for an efficient implementation. This is done in Section 5.1.

4.4.2 FFBP dependencies

The Fast Factorized Backprojection (FFBP) algorithm is a prestage for the GBP, to reduce processing complexity, by reducing the original raw input data for one image. The original raw data is reduced by merging multiple adjacent apertures to form new apertures. The new reduced set of apertures is the new raw input data for the GBP. The merging process can be split into two steps, first calculating the position of a new aperture, and second the rectification and summation of multiple apertures on the new aperture position. The process introduces geometrical errors which are stemmed by focusing each new raw data set on smaller image regions. Therefore, many sets are required to form the final image without a loss of quality. Therefore, the total amount of raw data to process the entire image is increased. Nevertheless, the initial statement is still valid, as the input data for each subimage is reduced and therefore the complexity is reduced for a smaller image, and therefore in total. The concept is to reduce overall complexity, by combining many small problems with reduced complexity, instead of having one problem with high complexity. The process to form the small sets of raw data can be split into stages, always generating new raw data out of the previously created raw data. The final sets form the final image after GBP processing. During this process, a trade-off between the number of merged apertures and the amount of formed subimages is possible. This trade-off condition and the impact on image quality and complexity is discussed further in Section 5.4.1.

The basic principle of the FFBP algorithm is described in more detail in Section 2.3.2. The three FFBP steps of aperture position calculation, subimage center calculation and aperture merging are formulated in architecture-independent pseudo-code in Algorithm 4.12, Algorithm 4.13 and Algorithm 4.14. Like for the GBP, the above-described dependencies need to be identified in the code, to either resolve or at least alleviate them as much as possible. The algorithms are always valid for a specific stage and subimage. Since processing can be implemented as a tree with multiple stages and subimages, every stage and the included subimages will rely on the subapertures and their positions, as well as on the subimage positions from the previous stage and subimage. This forms a relation from the parent node to the child node, which is not expressed in the presented algorithms and therefore, each subimage treats incoming data as original data. In the case of multiple stages and subimages, the algorithms need to be glued by additional parameters like stage and subimage counters. To analyze the dependencies a stand-alone projection of every step is sufficient.

In Algorithm 4.12 the center positions of all subimages in stage s are calculated. This is a necessary step before the actual factorization of apertures, as an entire set of new subapertures for one subimage needs to be rectified to the center of that subimage. The amount of subimages per stage is set by f_{sub}, while the center positions of the subimages are equally spread across the entire range of the image of the preceding stage. The loop iterations can be processed in parallel as they do not depend on each other. This means the order of processing does not matter, as only iteration internal dependencies exist. Full parallelization is therefore possible if needed, but would inflict parallel read access on the memory for some constants.

Algorithm 4.12: FFBP subimage center calculation per stage

```
input  : $N_x$ = number of image pixel in $x$
         $N_y$ = number of image pixel in $y$
         $f_{sub}$ = number of sub images for this stage
         $IP[N_x,N_y]$ = image pixel positions on ground
output : $I_{subC}[f_{sub}]$ = 3D subimages center vector of this stage (z=0)
1  $IX_{absD} = IP[N_x,0]$ - $IP[0,0]$                /* width of $I$ in $x$ */
2  $IX_{subC} = IX_{absD}/2$                    /* center of subimage in $x$ */
3  $IY_{absD} = IP[0,N_y]$ - $IP[0,0]$                /* width of $I$ in $y$ */
4  $IY_{subD} = IY_{absD}/f_{sub}$              /* width of one subimage in $y$ */
5  $IY_{subR} = IY_{subD}/2$               /* half width of one subimage in $y$ */
6  for ($i_{sub} = 0$ to $f_{sub}$-1) do               /* iterate over subimages */
7      $IY_{subR} = IY_{subD} \cdot i_{sub}$           /* startpostition for subimage */
8      $IY_{subC} = IY_{subR} + IY_{subR}$        /* subimage $y$ position increase */
9      $I_{subC}[f_{sub}] = [IX_{subC};IY_{subC}]$   /* center position for subimage */
10 end for
```

Besides the parallelization of iterations, it is possible to execute instructions of one iteration in parallel. This depends on the dependencies which are given in Table 4.3.

Based on Table 4.3, different subsets of instructions can be performed in parallel without creating any hazards. It is assumed that the architecture can solve input dependencies without additional delays by the additional use of resources. The first subset, formed by line 1 and 3, can be processed in parallel, as both instructions do not depend on any other instruction. The second subset is formed by line 2 and 4 as both depend on line 1 or 3 but not on each other. All remaining lines (5,7,8 and 9) are single subsets, which all depend on their preceding instruction. Based on this, the instructions outside the loop can be split into a pipeline of three stages. But as these instructions are only performed once, this is not performance relevant. It might only be of interest in case of processing all subimage centers of all stages at once in a loop construct. The whole inner loop cannot be parallelized but must be split into three stages as well.

In Algorithm 4.13 the center positions of the new subapertures for one subimage of stage s are calculated. Besides calculating the subimage center, this is the central step necessary for factorization of apertures. The amount of merged apertures is set

Table 4.3: Dependencies of FFBP subimage center algorithm (Algorithm 4.12)

Flow dependency:	$\{2\} \overset{f}{\mapsto} \{1\}$; $\{4\} \overset{f}{\mapsto} \{3\}$
	$\{5\} \overset{f}{\mapsto} \{4\}$
	$\{7\} \overset{f}{\mapsto} \{5\}$
	$\{8\} \overset{f}{\mapsto} \{5,7\}$
	$\{9\} \overset{f}{\mapsto} \{2,8\}$
Input dependency:	$\{3\} \overset{i}{\mapsto} \{1\}$; $\{1\} \overset{i}{\mapsto} \{3\}$

by f_{apt}, while the position of the new aperture is always the geometrical center of all merged aperture positions. The entire set of new apertures will be focused on the line between the center of a subimage and the new aperture center. The inner loop is a summation over all loop iterations and contains flow loop dependencies, wherefore a parallelization is not possible. The flow loop dependency could be resolved like in Algorithm 4.11, but with the same consequences. Wherefore full parallelization is only possible for the outer loop runs, as they are independent of each other. Still, this would inflict parallel read access on memory for some constants.

Algorithm 4.13: FFBP aperture position calculation per stage and subimage

```
input  : $N_{az}$ = number of incoming sensor echos in azimuth
         $f_{apt}$ = aperture factorization count for this stage
         $T[N_{az}]$= 3D aperture trajectory vector of this stage/subimage
output : $T'[N_{az}/f_{apt}]$ = 3D aperture trajectory vector of next stage/subimage
for ($i_{apt} = 0$ to $N_{az}/f_{apt}$-1) do                  /* all aperture merges */
    $T'[i_{apt}] = 0$
    for ($i_m = i_{apt} \cdot f_{apt}$ to $i_{apt} \cdot (f_{apt}$+1)-1) do       /* single merge */
        $T'[i_{apt}] = T'[i_{apt}] + T[i_m]$
    end for
    $T'[i_{apt}] = T'[i_{apt}] / f_{apt}$
end for
```

Besides the parallelization across iterations, parallelization of the instructions within one iteration could be possible as well. This depends on the dependencies which are listed in Table 4.4.

Based on Table 4.4, no proper subsets can be combined, as everything depends on each other in terms of output/anti/flow and flow loop dependencies. This means no parallelization besides for the outer loop level is possible.

Table 4.4: Dependencies of FFBP aperture position algorithm (Algorithm 4.13)

Flow dependency:	$\{4\} \overset{f}{\mapsto} \{2\}$
	$\{6\} \overset{f}{\mapsto} \{4\}$
Loop flow dependency:	$\{4\} \overset{fl}{\longmapsto} \{4\}$
Output dependency:	$\{4\} \overset{o}{\mapsto} \{2\}$; $\{6\} \overset{o}{\mapsto} \{4\}$
Anti dependency:	$\{4\} \overset{a}{\mapsto} \{4\}$; $\{6\} \overset{a}{\mapsto} \{6\}$

The most demanding step during FFBP processing is listed in Algorithm 4.14. It describes the process of merging f_{apt} neighboring apertures, across the entire set of N_{az} apertures of the preceding stage $s-1$. In the case of the first stage, $s-1$ refers to the original data. Merging can start as soon as the subimage center and the subaperture positions are calculated for this subimage and stage. The algorithm describes this process for one subimage out of f_{sub} subimages for the stage s. In a simple loop setup, the apertures are merged along a stage and subimage. Meaning, that all apertures, which belong to this subimage and stage, are processed subsequently (see Algorithm 4.14). Every range sample rg of a subaperture is processed with the input of all contributing apertures f_{apt}. Due to the order of processing, the geometrical constants for one subaperture need to be processed many times, or they need to be stored. But a range sample rg can be stored internally.

The opposite of this order of processing would be the processing across stages s. Like this, merging could start as soon as enough apertures are recorded (raw data) to merge them to one final new subaperture in the highest stage. To form a subaperture of the final stage, several raw data apertures is required that results from the product of the f_{apt} factors of all stages s. In the process of merging a subaperture of the final stage, the subapertures of all stages before, which act as parent nodes for this subaperture, are formed as well. The GBP can then project the subaperture of the final stage across the final subimage. This order of processing requires either multiple calculations of certain parameters for a subimage run or more memory to hold the already calculated parameters. Additionally, the GBP processing would induce memory access more often, as the image pixel would need to be reloaded more often (see Section 4.5.1.1 for more information on GBP processing strategies).

Another approach is to perform range rg processing in the inner loop. Distances between the preceding aperture and the subaperture need to be processed only once, but the entire subaperture line must be reloaded f_{apt} times from memory.

All approaches show that flexibility in processing order is given, although not to the extent of the GBP, where every pixel is completely independent. Nevertheless, this also shows, that processing in parallel is possible since data dependencies between subimages within one stage are not existing. Another option is the parallel processing

Algorithm 4.14: FFBP aperture merging per stage and subimage

```
input  : s = current stage number
         $pc_c$ = phase correction constant $(4\pi/\lambda)$
         $N_{az}$ = number of incoming sensor echos in azimuth
         $N_{rg}$ = number of incoming range samples per range line
         $r_{min}$ = minimum distance covered by sensor echo
         $r_{max}$ = maximum distance covered by sensor echo
         $f_{apt}$ = apertures factorization per stage
         $f_{sub}$ = number of subimages per stage
         $\delta rg$ = distance between each range cell
         $T[N_{az}]$ = 3D aperture trajectory vector of last stage/subimage
         $I_{subC}[f_{sub}]$ = 3D subimages center vector of this stage (z=0)
         $S[N_{az},N_{rg}]$ = sensor data array
         $T'[N_{az}/f_{apt}]$ = 3D aperture trajectory vector of this stage/subimage
output : $S'[N_{az}/f_{apt},N_{rg}]$ = new apertures for this subimage and stage

for ($i_{apt} = 0$ to $N_{az}/\ f_{apt}\ -1$) do                    /* all aperture merges */
    for ($i_r = 0$ to $N_{rg}-1$) do                               /* cross range samples */
        for ($i_m = i_{apt} \cdot f_{apt}$ to $f_{apt} \cdot (i_{apt}+1)-1$) do      /* single merge */
            $d_{cn} = I_{subC}[s] - T'[i_{apt}]$
            $d_{on} = T[i_m] - T'[i_{apt}]$
            $scа_p = d_{on} * d_{cn}$                                     /* scalar product */
            $sca_n = ||d_{on}|| \cdot ||\ d_{cn}||$                       /* scalar product norm */
            $\cos\ (d_{on} \measuredangle d_{cn}) = sca_p/sca_n$
            $r_{dist} = i_r \cdot \delta rg$
            $d_{ns} = r_{dist} + r_{min}$
            $c_1 = ||d_{on}|| \cdot ||d_{on}||$
            $c_2 = d_{ns} \cdot d_{ns}$
            $c_3 = 2 \cdot \cos(d_{on} \measuredangle d_{cn}) \cdot ||d_{on}|| \cdot d_{ns}$
            $rd_{cos} = c_1 + c_2 - c_3$
            $rd = \sqrt{rd_{cos}}$
            if ($r_{min} \leq rd \leq r_{max}$) then                   /* within line boundaries */
                $r_{range} = rd - r_{min}$
                $r_{cell} = r_{range}\ /\ \delta rg$
                $\delta d = rd - d_{ns}$
                $\phi_{corr} = pc_c \cdot \delta d$
                $r_{comp} = e^{j \cdot \phi_{corr}}$                    /* then follows interpolation */
                $S_{int}[i_m,r_{cell}] = int\{[i_m,r_{cell}],S[i_m,i_r]\}$
                $S_{comp} = S_{int}[i_m,r_{cell}] \cdot r_{comp}$
                $S'[i_{apt},i_r] = S'[i_{apt},i_r] + S_{comp}$
            end if
        end for
    end for
end for
```

Table 4.5: Dependencies of FFBP aperture merge algorithm (Algorithm 4.14)

Flow dependency:	$\{11\} \overset{f}{\mapsto} \{5\}$; $\{10\} \overset{f}{\mapsto} \{9\}$
	$\{6,7\} \overset{f}{\mapsto} \{4,5\}$
	$\{8\} \overset{f}{\mapsto} \{6,7\}$; $\{12\} \overset{f}{\mapsto} \{10\}$
	$\{13\} \overset{f}{\mapsto} \{5,8,10\}$
	$\{14\} \overset{f}{\mapsto} \{11,12,13\}$
	$\{15\} \overset{f}{\mapsto} \{14\}$
	$\{16,17,19\} \overset{f}{\mapsto} \{15\}$; $\{16\} \overset{f}{\mapsto} \{15\}$
	$\{17,19\} \overset{f}{\mapsto} \{16\}$
	$\{18\} \overset{f}{\mapsto} \{17\}$; $\{20\} \overset{f}{\mapsto} \{19\}$
	$\{21\} \overset{f}{\mapsto} \{20\}$; $\{22\} \overset{f}{\mapsto} \{18\}$
	$\{23\} \overset{f}{\mapsto} \{21,22\}$
	$\{24\} \overset{f}{\mapsto} \{23\}$
Loop flow dependency:	$\{24\} \overset{fl}{\longmapsto} \{24\}$
Anti dependency:	$\{24\} \overset{a}{\mapsto} \{24\}$
Input dependency:	$\{4\} \overset{i}{\mapsto} \{5\}$; $\{5\} \overset{i}{\mapsto} \{4\}$
	$\{6,7\} \overset{i}{\mapsto} \{13\}$; $\{6,13\} \overset{i}{\mapsto} \{6\}$; $\{7,13\} \overset{i}{\mapsto} \{7\}$
	$\{12\} \overset{i}{\mapsto} \{13\}$; $\{13\} \overset{i}{\mapsto} \{12\}$
	$\{17\} \overset{i}{\mapsto} \{19\}$; $\{19\} \overset{i}{\mapsto} \{17\}$
Control dependency:	$\{17,18,19,20,21,22,23,24\} \overset{c}{\mapsto} \{16\}$

on aperture merging. For the given loop order, the inner loop contains a summation over all outer loop iterations i_{apt} in line 24. This forms an indirect flow loop dependency, wherefore running inner loops in parallel is not possible. One could argue that this can be solved by rearranging the loop hierarchy, but however the loops are ordered, a indirect flow loop dependency will always exist due to line 24 or 22. The flow loop dependency could be resolved like in Algorithm 4.11, but with the same consequences. Wherefore full parallelization is only possible for the outer loop runs as they are independent from each other. Still this would inflict parallel read access on memory for some constants. Beside the parallelization along stages (or merges), parallelization of the instructions within one iteration could be possible as well. The possible dependencies are basically structured identically and are given in Table 4.5 based on the structure in Algorithm 4.14.

Based on Table 4.5, subsets for loop internal pipelining can be found. The instructions within a subset can be performed in parallel without creating any hazards.

Subsets, in general, depend on each other and must be processed sequentially, which means a delay must be introduced until all instructions in the subset are completely processed. Depending on the instruction type in the subset, this can take multiple cycles. It is assumed that the architecture can solve input dependencies without additional delays by the additional use of resources. Therefore, subsets are circumscribed by flow dependencies mainly. The first subset is formed by line 4,5 and 9 and can be processed in parallel, as they do not depend on any inner loop instruction. The second subset is formed by line 10 and 11, as they depend on line 5 and/or 9 but not on each other. The third subset consists of line 6 and 7 since they depend on line 4 and 5 each. Line 8 and 12 form the fourth subset by depending on line 6 and 7 or line 10. Line 13, 14 and 15 form the fifth, sixth and sevenths subset each. Line 16,17 and 19 all depend on line 15 and could be processed in parallel as a closed subset if line 17,18,19,20,21,22,23 and 24 would not be depending on line 15 in the form of a control dependency. Therefore, line 16 is the only instruction directly depending on line 15 forming the eights subset. While line 17 and 19 form a ninths subset depending on line 16. Line 18 and 20 depend on line 17 and 19 forming subset number ten. Line 21 and 22 form subset number eleven as they depend on line 18 or 20. The remaining lines 23 and 24 are single subsets number twelve and thirteen which both depend on their preceding instruction. Instruction 24 could also be reformulated to $a = b + c$; followed by $b = a$ statement. As this is a write after read behavior, this forms an anti dependency.

Based on this, the whole inner loop can be split into thirteen consecutive processing stages (flow dependencies), which would then define the maximum gain through parallelization. This is just a theoretical consideration, as it is not clear how an architecture could perform different instructions. While standard operations like addition, subtraction and multiplication are easy to define in terms of delays, other (more costly) instructions can be implemented in different ways. This is especially true for the instructions of line 4,5,6,7,11,15,21 and 22 which need to be analyzed in more detail for an efficient implementation. This is done in Section 5.1.

4.4.3 Conclusion of dependency analysis

The analysis of the GBP and FFBP algorithms show the potential of possible parallelization in dedicated hardware. Depending on the resources of a given FPGA and the interfacing periphery, all of the described subsets of instructions could be performed in parallel, which allows for an increase of throughput rate. In general, it can be stated that, according to the Bernstein condition, an instruction A can be executed in parallel to instruction B when no flow, output, anti or control dependency exits between them. Even if a dependency exists, a parallel execution might be possible when the instructions are in different time slots of a pipeline. But this has to be manually checked. Based on the results of such a manual analysis, the architecture is designed to max out the possible performance. Since the restrictions and capabilities of a given system (FPGA and the interfacing periphery) are not known in advance, the design needs to allow for a dynamic adaptation of parallelization degree and interfacing speed.

4.5 Analysis of streaming capabilities

An architecture can be defined as a streaming architecture when a constant flow of incoming and outgoing data exist. Although this is the general definition of a stream, this does not state anything about how often or how long some PEs wait (stall) for the result of another PE. The overall goal of a streaming architecture should be to create a stream of data with a minimum amount of stalls in any PE for any type of parallelism. The dependency analysis in the preceding section examines the logical boundaries to exploit parallelism. Depending on the given architecture such boundaries can only be reached to a certain extent, as memory bandwidth, processing power or other constraints will generate practical limitations. These limitations will also influence, if parallelization is implemented more as an intra loop (parallelization of instructions inside a loop iteration), or inter loop (parallelization of loop iterations) concept. This indicates that the degree of inter loop parallelization is not the only measured value, as parallel PEs might still be stalled for longer time intervals in case the design is not optimal. For an optimal design, which is capable of well-balanced streaming, three aspects in the following order need to be considered.

1. How to setup memory access patterns (loop ordering)
2. How to balance multiple parallel read/write operations (inter loop streaming)
3. How to segment loop internal pipelining (intra loop streaming)

The following section will explain how the mentioned mechanisms can be used, to implement a design with a maximum in parallelization, that works similar to vector or array processors in a streaming manner.

4.5.1 Memory access patterns (loop ordering)

The presented backprojection pseudo-codes (Algorithm 4.1; Algorithm 4.12 - Algorithm 4.14) show, that the algorithms demand plenty of data to process one final image pixel. The same data words are required (load, store) often to process the entire image since a cell of one echo line contributes to many image pixel. This results in a vast amount of data accesses and identifies the memory interface as a bottleneck for any chosen architecture. Especially, because SAR data sets, and the resulting images are usually in the range of many million pixels. Since high throughput is a key requirement of the backprojection algorithms, it is fundamental to blueprint the memory architecture, before considerations for intra or inter loop streaming are finalized. The big sets of raw data and image data exclude small and fast local memories (caches or buffers). Bigger and slower memories (RAM) are the only way to store the entire raw data and image data during the process. The technical features of such memories constrain the choices of how the architecture should be designed for fast processing. Read cycles are usually faster than write cycles for example. The design should mainly focus on the data access patterns of the GBP, as it is usually more data demanding than the FFBP and therefore could slow down the processing speed if not considered appropriately. The data access patterns in the GBP can be modified by switching the parameters in the three nested loops (loop ordering). It is important to isolate the permutation of parameters which matches best with the features of the given memory architecture. Nevertheless, slower memories might result in a memory bottleneck as the GBP is a data greedy algorithm. To close, or at least reduce bottlenecks, local memories can still be used to reduce the overall memory transfers to/from the RAM, how it is done with regular caches in GPPs for example. In connection with local memories and the concept of parallelization (inter or intra) data broadcast might be used as well, to reduce the overall bus transfers. Basically, all methods have to be considered in combination for a sophisticated design, as one aspect might benefit another.

4.5.1.1 General considerations for RAM access

As already mentioned, one aspect which has to be considered, is the order of loops, since this also affects the loop internal parallelization or the number of input dependencies. As presented in Algorithm 4.1, the raw echo data is read in the outer loop, which would mean a new line of data is received and projected on the image. The projection can be chosen (depending on the loop order) to be done fast in azimuth (y) direction, and slow in range (x) direction, or the other way around. The order of access pattern will not alter the result, but from the architectural point of view, this order can have a big effect on the design and throughput rate. This can be understood when typical RAM attributes are considered. RAM is required for storing the entire raw data and image data. Although typical RAM interfaces work in a way that a wide data word can be accessed at any location in any order, this does not imply that any access pattern will result in the same latency. Usually, the addressing scheme of RAM is a consecutive number, but internally these numbers translate in a matrix memory with lines (combined to pages) and columns. Also, the bigger the memory, the more likely it will be organized in physically separated memory banks which still appear as one big memory. Although RAM provides fast and broadband interfaces, data rates depend heavily on the linearity of memory access and might drop in any of the following cases:

1. Reading not in burst mode (consecutive read in one page)
2. An address jump across pages
3. A switch from one bank to another
4. Address conflicts in case of parallel access

The physical characteristics of a RAM define the timing parameters of the memory and describe the required time for any memory access pattern. Random access patterns should be avoided as they will inflict high timing penalties more often than regular access patterns. Also, a regular pattern might allow arranging access in a way that the predictable timing penalties are masked behind other memory operations. Especially, streaming systems can benefit from this.

For reading data this automatically implies that reading in one page and also reading pages, in a consecutive manner, is fast, while reading consecutive columns is the slowest access as address jumps across pages and banks are more likely to happen. As bigger images will be accessed through different pages and columns in the memory, this shows that reading fast in x (page direction) or reading fast in y (column direction) can have a big impact on throughput. Of course, an image can also be transposed if the loop order would require this, but this is only possible if the mathematical operations match with this pattern. While it might be possible to store the image transposed to read and write fast in y direction, mathematical operations might need access to neighboring data in x direction. The interpolation in the GBP algorithm is an example of this. In this case, raw data and image data could be stored in different directions, as all mathematical operations (besides the final accumulation for one image pixel) only require raw data. Nevertheless, this explains why such implications need to be considered for fast design.

Writing data to the RAM usually underlies even stronger restrictions. For example, in a GPP with a four-way associative cache, a read operation will perform faster than a write operation. To ensure fast data fetching into the cache, parallel read access to four RAM memory banks are performed since it is not determined where the required data word is stored. Although this includes several useless read operations, this access will not alter the data. Writing data from the cache to RAM is slower because four parallel write operations would alter data words that should not be altered. Therefore, the chip must wait until the memory bank address is determined, which might take several clock cycles. For writing data in a streaming system this is not necessarily true, as buffers do not have to be so complex. In case of data dependencies are not given, addresses can be determined without bigger effort as it follows a straight pattern. The write operation can then be pipelined. Nevertheless, buffers and the patterns of access need to be understood and planned in advance.

Considering that a RAM is required, the architecture should be designed to match general RAM attributes. Since many PEs should act in parallel, the question arises, if one PE should process a full image, a full image line or a pixel alone, or if any of the options should be divided between many PEs. All strategies bare pros and cons which show when the loop order is discussed.

4.5.1.2 Considerations for GBP RAM access optimization

As already discussed, the GBP treats every image pixel independent of all other image pixels. Therefore, the loop hierarchy is free of choice. Changes in the loop hierarchy and the choice of the right memories provide a lot of alternatives. For a simple and fast estimate of investigated alternatives, simplified image and echo parameters are used. These dimensions are, raw data of $N_{az} = 1000$ lines in azimuth, each with $N_{rg} = 1000$ echo samples in range, and image dimensions of $N_x = 1000$ and $N_y = 1000$ pixel. This set is treated as one synthetic aperture, meaning that every image pixel is illuminated by every aperture (pulse) of the raw data set (N_{az} raw apertures). A single image pixel would thereby require 1000 projections to be finished, one from every raw aperture in azimuth. An important parameter of the GBP is the number of projections for the entire image given by Eq. (4.1).

$$N_{pro} = N_x \cdot N_y \cdot N_{az} \tag{4.1}$$

The number of projections is proportional to the image dimension and the number of echo positions in azimuth. For the here used simplified parameters, 10^9 projections or 1G (Giga) projections. For each projection, the core of the loops of Algorithm 4.1 need to be processed. Therefore RAM data needs to be read and written. For FPGA implementations, parameters in the range between 256 and 8192 will be investigated. Therefore an exemplary size of $N_{az} = 1000$ is in the mid number range. Since these numbers are comparatively small raw data and image dimensions, it is obvious that the GBP core operation should be processed as efficiently as possible. Therefore, memory access patterns need to be optimized as well.

To do so, the permutation of loop parameters need to be checked. This results to six different configurations. These configurations are depicted in Fig. 4.3, while the corresponding memory access rates are given in Table 4.6. The configurations differ in running order $[1,2,3]$, where the first, second and third positions represent the outer, intermediate and inner loop. Every raw data sample and image data pixel (indicated as black) is accessed directly in the inner loop, while everything indicated with a darker grey is accessed slower in the intermediate loop. Everything indicated with a very light grey is accessed in the outer loop, so very slowly. The following considerations are thought without the concept of local memories in the PE. It is also assumed that the RAM bus word width B_{ww} is not equal to the data word width D_{ww}, which is used for processing raw data samples and image pixel. Like this, the system and/or memory controller might also have to take care about reading or storing a smaller data word within a larger bus word in memory. This requires more memory operations to handle address misalignment and would not allow for single data word addressing. The RAM access of the first iteration of the inner loop is represented with arrows. The load and store actions in Fig. 4.3 are indicated with lower and upper case letters. A lower case indicates the load or store of a single data word, while an upper case letter indicates the load or store of an entire line. Read operations from the raw data memory are indicated by the letter a or A. Load and store operations on the image memory are indicated by the letter b and B or c and C. Whether an operation is a load or a store can be seen by the direction of the arrow. An arrow from the image memory to a PE is a load operation, while an arrow to the image memory is a store operation.

Without any optimization, processing a full image requires $N_x \cdot N_y \cdot N_{az}$ raw data load, image data load and image data store memory operations each. This does not include additional load and store operations required for the mathematical operations. Every permutation of access strategies is characterized by different access patterns, which can perform differently on memory. Fig. 4.3 depicts the different possible patterns to calculate the image. Again, this can result in different access patterns on the image memory and the raw data memory, which are depicted as bigger blocks. A PE loads data from the raw data memory and loads/stores data from/in the image memory. This depends on the chosen strategy. A PE is depicted as one single square but can either hold one single pixel, parts of an image line, a full image line or the entire image, depending on available resources. It will be discussed that some strategies allow for certain advantages regarding the amount of load and store operations, while others do not have any effect. By this, certain resource-demanding setups for the PE can be excluded from further evaluation as no beneficial effects exist.

Configuration ① and ② project N_{az} raw data lines, one after another, on the entire image. This creates N_{az} slices of the image, which need to be accumulated for the full image (see Fig. 4.3). Loop parameters are accessed in the order of ① $= [az, y, x]$ or ② $= [az, x, y]$. This means, that for the processing of one image slice the same raw data line az will be accessed $N_x \cdot N_y$ times. Therefore, the range samples rg within that a range line az are loaded in total $N_x \cdot N_y$ times. Meaning that a specific sample rg will be accessed $\frac{N_{rg}}{N_x \cdot N_y}$ times in average. Whereby not every sample must be accessed to the same amount. This depends on the geometrical dependencies. Based on these dependencies, a sample might be required more often for interpolation then others. The geometric relationship between the range line and image line determine the access rate of each range sample. In total the amount of read operations (a) on raw data memory for the entire image sums up to $N_{az} \cdot N_x \cdot N_y = 10^9$ or 1G (Giga) loads. Image data is loaded continuously (b) as with every inner loop iteration a new pixel is partly processed. The result is written back (c) to memory as a small fraction of one image slice. To form the complete image, all slices need to be summed up coherently, wherefore one pixel must be loaded, summed up and stored az times. This results to the total amount of $N_{az} \cdot N_x \cdot N_y = 10^9$ or 1G (Giga) load operations and 1G (Giga) store operations on the image memory. Assuming that the x direction is the direction of fast access in the memory, configuration ① is advantageous over ② in terms of access patterns. This is due to the fact, that the access across memory is performed fast in x direction (inner loop). Therefore, this pattern exploits spatial locality better than a fast run in y direction. This reduces the address jumps within image memory and also the busloads.

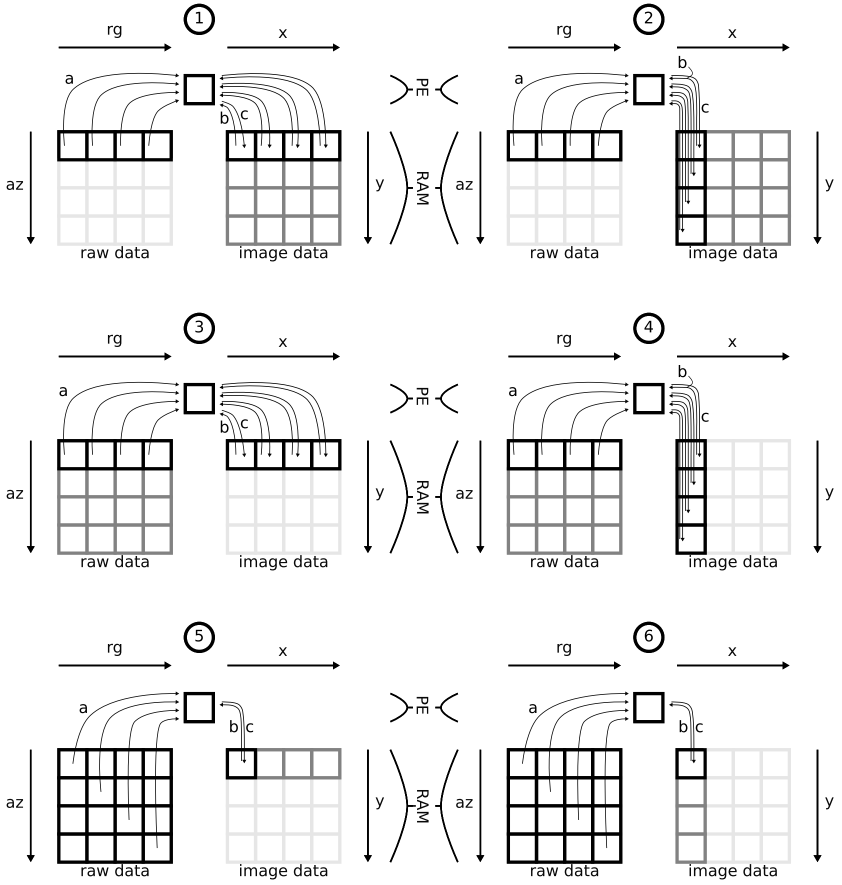


Figure 4.3: Memory access patterns without local buffers

Configuration ③=$[y,az,x]$ and ④=$[x,az,y]$ project raw data lines az one after the other on an entire image line (see Fig. 4.3). In contrast to ① and ②, one image line is finished after the inner loop is done. The amount of read operations (a) on raw data memory is the same as for configuration ① and ②. A smaller disadvantage is the access pattern of raw data memory since a new raw data line az is accessed in every iteration of the intermediate loop (indicated with darker grey). This results in a more frequent change of memory locations in the raw data memory. The amount of image loads (b) and stores (c) from and to the image memory is equivalent to configuration to ① and ②. This results from the fact that the inner loop accesses a new pixel in every iteration. But the order temporally restricts iteration space (black line instead of light grey) and reduces the address jumps on image memory to one direction until one image line is finished. Considering the order of x and y and the image memory direction, the pattern of configuration ③ is advantageous over ④.

Configuration ⑤$=[y,x,az]$ and ⑥$=[x,y,az]$ project the raw data lines az one after another on an image pixel (see Fig. 4.3). In contrast to ① and ② the pixel is finished after the inner loop is done. The amount of read operations a on raw data memory is the same as for any other configuration. A bigger disadvantage is the pattern of raw data memory access since a new raw data line az is accessed in every iteration of the inner loop (indicated black). But for image data these configurations show a major advantage, which leads to a reduction of read operations b to $N_x \cdot N_y = 10^6$ or 1M (Mega) loads for both configurations. Store operations c also reduce to $N_x \cdot N_y = 10^6$ or 1M (Mega) stores for both configurations. Configuration ⑥ is processed in y (intermediate loop) wherefore address jumps in image memory are more frequent. But the load operations are only required when storing the final pixel value. Since the initial value of any image pixel is zero, no load cycles on image memory are required. And since a single image pixel is processed in one sweep, intermediate results do not have to be stored and loaded for later accumulation in the image memory. However, this only applies in the case, that single data words can be written to the image memory. But usually, only full bus words with B_{ww} bits can be written to the image memory. Usually, $B_{ww} > D_{ww}$, which means that a data word (pixel) with D_{ww} bit must be stored inside of a full bus word. Therefore, it is necessary to read the memory location first, then include the pixel value and store the entire bus word back into image memory. For the case that $D_{ww} \geq B_{ww}$ read operations could be skipped completely for both strategies.

The configurations show how much influence the order of loops can have on the memory access patterns and thereby on access rates. Under the assumption, that tiny local buffers in an architecture are present, which is a standard feature in the form of a cache, load and store operations can be reduced significantly in some cases. This results from the fact that the tiny memories allow for storing enough pixel of D_{ww} bit to combine it to a full bus word of B_{ww} bit, which can then be transmitted directly. This results in a reduction of load and store cycles and can even result in a reduction to zero read cycles on image memory in the case of configuration ⑤. These mechanisms will be discussed in the following subchapters. Since the reduced memory access rates result from the possibility to hold data in a tiny local memory, the potential of even bigger local memories should also be discussed further.

4.5.1.3 Considerations for GBP RAM access reduction

For FPGAs a larger design space exists when compared to non-dedicated architectures. FPGAs allow for a high degree of freedom in the choice of how resources are implemented. This freedom can be used to implement local memories (in the form of

BRAMs) in a variety of sizes and bandwidth. The amount and size of such memories depend on the FPGA type. Therefore, a design must be able to adapt the amount and size of the buffers. Like caches for regular GPPs, buffers are an important factor for fast processing and can be vital for reaching a fluent stream of data. Using BRAMs gives several advantages compared to a system of only RAM.

First, compared to RAM, BRAMs can provide lower read/write latency of one clock cycle, a wider interface, true dual-port access (synchronous read/write) and deterministic timing. Especially for streaming architectures, a deterministic access behavior is important, to not run into stalls of the entire system. Due to the fast and deterministic access, every load/store operation shifted from external to internal memory, does not need to be considered further for the memory access rates.

Second, small buffers can be used to fully utilize RAM access. As already described in Section 4.5.1.2 for configuration ⑤ and ⑥, redundant data load operations occur when $D_{ww} < B_{ww}$ because individual word addressing is not possible. Reading one data word (when $D_{ww} < B_{ww}$) just results in a transmission of additional useless data. Writing one data word would require to read one bus word from the RAM, encapsulating the data word, and writing back the bus word in order not to overwrite neighboring data words. Therefore, write access will increase the redundant bus load because of the additional preceding read operations. Depending on the loop order, BRAMs can be used to solve both problems. They can store the additional loaded data for a short time interval if the data is required in the short term. In the case of writing, a buffer can amass adjacent data words until a complete bus word can be written. Smaller buffers can already help to increase bus load efficiency and therefore reduce the total amount of external memory accesses. Since such buffers are so small, it is assumed that they exist in every architecture. For non-dedicated architectures, this can also be assumed, as almost all architectures are equipped with caches. The existence of such buffers is indicated by the red factors in Table 4.6 and Table 4.7.

Third, if efficiently integrated, bigger buffers allow shifting a bigger portion of external memory access to internal memory access. This is possible if the loop order mainly accesses a spatially limited memory region for a longer period. BRAMs can be used to buffer that data region to reduce write and read operations to the RAM. But considering the size of SAR raw and image data, FPGAs can only store fractions of the data in BRAMs. Since resources are limited, the BRAMs must be dimensioned and placed thoughtfully. Larger buffers can reduce RAM access rates significantly.

Applied to the discussed permutations of the GBP algorithm, BRAMs can be very beneficial in some cases, while in other cases they do not show any improvement. Figure 4.4 shows how access pattern change in case BRAMs are used as buffers. Table 4.6 lists the amount of external memory operations with and without BRAMs for comparison.

For configuration ① and ② buffers are useful in two cases.

First, raw data can be buffered. For one outer loop iteration (image slice), one raw data line in the external memory is accessed $N_x \cdot N_y$ times. During this time,

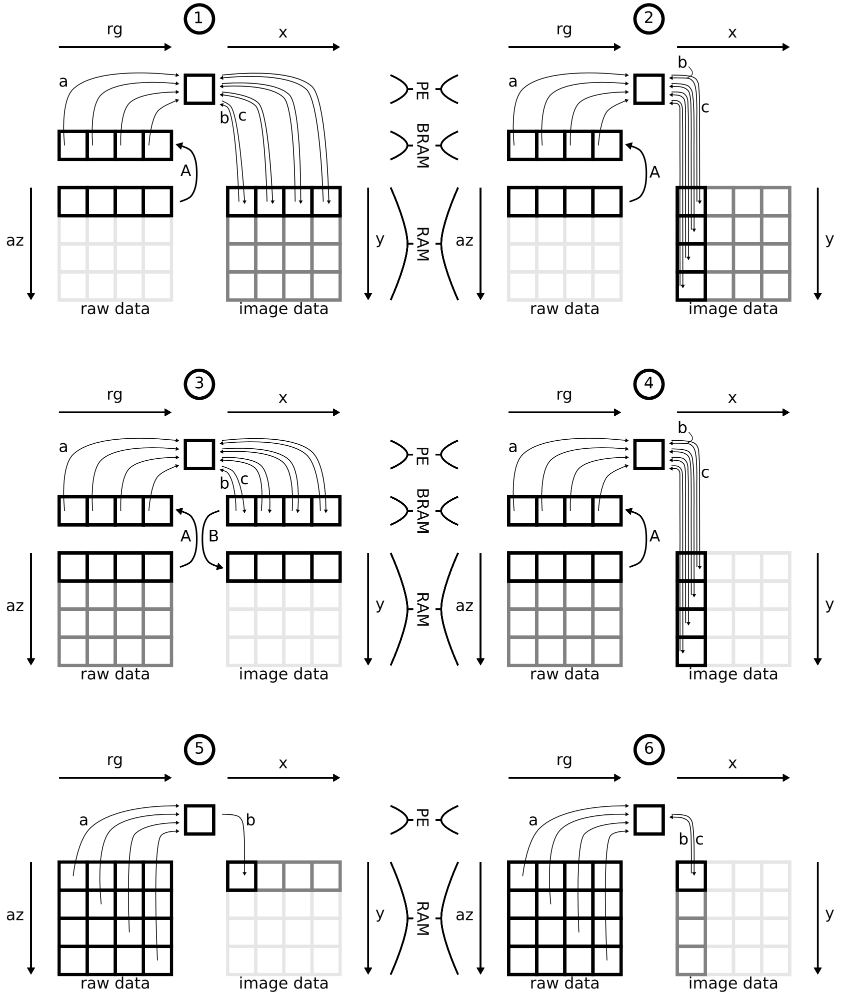


Figure 4.4: Memory access patterns with local buffers

the same raw data samples will be accessed repeatedly in irregular patterns. This generates small and/or big address jumps based on the geometric relation of the image and the raw data. For this reason, it would not be sufficient to buffer smaller coherent chunks of raw data, as the next access could already be out of buffer bounds. But in case the entire raw data line is buffered (indicated with A), one outer iteration run requires only N_{rg} external memory raw data load operations, to move the full line to the internal buffer. From this point, every access to this raw data line (indicated with a) is handled internally for this image slice, meaning that $N_x \cdot N_y$ load operations reduce to N_{rg} load operations for the processing of one line. For a full image, this sums up to $N_{rg} \cdot N_{az}$ raw data loads with buffers, compared to $N_y \cdot N_x \cdot N_{az}$ without buffers. For the exemplary dimensions, this reduces the load operations from 10^9 or 1G (Giga) loads to 10^6 or 1M (Mega) loads. In the case that the bus word width B_{ww} is bigger than the data word width D_{ww}, buffering is even more beneficial, since all additional data that is loaded is not lost due to full buffering. This results to $\frac{N_{rg} \cdot N_{az}}{\lfloor \frac{B_{ww}}{D_{ww}} \rfloor}$ raw data load operations in total.

Second, image data can be buffered. But line buffers are of no use, as an image pixel is only accessed once in one outer iteration run. Nevertheless, small buffers can be used to increase the efficiency of load (b) and store (c) operations in case of $B_{ww} > D_{ww}$. By storing a bus word instead of a data word, all loads to the following image pixel within the range of a bus word are avoided. The efficiency increases by the factor of $\lfloor \frac{B_{ww}}{D_{ww}} \rfloor$ which reduces the total amount of accesses from $N_x \cdot N_y \cdot N_{az}$ to $\frac{N_x \cdot N_y \cdot N_{az}}{\lfloor \frac{B_{ww}}{D_{ww}} \rfloor}$ for load and store operations each.

Additionally, burst modes, which are available in common external memory controllers, can be used to read multiple bus words consecutive in one direction. The amount of read operations is not reduced, but the overhead for memory setup times is smaller, whereby memory throughput is increased. Buffers need to be extended to the size of burst length $b_l \cdot B_{ww}$. Optimization considering B_{ww} and b_l are only possible for configuration ① which reads and writes data in memory direction (x,rg). When processing in y direction (image row) all additional intermediate pixel values which are loaded in x direction are obsolete. To avoid this, the buffer size would have to be increased from N_x to $N_x \cdot B_{ww}$ respectively $N_x \cdot B_{ww} \cdot b_l$. Otherwise, the same amount of read operations as without buffers is required.

For configuration ③ and ④ raw data can be buffered as well. But since one outer loop iteration completes an image line or image row, all raw data lines are accessed in shorter time. Since not all lines can be buffered, they need to be reloaded for the next outer iteration run. For a full image, this sums up to $N_{rg} \cdot N_{az} \cdot N_x$ raw data loads with buffers, compared to $N_x \cdot N_y \cdot N_{az}$ without buffers. In case of equal dimensions of rg and x, this leads to the conclusion that buffers are beneficial in the case $B_{ww} > D_{ww}$, resulting to $\frac{N_{rg} \cdot N_{az} \cdot N_y}{\lfloor \frac{B_{ww}}{D_{ww}} \rfloor}$ raw data load operations in total. Buffering also proofs to be very beneficial for image data. In case a full image line is buffered, no intermediate values need to be transferred to and from the RAM. This is because one image line is processed en bloc, meaning all raw data lines are projected on the line that is currently buffered. This reduces the number of load operations to zero. Only the final values of the line need to be transferred (indicated with B) in $\frac{N_x \cdot N_y}{\lfloor \frac{B_{ww}}{D_{ww}} \rfloor}$ cycles to the RAM. For the exemplary dimensions, this reduces the store operations from 10^9 or 1G (Giga) stores to 10^6 or 1M (Mega) stores. Write operations in burst mode can be used to reduce the setup time overhead. As for configuration ①, optimization considering B_{ww} and bl are only possible for configuration ③ due to operation in x direction. Operating in y direction bears another disadvantage for writing image data. While image rows can be processed en bloc in y direction, writing a row en bloc in y direction is not possible. Neighboring data in x would have to be read first, resulting in additional $N_x \cdot N_y \cdot N_{az}$ read operations. This is a drawback to processing without image data buffers.

For configurations ⑤ and ⑥ it is of no use to buffer raw data since every inner loop iteration addresses a new raw data line. If a line would be buffered, this would sum up to more raw data load operations then without buffers. This is why every raw data sample would have to be accessed separately like in a "unbuffered" setup. Buffering image data is beneficial. Although configuration ⑤ and ⑥ have different access patterns then ③ and ④, the numbers for reading and writing image data are identical. The difference is the size of the buffers. While ③ and ④ buffer a complete line, ⑤ and ⑥ buffer one image pixel only, which reduces the number of required resources for buffer size. Nevertheless, for writing data, a minimal buffer with the length of B_{ww} is required, for architectures where $D_{ww} < B_{ww}$. This is required to enable a PE to be able to write a complete bus word back to memory. This would also be necessary for "unbuffered" architectures. And since buffers for writing data could be used when reading image data this would change both image data access numbers. This is indicated by the red numbers in Table 4.6. This unshrinkable buffer shows, that already small amounts of additional resources can reduce the number of memory operations significantly. For each FPGA a trade-off between available resources and reduction of memory load operations has to be made.

It can be stated, that an access in y direction (configuration ②, ④, ⑥) for any architecture where $D_{ww} < B_{ww}$ is unfavorable, due to the perpendicular memory access. Especially for line buffered architectures all benefit is lost, as long as buffers are not two dimensional.

For a design decision, all configurations need to be compared. It is obvious that all configurations with data access in y direction do not have to be considered. Configuration ① requires reduced raw data read operations but much more image read/write operations. Considering the costs of a write access, this configuration is only favorable if raw data sets are too big for main memory or other memory models that are used, like for example in [76]. When comparing "unbuffered" architectures, configuration ⑤ offers the most benefit, as image read operations are reduced to zero and image write operations reach the theoretical minimum. For buffered architectures ⑤ and ③ are in strong competition. The only difference in data access rates is for raw data which is reduced for ③ in case $D_{ww} < B_{ww}$. Also, access patterns are fit better, as full lines are read in range direction instead of singe pixel in the azimuth direction. On the other hand, configuration ⑤ consumes much fewer resources, as no full lines need to be buffered to write image data. A design decision must either base on the available resources or on another advantage, which might be given in case of intra or inter parallel PE design. Moreover, the mathematical

Table 4.6: Memory access rates for GBP configurations, implemented with and without buffers, the red factors apply in case the size of a full B_{ww} can be stored locally, otherwise red factors apply with 1

	buffered			unbuffered		
Configuration	raw data	image data		raw data	image data	
	read	read	write	read	read	write
$1=[az,y,x]$	$\frac{N_{az}\cdot N_{rg}}{\lfloor \frac{B_{ww}}{D_{ww}} \rfloor}$	$\frac{N_{az}\cdot N_x\cdot N_y}{\lfloor \frac{B_{ww}}{D_{ww}} \rfloor}$	$\frac{N_{az}\cdot N_x\cdot N_y}{\lfloor \frac{B_{ww}}{D_{ww}} \rfloor}$	$N_{az}\cdot N_x\cdot N_y$	$\frac{N_{az}\cdot N_x\cdot N_y}{\lfloor \frac{B_{ww}}{D_{ww}} \rfloor}$	$\frac{N_{az}\cdot N_x\cdot N_y}{\lfloor \frac{B_{ww}}{D_{ww}} \rfloor}$
$2=[az,x,y]$	$\frac{N_{az}\cdot N_{rg}}{\lfloor \frac{B_{ww}}{D_{ww}} \rfloor}$	$N_{az}\cdot N_x\cdot N_y$	$N_{az}\cdot N_x\cdot N_y$	$N_{az}\cdot N_x\cdot N_y$	$N_{az}\cdot N_x\cdot N_y$	$N_{az}\cdot N_x\cdot N_y$
$3=[y,az,x]$	$\frac{N_{az}\cdot N_{rg}\cdot N_y}{\lfloor \frac{B_{ww}}{D_{ww}} \rfloor}$	0	$\frac{N_x\cdot N_y}{\lfloor \frac{B_{ww}}{D_{ww}} \rfloor}$	$N_{az}\cdot N_x\cdot N_y$	$\frac{N_{az}\cdot N_x\cdot N_y}{\lfloor \frac{B_{ww}}{D_{ww}} \rfloor}$	$\frac{N_{az}\cdot N_x\cdot N_y}{\lfloor \frac{B_{ww}}{D_{ww}} \rfloor}$
$4=[x,az,y]$	$\frac{N_{az}\cdot N_{rg}\cdot N_y}{\lfloor \frac{B_{ww}}{D_{ww}} \rfloor}$	$N_{az}\cdot N_x\cdot N_y$	$N_x\cdot N_y$	$N_{az}\cdot N_x\cdot N_y$	$N_{az}\cdot N_x\cdot N_y$	$N_{az}\cdot N_x\cdot N_y$
$5=[y,x,az]$	$N_{az}\cdot N_x\cdot N_y$	0	$\frac{N_x\cdot N_y}{\lfloor \frac{B_{ww}}{D_{ww}} \rfloor}$	$N_{az}\cdot N_x\cdot N_y$	$N_x\cdot N_y\cdot 0$	$\frac{N_x\cdot N_y}{\lfloor \frac{B_{ww}}{D_{ww}} \rfloor}$
$6=[x,y,az]$	$N_{az}\cdot N_x\cdot N_y$	$N_x\cdot N_y$	$N_x\cdot N_y$	$N_{az}\cdot N_x\cdot N_y$	$N_x\cdot N_y$	$N_x\cdot N_y$

operations for processing could tip the balance, when more than just one atomic location in memory is required. In the case of interpolation, neighboring samples will have to be loaded as well. Based on the width of such operations, buffers in the line direction might reduce external access additionally.

4.5.1.4 Considerations for FFBP RAM access optimization

Similar to the analysis for the GBP algorithm, strategies can also be optimized for the FFBP algorithm in order to reduce the RAM access rate. The given analyzes concentrates only on the factorization part of the FFBP (see Algorithm 4.14), as only in this part comparatively high bus load is induced. The parts for subimage center calculation (see Algorithm 4.12) and subaperture position calculation (see Algorithm 4.13) require mostly arithmetic operations and less memory transfers.

In terms of iterations, the merging of apertures to form a new subaperture is much less complex than a full GBP run. This is caused by the fact, that every sample in a raw data set is only accessed once to form all subapertures for a new data set. Compared to the GBP, where every sample is accessed N_{az} times, this reduces complexity from $O(N^3)$ to $O(N^2)$ (where N represents N_{az} and N_{rg}). But complexity is increased for the FFBP since the described process only represents the complexity of one out of many subimages in one out of many stages. Nevertheless, the FFBP would only reach (or even extend) the complexity level of the GBP for very small images combined with many stages and subimages. When comparing the general structure of the two algorithms, similarities are recognizable in terms of loop setup, even some instructions are shared. The major difference besides the complexity is that the range of accessed samples is reduced to the spatial locality of f_{apt} apertures for subaperture merging. This is contrary to the GBP, where each sample is treated more or less global. Combined with the right strategy, this can result in some advantages in terms of memory access rates.

Similar to the GBP the loop order of the FFBP is flexible, but the indexing of data arrays or vectors must be adjusted. The order describes how often raw data must be read and how often subaperture data must be read and/or written. Table 4.7 lists the general formulas for the access rates for generating of one subimage in one stage. Loop parameters are i_{apt}, i_r and i_m, where the index apt is the number of formed subapertures, m controls the number of merged apertures per subaperture and r is the number of range samples in a line. For example, the order of $[i_{apt}, i_r, i_m]$ means that i_{apt} is controlling the outer loop on the highest hierarchic level, i_r controls the intermediate loop and i_m is controlling the inner loop on the lowest hierarchic level. Although not directly visible through the loop factors, the factor N_{az} is included in these formulas, as it results from the product of merged apertures f_{apt} and the number of subapertures $i_{apt} = N_{az}\ /\ f_{apt}$.

Configuration ⓐ is processing the subaperture positions in the inner loop. Since every subaperture is merged from a different set of f_{apt} raw data apertures, address jumps in both memory sections (raw data read, subaperture write) are required for every inner loop iteration.

As already explained in the GBP section, a minimal buffer with the length of B_{ww} is required for writing data, for any architecture (“buffered” or “unbuffered”) where $D_{ww} \leq B_{ww}$. This changes access numbers and is indicated by the red factors in Table 4.7. In case of a $B_{ww} > D_{ww}$, the beneficial effect of reading more then one word per cycle is not given due to the continues jumps. This yields $N_{az} \cdot N_{rg}$ memory read operations on raw data memory, as every raw data sample must be read separately. The same applies to the writing of subaperture samples to subaperture memory. Since samples are merged, $N_{az} \cdot N_{rg}$ read operations on subaperture memory are necessary to read the current sum for further accumulation. As this is not necessary for the first summand of a range sample, this results in a given amount of read operations. The difference between configuration ⓐ and ⓑ is the faster run across merge positions. This causes even more address jumps in memory. But since the inner loop causes bigger address jumps by running fast across subaperture positions, there is no real difference between both configurations. Therefore the same amount of memory operations is required.

Configuration ⓒ is processing merge positions in the inner loop. This implies jumps to a new raw data line in every cycle. No bus words can be used, thus $N_{az} \cdot N_{rg}$ memory read operations on raw data memory are required. But since a merge for one sample of a subaperture is performed coherently in place accumulation is possible.

Table 4.7: Memory access rates for FFBP configurations, implemented with and without buffers, for one subimage and one stage, the red factors apply in case the size of a full B_{ww} can be stored locally, otherwise red factors apply with 1

	buffered			unbuffered		
Configu-ration	raw data	subaperture data		raw data	subaperture data	
	read	read	write	read	read	write
a=$[i_m, i_r, i_{apt}]$	$\frac{N_{az}\cdot N_{rg}}{\lfloor \frac{B_{ww}}{D_{ww}} \rfloor}$	0	$\frac{N_{az}\cdot N_{rg}}{\lfloor \frac{B_{ww}}{D_{ww}} \rfloor}$	$N_{az}\cdot N_{rg}$	$(N_{az}-\frac{N_{az}}{f_{apt}})\cdot N_{rg}$	$N_{az}\cdot N_{rg}$
b=$[i_r, i_m, i_{apt}]$	$\frac{N_{az}\cdot N_{rg}}{\lfloor \frac{B_{ww}}{D_{ww}} \rfloor}$	0	$\frac{\frac{N_{az}}{f_{apt}}\cdot N_{rg}}{\lfloor \frac{B_{ww}}{D_{ww}} \rfloor}$	$N_{az}\cdot N_{rg}$	$(N_{az}-\frac{N_{az}}{f_{apt}})\cdot N_{rg}$	$N_{az}\cdot N_{rg}$
c=$[i_{apt}, i_r, i_m]$	$\frac{N_{az}\cdot N_{rg}}{\lfloor \frac{B_{ww}}{D_{ww}} \rfloor}$	0	$\frac{\frac{N_{az}}{f_{apt}}\cdot N_{rg}}{\lfloor \frac{B_{ww}}{D_{ww}} \rfloor}$	$N_{az}\cdot N_{rg}$	0	$\frac{\frac{N_{az}}{f_{apt}}\cdot N_{rg}}{\lfloor \frac{B_{ww}}{D_{ww}} \rfloor}$
d=$[i_r, i_{apt}, i_m]$	$\frac{N_{az}\cdot N_{rg}}{\lfloor \frac{B_{ww}}{D_{ww}} \rfloor}$	0	$\frac{\frac{N_{az}}{f_{apt}}\cdot N_{rg}}{\lfloor \frac{B_{ww}}{D_{ww}} \rfloor}$	$N_{az}\cdot N_{rg}$	$\frac{N_{az}}{f_{apt}}\cdot N_{rg}$	$\frac{N_{az}}{f_{apt}}\cdot N_{rg}$
e=$[i_m, i_{apt}, i_r]$	$\frac{N_{az}\cdot N_{rg}}{\lfloor \frac{B_{ww}}{D_{ww}} \rfloor}$	$\frac{(N_{az}-\frac{N_{az}}{f_{apt}})\cdot N_{rg}}{\lfloor \frac{B_{ww}}{D_{ww}} \rfloor}$	$\frac{N_{az}\cdot N_{rg}}{\lfloor \frac{B_{ww}}{D_{ww}} \rfloor}$	$\frac{N_{az}\cdot N_{rg}}{\lfloor \frac{B_{ww}}{D_{ww}} \rfloor}$	$\frac{(N_{az}-\frac{N_{az}}{f_{apt}})\cdot N_{rg}}{\lfloor \frac{B_{ww}}{D_{ww}} \rfloor}$	$\frac{N_{az}\cdot N_{rg}}{\lfloor \frac{B_{ww}}{D_{ww}} \rfloor}$
f=$[i_{apt}, i_m, i_r]$	$\frac{N_{az}\cdot N_{rg}}{\lfloor \frac{B_{ww}}{D_{ww}} \rfloor}$	0	$\frac{\frac{N_{az}}{f_{apt}}\cdot N_{rg}}{\lfloor \frac{B_{ww}}{D_{ww}} \rfloor}$	$\frac{N_{az}\cdot N_{rg}}{\lfloor \frac{B_{ww}}{D_{ww}} \rfloor}$	$\frac{(N_{az}-\frac{N_{az}}{f_{apt}})\cdot N_{rg}}{\lfloor \frac{B_{ww}}{D_{ww}} \rfloor}$	$\frac{N_{az}\cdot N_{rg}}{\lfloor \frac{B_{ww}}{D_{ww}} \rfloor}$

When writing data to the subaperture memory, only the actual summed up values for all subapertures are written ($\frac{N_{az}}{f_{apt}}$). Since the merge is processed in range direction in the intermediate loop, a bus word width can be assembled and written back to memory. This results in 0 read operations on subaperture memory, as a full bus word is written, thus no redundant parts need to be read, to write a single sample back to memory. The difference between configuration ⓒ and ⓓ is the faster run across subaperture positions. This implies to disadvantages. First, no bus words can be assembled, wherefore the number of write operations cannot be optimized. Second, this makes read operations on subaperture memory necessary, as full bus words must be read to include a single sample value.

Configuration ⓔ runs across range positions in the inner loop. This implies a continues address scheme and therefore few jumps in memory. Bus words can be used, wherefore read operations on raw data memory is reduced. Since no in place accumulation is possible, the read and write operations on subaperture memory would result in the same amount of operations as for configuration ⓐ and ⓑ. But processing in range allows reducing this amount by reading full bus words in range direction. The difference between configuration ⓔ and ⓕ is the faster run across merge positions. This has no effect on the number of memory operations, as jumps across memory are necessary anyway and in place accumulation is still not possible.

In case of $D_{ww} = B_{ww}$ ⓒ is the most efficient configuration as it requires no read operation on subaperture memory. Configuration ⓔ and ⓕ increase in efficiency the shorter D_{ww} is compared to B_{ww}. This is to be expected for any system with common RAM interfaces. Between the two setups, configuration ⓕ is preferred, as the addressing scheme is causing smaller address jumps compared to ⓔ. Additionally, both configurations not only reduce memory operations but also the number of arithmetic operations. Due to the run in range direction, the cosine for the angle between subaperture and old aperture is constant for the entire inner loop iteration. A similar, but less effective, effect can be observed for configurations ⓑ and ⓓ due to the slow change in range direction. This reduces the computation time for every instruction related to i_r. However, for a regular SAR setup, the loop length for i_r will be greater than for the merge loop i_m or the run across apertures. Considering all aspects, configuration ⓕ seems to be the most efficient when it comes to memory operations. Especially in combination with a data greedy GBP algorithm using the same bus interface at the same time, the reduction of memory access is a vital criterion.

4.5.1.5 Considerations for FFBP RAM access reduction

Local buffers can only be instantiated in smaller numbers as already discussed in the GBP Section 4.5.1.3. Introducing buffers will change how often raw data is read and how often subaperture data is read and/or written. Table 4.7 lists the general formulas for the access rates for one subimage of generating one stage.

For the possible configurations, buffers show low effort whenever the outer loop is used for the actual merge process. This implies that the inner loops change subaperture position and range position fast in one of two cases. Any of the two would require $N_{az} \ / \ f_{apt}$ buffers with a length of N_{rg} each, to allow for the merge to be done in phase. This is equivalent to buffering the entire subimage, which is not resource-efficient and not even possible for bigger subimages. The configurations ⓐ and ⓔ are characterized by this combination. For configuration ⓐ buffers can still be of good use, but only when instantiating $N_{az} \ / \ f_{apt}$ buffers with a length of B_{ww} each. This helps to read and/or write raw data and subaperture data in bus words.

The same type of buffers can be used for configuration ⓑ and ⓓ with a different result compared to ⓐ. Both configurations have the same setup in the two most inner loops. Which means running fast across subaperture positions and merge sets. With a set of small buffers, a small set of intermediate results of a merge can be buffered for every subaperture position. This allows writing a bus word to memory, which makes read operations to subaperture memory obsolete. The raw data memory reads can be reduced as well, as the small memory can buffer the additional loaded samples until they are next in line for processing.

Another size of the buffer is the full range line buffer. This is beneficial for configuration ⓒ and ⓕ. Since the two most inner loops of both configurations run across range direction and merge direction. Therefore, a subaperture is always completely processed before the next subaperture is stored. A full range line buffer (or subaperture buffer), would allow to read bus words from raw data memory and buffer them. This allows merging samples to assemble bus words. Therefore, write operations to subaperture memory are reduced and all read operations to subaperture memory are obsolete. While the reduction of read operations to the raw data memory is only beneficial for configuration ⓒ, the reduction on aperture memory only affects configuration ⓕ.

Four configurations can be trimmed to the same level of memory operations. From all four, ⓕ seems to be most attractive as it maintains spatial locality in range and in merge direction. This allows for smaller address jumps in both memory sections. Nevertheless, due to the limited amount of BRAMs, the use off buffers is questionable for the FFBP as they are more beneficial for the design of a fast GBP implementation. In case of too few resources, buffers might only be possible for the GBP implementation.

4.5.2 Inter loop streaming (read write balancing)

To reach peak performance the processing of multiple data streams is vital. Depending on the algorithm, not every form of parallelism (e.g. instruction or data parallelism) is possible. Different processor designs are possible to suit a given class of algorithms at the best rate possible. According to Flynn's taxonomy [77], every design can be classified based upon the number of data streams or instruction streams available in the processor. The backprojection algorithms are characterized by the vast amount of data that is processed to form new sets of raw data or final image pixel. Since the instruction patterns are very regular and show almost no control dependencies, processing would fit best to a Single Instruction Multiple Data (SIMD) architecture. In a SIMD architecture each element of a set of similar data is processed simultaneously on multiple identical PEs in parallel. Such architectures are often realized in the form of vector or array processors. A block diagram of such a processor is depicted in Fig. 4.5. A control unit, which can be a softcore or a state machine, controls the entire flow of the system. This includes the input and output of data into main memory, the allocation of tasks to modules and it's inner PEs and the timing of data access. The PEs consists of an Arithmetic Unit (ARU) to perform arithmetic operations. They can have local buffers of variable size which are connected to main memory via a connection network, which can be a regular bus with different typologies. In a perfect system, the data would be streamed through the system without any stalls, resulting in maximal resource utilization.

The speedup of such designs does not only depend on the number of available resources. As already mentioned, the accessibility of data is critical. With a growing

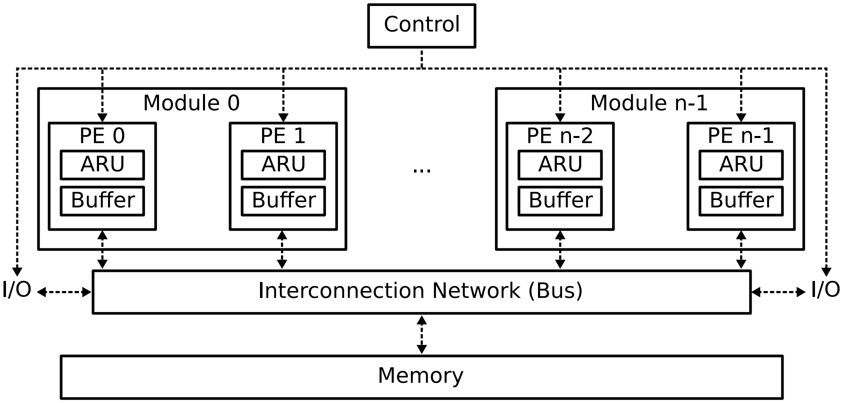


Figure 4.5: Generic block diagram of an array processor design

number of active PEs an exponential growth of read/write collisions is not unlikely since the memory is shared. This must, of course, be avoided by design choices. The focus of the inter loop streaming is to:

1. Reduce memory busload
2. Align data access across all implemented modules

Both algorithms work on different data in memory. The FFBP requires original raw data to create merged subaperture data. The GBP takes this subaperture data for processing. For this reason, the busload reduction should be handled on module level (GBP and FFBP). The alignment of data access is partly handled on PE level (as a module can contain several PEs which interfere with each other), partly on module level (as FFBP and GBP module are active at the same time but both interact often with main memory).

4.5.2.1 GBP inter loop streaming

As a first step on GBP PE level, one of the already discussed configurations in Section 4.5.1 is picked in order to reduce memory access. In case of insufficient resources for local buffers, configuration ⑤ requires the smallest amount of memory access. The disadvantage of all "unbuffered" architectures is the continuous access to memory, either to raw data or image memory. The continuous address jumps and the big amount of surplus raw data on the bus (in case a data word is not directly accessible) facilitates the bus as a bottleneck. Assuming that FPGAs with a considerable amount of BRAM resources are available, buffers can be discussed.

Between buffered mode configurations, ③ is the most efficient. Equipped with two full line buffers for raw data and image data, this configuration combines several advantages. These advantages are, a minimum of image data write operations, reduction of all image data read operations to zero, reduced raw data read operations and more linear raw data read patterns (in case of $B_{ww} > D_{ww}$). Besides the already discussed advantages, this configuration enables for an additional reduction of raw data reads by the factor p, which represents the amount of parallel working PEs. This is a major feature since more PEs would usually cause a heavier busload in a shorter time. This would increase the chances for conflicts and stalls. For a streaming design, even one PE is already enough to block access to memory alone. But with the chosen design, more PEs actually manage to keep the busload on the same level of one PE, and therefore to reduce the amount of total raw data read operations. This is possible through data sharing and is a major argument to have multiple GBP PEs working in parallel.

To process an image line, a PE reads the entire set of $N_{az} \cdot N_{rg}$ raw data. To process all lines of an image, this process needs to be repeated N_y times. A reduction is possible because p parallel PEs require the same raw data to process p different image lines. All p PEs run in parallel and can be synchronized to always process the same raw data section at a time. This allows to broadcast this section of raw data to all PEs where the data is buffered for full line processing. This reduces the N_x runs by factor p. Taking burst transfers of length b_l at a bus word width B_{ww} and data word width D_{ww} into consideration, the final amount is determined by Eq. (4.2).

$$Raw_{rd} = \frac{N_{az} \cdot N_{rg} \cdot N_y}{p \cdot b_l \cdot \lfloor \frac{B_{ww}}{D_{ww}} \rfloor} \tag{4.2}$$

This results in a very drastic reversal behavior. The more PEs work in parallel, the less traffic for raw data loads is caused. This is contrary to all other configurations, where more PEs would cause more load traffic in the same time interval.

Similar optimizations are not possible for other configurations, due to different access patterns on raw and image data. Configuration ① would not benefit from such measures as every PE would handle a different raw data line. Configuration ⑤ would not benefit either as every PE needs only a small specific section of a raw data line to project it onto the processed image pixel. Based on the geometric relation between the raw data line and image pixel, these sections can be different for every pixel. For reading image data such optimizations are impossible for any configuration, as they would inflict hazards in the form of output dependencies.

This is giving configuration ③ a clear implementation advantage in terms of bus load reduction. The linearity of the address patterns allows for the most efficient memory access. Additionally, no memory sections overlap, which avoids the chance of read or write conflicts. Nevertheless, it is important to try to balance read and write operations between multiple PEs. In this case not in terms of amount, but in terms of homogeneity. Every switch between read and write operations will add additional delays to memory access. A sequence of pure read or write operations will execute much faster than a sequence of operations consisting of a mixture of reads and writes. The concept of line buffers helps to meet these requirements once again. The strategy allows large blocks of data to be received or transmitted in one sweep. Nevertheless, PEs need to be scheduled to follow such patterns and to synchronize in order not to pile up read or write memory access.

This architecture is designed to work with standard monolithic memory components (eg. Double Data Rate (DDR) memory). Such components usually cannot be read and written at the same time. But they allow for wide memory access in one cycle. For example, one DDR3 Synchronous Dynamic Random Access Memory (SDRAM) module usually allows for $8 \cdot 64 = 512$ bit of parallel memory access. Compared with a standard data word width D_{ww} of 16, 32 or maximum 64 bit, multiple data words can be transferred in one cycle. Due to the heavy arithmetic load of the algorithm, it can also be assumed that loading a raw data line from memory will take a shorter time than the projection on an image line. This leads to the conclusion that the scheduled slots will be dominated by processing time. In combination with a configuration to reduced memory load, this allows to consecutively read from and write to a buffer line during processing.

A simple schedule for one PE with one raw data line buffer and one image data line buffer is depicted in Fig. 4.6. A raw data set R with two lines R_1 and R_2 is loaded from main memory, projected on an image I with two lines I_1 and I_2 and then written back to main memory. The length of a slot t is defined by the longest time, which is assumed to be the processing time. First, the raw data buffer of the PE is filled with the first raw data line R_1 in time slot t_1. In t_2 the ARU can start the projection of $R_1 \rightarrow I_1$. During this time new raw data cannot be transmitted to the buffer, as all raw data elements in the buffer are required for the projection. After the projection is finished, the next raw data line R_2 is loaded in t_3 and projected $R_2 \rightarrow I_1$ in t_4. In $t_5 - t_8$ this process is repeated with the second image line I_2. According to the assumption above, the finished image line I_1 can be transmitted from the image line buffer, back to main memory in the same cycle of receiving R_1 in t_5. The final image is transmitted back to the main memory in t_9. For the already mentioned exemplary setup of N_{az}, N_{rg}, N_x, $N_y = 1000$ this would result to $2 \cdot 10^6$ M (Mega) time slots t for processing, while one time slot can consist of many cycles of the ARUs. This calculation does not consider the time needed for filling the buffers at the very start. Fifty percent of the time is required for transferring data. Which means each component, the main memory, and the ARU are in idle state fifty percent of operation time.

As already claimed before, all resources should be utilized to the highest possible rate to achieve the maximum throughput rate. In theory, utilization can increase to one hundred percent by introducing additional buffers. The concept of a second buffer for raw and image data allows loading the next raw data line, while the first is

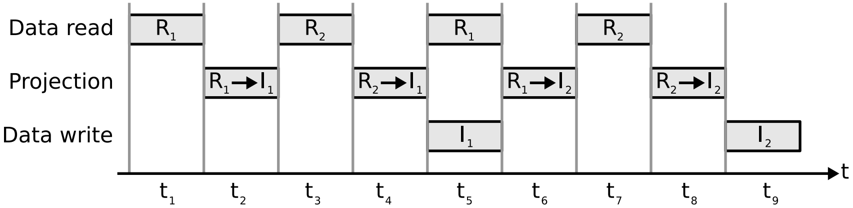


Figure 4.6: Schedule for GBP with one PE with single line buffers

still used for processing. Since this would enable for constant processing of raw data, a second image line buffer is required as well, to hold the already processed image line until it is fully transferred to the main memory. The ARU can swap the buffers, like this processing can continue seamlessly. A simple schedule for one PE with swap buffers is depicted in Fig. 4.7. The swap buffers are used for raw data and image data. In contrast to a single buffered setup, the next raw data line R_2 is loaded from main memory in t_2. As soon as projection $R_1 \rightarrow I_1$ is completed, buffers swap in t_3 and R_2 is projected on line I_1 in t_3 $(R_2 \rightarrow I_1)$. The final image line can be transferred to main memory in t_4. After this the last projections $(R_1 \rightarrow I_2$ and $R_2 \rightarrow I_2)$ follow. The whole process is finished in six time slots t. This concept implies that at least two full lines can be transferred during the processing of one line. This is possible as long as the interface of the main memory is at least twice as wide as a data word D_{ww} or bus clock frequency B_{cf} is at least twice as high as PEs clock frequency PE_{cf}. As long as this criterion is fulfilled, the total amount of time slots t can be reduced by fifty percent to $10^6 = 1\text{M}$ (Mega), just by doubling resources for buffers.

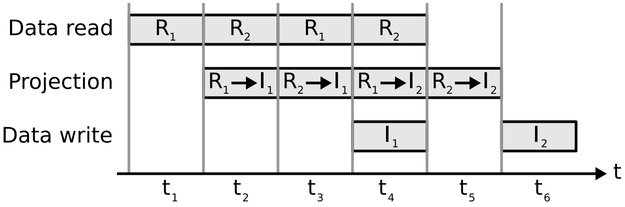


Figure 4.7: Schedule for GBP with one PE and swap line buffers

Constant processing (streaming) of a module can be established with the discussed methods of swap buffers and proper scheduling. A different way of optimization is the described parallelization of PEs. The PEs do not have to implement swap buffers for this. But the total amount of allocated resources is smaller for one PE with swap buffers then for two PEs with single buffers. More resources are consumed because a second PE doubles resources for buffering and for arithmetic operations (ARU), while a second buffer only doubles the amount of used BRAMs. However, a module should only contain PEs of the same design to smooth scheduling. The schedule for a multi-PE module with a parallelization degree of $p = 2$ including swap buffers is shown in Fig. 4.8. Following Eq. (4.2), the raw data bus load is reduced to fifty percent with $p = 2$. Both PEs receive the first raw data line R_1 in t_1 from memory with one transfer. In t_2 the swap buffers store the second raw data line R_2 during both PEs are processing. Each PE projects data on a different image line ($R_1 \rightarrow I_1$ and $R_1 \rightarrow I_2$). In t_3 buffers are swapped and the second raw data line is projected ($R_2 \rightarrow I_1$ and $R_2 \rightarrow I_2$). Both image lines are finished in t_4 and can be transferred to main memory. It has to be scheduled, which PE transfers its data to memory first, but the order will not alter the final image result since every PE processes a different image section. The line buffers, which are used for accumulation of all intermediate results can now act as a time buffer, until the PE is allowed to transfer data to the memory. All modules must finish transferring before the next range line can be requested from the main memory. For streamlined processing, which demands continuous memory transfer, this concept implies that $p + 1$ full lines can be transferred while processing. This is possible if $B_{ww} \geq D_{ww} \cdot (p+1)$, or if the bus clock frequency B_{cf} is higher then the PEs clock frequency PE_{cf}, so $B_{cf} \geq PE_{cf} \cdot (p+1)$. As long as this criterion is fulfilled, the total amount of time slots required for processing can be reduced by the factor of p. Otherwise, the average processing rate will be reduced because additional stalls for transmitting and receiving data would be required. In the given example, the required time slots t are halved by quadrupling resources for buffers and doubling resources for arithmetic.

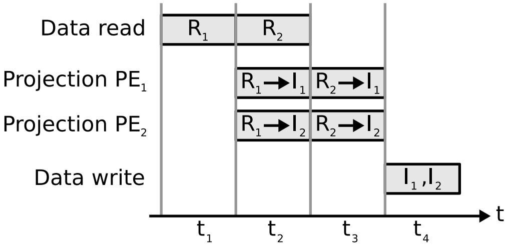


Figure 4.8: Schedule for GBP with two PEs and swap line buffers

These calculations are the theoretical minimum for seamless streaming. Different factors might interrupt the streaming and slow down the entire design. One of these factors could be the transmission of small chunks of control data and the geometrical parameters. Every PE receives this information to calculate the distance between a range line and the current image pixel, the interpolation index and a phase correction angle. All factors must, of course, be taken into consideration for the final architecture. At the same time, the design should be able to handle any interruption.

Based on the presented analysis, it is clear, that a concept containing multiple PEs with additional swap buffers offers the highest potential in terms of seamless streaming, as it is highly linear in terms of data access and reduces the busload to a minimum. The disadvantage is the higher resource consumption, therefore, the degree of p should be free of choice, to fit the design to different types of FPGAs.

4.5.2.2 FFBP inter loop streaming

Inter loop streaming is done on PE level, by a parallel instantiation of multiple processing cores. Naturally, this comes with an increase in resources. In the case of the GBP, the additional resources allow for faster processing and to help to reduce memory access rates. The flexibility in processing order allows to find an optimal parameter configuration of the algorithm, that allows combining p PEs leading to a reduction of read operations on memory by the factor p. This is possible by splitting the working set to parallel modules, which share the same raw data. Processing one subimage of the FFBP does not offer the same opportunity since each line of raw data for a subimage is only read once to contribute to one new subaperture. Nevertheless, more PEs can be instantiated to speed up the processing. More PEs can compensate for a lack of processing power in case incoming data for a subimage is transferred before a PE has finished the processing. Similar to the GBP PE, which can process one Pulse to Pixel Projection (PPP) per clock cycle, a FFBP PE is assumed to process one interpolation of a range sample per clock cycle. In contrast to the GBP, where high quality (sinc) interpolation is necessary for good projection results, nearest neighbor or linear interpolation is sufficient for the FFBP, to keep the error rate in acceptable boundaries [42]. Therefore, access to a wide range of neighbor samples for interpolation is not required. Additionally, it can be assumed, that the required interpolation positions will grow linear in equidistant steps, and not change as dramatic as it might be possible in the GBP interpolation. Therefore, input buffers of range line length are not necessary and incoming data can be interpolated in line. This only requires small buffers for input data. For common systems, it can be assumed that multiple data words will be transferred from memory in one clock cycle. Figure 4.9 shows how multiple PEs would be synchronized to work on different range samples of the same incoming range line data. In this example, two range lines (apertures) get merged to form one new range line (subaperture) for the subimage.

For the exemplary case of a bus transfer of four parallel samples, Fig. 4.9 shows four parallel PEs, which utilize four times the amount of resources. Each PE is assigned to one section (a,b,c,d) of the streamed input data. Range line R_1 and R_2 are merged to form interpolated line $I_{1,2}$. The problem of this configuration is the time that is required to write back the merged range line. During the transfer, all modules are blocked from processing and have to idle. Although the full data of the range lines are shared with all PEs, a PE only uses a quarter of the data. Therefore, total bus transfers are not reduced because the range line would, in general, have been required only once during the process.

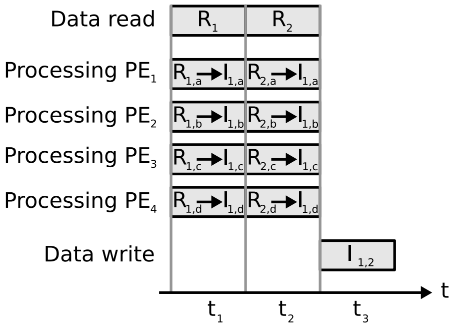


Figure 4.9: Schedule for FFBP PEs processing one range line for one subimage

To emphasize the importance of a reduction of bus transfers, the following example shows how often data is transmitted during the entire FFBP stage. Again the following exemplary dimensions are assumed, raw data of $N_{az} = 1000$ lines in azimuth, each with $N_{rg} = 1000$ echo samples in range. The FFBP runs three iterations $s = [1,2,3]$ with the following setup for subimages $f_{sub} = [16,8,4]$ and aperture factorization $f_{apt} = [2,2,4]$ per stage. The total number of read operations on memory results to Eq. (4.3):

$$Raw_{rd} = \frac{Naz \cdot Nrg \cdot f_{sub}[1] + \frac{Naz \cdot Nrg}{f_{apt}[1]} \cdot f_{sub}[1] \cdot f_{sub}[2] + \frac{Naz \cdot Nrg}{f_{apt}[1] \cdot f_{apt}[2]} \cdot f_{sub}[1] \cdot f_{sub}[2] \cdot f_{sub}[3]}{\lfloor \frac{B_{ww}}{D_{ww}} \rfloor} \tag{4.3}$$

While the total number of write operations on memory results to Eq. (4.4):

$$Raw_{wr} = \frac{\frac{Naz \cdot Nrg}{f_{apt}[1]} \cdot f_{sub}[1] + \frac{Naz \cdot Nrg}{f_{apt}[1] \cdot f_{apt}[2]} \cdot f_{sub}[1] \cdot f_{sub}[2] + \frac{Naz \cdot Nrg}{f_{apt}[1] \cdot f_{apt}[2] \cdot f_{apt}[3]} \cdot f_{sub}[1] \cdot f_{sub}[2] \cdot f_{sub}[3]}{\lfloor \frac{B_{ww}}{D_{ww}} \rfloor} \tag{4.4}$$

For the given example configuration, this results in $208\ M$ (Mega) read operations and $72\ M$ (Mega) write operations. Depending on how many data words fit into one bus word, these numbers are reduced by $\lfloor \frac{B_{ww}}{D_{ww}} \rfloor$. Although this is still few when compared to the amount of read and write operations which are required the GBP, especially read operations should be reduced to keep the busload low.

As depicted in Fig. 4.10 the processing scheme of the subimages for the FFBP forms a tree, where the root is the set of original raw data. $f_{sub}[1]$ sibling (child nodes or subimages) will generate from the root. In the next stage, the generated child nodes become the parents nodes for the next stage, each generating a new set of $f_{sub}[2]$ siblings. This continues until the last stage is reached and the GBP

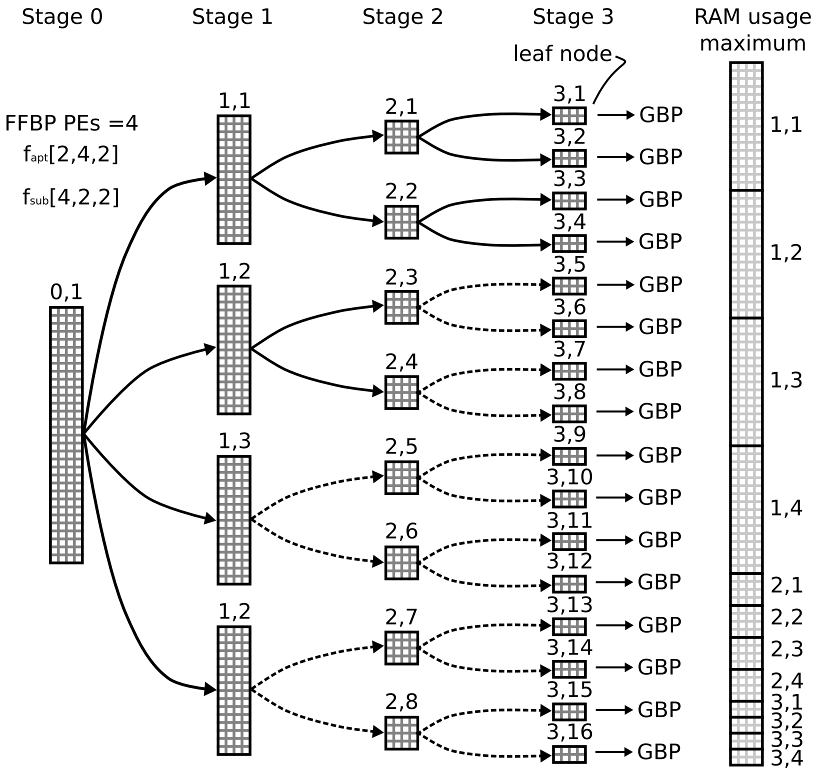


Figure 4.10: FFBP processing tree + memory usage with $s = 3$, $f_{apt}[2,4,2]$, $f_{sub}\,[4,2,2]$

can run on the leaf nodes. Inside this tree, the multiple defined PEs of Fig. 4.9 can be assigned to process siblings of subimages in parallel, instead of processing one subimage in cooperation. Since all siblings of a parent node share the same input data, the data can be broadcasted to all PEs. Another setup is the splitting of PEs to process multiple subimages in parallel. The splitting of PEs does not allow to process a continues stream of incoming data, because every PE needs more time for processing. The entire bus word must be processed before a new bus word can be transmitted. This results in clock cycles in which no traffic on the bus is generated. This schedule is depicted in Fig. 4.11. After the second range line R_2 is sent to the PEs, merging is complete for the first section in every subimage. These chunks (I_a, I_b, I_c, I_d) can be sent to memory during the free bus cycles, to transmit data to the memory. The required resources are the same, but the merging of one line takes

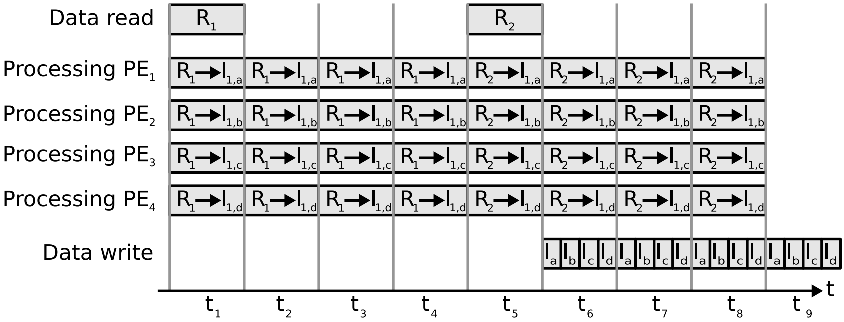


Figure 4.11: Schedule for FFBP PEs processing one range line on multiple subimages

four times longer than before. This is compensated by the four lines merged within this time interval. The distributed merging has two advantages compared to the collaborative merging. First, raw data is shared on the bus, for the given example the read operations are reduced from $208\ M$ (Mega) to $52\ M$ (Mega). The second advantage is the higher bus utilization as part of the merged data can already be sent during merging, only one additional cycle is required to transmit the final samples of the entire subimage. The disadvantage of the strategy shows whenever the amount of generated siblings per stage is smaller than the amount of available PEs. This would result in modules in idle status during a full stage.

Both strategies require full line buffers to merge (accumulate) the input lines. Due to the given advantages, the FFBP PEs are parallelized to perform distributed merging 4.11. As line merging is performed in range direction, this strategy is only possible with configurations ⓔ or ⓕ. This delivers one more argument for configuration ⓕ which is already the preferred configuration to reduce overall busload.

4.5.2.3 Inter loop streaming on module level

So far, the GBP and FFBP modules were discussed separately. For the processing of the GBP, all discussed aspects are valid for the design. For processing the FFBP this is different, as both modules are required in combination to generate a FFBP SAR image. This aspect will be discussed and raises questions about, how the modules should be synchronized. Two basic opposing strategies exist.

The first strategy is successive processing. The entire set of stages would need to be processed until all leaf nodes (subimages) are generated and stored in the RAM. Then the GBP can start loading each subimage from the RAM, process the data and transfer it back to memory. This would ease the synchronization and reduce bus conflicts. But one of the two modules would be stalled for the full runtime of the other module. Resulting in a low hardware utilization rate. This obviously results in the longest possible runtime as long as a simultaneous run would not inflict so many bus hazards, that a massive overhead of stalls is generated. Finally, this strategy causes a peak in memory consumption, as all subimages need to be held in memory before the GBP starts processing.

The second strategy is interleaved processing. To start the GBP, only one subimage of the final FFBP stage is required to be fully stored in memory. To start as early as possible with interleaved processing, the FFBP is set to process only one branch of the processing tree until the first leaf node is reached. This is free both modules, to process at the same time, as long as input data is available for both modules and the bus allows to transfer output data back to memory. Due to the data greedy behavior of both modules either the shared memory or the limited bus interface will reach its limitations at certain points. Then one of the modules is stalled because another module is loading or transferring data from/to memory. In case of slow FFBP processing, the GBP will also stall due to insufficient subimage data. As a result, the strategy requires a fine balancing of available hardware resources. In case of good balancing and synchronization, this strategy allows for the shortest possible runtime due to maximum resource and bus utilization. Additionally, the required memory to store subimages is kept on a stable level, because of the branch processing. This is shown in Fig. 4.10.

The shown example in Fig. 4.10 is optimal for a FFBP module containing 4 PEs. The configuration runs three iterations $s = 3$ (stage 1, 2 and 3). Each iteration (stage) has a individual configuration for merging apertures (aperture factorization) $f_{apt}[2,4,2]$, and a individual configuration for the amount of subimages that are spawned (image factorization) from the image in the previous stage f_{sub} $[4,2,2]$. Stage 0 represents the full raw data as it was received from the SAR system (containing 32 range lines in azimuth direction). This stage contains the root node and serves as the parent node for all siblings in stage 1 (1,1; 1,2; 1,3; 1,4). In stage 1 four subimages $f_{sub} = 4$ will be generated. Since $f_{apt} = 2$ two apertures get merged to form the new subapertures for the child nodes in stage 2. The data of the parent node must be kept in memory until the entire set of siblings for this parent node (stage 2) was generated. This means, that in case the number of PEs is less than the number of siblings (subimages) spawning from one parent, the PEs need to process the data of the parent node multiple times. In case an intermediate parent node is removed from memory before all siblings nodes are processed, a redundant stage 1 run for this parent node is necessary. In the case of too few available memory, this can become a necessary action. Depending on the configuration of the FFBP and how the tree is processed (walked), this can create vast amounts of intermediate data. For example when each subimage in each stage is processed before the next stage is triggered. On the other hand, when enough memory is available, the creation of multiple nodes in the same stage can be beneficial as the data broadcast option to all FFBP PEs is reducing busload. The concept of bus load reduction applies by sharing data during the creation of f_{sub} child nodes. Whether it is better to hold multiple parent nodes in memory, or to process a parent node multiple times, depends on the system resources. Therefore, a system-dependent compromise between memory usage and redundancy must be found.

The shown tree walk of recursive factorization (indicated by the dashed lines) shows a good solution to keep memory usage stable at the level indicated by the bar at the right side. One has to keep in mind that raw data sets of common SAR systems can produce considerable big raw data sets. For example the data set shown in Fig. 1.1 of Frankfurt airport [6], is a block of $16k \cdot 4k$ raw data samples resulting to $64\ M$ (Mega) samples of data. With a sample size of 32 bit one block is roughly 256 MB of data. Creating multiple subimages in the first stage with low aperture factorization will account to intense memory usage. For the given strategy it is sufficient to process every intermediate node only once and to keep a maximum of $\min(PE,\ f_{sub}\ )$ nodes per stage in memory. After the last iteration is reached, the GBP is executed, resulting in a first partial image, so that the raw data of all nodes that are not required for other subimages, can be deleted from memory. All siblings of the same stage, where the parent data is still present in memory, will be processed. After processing all child nodes, also the parent node can be deleted from memory. After a branch is finished the tree is walked recursively until a node with unprocessed child nodes is found. The whole procedure is repeated until all siblings are processed by the GBP. The total amount of memory access is of course not reduced by this (besides the reduction through broadcasting). The combination of stable memory usage, the use of broadcasting data and the faster processing make interleaved processing superior to successive processing and is therefore preferred for hardware implementation.

4.5.3 Intra loop streaming (pipelining)

An important measurement factor for high throughput architectures is the utilization rate of used resources, meaning how high is the percentage of stall cycles compared to processing cycles. This was already discussed for PEs on a higher level. Besides this general concept of a streamlined design on module level (inter loop streaming), optimized streaming on PE level is imperative for full resource utilization and a fast design. For comparison, inter loop streaming is based on the concept of parallel PEs that form of an array processor architecture, to process the same task in parallel on multiple data samples. This is a homogeneous design, since all PEs are atomic on module level and result in equal runtime. But inside of a PE the structure breaks down into buffers and the ARU with several Function Units (FUs) with heterogeneous tasks and varying runtime. The different FUs cover the diversity of arithmetic operations of the GBP. High resource utilization is more challenging on a lower level because of the big gaps in complexity between different FUs, but therefore it can create even greater speedup if done carefully and gaps are balanced. By implication this could mean that available resources be rather used to improve constant data flow through the PEs (intra loop streaming) then for a higher degree of parallelization (inter loop streaming) across PEs.

Without optimization, a PE generates a result per clock cycle in one continuous complex long step. This lowers the throughput of the entire PE as this cycle defines the clock rate of the PE which is considerably low, due to the long processing of the complex step. The clock frequency of a PE is defined as in Eq. (4.5).

$$PE_{cf} = \frac{1}{PE_{pt}} \tag{4.5}$$

Where PE_{pt} is the time a PE requires to project one raw data sample on an image pixel. During this process new raw data samples cannot be feed into the ARU as the single FUs are directly coupled and depend one the result of the preceding FU. PE_{pt} is the sum of all FUs processing delays $FU_{x_{pt}}$ and resolves to Eq. (4.6).

$$PE_{pt} = FU1_{pt} + FU2_{pt} + ... + FU n_{pt} \tag{4.6}$$

The linear processing is very inefficient because only one out of n FUs is active at once while all other parts are stalled as depicted in Fig. 4.12.

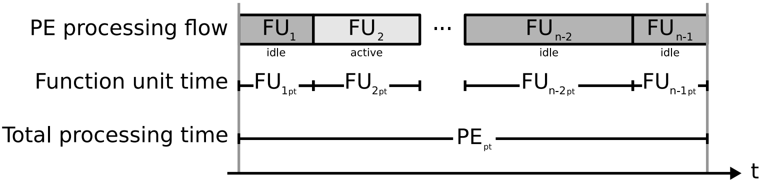


Figure 4.12: PE processing time without optimization technique

This is similar to Fig. 4.6, where only the buffer or the ARU can be active at once. To speed up the processing, buffer resources have to be doubled, to fetch data while processing, because the ARU depends on the access to the whole buffer range. In contrast, to speed up the ARU only an insignificant amount of resources is required by the use of pipelining. The complex process is split up in many shorter cycles and the process is divided into pipeline stages, which are less complex (down to atomic level). This is possible albeit all FUs always depend on their preceding FU, they never require access to all intermediate results of the preceding FU but only to the final result. To allow for parallel processing of the FUs, additional registers RE (buffer with the size of one data word) are injected, so that every FU can store its intermediate result. This allows every FU to act independently and process in parallel to all other FUs. The benefit is a higher throughput for the entire ARU, with only marginal additional resources for the added registers. This is called pipelining and is a standard concept in many processors. For a PE this is depicted in Fig. 4.13.

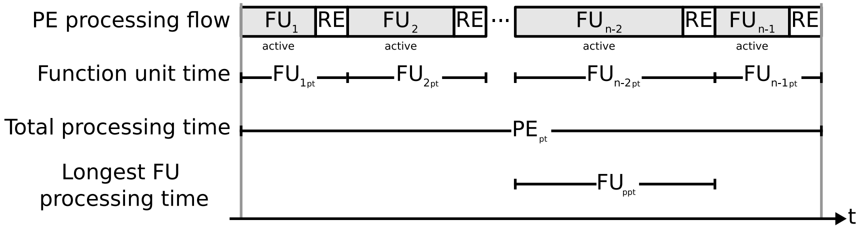


Figure 4.13: PE processing with added registers (RE) for pipeline technique

Pipelining can create high speedup values but implies a rather predictable sequence of instructions, as any jump would create stalls which result in performance drops. The pseudo-code listed for the backprojection algorithms in the preceding section fulfill this requirement in perfection, making them an ideal candidate. Pipelining can be understood as a conveyor belt, where every step of a process is handled by a different station. In a pipeline, the stages are filled only step by step, since the stages with the registered and the injected registers (RE) are clock synchronized. As soon as the pipeline has processed one full cycle and data continues to stream into the PE, all pipeline stages are busy. PE_{pt} is also called the core clock, as it represents the time a data word requires to pass the entire pipeline. In a pipeline of n stages, a data word is processed n cycles. This defines the latency for the processing of one data word. Since the steps are independent, every stage can load a new data word in every cycle. Data words are then sequentially sent through the entire pipeline. From the moment a data word has passed the entire pipeline, on average, every cycle one processed data word drops out of the pipeline. Whereby the slowest stage in the entire pipeline determines the clock frequency of the PE. Furthermore, the time needed to store the intermediate result in the register RE is added to the total processing time of every stage. The final clock frequency of a PE is given by Eq. (4.7).

$$PE_{pcf} = \frac{1}{max(FU1_{pt} + FU2_{pt} + ... + FUn_{pt}) + R_d} = \frac{1}{FU_{ppt}} \tag{4.7}$$

In case of Fig. 4.13, $FUn-2_{pt}$ is the slowest stage and therefore defines FU_{ppt}, as all other stages would need to stall until FU_{n-2} has finished processing. This is why balancing the internal pipeline is not as simple as balancing on PE level, where every PE requires the same time for processing. On ARU level, a multiplication might be one stage while the other stage performs an interpolation. The maximum theoretical speedup PE_s of a pipeline is limited to the number of stages if all stages would require the same processing time. For k data samples passing the pipeline, which is divided into n stages, with a cycle time of FU_{ppt}, the speedup is given by Eq. (4.8).

$$PE_s = \frac{PE_{pt} \cdot k}{n \cdot FU_{ppt} + (k-1) \cdot FU_{ppt}} = \frac{PE_{pt} \cdot k}{(n+k-1) \cdot FU_{ppt}} \tag{4.8}$$

Filling the pipeline takes n stages. As soon as the pipeline is filled, the first data sample has already passed all stages and is processed. The remaining data samples will be processed after additional $k-1$ steps. The less complex the stages are, the higher the possible PE clock. To balance the stages and thereby increase the throughput, pipeline stages can split down to the atomic level. This increases the maximum possible clock frequency, which of course increases resource consumption. The advantage in throughput is only diminished when execution hazards occur. A hazard requires the pipeline to stall or to restart the entire pipeline again. A non-pipelined processor is basically performing an unconditional restart for every instruction. With more stages the chances for hazards in the form of control, input or output dependencies increase. It should, therefore, be adapted to the available periphery and algorithms to reduce such hazards, as they lower the efficiency of the entire pipeline.

4.5.3.1 GBP intra loop streaming

Independent of the chosen configuration, the flow of operations in the GBP is fixed and is localized in the inner loop. The dependency analysis in Section 4.4.1 shows how to split up this flow into several stages, for pipeline processing in general. The stages determine which instructions can be parallelized, as the instructions of a subset do not dependent on each other. To reach the maximum speedup, the possible degree of parallelization should be exhausted. In the case of the GBP, a pipeline has mostly benefits besides the marginal additional resources consumption for registers. Since the algorithm is highly linear, the risk of hazards is only given for the control dependency. To reduce input dependencies, the chosen configuration should allow for a split of the outer loop on different PEs, so that data dependencies do not occur between the data accessed by the PEs. In the case of configuration ①, such input dependencies would already occur with the first level of loop parallelization, due to the simultaneous access on image pixel by the parallel PEs. To streamline the process and increase pipeline efficiency, configuration ③ with additional resources in the form of swap buffers, for raw data and image data, is optimal. This allows for splitting the outer loop so that every PE can process an isolated image line. Instead of creating data dependencies on PE level, a broadcast could be implemented, so that busload even reduces with a growing number of PEs. To optimize the flow, the three existing types of dependencies (input, control, flow) should be addressed individually. Two types of dependencies can be resolved directly in the design.

Input dependencies, created through the concurrent access on the pixel coordinates x and y, and the concurrent access on the range distance δr can be directly solved in hardware. A data word can be sent to an (almost) arbitrary amount of FUs through direct routing. Any input dependency can be resolved like this with few additional resources, as long as the passed data word is short enough. This is contrary to GPP architectures, where the memory architecture, the interface, and the caching strategy influences if a data word is accessible in parallel.

Control dependencies in the GBP only exist in one case, to decide if the currently processed image pixel is covered by the echo of the sensor at the current range line position. A pixel could either be too close to the sensor, so that $\delta r \leq r_{min}$ or to far away so that $r_{max} \leq \delta r$. If any of the conditions is true, further processing is pointless since the current echo will not contribute any information to the current pixel. The consequence would be a pipeline flush which would cause all following FUs to stop processing for the time a new data word passes the given criteria. Given the fact that the implemented FUs would turn to idle mode, as they could not be allocated to another PE, the control dependency decision could be shifted to the end of the pipeline, to avoid a flush, continue processing and instead to zeroize the result of the projection at the very end. Since all required data is held in local buffers the processing would not cause additional busloads. The final image line would be written back to memory anyway. This would not reduce the number of required resources to resolve the control dependency, but it would not become obsolete to invest resources to partly stall the pipeline. Other platforms like GPPs have the advantage of branch prediction and dynamic resource allocation in order not to process useless data. Nevertheless, the chances of running into such conditions can be reduced by a reasonable choice of the image origin, to the data origin. When the area that is covered by radar echos is smaller then the image area, the question of, if a pixel was illuminated by the sensor echo, will result in false often. To achieve a sharp image in all sections, the dimensions of the area covered by sensor echos should be greater than image dimensions. Both scenarios are depicted in Fig. 4.14. In case of a smaller echo area, the outer image pixel can be assigned to an echo sample by a distance r. In case of a bigger echo area, all image pixel can be assigned to an echo sample.

Flow dependencies are the major constraint for parallel processing of the GBP. They cannot be resolved but the instruction within one stage described in Section 4.4.1 can be performed in parallel. Further parallelization is only possible in case a second full-featured pipeline is implemented in a second ARU. This is only applicable in case the BRAMs, used to implement line buffers for raw data and image data, offer more bandwidth than needed by a single ARU. This is discussed further in the GBP chapter. Nevertheless, the central feature of a pipeline is the increase of throughput through parallel stages. High throughput asks for a high pipeline frequency PE_{pcf}, which is defined by the longest critical path FU_{ppt} of all pipeline stages. Simple instructions like a simple addition mark the lower boundary of a critical path and therefore mark the upper limit for PE frequency and throughput. For the given pipeline, the more

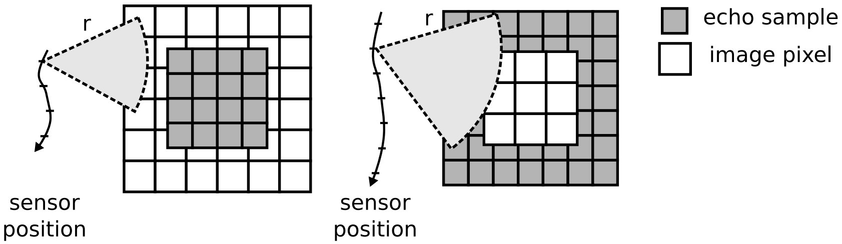


Figure 4.14: Overlap of raw data and image grid

complex instructions, like square root or interpolation, mark the upper boundary and therefore need to be chopped to smaller stages or sub-pipelined, to shorten the critical path. This process is, of course, limited again due to other flow dependencies.

To efficiently map an instruction on an FPGA, regarding the consumed resources and the critical path, it is expedient to consider approximations instead of the exact mathematical solution. To gain an advantage, the approximation should be superior in speed (degree of parallelization and/or critical path) or resource consumption. Operations like division and multiplication are of course possible and also common to implement on FPGAs (some are even specialized on this), but they are always more consuming in terms of resources and processing time. If an approximation can be broken down to use mainly very basic operations, like additions and shifts, this is advantageous, as they can be implemented with few resources and therefore feature small propagation delay (critical path). For an efficient implementation, suitable approximations should be discussed for the square root calculation, interpolation, polar format transform, and complex multiplication.

4.5.3.2 FFBP intra loop streaming

The general mechanisms and benefits of pipelining were already explained for the GBP. These mechanisms are also applied to the FFBP PE to gain a maximum in clock frequency and parallelism. Although the flow of operations is mostly fixed in the inner loops of the three different parts of the FFBP algorithm, the chosen configuration can reduce certain dependencies. The prior analysis prefers configuration ⓕ for FFBP implementation due to the given reasons. The dependency analysis in Section 4.4.2 shows how to split up the flow of the three different algorithms parts into several stages for pipeline processing in general. Since it was already explained in detail in the Section 4.5.3.1, how certain dependencies can be resolved, this section will not get into the details of these aspects and only draw a sketch if and how pipelining should be applied for each part.

Based on Algorithm 4.13, the calculation of subimage centers can be split into six subsets, three for the outer loop instructions and three for the inner loop instructions. The outer part is only performed once per stage s, wherefore no additional resources should be invested to speed up processing. The whole inner loop could be parallelized to compute subimage centers faster if required. As this is only required rarely, this part might not be parallelized at all. Algorithm 4.13 forms a single subset for inner and outer loop instructions each. Parallelization should be performed on the outer loop level to compute subaperture positions faster. But similar to subimage center computation, it is required rather seldom compared to the core operation of merging subapertures in Algorithm 4.12. The actual compute-intensive work, in terms of massive interpolation, to create subimage sample data is performed in this part of the FFBP. As stated in Section 4.4.2, this can be split into thirteen consecutive subsets if maximum speedup must be reached. It was also mentioned that a set of instructions (line 4,5,6,7,11,15,21,22 of Algorithm 4.12) need to be analyzed in more detail, in order to understand the optimal design for an efficient implementation. Instructions 21 and 22 are the same as in the GBP. The analysis of these instructions and possible approximations is presented in Section 5.1. The same applies to the first instructions until line 16, which address mainly vector operations, like forming a vector sum or obtaining the scalar product. The basic goal of the instructions is to find the distance between a specific range cell of the old aperture and the antenna position of the new subaperture. This is done to find the corresponding range cell in the old aperture. The calculation is performed, based on the law of cosine. The entire range calculation (all instruction until the control instruction in line 16) can be substituted with a approximation (see Section 5.1). A major benefit of the approximation, besides the simplification, is that the general characteristics of a SAR setup, allow to generalize the whole computation of range cell-specific range computation, by dissolving the actual dependency on the range cell position. This allows moving the entire block of instructions up to line 16 out of the actual loop construct. The result is that the range value can be processed once for the whole range line. A comparison between the versions with and without approximation is shown in Fig. 4.15.

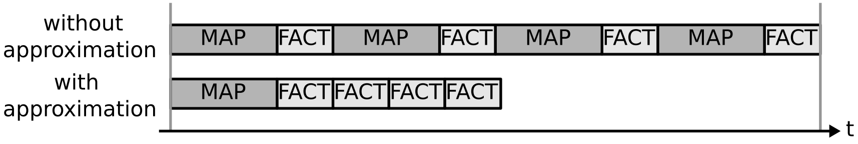


Figure 4.15: Runtime comparison with/without range approximation for a FFBP line

In Fig. 4.15 an exemplary range line with four samples is interpolated or factorized (FACT). Without approximation, every interpolation would require the range cell to be mapped (MAP) to the corresponding distance. Compared to the linear interpolation of one cell, the mapping process requires much more time. With the range approximation, the entire range line can be interpolated with the same range value. Since common range lines lengths in SAR data sets can reach up to multiple thousands of samples, the saving in processing time is massive. To speed up processing, the pipelines can be fully separated. As mapping is now a process that is required less often, it can be merged with the other parts of the FFBP algorithm, which are processed less often. A compound of steps is formed (subimage center calculation, subaperture position calculation, range calculation). Because this compound manifests a rather unbalanced workload when compared to the interpolation part, a method to uncouple the actual parameter calculation from the interpolation stage should be developed. One way to do this is a loose coupling over RAM memory as depicted in Fig. 4.16.

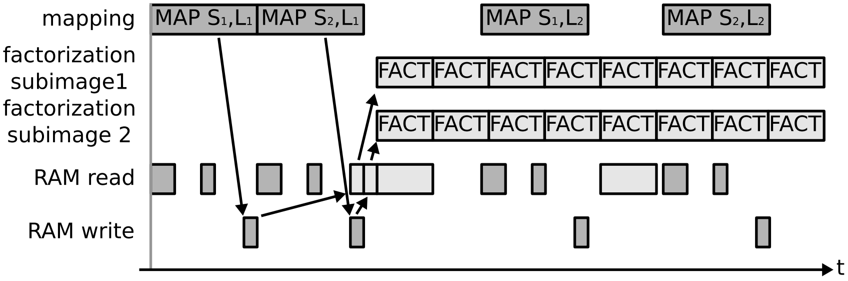


Figure 4.16: FFBP with separated mapper and factorization for two subimages

The schedule shows the processing of the first line (eight samples in range direction) of two subimages of the same parent. Before factorization in the according PE can start, which from now on will be referred to as **factorizer**, the entire calculations for both subimages need to be finished in the according PE, which from now will be referred to as **mapper**. The mapper loads several constants, position data and the FFBP configuration from RAM memory. After processing is finished the range data required from the factorizer is stored into RAM memory. As soon as the sets for both subimages are processed, both modules can start by loading the range data and sample data from RAM and start processing. The factorizer will load chunks of sample data from RAM in equidistant time frames to continue processing. In between those time frames, the bus might be in an idle state. The mapper can use these free time-slots to process upcoming range data sets for the upcoming lines. It is called loose coupling, as the modules of course still depend on a quite linear schedule, but the time frame is wider and therefore fewer restrictions apply. By introducing a distributed memory on PE level, which holds all range data sets and further configuration settings, this principle can be optimized as depicted in Fig. 4.17.

An internal distributed memory is accessed faster and can provide a wider interface if necessary. As only parameters are stored, it can be smaller by orders of magnitude when compared to the external RAM. The configuration memory, which from now will be referred to as **config mem**, helps to reduce the busload, whereby the development

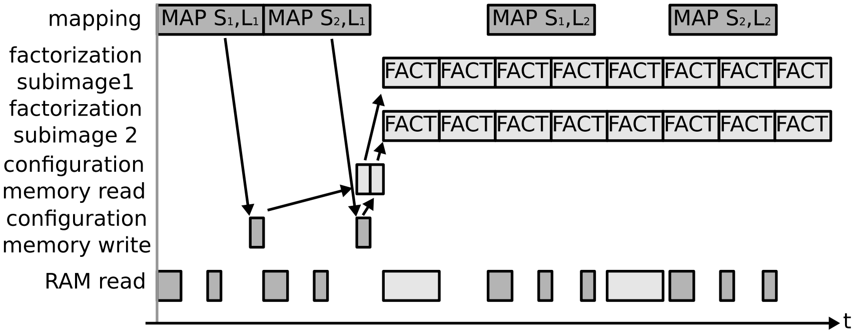


Figure 4.17: FFBP with separated mapper, factorization and configuration memory

of hot spots is reduced. Additionally, bus timing is improved due to less alternating access patters. With a decoupled setup, the mapper can be implemented in a resource-saving design, since a slow mapper will not cause any delays for the factorizer, as long as it is guaranteed that range data sets will be ready on time. Saved resources might be used to implement more factorizer PEs. But it has to be mentioned that only a limited amount of factorizer PEs is useful. As the modules are on the same stage to reduce busload, only $max[f_{sub}]$ are useful. Depending on the FFBP setup, it can only be guaranteed that a minimum amount of two factorizer modules are constantly used.

This results in a setup of two different PEs with separated pipelines, which are loosely coupled over a distributed memory. The pipeline of the factorizer is shortened by this to six subsets, corresponding to six stages. The mapper has to be discussed based on the applied approximation presented in Section 5.1.

5 Hardware implementation

The detailed implementation of the concepts discussed in Chapter 4 are explained in the following. This includes the basic signal processing blocks, which are partly used for the Global Backprojection (GBP) and the Fast Factorized Backprojection (FFBP) architecture. Additionally, the concept for the integration of the entire architecture, in a framework to connect to the outside periphery, is explained. All decisions are driven by the requirement to combine and fulfill the given constraints at the best possible rate and to reduce processing time to a minimum, by implementing a highly efficient and resource exhaustive architecture. The actual implementation can differ slightly from the conceptual phase as the approximation of the basic signal processing blocks simplify certain steps.

5.1 Basic dedicated signal processing function units

Besides the basic arithmetic operations, time-domain Synthetic Aperture Radar (SAR) algorithms rely on three different classes of more costly operations. These operations include trigonometric functions, interpolation, and complex-valued functions. As the algorithms mainly based upon these core operations they should be implemented efficiently. Efficiency primarily means fast and/or resource-saving implementations. General Purpose Processors (GPPs) perform a great part of operations in single or double-precision floating-point format. While this format allows for high precision and high dynamic range due to the adapting decimal point, it is resource (area) costly. This is because floating-point arithmetic uses fixed-point logic at its core, but resources are added to manipulate the data word before and after (normalize) core processing. While implementing operations in floating point is, of course, possible on Field Programmable Gate Arrays (FPGAs), however, it is more efficient to implement an operation in fixed-point format. Fixed point operations are also less resource consuming because FPGA designs can be adjusted to the required accuracy bitwise. In most cases, it is sufficient enough to represent a result in a fixed-point number representation also because an algorithm might not even require an arithmetic operation to be highly precise. To use fixed point instead of floating-point, the data path in a Function Unit (FU) might have to be adjusted in width. This also includes the required dynamic range of every step, in order not to truncate intermediate results unintended. Beside preserving resources, efficiency is also bound to throughput. Wherefore the FU should allow for parallelism and sub-pipelining.

5.1.1 Range distance approximation

The processing of subapertures for the FFBP algorithm includes the interpolation of range lines. Each range sample must be interpolated in order to be mapped on the new range sample position as depicted in Fig. 2.7 of the artificial formed subaperture which is aligned to the new subimage center c. To obtain the actual interpolation index, the range distance d_{ns} between the sample (range cell) of the old aperture and the antenna position of the new subaperture must be calculated. This is done with the help of the law of cosine (Eq. (5.1)), where the length of the third side of a triangle can be calculated if one knows the lengths of the two remaining sides and the angle between them:

$$c = \sqrt{a^2 + b^2 - 2ab\cos\phi} \tag{5.1}$$

To simplify the calculation a Taylor series can be used. A Taylor series [78] represents an arbitrary function as a sum of terms, which are calculated based on values of the same function derivatives at a single point a. This is an approximation, as a Taylor series converges only in a certain interval around this point a. The universal formulation of a Taylor series for $f(x)$ that is differentiable infinitely at a real or complex number a is a power series and results to Eq. (5.2).

$$Tf(x;a) = f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + ... \tag{5.2}$$

A more complex notation is given in Eq. (5.3) where $n!$ denotes the factorial of n and $f^n(a)$ denotes the nth derivative of f.

$$Tf(x;a) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n \tag{5.3}$$

The higher the order of this power series, the higher the accuracy around point a. Nonetheless, the power series can be also truncated after any term. The function $rd(d_{ns})$ (Eq. (5.4)) calculates the range distance with the law of cosine.

$$rd(d_{ns}) = \sqrt{d_{on}^2 + d_{ns}^2 - 2d_{ns}d_{on}\cos\phi} \tag{5.4}$$

The range distance $rd(d_{ns})$ results to Eq. (5.5) when it is approximated with a Taylor series truncated after the first term. Instead of the actual calculation of the distance to the range cell, the range distance function rd is developed around the point of distance between the old aperture and the new subimage center d_{oc}. The subimage center is assumed to be the optimal development point, as the error at this point is minimal in average, since it represents the average distance for all range cells in the subimage.

$$rd(d_{ns}) \approx rd(d_{oc}) + \frac{rd'(d_{oc})}{1!}(d_{ns} - d_{oc}) \tag{5.5}$$

Taking d_{oc} as a development point, the outer function for $rd(d_{oc})$ is given with $g(x) = \sqrt{x}$, while the inner function results to $h(x) = d_{on}^2 + d_{oc}^2 - 2d_{oc}d_{on}\cos\phi$. The first derivation of the outer function results to $g'(x) = \frac{1}{2\sqrt{x}}$, while the first derivation of the inner function results to $h'(x) = 2d_{oc} - 2d_{on}\cos\phi$. Applying the chain rule $f'(x) = g'(h(x)) \cdot h'(x)$ the first derivation of $rd(d_{oc})$ results to Eq. (5.6).

$$rd'(d_{oc}) = \frac{2d_{oc} - 2d_{on}\cos\phi}{2\sqrt{d_{on}^2 + d_{oc}^2 - 2d_{oc}d_{on}\cos\phi}} \tag{5.6}$$

Inserting Eq. (5.6) into Eq. (5.5) results to Eq. (5.7).

$$rd(d_{ns}) \approx \sqrt{d_{on}^2 + d_{oc}^2 - 2d_{oc}d_{on}\cos\phi} + \frac{2d_{oc} - 2d_{on}\cos\phi}{2\sqrt{d_{on}^2 + d_{oc}^2 - 2d_{oc}d_{on}\cos\phi}}(d_{ns} - d_{oc}) \tag{5.7}$$

According to the law of cosine, the substitution of d_{ns} with d_{oc} results to the range distance between the new aperture and the subimage center. This corresponds to the vector d_{nc} in Algorithm 4.14, wherefore $\sqrt{d_{on}^2 + d_{oc}^2 - 2d_{oc}d_{on}\cos\phi}$ equals d_{nc}. Inserting d_{nc} in Eq. (5.7) plus cutting out the factor 2 in the fraction, results to a Taylor approximation for the range distance rd which is developed around the point d_{oc}. The result can be expressed as Eq. (5.8)

$$rd(d_{ns}) = d_{nc} + \frac{d_{oc} - d_{on}\cos\phi}{d_{nc}}(d_{ns} - d_{oc}) \tag{5.8}$$

Equation (5.8) still contains four vectors and requires the calculation of $\cos\phi$. Since we are in a SAR scenario, it can be assumed that the distance in range rd is by orders of magnitudes greater then the distance d_{on} between the old and new antenna positions. This necessarily means that also all vectors d_{nc}, d_{ns} and d_{oc}, are far greater than d_{on}. Therefore, d_{on} can be assumed to tend to zero, which allows for the simplification of Eq. (5.9)

$$d_{nc} \approx d_{oc} - d_{on}\cos\phi \tag{5.9}$$

To prove this behavior, a second Taylor approximation is taken as a proof. For the second approximation, $rd = d_{oc}$ will stay constant while d_{on} will vary and can be set to 0. This results to Eq. (5.10).

$$d_{nc}(d_{on}) = d_{nc}(d_{on} = 0) + \frac{d'_{nc}(d_{on} = 0)}{1!}(d_{on} - 0) \tag{5.10}$$

The first derivation of $d_{nc}(d_{on})$ is given in Eq. (5.11).

$$\begin{aligned} d'_{nc}(d_{on}=0) &= \frac{2d_{on}-2d_{oc}\cos\phi}{2\sqrt{d_{on}^2+d_{oc}^2-2d_{oc}d_{on}\cos\phi}} \\ &= \frac{-2d_{oc}\cos\phi}{2\sqrt{d_{oc}^2}} \\ &= \frac{-d_{oc}\cos\phi}{d_{oc}} \\ &= \frac{-d_{oc}\cos\phi}{d_{oc}} \\ &= -\cos\phi \end{aligned} \tag{5.11}$$

Inserting Eq. (5.11) into Eq. (5.10) results to Eq. (5.12).

$$d_{nc}(d_{on}) = d_{nc}(d_{on}=0) + \frac{-\cos\phi}{1!}(d_{on}-0) \tag{5.12}$$

With $d_{nc} = \sqrt{d_{on}^2+d_{oc}^2-2d_{oc}d_{on}\cos\phi}$ and setting $d_{on}=0$, this results to $d_{nc}=d_{oc}$. Substituting d_{nc} with d_{oc} gives the final results of Eq. (5.13).

$$\begin{aligned} d_{nc}(d_{on}) &= d_{oc} + \frac{-\cos\phi}{1!}(d_{on}-0) \\ &= d_{oc} - d_{on}\cos\phi \end{aligned} \tag{5.13}$$

This proves Eq. (5.9) under the given assumptions and allows to simplify Eq. (5.8) to the final approximation result given in Eq. (5.14).

$$\begin{aligned} rd(d_{ns}) &= d_{nc} + \frac{d_{nc}}{d_{nc}}(d_{ns}-d_{oc}) \\ &= d_{nc} + (d_{ns}-d_{oc}) \end{aligned} \tag{5.14}$$

This is a significant simplification in terms of complexity, as all lines from 4-15 in Algorithm 4.14 can be replaced with the computation of Eq. (5.14). This can be performed by simple adders.

5.1.2 Square root and trigonometric functions (CORDIC)

Time-domain-based SAR algorithms require two arithmetic operations in consequence of the varying distance between the signal source and the projected area. Those two operations are:

1. Square root calculation
2. Sine and cosine calculation (trigonometrical functions)

The square root calculation is required to determine the direct distance of δr between the sensor and the pixel position. This distance is required to allocate the corresponding range sample i_{rg} of a sensor echo i_{az} to an image pixel (i_x, i_y). Based on δr and the phase correction constant pc_c (which depends on the radar mid-frequency), a correction angle ϕ_{corr} is calculated. This factor is used to adjust the phase and amplitude signal of the complex-valued range sample. The correction factor can be interpreted as a complex number with a magnitude of 1 and a phase angle of ϕ. This polar coordinate representation must be transformed to a Cartesian coordinate representation to perform the raw data phase correction. This transformation delivers the sine and cosine component of ϕ_{corr} to form the complex-valued correction factor $e^{j\phi_{corr}}$ according to Euler's formula as given in Eq. (5.15).

$$e^{j\phi_{corr}} = \cos(\phi_{corr}) + j\sin(\phi_{corr}) \tag{5.15}$$

A complex multiplication performs the actual phase correction. The computation of these functions is usually costly as the used mathematical operations are not designed for dedicated hardware implementation. Well know methods for the calculation of the square root are the Babylonian method [79], Bakhshali method [80], Vedic duplex method [81], Goldschmidt's algorithm [82] and Taylor series [83]. Common methods to obtain the sine and cosine are the use of tables. A Taylor series can also be combined with tables and polynomial or rational approximation such as Chebyshev [84] or Padé [85]. The disadvantage of implementing such methods in hardware is the extensive use of costly multiplications and/or the significant amount of required memory (for table operations).

An alternative for computing both functions exists in the form of an iterative algorithm named Coordinate Rotation Digital Computer (CORDIC), developed by Volder in 1959 [86] and extended by Walther in 1971 [87]. This is especially interesting for devices lacking in multipliers since the algorithm mainly uses addition, subtraction and shift operations, combined with small lookup tables. The basic algorithm is based on the principle of iterative vector rotation to transfer polar coordinates (R,ϕ) to Cartesian coordinates (x,y) (rotation mode) and vice versa (vectoring mode). Either the input is a vector with magnitude R, which is rotated to reach the angle ϕ, resulting in x and y, or the input is a vector at x,y and is rotated to the abscissa, resulting in magnitude R and angle ϕ.

But it is not trivial to directly rotate a vector by an arbitrary angle, because of the values for sine and cosine. Instead of the direct rotation, the idea of CORDIC is to use predefined angels for the rotation. The angles are precomputed and stored in a Look Up Table (LUT). Basically, the first angle for rotation is 45°. Following rotations ideally (but not necessarily) always reduce the rotation by fifty percent. As this would require multiplications, the set of angles is chosen in a way that the multiplication can be substituted with a shift to the right (division by two). But the set of chosen angles must allow for a vector to be rotated by a full quadrant maximum. The total angle of rotation is then constructed from the sum of small rotations, whereby each rotation increases the convergence gradually. The direction of rotation

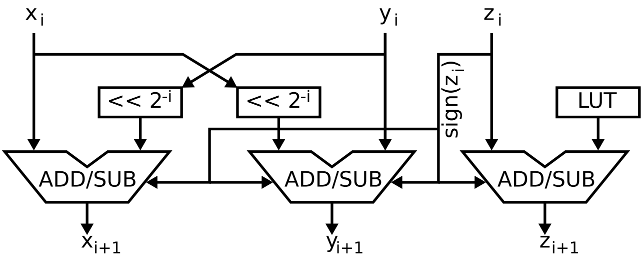


Figure 5.1: Block diagram of a single CORDIC unit FU_{CO}

(clockwise and/or counter-clockwise) is determined by the current vector quadrant. If the vector drops from quadrant I to quadrant IV, so below the abscissa, rotation direction is inverted. The CORDIC can realize different operations, depending on the input and initial values, the chosen coordinate system (circular, hyperbolic or linear coordinate) and the CORDIC mode (vectoring, rotation). A full overview of the possible operations is given in [88]. A factor K is required to correct the final result, which is the only multiplication required in the whole algorithm.

A block diagram of a single Cordic Element (CE) is depicted in Fig. 5.1. Table 5.1 lists how input ports need to be connected to receive the required results on the output ports. An additional control block is necessary to alter how the CORDIC is wired to set the appropriate modes and initial values.

A single CE requires a small LUT to store the precomputed angles, three Arithmetic Logic Units (ALUs) to perform the actual addition or subtraction and two shift register to manipulate the argument of the previous calculation. The shift registers need to be able to perform a variable shift depending on iteration count i. The ALUs need

Table 5.1: Required CORDIC operation modes

performed operation	coordinate system	selected mode	initial values	output values
$\cos\phi$ $\sin\phi$ $\tan\phi$	circular	rotation	$x_i = 1$ $y_i = 0$ $z_i = \phi_{corr}$	$x_{i+1} = \cos\phi_{corr}$ $y_{i+1} = \sin\phi_{corr}$ $z_{i+1} =$position
cartesian to polar $\sqrt{a^2+b^2}$ $\tan^{-1}(b/a)$	circular	vectoring	$x_i = a$ $y_i = b$ $z_i = 0$	$x_{i+1} = \sqrt{a^2+b^2}$ $y_{i+1} =$position $z_{i+1} = \arctan(b/a)$

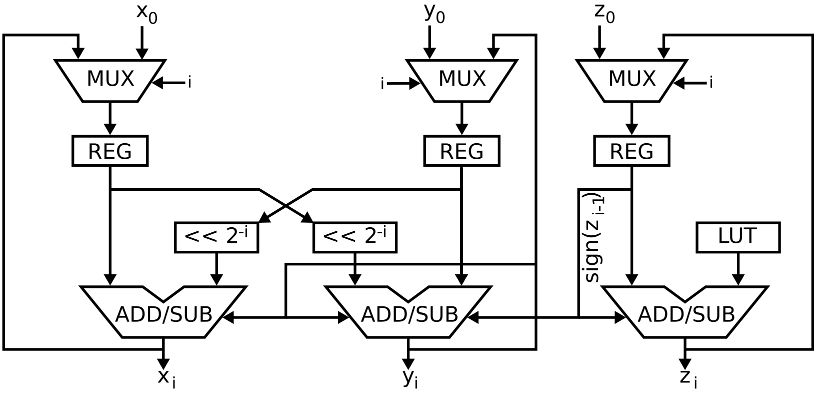


Figure 5.2: Block diagram of a recursive CORDIC unit FU_{COR}

to be controllable to perform a subtraction or addition depending on the extracted sign of z_i. To perform a sequence of iterations, the CE is wired in a recursive manner as depicted in Fig. 5.2. Only two types of additional hardware elements are required for the single CE to work in recursion. Three multiplexers (MUX) are required for flow selection during iterations. The start values are feed to the CORDIC in the first iteration. For every consecutive iteration, the multiplexer feeds the intermediate results of the previous iteration into the current iteration. Also, three registers (REG) are required to hold the intermediate values to wait for the next ALU cycle to begin. One CE in a loop only allows for a throughput of one result every i cycles. Since a CE only consumes few resources it is therefore attractive to implement the module in a pipeline as shown in Fig. 5.3.

In a pipelined CORDIC implementation with i CEs, a initial latency of i cycles exist for the first result. After i cycles, one result per cycle is processed on average. The resource consumption of a recursive CE does not multiply with i for the pipelined version. The LUT is only required once (hard-wired to every stage) and shift operations do not need to be variable (hard-wired shift). Multiplexers are not required at all.

As an example for the quality of a CORDIC the error for sine and cosine approximation is evaluated in Fig. 5.4. As already shown in [73], the absolute error ε_{max} (which represents the worst-case during processing) decreases with the number of iterations and the bit width of the data word width dw_w (wordlength). The trade-off between approximation precision and resource consumption leads to an operation point for the final hardware configuration. As presented in [89], each iteration roughly allows for an additional accuracy of 1 bit in the fraction part. With a wordlength of 16 bit and 12 iterations, the error level is almost dropping below 10^{-3} and is therefore comparable with polynomial approximation.

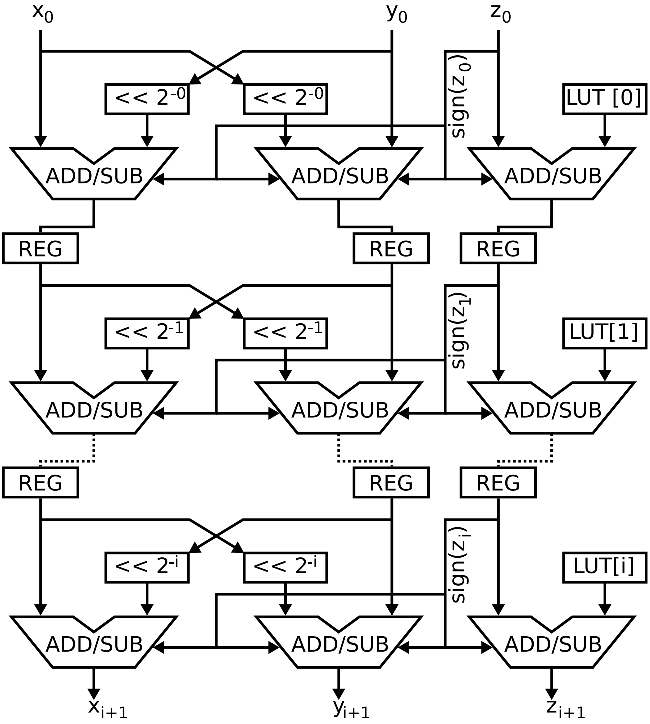


Figure 5.3: Block diagram of a pipelined CORDIC unit FU_{COP}

A pipelined CE with the required accuracy would result in: $12 \cdot 3 \cdot$ ALUs, $12 \cdot 3 \cdot$ registers, one LUT with at depth of 12 words, and a considerable amount of resources for routing. The fact that multiple iterations increase the delay is not relevant as the entire processing of the GBP follows a pipelined layout with higher initial latency. Besides this, every other method would result in long latency processing as well, since multiplications usually consume more time then additions/subtractions and shifts. Since the longest critical path defines the clock frequency of the entire design, it is even preferable to implement many simple elements than a few complex elements. This is important when registers are injected to form a pipeline. As the critical path is defined by the most complex element, simple elements allow for a significant shortening of the critical path of the single element, which directly translates into an increased throughput. Additionally, the CORDIC allows for direct computation of the euclidean magnitude $\sqrt{a^2+b^2}$, which is required for the three-dimensional distance calculation of δr. All these pros, make the CORDIC a preferable algorithm for a dedicated SAR implementation.

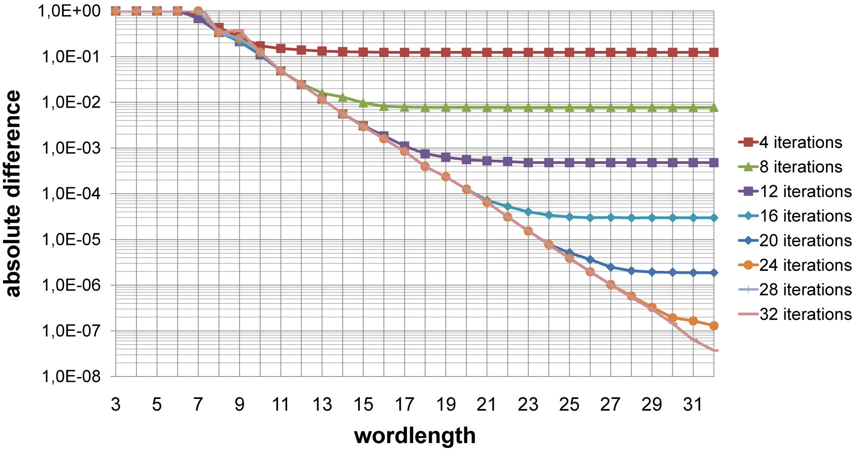


Figure 5.4: Evaluation of CORDIC error for sine and cosine [73]

5.1.3 Interpolation

In the GBP algorithm, the distance vector δr between the image pixel (x,y) and the antenna position (i_{az}) defines the position of the range sample i_{rg}, which covers the current image pixel. Most likely, the distance vector will not align with the equidistant sample positions of the range samples. Interpolation is required to create this off-grid (intermediate) echo data. For the FFBP algorithm a similar problem exists due to the off-grid antenna positions created for the subapertures. As the raw data is the base for a sharp image, the interpolation method must sustain a certain quality, to not create errors which would amplify during processing.

5.1.3.1 Evaluation of interpolation methods

Literature provides a broad set of interpolation methods which differ in complexity and accuracy. Starting from the simple nearest neighbor and linear interpolation (curve fitting among limited spatial points), up to complex polynomial interpolation (finding the polynomial of the lowest possible degree that passes through all points). The interpolation method must also fit to the data that is interpolated. According to [20] a common technique to interpolate the band-limited SAR data, is by the use of ideal low pass filters (also called sinc filters). A sinc filter removes all frequency components above a given cutoff frequency, without affecting lower frequencies. The name originates from the impulse response in the form of a sinc function in the time-domain, while its frequency response is a rectangular function. This behavior can only be approximated, as the ideal version has an infinite impulse response. As a consequence, the sinc filter kernel is often tapered (windowed) when implemented. One way to approximate a sinc filter is by Finite Impulse Response (FIR) filters which use linear convolution after Eq. (5.16).

$$y[n] = \sum_{k=0}^{T-1} h[k] \cdot x[n-k] \tag{5.16}$$

A T tap wide window $h[k]$ slides across the incoming data $x[n]$, creating a sum of k weighted input samples per output sample $y[n]$. The weighting function is characterized by the number of taps and the coefficients c used for each tap. The amount of t coefficients affects the slew rate of the filter. They are precomputed and stored in a LUT for direct access. For interpolation, these coefficients need to be ordered in $S-1$ separate sets, where every set represents a sub-sample position between two discrete samples. The number of taps and sets form a trade-off condition. This condition must be evaluated based on the interpolation quality and resource consumption. While more sets increase the amount of required LUTs, more taps increase the required amount of resources for Multiply-Accumulate (MAC) operations and LUTs. The final amount of LUTs results from the product of taps T and S.

The quality of different sinc setups was already evaluated in [90] based on $f_r = \frac{f_{max}}{f_N}$. Where f_{max} is the maximum frequency of the radar signal bandwidth B, and f_N is the Nyquist frequency for this signal. $f_r > 1$ represents a undersampled radar signal, while $f_r < 1$ represents oversampling. The evaluation compares sinc, nearest neighbor and linear interpolation in floating-point format. It shows, that a sinc filter with a setup of $T = 8, 16, 32$ and $S = 64$ deliver good results (mean error) for all ratios $f_r < 1$. Within the same range, simple interpolation methods lack in quality due to less oversampling when f_r is approaching 1. Since a setup with $T = 8$ shows good results, it is preferred in the following study, as this will reduce the required memory depth $(T \cdot S)$. Additionally f_r is assumed to be close to 1, as this represents a realistic SAR system setup.

To justify the use of the sinc interpolation, the sinc setup of $(T = 8; S = 64)$ is compared in Fig. 5.5 with more complex methods of spline interpolation and ideal sinc interpolation. The evaluation is based on a random simulated and time-discrete sampled baseband limited waveform. As a comparison to [90], linear and nearest-neighbor interpolation results are compared as well. Every interpolation is performed in floating-point number format and is based on a set of 16 sampling points. The chosen metric for quality evaluation is the Root Mean Square (RMS). The relative error $y_{err_{rel}}$ is defined as absolute difference of the interpolated sample y to the ideally interpolated sample ref divided by the maximum absolute value of the ideally-interpolated signal ref. The values given in dB are calculated as $20 \cdot \log_{10}(y_{err_{rel}})$.

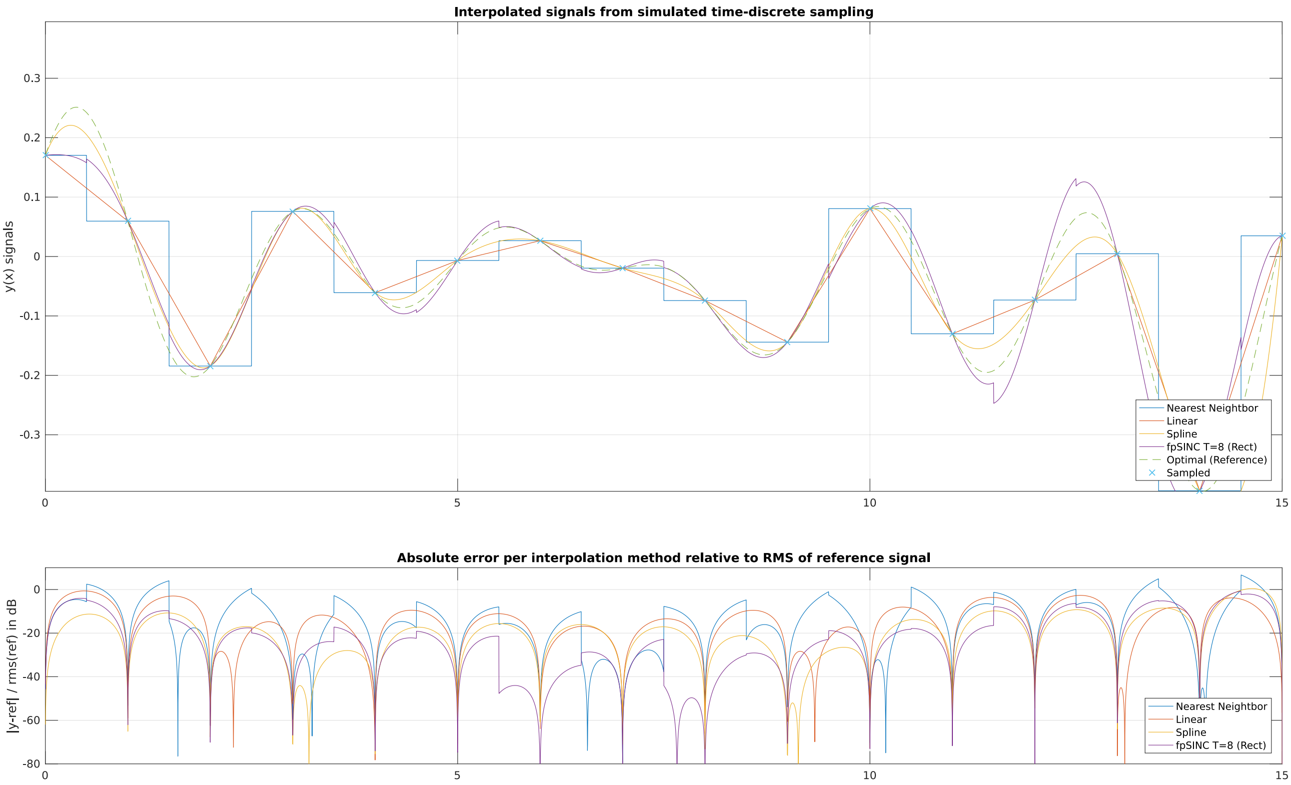


Figure 5.5: Evaluation of interpolation methods on random band limited signal

Table 5.2: Average, median, maximum error rates for interpolation methods in dB

interpolation method (setup)	average error in dB	median error in dB	maximum error in dB
nearest neighbor	-21.24	-24.90	-3.28
linear	-24.83	-28.04	-10.65
spline	-31.52	-35.21	-11.80
ideal sinc	-57.27	-65.49	-26.80
sinc (T=32;S=64)	-44.77	-49.93	-27.17
sinc (T=16;S=64)	-42.26	-44.62	-25.16
sinc (T=8;S=64)	-38.11	-41.35	-20.91
sinc (T=32;S=32)	-40.38	-42.99	-25.69
sinc (T=16;S=32)	-39.08	-41.31	-23.37
sinc (T=8;S=32)	-36.32	-38.71	-19.71
sinc (T=32;S=16)	-37.08	-39.89	-22.69
sinc (T=16;S=16)	-36.29	-38.68	-22.47
sinc (T=8;S=16)	-34.43	-36.40	-20.07

The evaluation confirms that linear and nearest-neighbor interpolation is not applicable for SAR echo data. The main result is that on average, a more complex (resource costly) spline interpolation is not superior to a sinc interpolation. While the spline interpolation tends to undershoot, the sinc interpolation tends to overshoot. The relative error $y_{err_{rel}}$ for the spline and the sinc interpolation are comparable. Often the relative error for sinc interpolation is even below the relative error for spline interpolation. Since the results in [90] only compare setups with $S = 64$, Table 5.2 lists the values for all permutations of sinc setups for $T = 8, 16, 32$ and $S = 16, 32, 64$ as well as all other mentioned interpolation methods.

The results in Table 5.2 confirm, that reducing the sets S can be compensated by increasing the taps T and vice versa. Additionally, the results show, that although a sinc setup of $T = 8;\ S = 16$ shows the lowest performance between all sinc permutations, the results of this configurations are still comparable with the performance of spline interpolation. As this setup consumes the fewest resources between all permutations, it is preferred for implementation. To be able to adapt to different SAR signals, the implemented interpolation Processing Element (PE) should allow the adjustment of either the filter behavior (by changing the number of taps) or the quantization error for sub-sample positions (by changing the number of sets). As both parameters adjust the quality, while changing S only affects the memory depth, this parameter is selected to be variable in the PE.

All presented results base on filter coefficients $c \in S$ formatted in floating-point number representation. This allows for high precision and wide dynamic range. But since fixed-point number format is used for PE implementation, an evaluation of the required fixed-point data width for coefficient quantization is mandatory. This gap from floating to fixed-point is closed in [90]. The results show that reducing the quantization width to 14 bit has only a marginal impact on the interpolation quality and can pin general keep up to the floating-point number representation. A further reduction would reduce memory depth but also reduce interpolation quality noticeably. Based on all presented results, a filter setup of 8 taps (T) and 16 sets (S) with a coefficient quantization of $q_w = 14$ bit is chosen for the final interpolation PE implementation.

5.1.3.2 Implementation of interpolation

The preferred configuration ($T = 8$, $S = 16$, $q_w = 14$) for the interpolation unit FU_{INTP} is depicted in Fig. 5.6. The unit is designed to process one interpolation per clock cycle in a pipelined manner, to reduce the critical path and increase throughput. This is the most complex unit between the basic signal processing blocks, which shows the longest latency from input to output, as it requires to be split into many stages by registers. As already explained in [53], the unit FU_{INTP} splits the sample into a real and imaginary part which are processed on two separate unit cores. The unit receives two data streams while sending (pumping) two data streams at the same time.

1. IN 1: real valued interpolation position data to both cores
2. OUT 1: real valued block address data for line buffers
3. IN 2: complex valued sensor data ($Re;Im$): Re to core1, Im to core2
4. OUT 2: complex valued sensor data ($Re;Im$): Re from core1, Im from core2

The unit receives (IN 1) a real-valued stream of interpolation positions (DI_{idx}), each composed of an integer and a fractional part $[int, frac]$. The int part of DI_{idx} is the base to determine the actual read position DO_{addr} for the line buffers. For interpolation, a block position calculation unit shifts the int part, to load also neighboring samples. The $frac$ part is the base to determine the correct set of coefficients for sub-sample interpolation. The generated read address DO_{addr} is applied (OUT 1) to the line buffers which are located outside the interpolation unit (one new address each clock cycle). Interpolation with t taps, requires $t-1$ neighboring samples. The set of t complex-valued (Re,Im) samples is streamed (IN 2) from the N_{rg} samples wide line buffers. The data is split upon the two cores, the

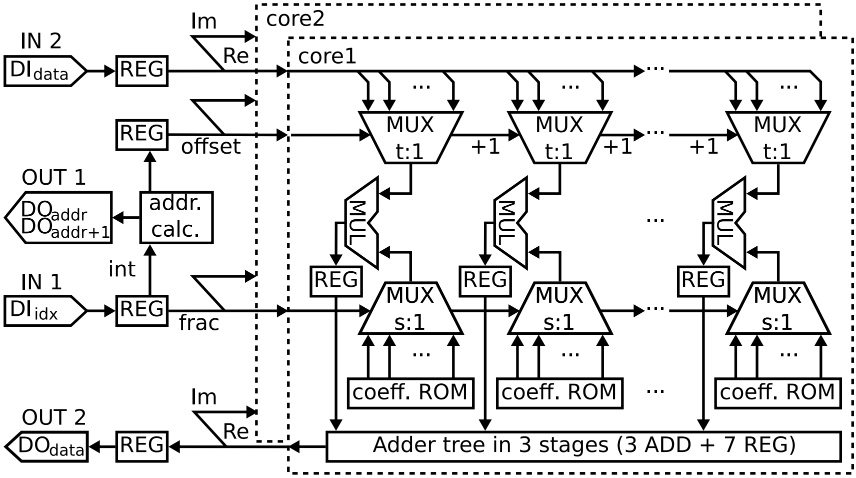


Figure 5.6: Block diagram of interpolation unit FU_{INTP}

real part Re is streamed to core1, the imaginary part Im part is streamed to core2. The final interpolated sample (Re,Im) is merged from the two separate interpolation cores. The interpolated sample (one per clock cycle) is streamed (OUT 2) to the functional unit for complex multiplication.

Since an interpolation index DI_{idx} depends on different geometrical parameters, any position within a range lane can be addressed. To ensure one interpolation per clock cycle, the line buffers need to contain the entire range line with N_{rg} samples, before the first interpolation is triggered. The line buffers outside the interpolation unit are assembled of Block Random Access Memorys (BRAMs). Each BRAMs can store up to 36 $Kbit$ and allows for a 36 bit wide output interface. As the data word width for samples is 32 bits, only 32 of the 36 bit of the output interface are relevant. This allows to store and load $1024 \cdot 32$ bit samples in each BRAM. To establish a wide interface to the line buffers (to fetch all neighbor samples for interpolation in a single clock cycle), the interface of the line buffers has to be t samples wide. This results to a $t \cdot 32$ bit $= 256$ bit wide interface, wherefore t parallel BRAMs each with a 32 bit interface need to be instantiated in parallel. Due to the limited interface of one sample per BRAM, the vector of t incoming samples must be stored interlaced inside the line buffer. This causes, that each sample from the linear stream of t samples, is stored in a different BRAM as shown in Fig. 5.7. This allows for parallel access to t samples at once, resulting in a 256 bit wide interface. Increasing the width has the disadvantage of reducing the address space by the factor t to $\frac{\log_2(N_{rg})}{t}$. Therefore, only discrete blocks, aligned to multiples of t samples, can be addressed.

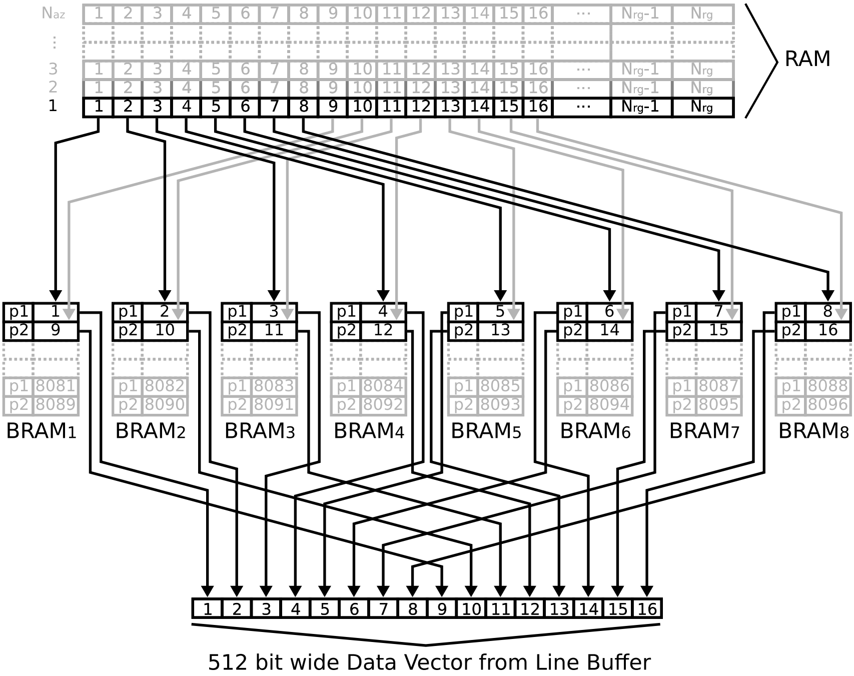


Figure 5.7: Interlaced sample storage in line buffers for parallel 512 bit access

But unaligned addressing of samples is necessary, as the t consecutive samples might overlap a block address. This is solved by the concatenation of two adjacent blocks, resulting in a $2 \cdot t$ sample vector loaded from the line buffers, encapsulating the t samples as shown in Fig. 5.7. The mentioned setup of the BRAM interfaces only offers a $t \cdot 32$ bit $= 256$ bit access. But the BRAMs can also be instantiated as true dual-port memory. A true dual-port setup allows doubling the interface width to 72 instead of 36 bit. This enables for reading two blocks (each $256 = 2^t$ bit wide) from the line buffers simultaneously, resulting in a 512 bit wide interface.

The address calculation block generates the two aligned block addresses DO_{addr} for accessing the line buffers. Address calculation is based on the integer part int of the sample index DI_{idx}. The int part represent the middle address of the $2 \cdot t$ linear samples. Subtracting $\lfloor t/2 \rfloor$ from the int value, results to the address of the first sample that is required for interpolation. Since the address can be unaligned with the block addresses, a division with a remainder with the sample block size t is applied, which results in the first block address DO_{addr}. The remainder is stored as an $offset$ value to extract the start position within the $2 \cdot t$ wide vector. As this is a division by the power of two, it can be realized by a simple right shift by $\log_2(t)$ bit. The second block address is just the next linear address to obtain two linear blocks from the line buffers.

The two blocks (DO_{addr}, DO_{addr+1}) form a data vector DI_{data} of 512 bit. The vector is read from the line buffers at once and is divided into a real Re and an imaginary part Im. Both parts are streamed separately to both cores. The samples needed for interpolation are selected by t parallel multiplexers (MUX) from the 512 bit and fed into t multipliers. Although the received data vector is $2 \cdot t$ samples wide, the multiplexer can be reduced to $t : 1$ instead of $2t : 1$. This is possible because at least the first sample must be located within the first block of the vector ($DO_{addr} + offset$), so one out of t position. Every sample requires the same selection window length, only the $offset$ value is increased by 1 for every consecutive sample. Every sample is assigned to its correct coefficient for multiplication in each core. For a correct assignment, the $frac$ part of $DI_{idx} = [int, frac]$ is passed to another set of multiplexer in each core. One out of s polyphase subsets of precomputed coefficients is selected from the Read Only Memorys (ROMs). The amount of subset influences the quantization precision and therefore controls how accurate the $frac$ part can be mapped to the subsets. This mapping is done by rounding the $frac$ part to the closest power of two of $\log_2 s$ bits. Based on this mapping, the t multiplexer pick one out of s sets ($s : 1$) which are then multiplied with the corresponding samples. All Re products and Im products are summed up separately in each core. Instead of the incremental summation of all products, an adder tree is implemented. The adder tree contains $t - 1$ adder and is divided in $\log_2 t$ stages. The tree consumes the same amount of resources as a incremental summation (when pipelined), but allows for a simultaneous processing of each product. This adds additional $t - 1$ registers in the DI_{data} path in order to delay every product for a correct timing for incremental summation. The same applies to the $offset$ path and $fraction$ path. The results of the two cores are then merged to a complex number and pumped outside (DO_{data}) for further processing.

The total amount of resources for each core sum up to, t ALUs for multiplication, $t - 1$ ALUs for the adder tree, $2 \cdot t$ multiplexer for data and coefficient selection, t ROMs each $q_w \cdot s$ bit in size. Additional resources for routing all signals and the address calculation unit need to be added. By adding $(t - 1)$ registers for the adder tree, t registers for the MUL operations and 4 registers for streaming, a pipeline implementation is achieved. By this, the critical path of the entire module is reduced to the time needed for multiplication, as it is the time dominant part.

5.1.4 Complex multiplication

To manipulate complex valued SAR data, complex multiplication is required for the phase correction of Radio Detection And Ranging (Radar) echos, in order to coherently sum up location invariant echos in one spot. This is a key element in processing SAR data, as through this correction, it is possible to generate a focused from unfocussed SAR. Since a complex multiplication is a costly function, it has to be discussed how this operation can be implemented efficiently. A complex number z is expressed by a real and imaginary part $z = Re + jIm$. The multiplication of two complex numbers, $a + jb$ and $c + jd$ is defined by Eq. (5.17).

$$\begin{aligned} z_1 \cdot z_2 &= Re + jIm = (a + jb)(c + jd) \\ &= (ac - bd) + j(bc + ad) \\ &= [a(c + d) - d(a + b)] + j[c(b - a) + a(c + d)] \end{aligned} \tag{5.17}$$

According to [91] one can calculate the intermediate values $k1$, $k2$ and $k3$ to:

$$\begin{aligned} k1 &= a(c + d) \\ k2 &= d(a + b) \\ k3 &= c(b - a) \end{aligned} \tag{5.18}$$

Using the intermediate values from Eq. (5.18), the real part Re and the imaginary part jIm can now be expressed to:

$$\begin{aligned} Re &= k1 - k2 \\ Im &= k1 + k3 \end{aligned} \tag{5.19}$$

An imaginary number can now be expressed to:

$$\begin{aligned} Re + jIm &= (k1 - k2) + j(k1 + k3) \\ &= [a(c + d) - d(a + b)] + j[a(c + d) + c(b - a)] \end{aligned} \tag{5.20}$$

The two formulations of a complex multiplication can now be compared for a dedicated implementation. The direct formulation Eq. (5.17) requires four multiplications (MUL) and three additions/subtractions (ADD/SUB) in total to obtain Re and Im. Equation (5.20) requires three multiplications (MUL) and five additions/subtractions (ADD/SUB). This is because the intermediate-term $k1$ is used twice, but must be computed only once. In the case of pipelining, nine registers are added to the indirect version (3M;5AS) while only four registers are required for the direct version (4M;2AS). The two block diagrams in Fig. 5.8 show how the two alternatives (4M;2AS) and (3M;5AS) can be transferred to a dedicated hardware unit FU_{CM}.

The decision between the two alternative implementations is based on resource consumption and performance. In terms of latency, the direct implementation (4M;2AS) is the better alternative, as processing requires only two consecutive arithmetic blocks instead of three. Without pipelining the direct implementation would also generate higher throughput rates as it has a shorter critical path. This advantage is nullified in case of pipelining the implementations in two stages for (4M;2AS) and three stages for (3M;5AS). (4M;2AS) still is advantageous in terms of initial latency and register consumption, but the critical path is now reduced to

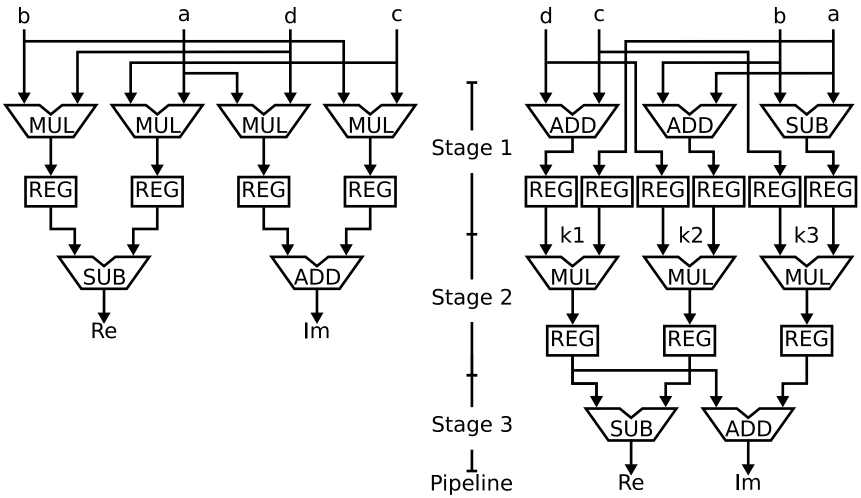


Figure 5.8: Block diagram of complex multiplication units FU_{CM} in different implementation with (4M;5AS) left and (3M;5AS) right

the MUL stage for both implementations. Passing through the stages, the data path will increase in width, based on the performed arithmetic operation. This influences either the critical path or the required resources, or even both. While the number of stages differs, both alternatives have a similar positioning of MUL and ADD/SUB operations. Therefore, the arithmetic blocks of MUL are assumed to have a similar critical path and resource consumption in relation to its pipeline stage. This results in the same critical path length for both implementations, as they are defined by the most complex block in the pipeline (MUL). A decision is therefore based on resource consumption. Although (4M;2AS) requires fewer arithmetic blocks, (3M;5AS) is picked for implementation. This is based on the considerations, that ADD/SUB operations can also be mapped to logic elements (LUT) instead of Digital Signal Processor (DSP) slices. This allows for an additional degree of freedom during the synthesis process and widens the range of FPGAs for implementation.

5.2 Dedicated GBP implementation

Several possibilities exist to implement the GBP in hardware. To reach maximum performance, the processing has to follow the already described concepts of pipelining (intra line) and parallelization (inter line). Due to the high regularity of the algorithm and the freedom to compute all Pulse to Pixel Projection (PPP) in any order, the architectural concept is optimized for memory management, scalability, and platform versatility. The concept for the GBP PE implementation is described in Section 5.2.1. Additionally, to the GBP core, a wrapping structure is required to control the core, the overall concept is described in Section 5.2.2.

5.2.1 GBP PE

A trivial approach to implement the GBP is to take one rangeline (the echoes of one pulse) and project it to all pixels. To be efficient with this approach, the whole image must be present in fast memory, due to the vast number of read and write access to the output image memory. This has been solved for CT in [76] by alternately using two dedicated static Random Access Memorys (RAMs) as accumulation memory. This is not possible for FPGAs, as the internal memory is too small to hold the whole image data at once. The here described architecture always generates one complete image line I_y of the final image $I[i_x, i_y]$ at a time, by looping over all pulses N_{az}. In the case of only one PE, every line will be processed consecutively by the PE. In case of p multiple PEs the lines are divided equally among all available PEs.

The concrete strategies to enable constant streaming of data were discussed in the previous chapters. The architecture for the time and resource costly functions were also described in the form of different FUs for complex multiplication, interpolation, square root calculation, and Euler transformation. All other operations during GBP processing can be covered by standard elements. All elements need to be interconnected in order to form a GBP PE. This PE should be flexible to allow for different rangeline lengths. Furthermore the PE should allow for inter loop streaming as depicted in Fig. 4.5, so that multiple PEs can utilize every FPGA.

The architecture for the GBP PE is depicted in Fig. 5.9. A PE is connected to a control unit, which is responsible for generating the control parameters of the internal loop. This includes the splitting of such parameters to several instantiated GBP PEs if existing. A connection to the RAM is provided over a bus that can transmit k data words per clock cycle. The proposed architecture reduces the bus load through internal buffers. Also, as already described, the busload is reduced by the factor p which represents the amount of parallel GBP PEs. Nevertheless, a constant stream of data is required from the RAM, including trajectory data, height information of the terrain (Z), constants and mostly raw data S. Of course, data is not only loaded from the RAM, but a fully processed image line is also transferred back as soon as the bus allows for a complete transfer.

Due to the parallel operation of GBP PEs, and the fact that RAM can only be accessed by one PE at once, every GBP PE is designed to be able to stall at any point during processing. Internal buffers with the size of one full range line are instantiated to avoid stalls. Holding the entire range line shifts the massive read access to the internal BRAMs of the FPGA, while write access to external memory is reduced to the minimum of one image size. This reduces the busload to the initial load of the line itself. To additionally cancel the waiting period for the initial load, the buffer is

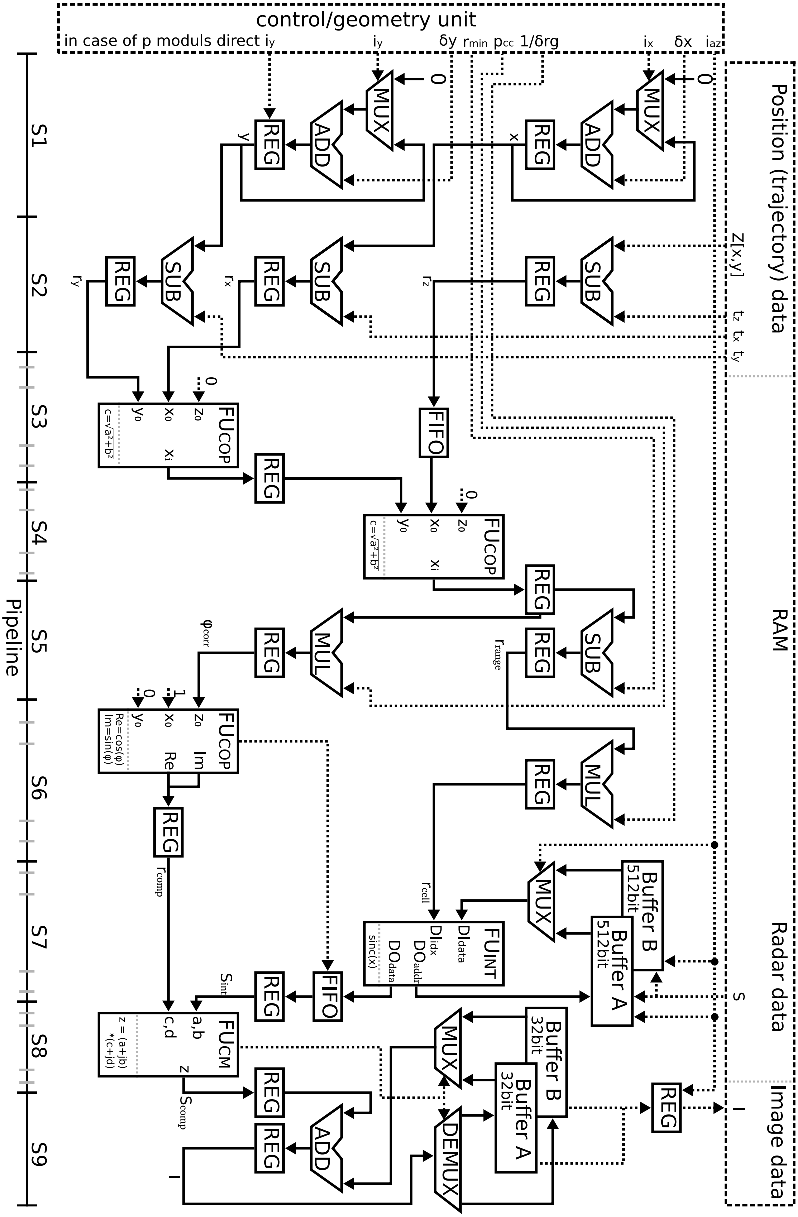


Figure 5.9: Block diagram of GBP PE

implemented in pairs, to form a swap buffer (buffer A+B). This allows to load the range line $az+1$ in advance into the background buffer, while the PE is processing the data from range line az in the foreground buffer. As soon as line az was projected on image line y, the buffers are switched from A to B (foreground to background) or vice versa. This allows for constant processing of raw data, as long as the processing time PE_{pt} is smaller then the line load time PE_{llt} to load one full range line from RAM. The same concept is used to hold intermediate results. A full image line is held in a foreground buffer, cutting all read and write accesses to RAM, besides the final transmission of the processed image line. To ensure access to a buffer for processing the next image line, the foreground buffer is switched to the background buffer. The data can then be transmitted to the main memory, as soon as the bus is free. Since all az range lines need to be loaded completely to generate one image line, the raw data buffers are switched N_{az} times more often than image line buffers.

The whole PE follows a pipeline concept (intra line streaming) to gain maximum throughput. The pipeline stages on PE level are indicated in Fig. 5.9 (S1 - S9). This does not include the sub pipeline stage (indicated grey) of the more complex FUs, wherefore they need to be looked up in the preceding sections. In contrast to the theoretical considerations (11 pipeline stages), the total amount of stages results to 9. This is the result of an optimization of control dependencies. It is not evaluated if the current distance is within the actual range line limits. Instead, the processing is continued, as the resources are available anyway. Outliers will try to access data that is not available in the range line buffer, which will simply result in zero. Another reduction is the result of a mathematical approximation in the form of CORDIC modules. The CORDIC directly processes the differential vector in fewer steps.

The core operation of a single PPP starts by calculating the distance from the antenna position (including motion deviations) to the ground point (pixel) position δr in the first four stages. The ground position (pixel position x and y) is generated in stage S1 by the loop values i_x and i_y using start (0) and increment values (δx,δy). In S2 three subtractors calculate the difference to each component of the current antenna position (t_x,t_y,t_z) based on x and y and the height value $Z[x,y]$. The third and fourth stage complete the calculation with two CORDIC FU_{COP} units by directly computing the differential vector norm δr. The CORDIC unit FU_{COP} is seen as one pipeline stage, while it is actually sub pipelines as described in Section 5.1.2. This adds more pipeline stages which are actually not listed but indicated as small grey dashes on the pipeline bar. To sync the input values for the CORDIC in S4, the path for r_z is buffered with a First In – First Out (FIFO) buffer.

After stage S4 the path splits. The upper path is the interpolation path. To obtain the actual interpolation position from the range distance δr, the distance r_{min} to the first range sample is subtracted in stage S5 resulting in r_{range}. This value must be divided by the constant distance δrg between range cells, to obtain the actual interpolation position r_{cell} in the range line. The division corresponds to multiplication with the reciprocal of δrg in stage S6. The real-valued distance is fed into the sub pipelined interpolation unit FU_{INT} in stage S7, which computes the interpolated but not yet phase-corrected raw data sample. The lower path generates the phase correction angle ϕ_{corr} by multiplication with the phase correction constant p_{cc} in stage S5. The phase correction value is constant over the entire image, wherefore no subtraction of the r_{min} is necessary. In stage S6 another instance of FU_{COP} is used to approximate the cosine (Re) and sine (Im) values of ϕ_{corr} as the complex phase correction factor.

Upper and lower path unite again in stage S8 for complex multiplication in FU_{CM}. Due to the different operations, both data streams feature a different latency and must be synchronized to generate a correct result. A handshake mechanism with two signals is used to regulate the data flow. A *req* signal indicates that data is requested from the preceding stage or memory for the next clock cycle. A *val* signal confirms (usually one clock cycle later) that the triggered unit is ready. The *req val* protocol gives every unit one cycle to have the requested data ready. This is especially important for internal buffer access, as data is only valid one cycle after the address is requested. In the special case of upper and lower path, the synchronization is realized through a FIFO memory of dynamic length in the upper interpolation path. The FIFO length (delay) is set according to the accuracy (iteration depths) of the CORDIC unit in the phase angle path. The FIFO buffers the interpolated samples S_{int}, until FU_{COP} sends a *req* signal, which is confirmed with a *val* signal to show that processing can continue. This and the possible lack of bus bandwidth when writing or reading data, requires to stall the entire design also just partly to guaranty a flawless processing scheme under all conditions. From that point, all pipeline stages are filled and synchronized continuously, allowing for constant processing. The complex multiplication of S_{int} and S_{comp} is carried out in S9, generating a phase-corrected and interpolated sample. This stage marks the end of a PPP. $N_{az} \cdot N_x$ PPPs are required to form one finished image line. To cancel the continuous read and write operations to the external memory, one of the buffers (selected by the FU_{CM} as foreground buffer) holds the entire line for in line pixel accumulation $(I[x,y]+S_{int})$. As this accumulation can lead to overflows of the fixed-point number, the data path width to the buffers is extend by $log_2(N_{az})$ bit for the real and imaginary part of a sample. When the entire line is transferred to the main memory, $log_2(N_{az})$ bits of each value are truncated. To not alter the maximum value, a dynamic right shift by $log_2(N_{az})$ bit is performed. This results in truncating the least significant bits, which corresponds to multiple divisions by factor two. By this, the dynamic range can be adjusted. The chosen structure of pipelining and path synchronization allows for easy enhancement of the module to adjust the algorithm.

At this stage, a full image generation takes about $N_{az} \cdot N_x \cdot N_y$ PPPs (clock cycles) with a small overhead for pipeline filling and double buffer swapping. Altogether this concept enables for an exhaustive utilization of available resources, while no special requirements are imposed on the platform itself or its memory periphery, while all required hardware units are embedded on the FPGA.

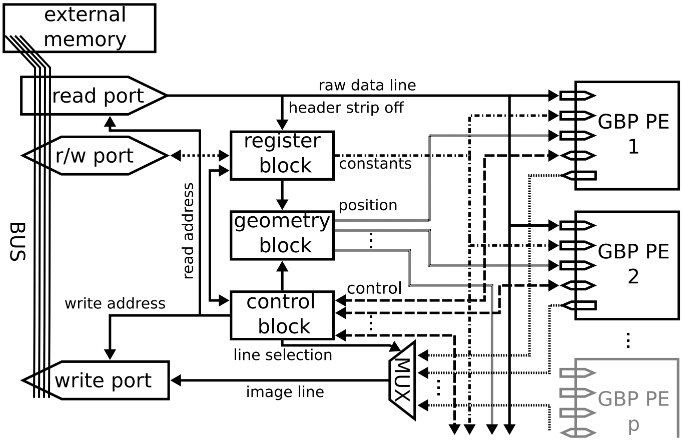


Figure 5.10: Block diagram of the control instance of p GBP PEs

5.2.2 GBP control structure

To exploit maximum performance on any FPGA, the GBP PE is instantiated p times. Each instance of p processes an individual image line at a time. Every instance iterates synchronously across the raw data lines. Wherefore raw data can be shared between all p instances. This reduces the number of total read operations by a factor p when compared to operation with only one PE. Concurrently, the write access rate is increased by the same factor. The proposed PE has five potential key features:

1. one PPP per clock cycle (after initial latency)
2. the flexible design can handle any arbitrary flight paths
3. the massive shift of slow external read/write operations to fast internal buffers
4. the reduction of external read access by parallelization factor p
5. the reduction of complexity through function approximation

The GBP PE is the core of the FPGA design for fast processing. Nevertheless, a higher level instance is required to control the PE and to synchronize multiple PEs if existing. The user space is setting all necessary parameters for this, including image size, the number of parallel modules p and accuracy parameters. Figure 5.10 shows this higher instance in a block diagram. The diagram shows a setup of p parallel GBP PEs.

First of all, it must be noted that no element inside a GBP PE has direct access to the bus, which is connected directly to the external memory. For simplification, this was indicated differently in Fig. 5.9, where constants and trajectory data were always loaded directly from memory. This would inflict a high risk of stalling the pipeline. Therefore, all user settings and constants (which are universal for all PEs) are stored in special registers (small local buffers) for a broadcast to all p PEs. The registers are set before processing starts and can be directly accessed from the outside world over a special slave port to the external memory. Next, raw data is requested by a control block (Finite State Machine (FSM)) over a bus master port from external memory. The control block generates the raw data line addresses. How often a line is requested is defined by the number of image lines and factor p. Trajectory parameters are included in each header of a raw data line. The parameters for every line are stripped off in a geometry block. These parameters are broadcasted to every PE, as all PEs process the same raw data and therefore require the same parameters. In case of multiple PEs, the geometry unit additionally calculates all start positions y for every PE before processing starts. This is necessary as every PE is processing a different image line. Only in case of one single PE the position can be calculated inside the PE by constant accumulation. p processed image lines will be available at the same time in the local PE buffers after N_y clock cycles. The control unit sets the memory address to transfer each line successively to main memory over an additional master port on the bus. The control unit controls the entire process independently until all loop parameters are processed.

5.3 Dedicated FMCW implementation

While the GBP is a very accurate algorithm with inherent compensation in case the recorded trajectory is accurate, it was designed for short pulsed Radar. The use of Frequency Modulated Continuous Wave (FMCW) systems imposes other restrictions, so that the start-stop-approximation becomes invalid. In order to deal with this problem, the GBP architecture in Section 5.2 needs to be modified according to [56].

The difference between pulsed and FMCW Radar is the geometrical offset between pulse transmission and echo reception. While for pulsed systems, the difference is so small (due to the short pulse) that it is negligible, for FMCW systems this distance increases with growing velocity. Since the correct projection of a range line onto the image is based on the distance Δr between a pixel and antenna position, the continuous change in distance during one projection results in erroneous projections. The GBP, with the pseudo-code given in Algorithm 4.1, is modified to correct this process. The steps required for correction are shown in Algorithm 5.1. Instead of listing the entire code from Algorithm 4.1, only the loop structure and the modified lines for correction are shown.

To adapt to motion, the distance is corrected for every range cell ($\delta r_{corrected}$) as explained in Section 2.3.3. For the correction, the influence of the platform velocity during ramp transmission must be considered. For this, the quotient of δr and δr_x must be calculated. Several system constants (ramp-length, velocity,

Algorithm 5.1: GBP algorithm modified for FMCW Radar after [56]

```
input  : $pc_c =$ phase correction constant $(4\pi/\lambda)$
         $N_{az} =$ amount of incoming sensor echos in azimuth
         $\delta rg =$ distance between each range cell
         $r_{min} =$ minimum distance covered by sensor echo
         $r_{max} =$ maximum distance covered by sensor echo
         $S[N_{az}, N_{rg}] =$ sensor data array
         $FMCW_{const} =$ pre calculated FMCW constant
         $N_x =$ amount of image pixel in x
         $N_y =$ amount of image pixel in y
         $I[N_x, N_y] =$ initial image array on ground (set to 0)
output : $I[N_x, N_y] =$ processed image
for ($i_{az} = 0$ to $N_{az} - 1$) do        /* iterate over echos in azimuth */
    for ($i_x = 0$ to $N_x - 1$) do      /* iterate over image pixel in x */
        for ($i_y = 0$ to $N_y - 1$) do  /* iterate over image pixel in y */
            /* correction for FMCW radar                               */
            $\delta r_{corrected} := \delta r + FMCW_{const} * r_x / \delta r$
            if ($r_{min} \leq \delta r_{corrected} \leq r_{max}$) then   /* in sensor range?  */
                $r_{range} = \delta r_{corrected} - r_{min}$
                $r_{cell} = r_{range} / \delta rg$
                $\phi_{corr} = pc_c \cdot \delta r_{corrected}$
                $r_{comp} = e^{j \cdot \phi_{corr}}$        /* then follows interpolation */
                $S_{int}[i_{az}, r_{cell}] = int\{[i_{az}, r_{cell}], S[i_{az}, i_{rg}]\}$
                $S_{comp} = S_{int}[i_{az}, r_{cell}] \cdot r_{comp}$
                $I[i_x, i_y] = I[i_x, i_y] + S_{comp}$
            end if
        end for
    end for
end for
```

wavelength and cutoff frequencies) are combined in a pre-calculated factor $FMCW_{const}$. Multiplied with this factor, the sample distance (range) is corrected for each range cell individually. After this, the interpolation and the phase correction for this specific position are applied and the value is accumulated to the corresponding position value [54]. The modification of the GBP architecture is depicted in block diagram Fig. 5.11.

Compared to the GBP PE in Section 5.2, the presented architecture shown in Fig. 5.11 consists of additional elements (marked in light grey) to correct the influence of the platform velocity during ramp transmission. Besides these additional elements the processing flow is identical to the GBP PE. An extra CORDIC FU is added to perform the division of r_x and δr. An additional multiplier is instantiated for the correction factor $FMCW_{const}$. An extra adder sums up the multiplier result with the originally calculated range distance δr to obtain the corrected range value $r_{corrected}$. The adjusted range is transferred to the two paths, one for interpolation with the 8-tap low-pass sinc filter, the other for phase correction with complex multiplication. The additional steps increase the pipeline length by three stages. In case of a pulsed Radar these blocks can either be chosen to not be included in the entire design or if flexibility is required, the modules can be bypassed in case the system is used for FMCW and pulsed Radar.

5.4 Dedicated FFBP implementation

The implementation of the FFBP algorithm is less straight forward then implementing the GBP algorithm. Earlier analyzes showed, that the actual FFBP module should be split in two PEs. The **mapper** PE which calculates all parameters, distances, positions and addresses for the actual factorization process. The **factorizer** PE conducts the merging of apertures to form new subapertures for all subimages of each FFBP stages. Both PEs work unsynchronized, meaning that they are not directly interconnected to form a linear processing pipeline. The PEs are loosely coupled over the **config mem**. The **mapper** stores multiple sets of computed parameters for a merge process in the **config mem**. All instantiated **factorizer** PEs can access the memory whenever ready and the bus is in idle state. Whenever a subimage in the final stage is ready, the responsible **factorizer** PE triggers a GBP PE to start the processing of the subimage. To reach maximum performance, the design of the PEs has to follow the already described concepts of pipelining (intra line) and parallelization (inter line). The architecture concepts are developed to optimize memory management, scalability, and platform versatility. The concept for the implementation of the FFBP PEs is described in Section 5.4.2 and Section 5.4.3. The structure to connect both PEs is described in Section 5.4.4. Since the FFBP is a flexible algorithm in terms of iterations, optimized configurations of subimages and aperture factorization, a pre-analysis for a set of permutations is conducted in Section 5.4.1. This is required, to understand which parameters have the highest impact on quality, etc. and if this will affect the actual implementation.

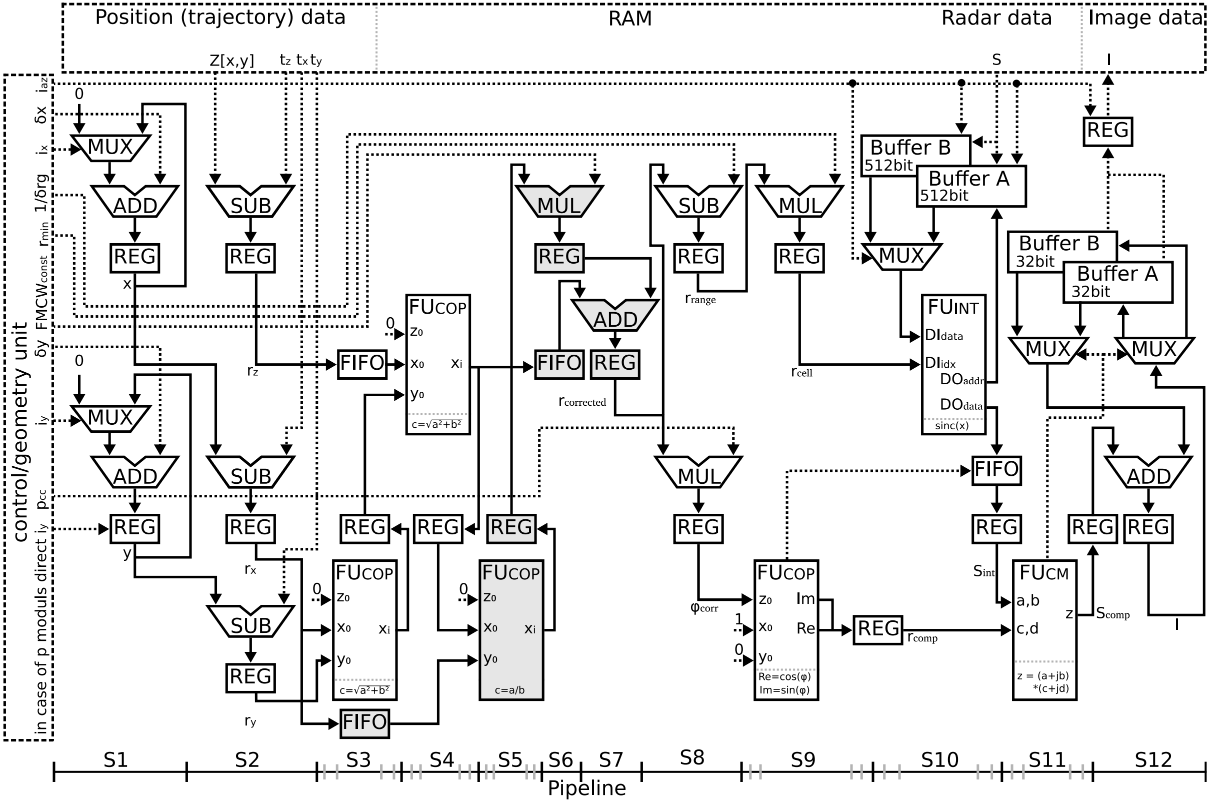


Figure 5.11: Block diagram of GBP PE modified for FMCW Radar according to [56]

Table 5.3: System parameters of the here used high bandwidth FMCW SAR sensor devolped by Pohl et al. [92] for near field measurements

parameter name	value
frequency	80 GHz
bandwidth	25.6 GHz
max. resolution (-6 dB)	$8.08\ mm \times 10.73\ mm$
ramp-lenght	2-16 ms

5.4.1 Analysis of factorization impact on image quality

The FFBP can be processed in many iterations or stages s, while each stage is controlled by different parameter permutations of aperture factorization f_{apt} and subimage splitting f_{sub}. The combinations of different parameter permutations per stage s are free of choice and affect the final image quality and the overall runtime (Filtered Backprojection (FBP) + GBP). It is necessary to get a clear understanding of the effects, to implement hardware that creates a good trade-off between both aspects. A comprehensive study was conducted where the same set of parameter permutations was evaluated for different amounts of iterations s. While one can state that an increased runtime results in better image quality, this is not always true when different iterations are compared. The results of this study were also discussed in [58], [59]. To classify the relevance of the study, two aspects need to be discussed. First which type of SAR data was used for image quality evaluation. Second which metric was used for evaluation.

The type of data used for evaluation was obtained from a near field sensor system which parameters are listed in Table 5.3. The sensor was used in a range distance of $1.2\ m$. The near filed setting simulates the worst case possible, to check if also harsh conditions would still allow a working design.

The nature of SAR signals is problematic for quality evaluation. First, no ground truth data is available for any set of data. This always raises the question, if a change in processing enhances or worsens the image. Second, without any ground truth data, it is hard to benchmark if a chosen metric is suited for quality evaluation. Since no metric was found so far that corresponds with human evaluation, SAR images are still evaluated manually by humans. For automated evaluation metrics like the Integrated Sidelobe Ratio (ISLR) and Peak Sidelobe Ratio (PSLR) are used. These metrics rely on corner reflectors or objects with similar characteristics that outshine their surroundings through intense reflections. The mentioned metrics rely on measuring the size of the main lobe and the surrounding side lobes, to evaluate the quality in the image. Due to the characteristics of the FFBP, these metrics can lead to misinterpretations during evaluation. Since the FFBP divides the image into subimages, edges start to emerge in the image. In case a strong reflector is located on an edge of a subimage, the quality of the signal will be reduced in comparison to the quality of the reflector in case it was located in the middle of a subimage. A possible metric is the Peak Signal to Noise Ratio (PSNR) which is based on the Mean Squared Error (MSE). The MSE measures the average of the squares of the deviations (or the difference) between the estimator and what is estimated. The MSE results to Eq. (5.21) for a two dimensional image I.

$$MSE = \frac{1}{N_x \cdot N_y} \sum_{k=0}^{N_x-1} \sum_{l=0}^{N_y-1} (I_{GBP}(k,l) - I_{FFBP}(k,l)) \tag{5.21}$$

Since the GBP can also be processed alone without any factorization, the resulting image can be used as a golden reference (or ground truth) for a comparison of different FFBP configurations. The PSNR in Eq. (5.22) is using the MSE, divided by a max value p_{max} which is the maximum pixel value found in the entire image. This metric is also used in the evaluation of SAR images according to the literature [20]

$$PSNR = -10\log_{10}(MSE/p_{max}) \tag{5.22}$$

The tested permutations in this chapter underlie a very specific restriction owed to the implementation in hardware. While in general $f_{apt}, f_{sub} \in \mathbb{N}$, which could be realized in hardware, it is more efficient to reduce the set from $f_{apt}, f_{sub} \in \mathbb{N}$ to $f_{apt}, f_{sub} \in 2^n$ where n is an integer. If the parameters for raw data sets and processed image dimensions respect the same restrictions, computations can be simplified, as some operations can be broken down to shifts instead of multiplications or adders. Furthermore, edge cases and odd parameters are reduced, since multiplications and division in this number of space will always reside in the same number space of 2^n.

Factorization of the image is possible in x (range) and y (azimuth) direction. The impact of splitting the image in y direction should have a greater effect then splitting in x direction. This is because the distances between two subimages in y direction should be much greater than the distance between two subimages in x direction. This should reduce the induced error in range at the same time. The tested permutations are plotted in Fig. 5.12 for a different amount of iterations. Within a group of constant iterations (1, 2, 3 or 4) all parameters (e.g. factorization factors in x and y, size of subapertures) are combined to different permutations. The vast amount of possible permutations across different iteration groups does not result in a plot that would allow identifying the individual parameter permutations. Instead, every configuration is ranged according to its PSNR value in dB against the runtime in seconds. The PSNR values are calculated by comparison to the same image processed

Table 5.4: Runtime and PSNR of FFBP Pareto configurations

config C/ points P	iterations i	f_{apt_i}	f_{sub_i} in x	f_{sub_i} in y	PSNR in dB	runtime in s
P1	1	2	8	2	58	52
P2	1	4	32	2	53	30
P3	1	4	16	1	52	26
P4	2	4,2	16,4	2,1	46	18
P5	3	2,2,4	4,4,4	1,2,1	40	11
P6	4	2,2,2,2	8,2,4,1	1,2,2,1	42	19
GBP	1	1	1	1	∞	101

with the GBP as a golden reference. The runtime of roughly 101 seconds for the GBP is marked as a vertical line for comparison, while every iteration group is colored differently.

The factorization across the image is creating a tree-like structure for every single permutation and iteration. Walking down a branch of a tree will always result in a reduction of image quality. This is owed to the increasing aperture factorization, which always results in a higher error rate due to the growing distance between actual aperture positions and subaperture position. Naturally, this error is increasing with f_{apt}, but it can be circumscribed to a certain degree by increasing the subimages f_{sub} for a node, as it limits the distance between the subaperture position and the subimage center. The compulsory loss in quality by aperture factorization and the fact that this can only be partly contained or limited by a higher image factorization is manifested by the imaginary horizontal boundary for each iteration group (1,2,3, and 4) in Fig. 5.12. Although the elements of different iteration groups (1,2,3 and 4) mix across the plot, it is obvious, that all elements of one iteration group cannot pass a certain horizontal boundary. The downgrading in quality between the iteration groups can also be understood through the reduction of raw data. With an increasing number of iterations, more apertures are merged (factorized). The reduction of aperture data has the same effect as undersampling, which introduces equidistant attenuated double images (or so-called ghost images). The final resolution of an image, which can also be chosen as a parameter, has only a small effect on image quality. This is because image formation is the last step and has no effect on how subapertures are generated. The processed image in Fig. 5.12 has a resolution of $1024x1024$ pixel. As already mentioned, this evaluation represents a worst-case scenario according to [42]. This is explained through the λ/f boundary, which indicates the maximum error in the distance that does not affect the image quality noticeably. For the near field sensor data, with a mid-range frequency of $80\ GHz$ the maximum unnoticed error in range results to $\lambda/4 = 0.936\ mm$. Additionally, the close range results in quite steep ratios between the radii of different antenna positions.

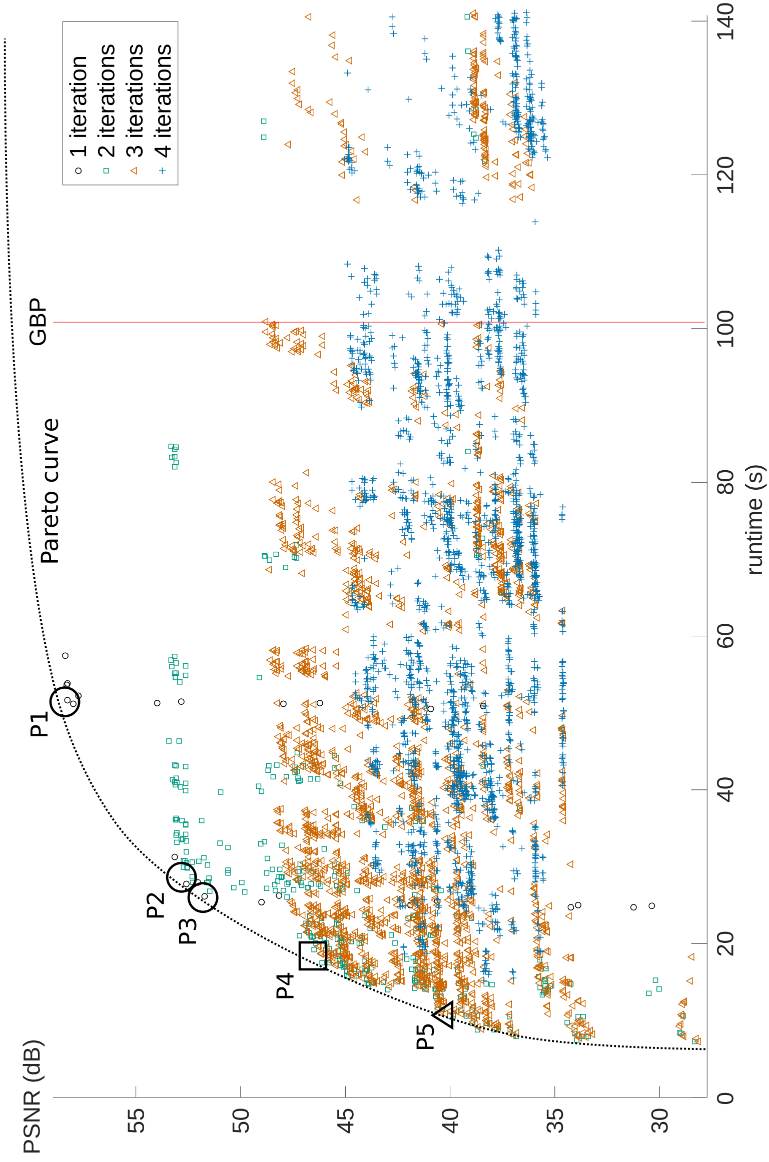


Figure 5.12: PSNR in dB vs. runtime in s for different FFBP configurations

Additionally, to the plotted configurations, a Pareto curve [93] is plotted in Fig. 5.12. The curve fits the outer frontier of configurations C that are most efficient. This means the Pareto curve marks the optimal trade-off between runtime and quality (PSNR) for a pair of iteration i and configuration C (parameter setting). The corner points P lay on this curve which marks the Pareto optimum. This means, a point in P represents a special configuration within an iteration i, in which no parameter can be changed to improve runtime or image quality, without degrading the other. $P1-P5$ mark points at which the slope of the Pareto curve changes, and therefore marks a local optimum point with the highest trade-off between relativity low computation and relativity high PSNR. This set of points is used to derive the actual constraints for the hardware implementation and therefore are used to compare the implementation with the results of this quality analysis.

It can be observed, that the most left-bound configurations belong to the iteration group of 2 and 3. This makes sense, as higher iterations tend to factorize a higher number of apertures. Following this logic, some configurations within the iteration group 4, should allow for even shorter runtimes then configurations with lower iteration count. This is not the case, since iteration counts of 4 and higher create a growing overhead in processing time for splitting images, combing apertures and such. Therefore, it takes even longer to process such configurations than configurations within 3 iterations. Depending on the raw data set and the sweep of parameters, this tipping point will be reached sooner or later for a FFBP run. A hardware implementation profits from this result, as the required memory can be reduced with fewer iterations.

The simulations show, that creating more subimages from the start is more important for image quality while keeping the aperture factorization low at the same time. For a hardware implementation, this is useful. The low aperture factorization can be used to limit the size of line buffers in the factorizer, due to the merging of subapertures. Since fewer summands create smaller sums, bits can be saved, which results in a reduction of buffer size. The high number of required subimages from the start also comes with a downside, as this results in higher memory consumption for storing the images. Such aspects need to be considered for the hardware implementation as the external RAM is, of course, limited in size as well.

Concluding this analysis, it has to be mentioned, that the differences in runtime between the corner points P are small in absolute numbers because the images are small in the presented example. The relative distance between the corner points would stay the same for bigger images, but the absolute numbers would be more significant than the here presented values. For a given SAR scenario it might be useful to switch between different configurations, as the factorization has different effects depending on the setting of the scenario.

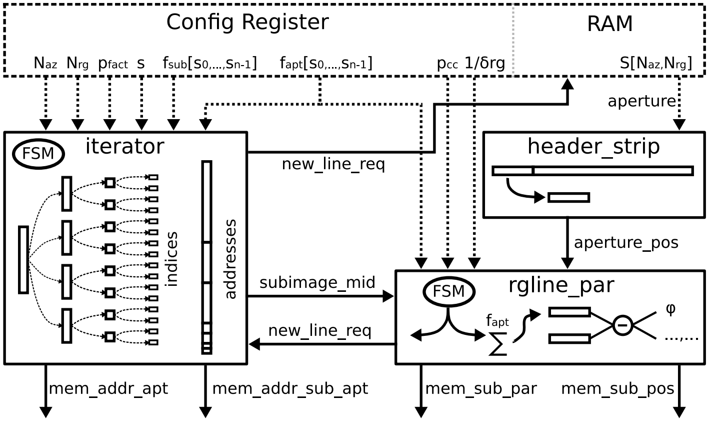


Figure 5.13: Block diagram of the mapper PE

5.4.2 Mapper

As already mentioned, the mapper and the factorizer are softly coupled over a configuration memory (config mem). The mapper is the controlling PE since all processed parameters direct the factorizers actions. The mapper receives certain information about the SAR system and the FFBP configuration. This information is used to calculate the geometrical parameters for subimage and subaperture processing for the entire tree walk. Additionally, read and write addresses are generated for the factorizer PE. The addresses indicate where old apertures are located in the main memory and where newly merged subapertures should be stored. The factorizer PE is a plain number cruncher and calculates the results according to the given parameters. The processing of one new range sample of a subaperture is performed once per clock cycle. Since parameters and addresses only need to be send once per line, the mapper has N_{rg}/p_{fact} clock cycles to process a new set, where p_{fact} is the number of parallel factorizer PEs. The mapper PE is depicted in Fig. 5.13. Since the merging of range lines usually requires significantly longer time, the mapper is not time-critical. This allows implementing the mapper in a resource-saving manner. Therefore, the following explanations focus more on the process itself, than on detailed diagrams of the actual hardware implementation.

The mapper is split into three parts. The major part is located in the iterator, which calculates all read and write addresses for the subaperture factorization. This is done for all stages s, including all submimages in x and y direction. Additionally, the subimage center positions are calculated, which are processed further in the rgline_par step of the mapper. All calculated addresses and positions are stored in

Table 5.5: Indicies for an exemplary FFBP tree run

m	p_i	i	x,y	p_i	i	x,y	p_i	i	x,y	p_i	i	x,y	p_i	i	x,y	p_i	i	x,y	p_i	i	x,y
0	0	1	0,0	0	2	0,0	0	3	0,0	1	3	4,0	1	2	0,1	2	3	0,1	3	3	4,0
1	0	1	1,0	0	2	1,0	0	3	1,0	1	3	5,0	1	2	1,1	2	3	1,1	3	3	5,1
2	0	1	2,0	0	2	2,0	0	3	2,0	1	3	6,0	1	2	2,1	2	3	2,1	3	3	6,1
3	0	1	3,0	0	2	3,0	0	3	3,0	1	3	7,0	1	2	3,1	2	3	3,1	3	3	7,1

so-called sets in the config mem to control the factorizer PE based on these sets. One set includes all the required information for the processing of one entire subimage, which then will be processed alone by one factorizer PE. To keep the config mem small, only $p_{fact} \cdot s$ sets are stored. As soon as the memory is filled, the entire mapper PE is stalled until memory is freed by the factorizer PE. Based on the strategy in Section 4.5.2.2, only one branch of the FFBP tree is processed until the final stage is reached. A FSM iterates the FFBP tree based on the parameters ($f_{apt}[s_0, ..., s_{n-1}]$, $f_{sub}[s_0, ..., s_{n-1}]$, s, N_{rg}, N_{az}, p_{fact}) which are set in the dedicated config registers of the FFBP module.

To calculate the addresses for input data and output data, the entire image is split and indexed across the x and y direction. Based on the index of a subimage, the aperture and the image factorization (f_{apt},f_{sub}), as well as the read and write addresses in main memory for this subimage, are calculated. Every index is paired with one available factorizer PE. The paring allows to write the calculated set of addresses and parameters in fixed addresses (direct mapping) in the config mem which simplifies the process of allocating. This process is exemplary shown in Fig. 5.14. In contrast to Fig. 4.10, the nodes are not indexed according to their position in the tree, but according to the image splitting, which is based on iterations and subimage factorization. Additionally, the bar does not indicate the amount of RAM used, but the position for the sets for each subimage in the config mem. The config mem bar also shows the fixed allocation of a FFBP PE to a fixed address in the config mem. Each color in the image indicates another iteration. As indicated by the continuous lines in the tree, the iterator finishes one branch to broadcast data from the main memory to multiple PEs. This is most efficient in the first stage where the amount of PEs fit f_{sub}. In Table 5.5 the exemplary run is listed from top to bottom, with every index and parameter required for address calculation. The table lists the run, split into lines for each module m. It is worth mentioning that it must be logged (p_i), how often one module passed a certain iteration i. This is required to track the tree position for each module, to generate the correct read and write address for the next stage.

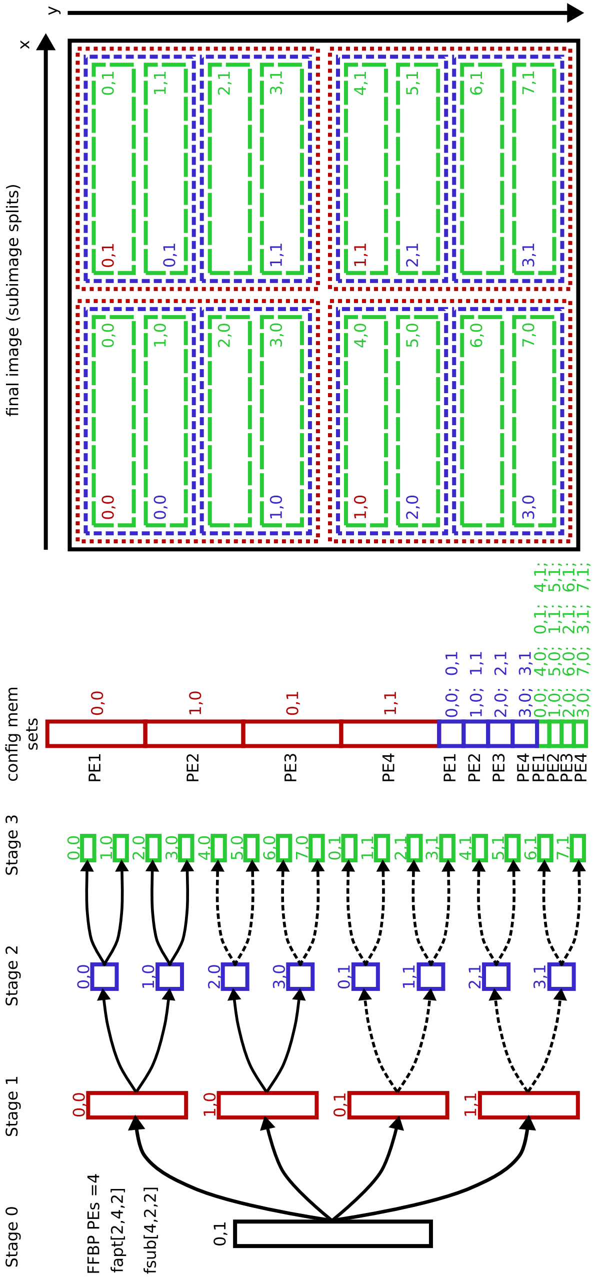


Figure 5.14: Image splitting for subimage address calculation and PE/config mem allocation

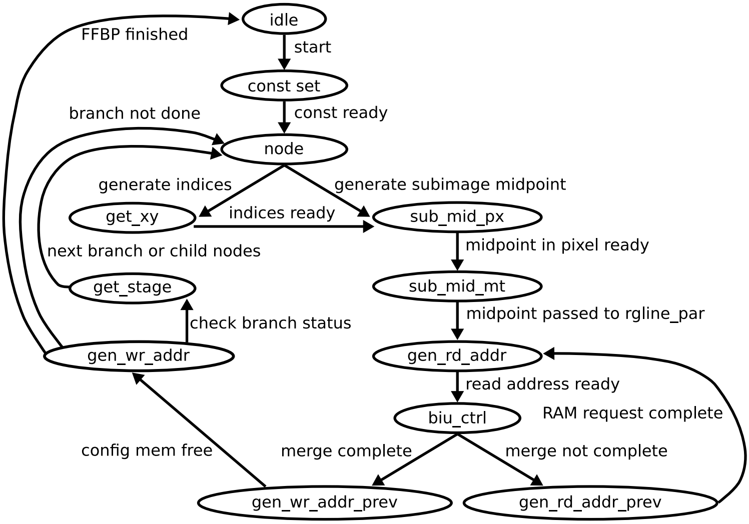


Figure 5.15: FSM to control tree processing in one iterator step of the mapper PE

The FSM that controls the tree walk is depicted in Fig. 5.15. From *idle* state, the setting of a control register flag initiates the FFBP, which will trigger the interwoven GBP. Required constants are calculated first in *const set* state. The *node* state is triggered and starts the FSM loop. Processing the FFBP tree is controlled from here in terms of the current branch and node position. In case of a new subimage, a new set of indices is generated in the *get_xy* state. The indices are used by the *sub_mid_px* state for subimage midpoint calculation in pixel (according to Algorithm 4.12). Since Algorithm 4.12 only represents center calculation in one dimension, this must be done separately for each dimension in *sub_mid_px*. State *sub_mid_mt* transforms the pixel position to meters for the *rgline_par* step. The subimage indices are required in state *gen_rd_addr* for RAM read address calculation of one range line (aperture) that contributes to this subimage. The address is stored in the config mem (if not full) over the **mem_addr_apt** signal as a part of the merge set for the factorizer PE. Furthermore the address triggers a read of the aperture in external RAM for the rgline_par stage to get the geometrical aperture position. The *biu_ctrl* state synchronizes this access with the rgline_par stage, whereby the merge set is generated. This is done $f_{apt}[s]$ times in a loop with the *gen_rd_addr_prev* state so that all addresses for one subaperture merge are calculated. To merge a new subaperture, a write address is required as well, which is calculated in the *gen_wr_addr* state and written to the config mem (if not full) over the **mem_addr_sub_apt** signal as a part of the merge set. A full merge set in stage s contains $f_{apt}[s]$ read addresses and one write address. A full set of merge sets is called a subimage set. It contains

$N_{az}/\prod_{i=0}^{s-1} f_{apt}[i]$ read addresses and $N_{az}/\prod_{i=0}^{s} f_{apt}[i]$ write addresses. Therefore, the FSM loops until the addresses for a subimage set are complete. The parameters of $f_{apt}[s]$, $f_{sub}[s]$ are passed over the **mem_addr_apt** signal, the subimage indices x, y and the flags for last iteration and finishing are passed over the **mem_addr_sub_apt** signal to contribute to the merge set. The set-specific information is required by the factorizer PE and the following GBP PE.

In case the *node* state detects that a subimage is done, new indices are generated and the loop performs another run to form the next merge sets to assemble the next subimage set. Only if the last iteration is reached and all line addresses for all p modules were generated, *get_stage* is triggered from *gen_wr_addr*, to proof if more then p child nodes depart from the current parent node. In this case, the current iteration stage is kept and the loop will finish all subimage sets. Otherwise, the tree is walked backwards to the next node with unprocessed child node, so that the next branch is processed. This is continued until all subimage sets for the entire image are processed. The sets are completed (during set construction) with the parameters processed by the rgline_par stage (**mem_sub_par**, **mem_sub_pos**), so that the factorizer PE can stream all required addresses, parameters, positions and control information from the config mem, in order to just load the actual raw aperture data from external RAM. merge sets are allocated to the factorizer PEs over the position in the config mem.

The rgline_par step is required for processing the interpolation parameters for each aperture and for processing the new subaperture position. Although all steps in the mapper are directly coupled, every step retains some cycles of flexibility due to buffers. The FSM depicted in Fig. 5.16 describes the flow of the steps.

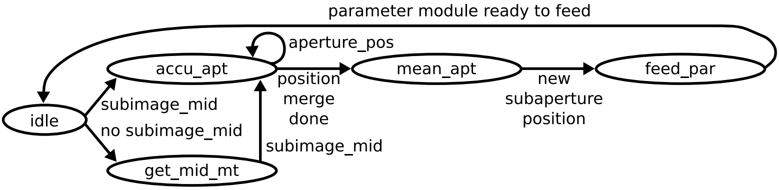


Figure 5.16: FSM for processing subaperture positions and parameter approximation

The rgline_par step waits in *get_mid_mt* until a subimage midpoint has been received. As soon as a midpoint is received, aperture position merging starts in *accu_apt* according to Algorithm 4.13. So f_{apt} aperture positions received from the header_strip step are processed. When the position merge is complete, the mean value is generated in *mean_apt* and is transferred to the parameter processing step in *feed_par*. The parameter processing step includes the processing of the actual distances between the new subaperture, each old aperture, and the subimage center. The difference in distances gives the parameters for the interpolation in the factorizer PE. Since those calculations are rather complex, (but were approximated in Section 5.1.1) a detailed block diagram of this approximation is depicted in Fig. 5.17 with the parameter calculation.

The approximation results to Eq. (5.14) with $rd \approx d_{nc} + (d_{ns} - d_{oc})$. To process d_{nc} and d_{oc} three positions are required. The header_strip extracts the old position (o) of each aperture triggered from the iterator step and passes it to the rgline_par module. The subimage middle position (c) is calculated by the iterator. The new subaperture position (n) is calculated inside the rgline_par module. Muxing o and n, d_{nc} and d_{oc} are calculated in turn by CORDIC FUs. Demuxing with a register as delay, provides both values in one cycle to interconnect them to a subtraction to result to rd. The value for d_{nc} is a equidistant linear growing range cell distance. In order to not generate the amount of N_{rg} values, adding this values is done later in the factorizer PE. This reduces the config mem by the factor of N_{rg} in size, as only one value is required to correct an entire line. This value is then split into a phase correction value and an interpolation index. This is similar to the GBP PE. The interpolation index is split into a fraction (r_{frac}) and an offset (r_{offset}) value. The fraction value defines the position for linear interpolation in the factorizer PE. The offset defines the number of the first sample to start the linear interpolation from. Basically lines are shifted against each other by this value. This is a significant difference to the GBP PE, where every range cell required a interpolation with unique values. This shifting against each other allows for another simplification when compared to the GBP PE. In the GBP the distance from the sensor to the ground r_{min} must be subtracted for correct calculation. As all lines share the same r_{min} the shifting against each other allows to drop r_{min} from the equation.

By using this approximation, one parameter set is sufficient for an entire line. Those calculated values are composed in the mem_sub_par set, while the mem_sub_pos set just contains the subaperture center position. Both are passed to the factorizer PE over the config mem. The whole mapper PE generates a set, which is required for one range line processing, which takes 49 cycles. This is fast compared to the interpolation length of N_{rg} samples. Even for a high parallelization factor p of the interpolation FUs, 49 cycles are more than sufficient.

5.4.3 Factorizer

The factorizer PE is used to create the new subapertures by linear interpolation. The interpolation can be carried out in parallel to process multiple range samples of one new subaperture at once, and/or multiple subapertures can be processed parallel to each other. The type and amount of parallelization can be parameterized and increased/decreased according to the available resources or desired throughput. The interpolation FUs have a fully pipelined design to achieve the highest throughput. Apertures from previous stages are loaded from external RAM and distributed to the factorizer via the State In (SI) FSM. The State Out (SO) FSM collects the

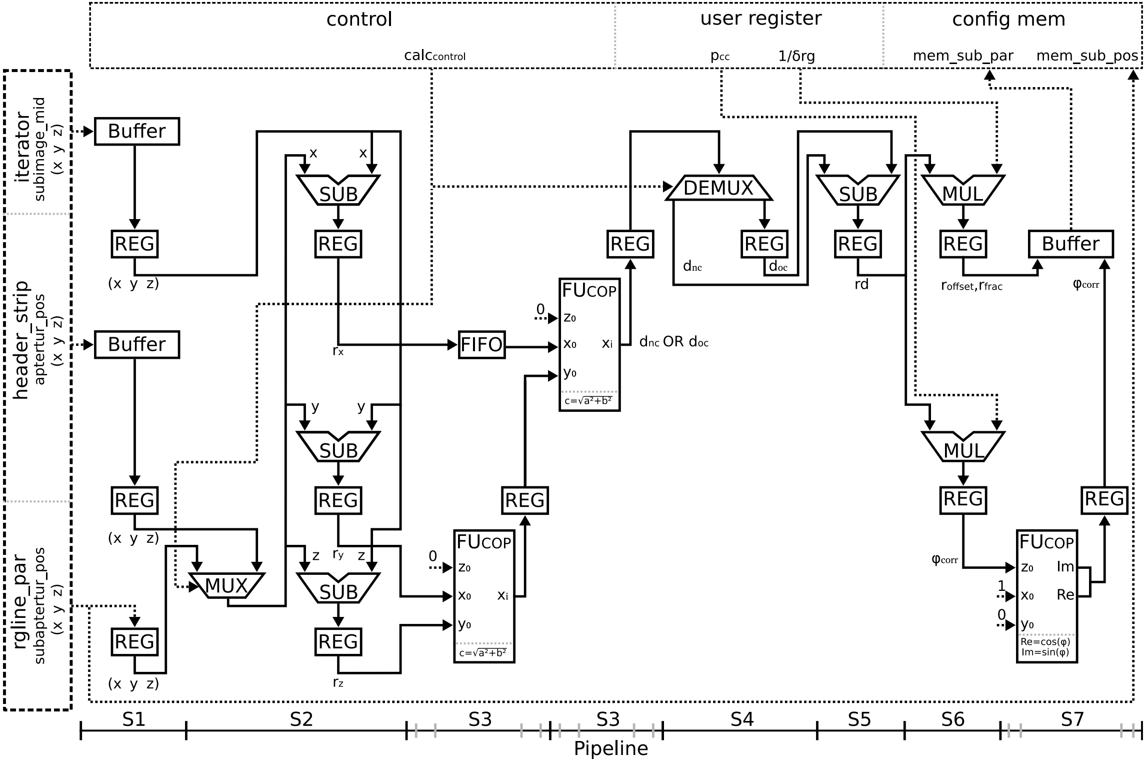


Figure 5.17: Block diagram of parameter approximation plus interpolation parameter calculation in rgline_par

subapertures from the interpolation modules. Both FSMs are controlled by the data in the config mem.

The SI FSM depicted in Fig. 5.18 checks in *idle* for prepared merge sets in the config mem. Two possible paths exist from there. Either the special case of GBP processing is activated ($f_{apt} = 0$, $f_{sub} = 0$, $s = 0$) which bypasses all FFBP steps and leads to the *rdaddr* state, or regular FFBP processing is activated which leads to the *same_module* state. The state checks for p merge sets in the config mem in order to distribute the same aperture data to p factorizer PEs that share the same parent node in the processing tree. In case $p > f_{sub}$, $p - f_{sub}$ PEs will set to be bypassed in the *no_wait* state for this specific run, meaning that the later states do not have to wait for this PE. The bypass flags are set by the iterator and passed to config mem. When all PEs are set, the merge starts by going to the *rdaddr* state. From here the aperture address is setup, either for GBP or FFBP processing. The *biu_ctrl* state will request the aperture from external RAM. The *wait_rlast* checks if the aperture was streamed completely through the factorizer PE, only to trigger the next aperture for the subaperture merge loop. When the merge is complete, the next merge sets will be checked in the *same_module* state.

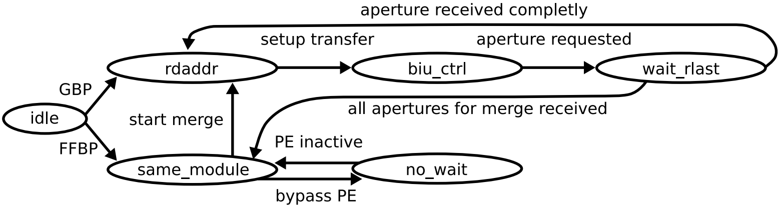


Figure 5.18: SI FSM for data in control of factorizer PE

The factorizer PE is shown in Fig. 5.19 and can be instanced in parallel as indicated in Fig. 5.21. The PEs are controlled by the FSMs and are feed with the same aperture data, but with individual parameter sets from the config mem. Throughout the entire architecture, all data words are requested with a req signal, as soon a data word is valid a val signal confirms the request and transmission. This val signal is masked for every PE individually based on the r_{offset} values calculated by the rgline_par step in the mapper PE. This offset value is transmitted over the config mem and indicates how many samples the actual aperture is shifted against the currently merged subaperture.

The design of the interpolation for the factorizer PE is very different from the GBP interpolation. As the linear interpolation does not result in address jumps and does not require many neighboring samples like in a sinc interpolation, the input buffer can be reduced to a minibuffer with the size of $b_{ww} + d_{ww}$ bit. The additional space for one data word is required so that every interpolation has a neighbor on the left side.

The core can use internal parallelization to interpolate all incoming samples at once. This depends on the available resources and bus capacity as an internal parallelization degree of $p_{fint} > \frac{b_{ww}}{d_{ww}}$ would remain some internal areas without data words for interpolation. A core receives the aperture specific parameters from the config mem. Each interpolation path streams two samples from the minibuffer to apply linear interpolation according to the r_{frac} value. A FU for complex multiplication performs phase correction according to ϕ_{corr}. Each sample of an aperture is corrected using the same r_{frac} value. After phase correction, the sample value is added to the sample value of the merged subaperture. The generated subaperture is being held in a buffer of N_{rg} samples in size until a merge is complete. Like this, no external bus access is required until the subaperture transfers to RAM. Switch buffers allow continuous processing in case the bus is busy. Buffers and parameter registers are controlled by counters to switch data and parameters when a line is finished. In case of one interpolation path, the core needs $\frac{b_{ww}}{d_{ww}}$ clock cycles for the interpolation of one bus word. Meaning that data is not continuously streamed through the unit.

The actual merge is performed in an extra unit in the form of a pipelined accu that contains one swap buffer and multiple adders. The unit also contains a position logic, that controls which part of the currently streamed aperture contributes to the subaperture. The incoming r_{offset} value determines which cell of an aperture is the first that contributes to the subaperture. The position cannot be loaded directly as the same data streams through all parallel factorizer PEs at the same time. Therefore, a selection mechanism for valid data is required. As the offset value results from the distance between the midpoint of the subaperture and the aperture, the merge can be seen as shifting, the aperture an individual value against the subaperture. After shifting the cells are aligned and can be accumulated. As a result of the shit, some values of the aperture are not relevant for the subaperture. Since both lines have the same length, this also means that some subaperture position will not be updated.

The position logic also tracks the offset values of all preceding apertures for this merge to compare them against each other. Each aperture that is added will be shifted a different amount of cells, wherefore different zones can exist within a subaperture after all apertures were accumulated. A zone where no values were added (should be 0), a zone where only one value was added and a zone where more than one values were added. The zones are required to either null an area, just write sample values to an area, or to add sample values to an area.

The SO FSM depicted in Fig. 5.20 is responsible for the transfer of merged subapertures to the external RAM and is triggered $f_{apt}[s]$ less often per iteration s then the SI FSM. The *idle* state checks for prepared merge sets in the config mem as they contain the write address, header information for the new subaperture and the flags for GBP control. When a merge set is present, the bus transfer is prepared and the *write_back* state is activated in case the bus is free. Write back starts but might be stopped in case the bus is busy, which would lead to a ping pong between *fifo_full* and *write_back* state. From both states, the same transitions are possible, as both could mark the last sample transfer of the merge set. The loop over the *idle*

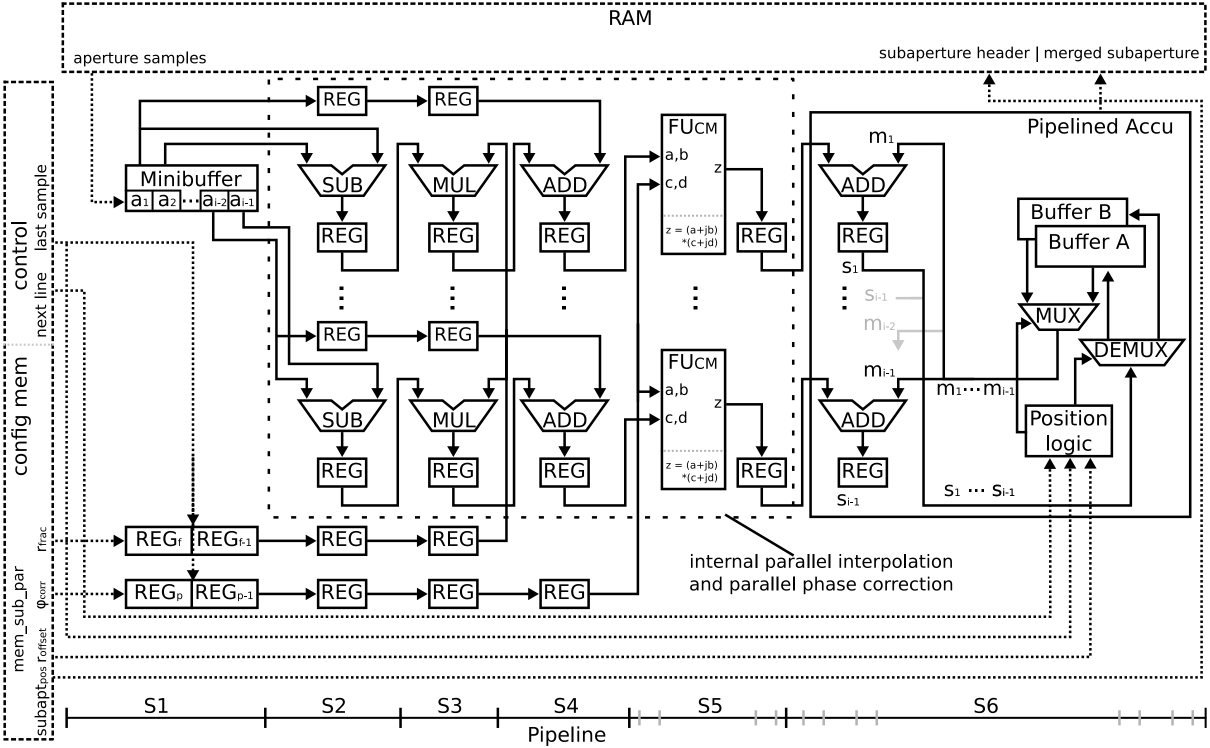


Figure 5.19: Block diagram of factorizer PE

state runs until the subimage set is finished. When finished and the last iteration stage is reached, the GBP is triggered in *gbp_start* to start processing the subimage data as a part of the final image. The *gbp_ctrl* state waits until the GBP is done, it either goes back to *idle* in case of upcoming subimages or stops in the *finished* state. This state can also be reached from *idle* in case the last subimage is a dummy image that was set to be bypassed from the iterator. The bypass, last iteration and finish flags are all set by the iterator. After finishing, the *idle* state is triggered to wait for the next FFBP run.

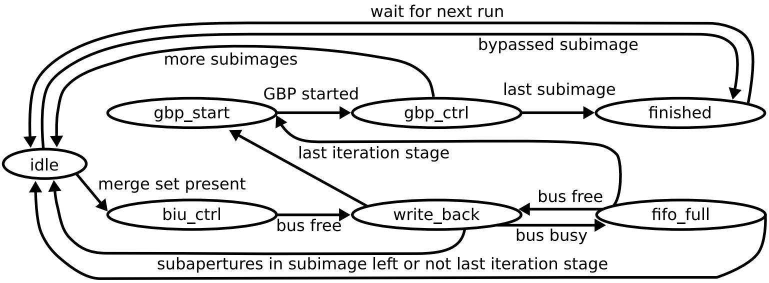


Figure 5.20: SO FSM for data out control of factorizer PE

5.4.4 Full FFBP system

Figure 5.21 shows the combination of the FFBP and GBP PEs to process the FFBP algorithm. Both modules receive data from user registers including constants, image dimensions, configurations, etc. Both modules interact with external RAM to either form new subapertures or subimages. Both modules can be instanced multiple times while the degrees of parallelization p of each module do not depend on each other. Parallelization in case of the FFBP means multiple factorization PEs since the mapper is not a time-critical element. The config mem is also required only once and can feed all p_{fact} factorizer PEs. Only the size needs to be adjusted to hold all required subaperture merge information for multiple PEs. The config mem is split into two memory banks, which are separated to store the different sets from the iterator and rgline_par stage. Based on this information the factorizer PEs create new subapertures. The control information is included in these sets, telling the iterator when to trigger the entire set of GBP modules. It is important to understand that each factorizer PE can only trigger the entire set of GBP modules and not a single PEs. Like this, the subimage is processed by all PEs simultaneously, which allows sharing the input data from the RAM. The parallel factorizer PEs and GBP PEs might block each other when writing the data back to memory, since both need excessive RAM access. To reach maximum performance, all modules listen on the bus to transfer data as soon as possible.

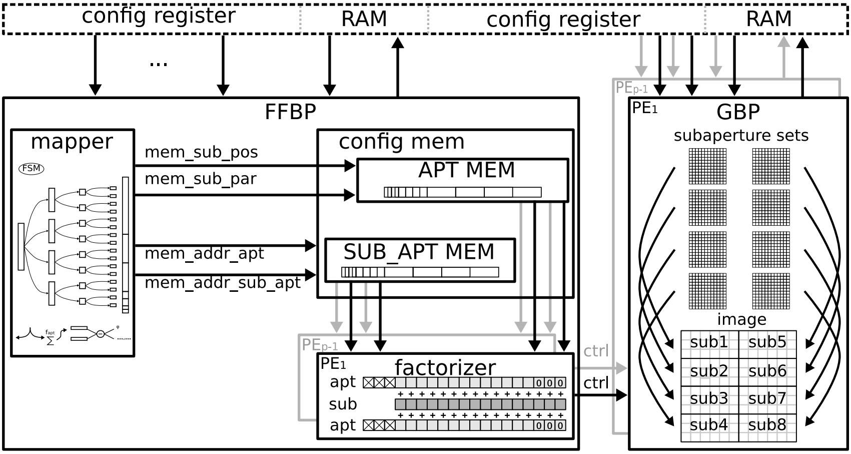


Figure 5.21: FFBP and GBP modules combined over the RAM

5.5 Integration of dedicated modules

The architectural elements described before were transferred to an FPGA system. To be independent of a specific FPGA or emulation platform all modules are embedded in a special framework which is designed to make future technologies easier to use. The so-called Unified EMUlation Framework (UEMU) [94] is used for this purpose. Developed architectures can easily be ported to various commercial or self-designed platforms. This environment, which has already been used in [53] for flexible platform integration enables for simple control of various hardware modules on the software side. The UEMU which is depicted in Fig. 5.22 combines high-level software with an FPGA emulation system. The control is possible via MATLAB or C/C++ routines. Gigabit Ethernet is used to transfer sensor and control data. The main advantage is the simple switching of emulation systems by changing the synthesis options referring to the used emulation platform. As no special hardware or memory requirements are imposed by any of the developed GBP or FFBP PEs, e.g. by the use of proprietary IP cores, it is possible to map this architecture on different FPGA platforms without modification. This feature is possible by using system abstraction layers. While platform-independent modules like the GBP or FFBP PEs are encapsulated in a subsystem (SUB) layer, platform-specific modules (e.g. memory controllers) are located in the interchangeable main system layer. This decouples the subsystem from the particular platform type and offers high flexibility in terms of technology. Modifications in the subsystem will automatically affect all platforms [58].

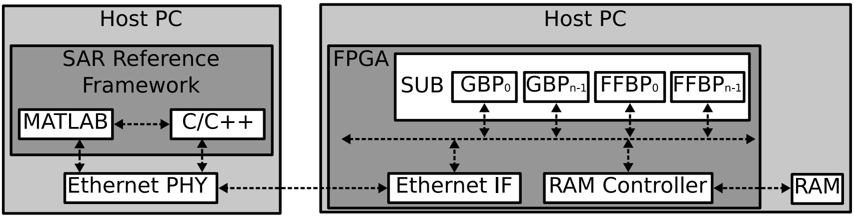


Figure 5.22: Coupling of emulation system and Host-PC with UEMU [53]

The supported platforms are the Xilinx ML605 [95], the BEEcube BEE4 rapid prototyping system [96], the Xilinx VC707 [97], some Altera FPGA based platforms, and the Virtex 5 MCPA board developed at the Institute of Microelectronic Systems (IMS) [98]. But other platforms can be integrated as well.

6 Results and evaluation

This chapter will present and evaluate the results for the three presented architecture implementations of the Global Backprojection (GBP) architecture, the modified Frequency Modulated Continuous Wave (FMCW) architecture and the Fast Factorized Backprojection (FFBP) architecture. The evaluation will cover the utilized resources for differentCommercial Of The Shelf (COTS) development boards with mid-sized Field Programmable Gate Arrays (FPGAs). The presented runtime results of all architectures refer to the same development boards and FPGAs while being compared to a software implementation running on a standard General Purpose Processor (GPP). The results in image quality are independent of resource consumption and runtime, but the fixed-point format induces loss in quality. Therefore, all image results are compared to the floating-point format software implementation. All results are discussed in the closing remarks of the chapter.

6.1 Resource utilization

As a proof of concept and for resource utilization comparison, the results for the different architectures, mapped on one of the FPGA platforms currently supported by the Unified EMUlation Framework (UEMU) framework, are presented. As described in Section 4.1 FPGA types differ in many aspects, therefore they are not 100 percent comparable in terms of resources. The here presented results all refer to one specific FPGA type, the Xilinx XC6VLX240T Virtex 6 [99]. The FPGA was used in combination with the Xilinx ML605 development board [95] for accessing external Random Access Memory (RAM) and general data I/O interfacing. The FPGA type was picked due to the moderate ratios between Block Random Access Memory (BRAM), Digital Signal Processor (DSP) and Look Up Table (LUT) elements. Nevertheless, any FPGA is a potential host for the presented architectures as the system is described in a generic way, which offers the possibility to adapt the amount of Processing Elements (PEs), processing accuracy as well as raw data size and image dimensions. The results for this FPGA can serve as a base for extrapolation to other FPGA types based on the resource ratios. As mentioned in Section 4.1, the used design tool can slightly alter the results within each run and especially when mapped on other FPGA types, due to the complex nature of the algorithms used for mapping.

6.1.1 GBP resource utilization

The exemplary synthesis results for the presented concept of a dedicated GBP architecture on a XC6VLX240T Virtex 6 FPGA [99] are discussed in the following.

Table 6.1: Resources of Xilinx XC6VLX240T Virtex 6 FPGA [99]

FPGA		year	process	BRAM		LUT		DSP	
family	type		$[nm]$	size $[Kb]$	#	type	#	type	#
Virtex6	XC6VLX240T	2009	40	36	416	6-LUT	150720	25x18	768

The resource capacity of this FPGA type are listed in Table 6.1. The utilized hardware resources for the GBP architecture depend on different parameters which are pre synthesis customizable over generics in a certain range. These parameters are listed in Table 6.2, with their possible range and with the resources that will be affected by them. Due to the vast amount of possible permutations, only a selection of these parameters will be backed up with synthesis results. For better comparison, all presented results rely on identical sized quadratic raw and image data. Meaning that any of the parameters is adjustable given that $N_{az} = N_{rg} = N_x = N_y$, which affects the maximum processable image dimension. For the presented setups, the GBP architecture was synthesized with $100\ MHz$ clock rate, while the bus and the external RAM run a higher clock rate. The relation between parameters and resource utilization for different parallelization degrees p and varying image and raw data size N_{max} is shown in Table 6.3. Accuracy is scaled by the number of filter taps N_{taps} and sets N_{sets}, the number of Coordinate Rotation Digital Computer (CORDIC) iterations C_i and the data word width d_{ww}. Processing is done in fixed-point format with complex valued data at $2 \cdot d_{ww}$ bit (real,imaginary). In the following evaluation $N_{taps} = 8$, $N_{taps} = 16$, $C_i = 14$ (14 bit fraction accuracy) and $d_{ww} = 16$ for real and imaginary part of the sensor data stay fixed. The geometrical data were set to 32 bit accuracy (fixed-point arithmetic).

Table 6.2: Generics of dedicated GBP architecture for pre synthesis

generic	value range	affected resource type
apertures N_{az}	2^n, $n =$ 4,5,6, ...	LUTs
range samples N_{rg}	2^n, $n =$ 4,5,6, ...	BRAMs
image pixel in x N_x	2^n, $n =$ 4,5,6 , ...	LUTs
image pixel in y N_y	2^n, $n =$ 4,5,6, ...	BRAMs
parallel GBP PEs p	1,2,3, ...	BRAMs, DSPs, LUTs
CORDIC iterations C_i	5,6,7, ...	LUTs
FIR tabs N_{taps}	2^n, $n =$ 1,2,3, ...	BRAMs, DSPs
FIR tabs N_{sets}	2^n, $n =$ 0,1,2, ...	LUTs

Table 6.3: Resource utilization for the GBP architecture on XC6VLX240T Virtex 6

N_{max}	p	LUT	BRAM	DSP
1024	8	57k (38%)	128 + 32 (38%)	272 (36%)
2048	8	57k (38%)	128 + 64 (46%)	272 (36%)
4096	8	58k (38%)	128 + 112 (57%)	272 (36%)
4096	4	29k (19%)	64 + 56 (29%)	136 (8%)
4096	1	8k (5%)	16 + 14 (7%)	34 (5%)

The results shown in Table 6.3 prove that p scales every resource type by the same factor. This is just logical as every physical instance of a PE requires always the same amount of each resource. This also reveals that the resources required for the control structure around the PEs only requires a marginal amount of logic, which results to $((4 \cdot 8K) - 29k)/(4-1) = 1k$ LUTs and no BRAMs or DSPs. This results in $8k - 1k = 7k$ LUT elements per PE.

A modification of N_{max} only affects control signals which have only marginal impact on logic (LUTs). The same applies to the number of used DSPs as they are only required for arithmetic operations. Only BRAMs are directly affected by p and N_{max}. To process one valid sample (32 bit) in each clock cycle, the internal 8-tap Finite Impulse Response (FIR) filter of a PE requires 8 physical BRAMs to access 8 samples as a single word. As one single BRAM primitive can store up to 1024 32 bit samples, the number of BRAMs allocated by the FIR filter would not increase for $N_{max} \leq 8 \cdot 1024$. To maintain data streaming, double buffering is required which results in 16 BRAMs per parallelization degree p, resulting to 128 BRAMs for $p = 8$.

Additional BRAMs are required for line accumulation and depend on N_{max}. Accumulating N_{max} samples with $2 \cdot 16$ bit requires $2 \cdot (\log_2(N_{max}) + 16)$ bit in order to not create an overflow. $N_{max} = 1024$ this results to $2 \cdot 26 = 52$ bit. Storing $N_{max} = 1024$ accumulations á 52 bit requires 53248 bit which fit in two BRAMs, each holding 36000 bit max. Multiplied again by two (for double buffering) this results two four BRAMs. With $p = 8$ this results to $8 \cdot 2 \cdot 2 = 32$ BRAMs. Accumulating $N_{max} = 2048$ samples requires $2 \cdot (\log_2(N_{max}) + 16) = 54$ bit. Storing $N_{max} = 2048$ accumulations á 52 bit requires 110592 bit which fit in four BRAMs each holding 36000 bit max. With $p = 8$ and double buffering this results to $8 \cdot 4 \cdot 2 = 64$ BRAMs. Accumulating $N_{max} = 4096$ samples requires $2 \cdot (\log_2(N_{max}) + 16) = 58$ bit. Storing $N_{max} = 4096$ accumulations á 52 bit requires 238568 bit. This fits in seven BRAMs each holding 36000 bit max. With $p = 8$ and double buffering this results to $8 \cdot 7 \cdot 2 = 112$ BRAMs. As a result, line length N_{max} can be extended without significant effects on resources for $p = 1$ (non-parallel processing). Whereas increasing $p > 1$ has a clear impact on logic cells, BRAMs and DSPs even for comparatively short lines N_{max}.

Table 6.4: Resource utilization for the GBP architecture including the UEMU framework on a XC6VLX240T Virtex 6

N_{max}	p	LUT	BRAM	DSP
1024	8	57k + 29k (57%)	128 + 32 + 18 (43%)	272 (36%)
2048	8	57k + 29k (57%)	128 + 64 + 18 (50%)	272 (36%)
4096	8	58k + 29k (58%)	128 + 112 + 18 (62%)	272 (36%)
4096	4	29k + 29k (38%)	64 + 56 + 18 (33%)	136 (18%)
4096	1	8k + 29k (24%)	16 + 14 + 18 (11%)	34 (5%)

Changing N_{max} has only little impact on the LUT utilization and the number of used DSPs, since DSPs are only required for arithmetic operations. The number of BRAMs is determined by N_{max} (and p certainly), as they are mainly used for the in/out buffers and the image line accumulation. One PE includes 38 DSPs. The determination of δr uses 4 DSPs: 2 for each CORDIC module. The calculation of the correction, the interpolation index and phase angle calculation requires 9 DSPs in total for all multiplications. In order to determine the phase correction factor (exponential function with sine/cosine) 2 DSPs are required within the CORDIC module. 16 DSPs are part of the FIR interpolation: 1 for the real and 1 for the imaginary part for each of the 8 taps. Finally, 3 DSPs are utilized for the complex multiplication for phase correction. This sums up to 34 DSPs

As already mentioned in Section 5.5, each module is embedded in a subsystem of a framework called UEMU. This framework allows to switch development boards, and thereby FPGAs, without bigger effort. This framework basically enables the system to access the external RAM and to interface with the periphery of the outside world. Of course this requires additional resources. Compared to the GBP PE the framework is not affected by scaling and therefore only utilized a fixed amount of resources.Table 6.4 gives an overview on the resource utilization of the entire system (UEMU + GBP) with the same GBP configurations as in Table 6.3. The constant amount of resources for UEMU sums up to $29k$ LUTs and 18 BRAMs. Overall, the combination identifies LUTs as a limiting factor for parallelization p, while BRAMs are the limiting factor in increasing line lengths N_{max}.

The GBP PEs can be mapped on different FPGAs to utilize the resource maximum. While the percentages of resource utilization after synthesis are final for the given configurations in Table 6.4, it has to be mentioned that the required LUT elements (logic) need to be mapped on LUT slices (areas with a fixed amount and assembly of LUT elements). The mapping creates an overhead and can leave a considerable amount of LUT elements unused. A real 100% utilization is therefore never achieved. This is called a routing limit which is assumed to be at roughly 90 % of LUT resources. This limit is soft and is crossed in some cases. But if a FPGA design is too large, the required timing constraints might not be met, as the delays of the logic and the routing becomes too large. This results in a reduction of clock frequency and thereby in a reduction of throughput.

Depending on the system size and the resource distribution on a specific FPGA, the process of mapping can vary in efficiency. For example, for the GBP architecture on a Xilinx XC6VLX240T Virtex 6 FPGA, LUT utilization is increased by roughly 20% by the mapping process. The limitation in p on this FPGA is caused by LUTs. Mapping increases the limitation by LUTs even more, as the 58% of LUT utilization ($p = 8$, $N_{max} = 4096$) are increased to 78%. This can be observed in Fig. 6.1, which shows the maximum parallelization degree p of the GBP PE on different developments boards with different FPGA types and size.

Figure 6.1 classifies the platforms in respect of their limitation in parallelization p for a fixed line length $N_{max} = 4096$. Since the UEMU framework is included in the system the curves are characterized by a offset value that is induced by the framework, which is only required once. For every FPGA, classification always depends on the dominating resource in percent on the y-axis. Depending on this resource ratio, p (on the x-axis) is either limited early by BRAMs or LUTs or DSPs. Since the ML605/VC707/VCU118, as well as the MCPA/PCIE 385N, provide a comparable LUT to BRAMs ratio, their curves show a similar shape. While the limiting factor in the Xilinx FPGAs are either BRAMs or LUTs, the Altera FPGA and MCPA board are limited by the number of DSPs. This shows in the crossing of ML605 and the PCIE 385N curve at roughly $p = 5$. As the UEMU does not require any DSPs, but mostly logic (ALM for Altera), the PCIE 385N performs better than the ML605 for low parallelization degree p, while higher degrees of p hit the DSP limit. Since the PCIE 385N belongs to another family of FPGAs with a different architecture, the resources are not directly comparable to the FPGAs from Xilinx.

6.1.2 FMCW resource utilization

The architecture for FMCW processing is a modified GBP architecture with additional elements to correct the failing assumption of the start-stop-approximation for pulsed Radio Detection And Ranging (Radar). The resource utilization of the FMCW PE is therefore quite similar to the architecture of the GBP PE. The extra steps to correct this assumption are highlighted grey in Fig. 5.11. Table 6.5 lists the resource utilization after the mapping of the FMCW PE for varying N_{max} at $f = 100\ MHz$ clock frequency at different parallelization degrees p. The configuration of the CORDIC module was identical to the others in the system, so 14 pipeline stages. Also the filter was configured as an 8-tap filter.

Since a major part of the resources is allocated by the regular GBP architecture, the results show the same behavior when scaling p or image size. The number of BRAMs stays the same, only LUTs and DSPs results change. The amount of utilized LUTs is increased by roughly $1\ K$ elements. This is caused mainly through the added

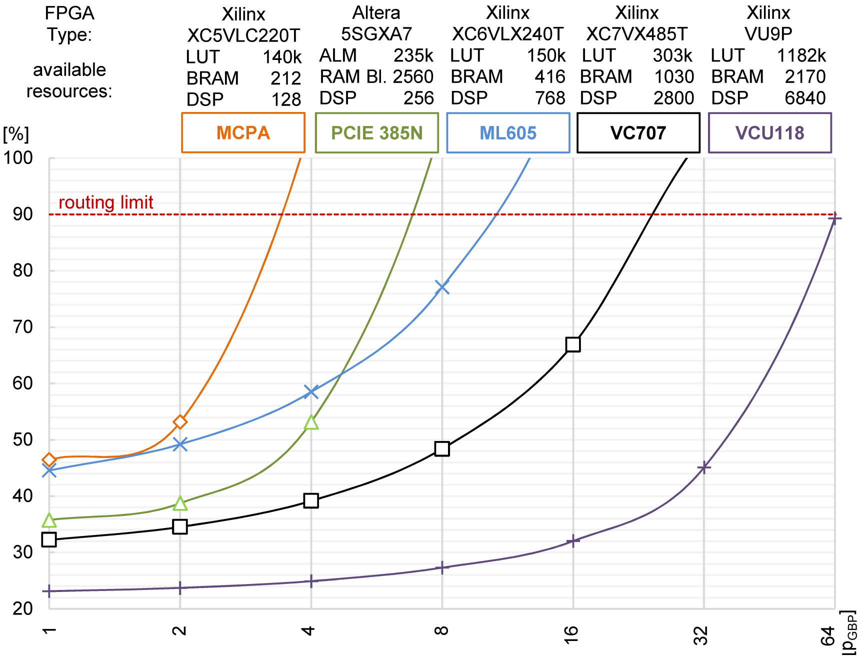


Figure 6.1: Resource utilization for different parallelization degrees p for GBP+UEMU with N_{max}=4096 on different FPGAs [54]

First In – First Out (FIFO) buffers, the adder and registers. The utilization of DSPs raises to 36. This is 2 more then a GBP PE utilizes. The standard determination of δr uses 4 DSPs: 2 for each CORDIC Function Unit (FU). By computing $\delta r_{corrected}$ with another CORDIC module, 2 more DSPs are utilized compared to the GBP PE.

Table 6.5: Resource utilization of FMCW architecture on a XC6VLX240T Virtex 6

N_{max}	p	LUT	BRAM	DSP
1024	8	65k (43%)	128 + 32 (38%)	288 (38%)
2048	8	66k (44%)	128 + 64 (46%)	288 (38%)
4096	8	66k (44%)	128 + 112 (57%)	288 (38%)
4096	4	34k (23%)	64 + 56 (29%)	144 (19%)
4096	1	9k (6%)	16 + 14 (7%)	36 (5%)

Table 6.6: Resource utilization for the FMCW architecture including the UEMU framework on a XC6VLX240T Virtex 6

N_{max}	p	LUT	BRAM	DSP
1024	8	65k + 29k (62%)	128 + 32 + 18 (43%)	288 (38%)
2048	8	66k + 29k (63%)	128 + 64 + 18 (50%)	288 (38%)
4096	8	66k + 29k (63%)	128 + 112 + 18 (62%)	288 (38%)
4096	4	34k + 29k (42%)	64 + 56 + 18 (33%)	144 (19%)
4096	1	9k + 29k (25%)	16 + 14 + 18 (11%)	36 (5%)

Table 6.6 lists the resource utilization with UEMU framework. Of course, only a fixed amount for the UEMU framework is added to the system regardless of p. The results show that the modification did not alter the results which resources are limiting for parallelization, wherefore no graph is presented. The LUT elements are still the binding resource, given the fact that they increased with the modification. Since overall resource consumption is similar to the GBP also the parallelization on different platforms is similar as DSPs show the lowest consumption rate. The platforms that are handicapped by an increase of DSP utilization, are the MCPA and the PCIE 385N platform, as the used FPGAs are characterized by a smaller ratio between DSPs and LUT elements. In general, it can be stated that the modification can be implemented without changing the maximum possible parallelization degree p when compared to the GBP.

6.1.3 FFBP resource utilization

Like the GBP module, the FFBP module is scalable in terms of throughput and processible data dimensions. For a fair evaluation, the same assumptions for image and raw data dimensions apply as for the GBP evaluation. Maximum processible configurations of f_{apt} and f_{sub} depend on the raw data size, as RAM memory of max $4\ GB$ on the used ML605 might be exceeded by too many subimages. The utilized hardware resources for the FFBP architecture depend on different parameters, which are customizable pre synthesis over generics in a certain range. These parameters are listed in Table 6.7 with the range of possible customization and the type of resources which will be affected by them.

Since the FFBP module contains three different PEs of which only the factorizer PE is instanced multiple time, the resources for each PE are listed in Table 6.8 for a line length of $N_{rg} = 4096$. Resources in the form of LUTs and DSPs are used for processing the set information, generated by the iterator and the rgline_par step of the mapper. Due to the limitation on the factors to the power of two for raw data dimensions and aperture factorization, address calculations of the iterator can be

Table 6.7: Generics for thes dedicated FFBP architecture for pre synthesis

generic	value range	affected resource type
apertures N_{az}	2^n, $n =$ 4,5,6, ...	LUTs
range samples N_{rg}	2^n, $n =$ 4,5,6, ...	BRAMs
aperture factorization f_{apt}	2^n, $n =$ 0,1,2, ...	BRAMs
subimage factorization f_{sub}	2^n, $n =$ 0,1,2, ...	LUTs
iteration stages s	n, $n =$ 0,1,2, ...	none
parallel FFBP PEs p	1,2,3, ...	BRAMs, DSPs,LUTs

handled by LUTs with shifts and adds. For the parameter module, CORDIC cores add more LUTs while other parameter calculations induce DSPs. Beside the fact that no BRAMs are used, resource consumption is similar to one GBP PE. But a mapper is only instanced once and does not depend on the range line length. According to Fig. 6.1 approximately 12 GBP modules can fit on a ML605 before the routing limit is reached. Substituting one out of 12 GBP PEs would reduce processing speed by $1/12$ while adding a mapper with additional factorizer PEs has the potential to speed up processing significantly based on the chosen configuration. Therefore, resource consumption is considerably low. For the config mem, only a small memory depth is required, as only small sets of parameters are saved for every aperture in a subimage set. Therefore, no BRAMs are used because small memories are more efficient to be mapped to LUT elements.

The factorizer is the only PE utilizing BRAM elements in order to establish the swap buffers with N_{rg} samples. Accumulating f_{apt} apertures with a sample width of $2 \cdot 16$ bit requires $2 \cdot (\log_2(f_{apt}) + 16)$ bit in order to not create an overflow. For $f_{apt} = 16$ this results to $2 \cdot 20 = 40$ bit. Storing $N_{rg} = 4096$ accumulations á 40 bit requires 163840 bit, which fit in five BRAMs each holding 36000 bit max. Multiplied again by two for swap buffering this results two 10 BRAMs. LUT and DSP resources are only affected by parallelization p. This is because the scaling is only made in the factorizer and not in the mapper. This means that the resources for the address calculations and the determination of the parameters remain constant. When checking Table 6.9, it is obvious, that with a growing number of parallel modules, the resource consumption is low compared to a GBP module since only the factorizer would be instanced multiple times. This allows for wide parallel use of the FFBP PE also for longer range lines.

For the presented setups, all FFBP PEs can be synthesized with $100\ MHz$ clock frequency, while the bus and the external RAM run at a higher clock frequency.

Figure 6.2 classifies the platforms in respect to their limitation in parallelization p for a fixed line length $N_{max} = 4096$. Here parallelization is divided in p_{GBP} for the GBP module and p_{FFBP} for the FFBP module for clear understanding. p_{FFBP} refers

Table 6.8: Resource utilization for one FFBP PE broken down for the different units with $N_{max} = 4096$ on a XC6VLX240T Virtex 6

PE	LUT	BRAM	DSP
mapper	9k (6%)	0 (0%)	32 (4%)
factorizer	2k (2%)	10 (2%)	6 (1%)
config mem	400 (<1%)	0 (0%)	0 (0%)
total	12 (8%)	10 (2%)	38 (5%)

to the external parallelization of factorizer PEs and not to the internal parallelizaiton of a single factorizer PE. The x-axis splits in segments with different p_{GBP} which are again split in different segments of p_{FFBP}. Like in Fig. 6.1 every curve starts with a bigger offset for resource utilization in percent (on the y-axis) because the UEMU framework is included. For the FFBP another offset is introduced by the mapper and config mem PEs which are instanced only once. Segments with a low p_{GBP} show a significant growth in utilization when p_{FFBP} is increased. This impact fades with growing p_{GBP}, which indicates that especially for bigger images (where a high degree of p_{GBP} is required) the FFBP can be added with a small growth in resource consumption. Similar to the GBP, utilization on different FPGAs reveals a similar behavior between the ML605/VC707/VCU118 and MCPA/PCIE 385N platforms. Especially for the MCPA platform, the additional FFBP modules limit the parallelization even further, due to the considerably low amount of DSPs.

A full FFBP systems is combined with the GBP and embedded in the UEMU framework. Picking a combination of GBP and FFBP modules depends on the FFBP configuration. In general, since the GBP has a higher degree of complexity, more GBP then FFBP modules should exist in a system. Only a vast amount of subimages would back up a similar or even higher amount of FFBP modules. But also this is

Table 6.9: Total resource utilization for the FFBP architecture with multiple PEs and different N_{max} on a XC6VLX240T Virtex 6

N_{max}	p	LUT	BRAM	DSP
1024	8	26k (18%)	32 (8%)	80 (11%)
2048	8	26k (18%)	48 (12%)	80 (11%)
4096	8	26k (18%)	80 (20%)	80 (11%)
4096	4	18k (12%)	40 (10%)	56 (8%)
4096	1	12k (8%)	10 (2%)	38 (5%)

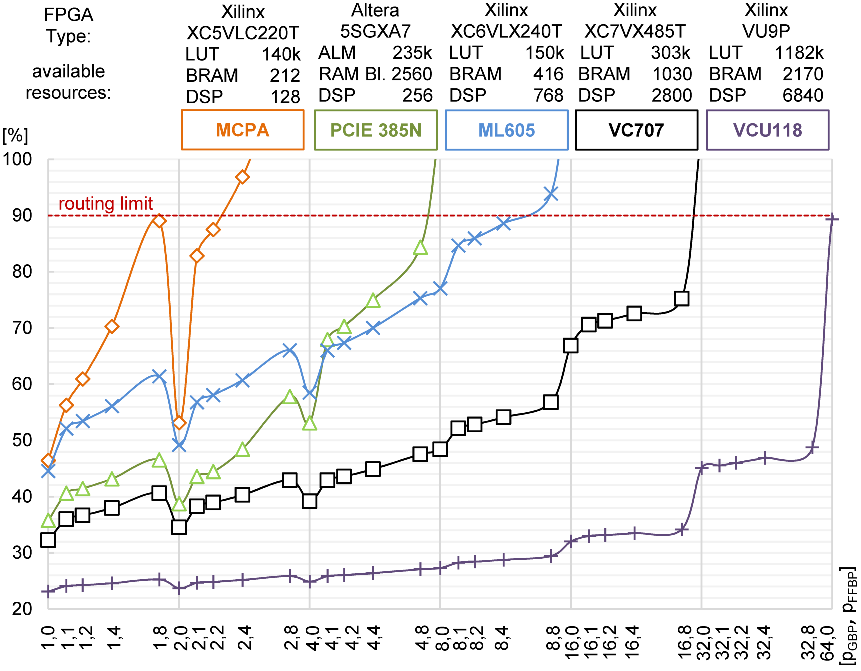


Figure 6.2: Resource utilization for different parallelization degrees p for GBP+FFBP+UEMU with N_{max}=4096 on different FPGAs

limited, as the FFBP becomes equivalent to the GBP when the amount of subimages approaches the number of image lines. In terms of quality, this setup would result in an image similar to an image processed the GBP, but with an increased runtime due to the FFBP overhead. Furthermore, no Pareto point in Table 5.4 exceeds a factorization $f_{apt} > 4$ in a single iteration stage. All results support that resources for the FFBP should only be invested to a certain boundary in relation to the GBP. Based on this information, two realistic full system configurations are selected and listed in Table 6.10 with full resource utilization (including UEMU) for the ML605 emulation platform. The resources for the FFBP PEs correspond to the first summand of a column, the second summand describes the resources of the GBP PEs and the third summand the part of the UEMU. The configurations were also picked with the assumption that image size and raw data size are equal ($N_{max} = 4096$). Therefore, they represent a regular Synthetic Aperture Radar (SAR) case where both dimensions are not split by orders of magnitude. In case of extreme conditions, for example, exceptional long-range lines, other configurations can perform better. The picked configurations are also used for runtime measurements.

Table 6.10: Resource utilization for different parallelization degrees p for FFBP and GBP including UEMU with N_{max}=4096 on XC6VLX240T Virtex 6

p_{GBP}	p_{FFBP}	LUT FFBP+GBP+UEMU	BRAM FFBP+GBP+UEMU	DSP FFBP+GBP
8	2	14k + 58k + 29k (67%)	20 + 240 + 18 (67%)	44 + 272 (41%)
4	4	18k + 29k + 29k (51%)	40 + 120 + 18 (43%)	56 + 136 (25%)

6.2 Performance results

Resource utilization is important to understand the physical limitation of a design on a specific FPGA. The results can be used to approximate runtime on a device for a streaming design. Such approximations can be very accurate as long as the surrounding periphery can keep up data I/O. Especially for data greedy applications this can be challenging and might limit processing speed although resources would allow for a higher speedup. Furthermore, an approximation can be accurate for one PE but multiple PEs can cause other effects that slow down a design. This can have multifactorial roots like dependencies or a lack of bandwidth due to increased parallelism. Such effects or not easy to foresee in complex systems. Therefore, runtime results are measured for different configurations of all presented designs and compared to the expected results.

6.2.1 GBP performance results

The theoretical computational runtime for the GBP can be calculated ($calc_{GBP}$) with the amount of required projections based on raw data size $N_{rg} \cdot N_{az}$ and image data size $N_x \cdot N_y$. Assuming that a system can manage one Pulse to Pixel Projection (PPP) per clock cycle at clock frequency PE_{cf}, the runtime in seconds results to Eq. (6.1)

$$calc_{GBP} = \frac{N_x \cdot N_y \cdot N_{az}}{PE_{cf}} \tag{6.1}$$

In an ideal scenario, runtime for p multiple PEs results to Eq. (6.2).

$$calc_{GBP} = \frac{N_x \cdot N_y \cdot N_{az}}{p \cdot PE_{cf}} \tag{6.2}$$

The calculation shows that for the GBP the range line length does not matter. This is because only one sample from each range line is required to process a pixel. The single pixel across all range lines form the semicircle around the antenna position that cross the calculated pixel position.

Table 6.11: Comparison of hardware, software and theoretical runtime for the GBP with $p = 8$ and different configurations for image size and raw data size

$N_x \cdot N_y$ [px]	$N_{az} \cdot N_{rg}$ [samples]	sw runtime [s]	hw runtime [s]	$calc_{GBP}$ runtime [s]	efficiency change $calc_{GBP}$/hw	speed up factor
1024*1024	256*256	35.87	0.36	0.34	0.94	99
	512*512	72.21	0.71	0.67	0.94	101
	1024*1024	136.00	1.42	1.34	0.94	96
	2048*2048	282.67	2.83	2.68	0.94	100
2048*2048	256*256	133.83	1.38	1.34	0.97	97
	512*512	263.31	2.76	2.68	0.97	95
	1024*1024	537.21	5.54	5.37	0.97	97
	2048*2048	1101.75	11.05	10.74	0.97	100
4096*4096	256*256	525.74	5.42	5.37	0.99	97
	512*512	1052.55	10.84	10.74	0.99	97
	1024*1024	2118.67	21.68	21.47	0.99	98
	2048*2048	4355.15	43.52	42.95	0.99	100

The GBP architecture was designed to allow for constant streaming of data even with multiple PEs due to the data broadcasting that was possible through a specific processing order. As a proof of concept, the real runtime is measured on a ML605 platform. Different combinations of image and raw data size are measured for the maximum parallelization degree of $p = 8$, on the ML605 at $PE_{cf} = 100\ MHz$ clock frequency. The real hardware runtime was measured with counters that increment each clock cycle from the start of processing until the entire image is generated. For a rating of the computational runtime, a software reference in MATLAB after [52] is used for comparison. The software (sw) runtime was measured on an Intel i5 running at 3.2 GHz without parallelization. Table 6.11 lists the software (sw) runtime together with the hardware (hw) runtime, as well as the calculated ($calc_{GBP}$) ideal runtime. The calculated runtime in the table is based on Eq. (6.2). The ratio of hw and $calc_{GBP}$ shows the losses between the minimum possible runtime and the actual time required in hardware. The listed speedup factors represent the speedup from software to hardware.

Table 6.11 reveals that doubling image dimensions results in quadrupling runtime while doubling raw data dimensions only doubles the runtime when image dimensions stay constant. This is caused by the fact, that during GBP processing only one range sample from each range line is required for the projection of one pixel. The range line length N_{rg}, therefore, has no impact on runtime. Nevertheless N_{rg} can not be increased without consequences as it has a significant impact on BRAM usage. Comparing hw and sw runtime shows a dramatic reduction in absolute runtime especially for high N_{az}, N_y and N_x values, resulting in exponential time savings. An average speedup of 98 is achieved for $p = 8$, while much less space and energy are consumed with an FPGA when compared to a GPP.

In general, the measured runtime indicates that the implemented architecture is close to the calculated (optimal) runtime. Nevertheless, a change in efficiency can be observed. This divergence increases with decreasing image dimension N_y. This results from the time (latency) needed to fill all pipeline stages of the backprojection module. This latency sums up to 55 cycles, from which 42 cycles apply to the three CORDIC modules (each 14 bit accuracy), plus 6 cycles for interpolation, 3 for complex multiplication and 4 for the remaining stages of the GBP module. This latency adds an overhead to each processed image line since the buffers of a GBP module can only be switched when the interpolation is finished. Therefore, the pipeline needs to be filled again with the beginning of a new line after the buffers are switched. This has a stronger impact the shorter the processed image line length N_y. Adding the delay to the calculated runtime $calc_{GBP}$ gives almost the real measured hardware runtimes hw_{GBP} as given in

$$hw_{GBP} = \frac{N_x \cdot (N_y + delay) \cdot N_{az}}{p \cdot PE_{cf}} \tag{6.3}$$

For example, adding 55 cycles to 1024 (N_y) results in 1079 cycles for a full projection. Dividing 1024 by 1079 results to an efficiency factor of 0.97, this factor can be applied to the entire processing time. Depending on the chosen dimension, the speedup factor will vary with the efficiency factor. This shows that the system does not inflict any bigger stalls caused by the limitations of the bus. This is mainly caused by the possible broadcasting of raw data to p PEs. Increasing p will result in p finished lines at a certain point. Since broadcasting only works for reading but not for writing data, more modules increase the waiting time for each PE to transfer data back to memory. For the ML605 four data words with 32 bit, each can be transferred in one bus cycle. This means the bus can handle four PEs without stalls. In case of $p > 4$ the system will get faster but efficiency will reduce since the required time to empty all PE buffers will increase, which will cause longer stall times.

Figure 6.3 depicts the runtime development across different raw data sizes on an ML605 at $PE_{cf} = 100\ MHz$ for $p = 4$ (dashed lines) and $p = 8$ (full lines) while image size is constant. As expected, runtime scales linearly with $1/p$ and N_{az}. What becomes obvious is the huge absolute difference in runtime when only one dimension is increased. This results in exponential growth in case both image dimensions are increased. Since SAR data and images are usually in the region of multiple thousand samples/pixel, the benefit of parallelization shows in the absolute reduction of processing time. Especially, when hardware runtime is compared to software

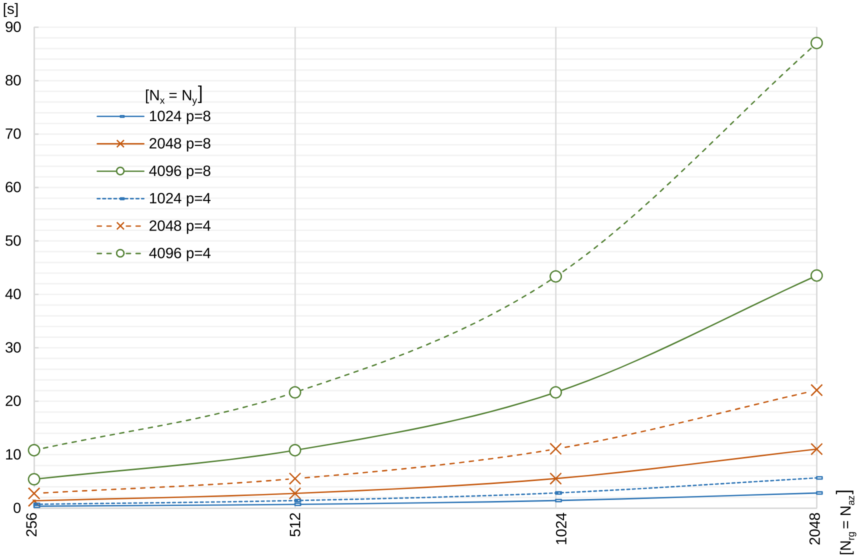


Figure 6.3: Runtime results for different p at different N_{az} with fixed image dimensions

runtime, the change in absolute time saving can range between minutes and hours depending on the dimensions.

6.2.2 FMCW performance results

The FMCW architecture is similar to the GBP architecture. The main difference is the additional changes in the pipeline to compensate for motion during ramping. These additional elements (mainly the additional CORDIC) increases the pipeline length of the FMCW PE. As discussed in Section 6.2.1, the pipeline length has an effect on efficiency that increases with decreasing image line length. Since the pipeline length is increased by a few stages only, the increase in delay is low, wherefore the effect on efficiency is negligible and not discussed further. The FMCW PE is therefore considered to be as fast and efficient as the GBP PE.

6.2.3 FFBP performance results

Processing of the FFBP includes the GBP step. Basically, the FFBP preprocessing is used to reduce GBP runtime by problem size reduction. Therefore, the performance is evaluated by comparing the GBP runtime without FFBP preprocessing, with the

Table 6.12: Runtime results for different FFBP+GBP PE parallelization degrees p at different Pareto points (configurations) and different image dimensions on a XC6VLX240T Virtex 6

algorithm/ configuration	i	f_{apt_i}	f_{sub_i} in x	f_{sub_i} in y	image and data dimensions parallelization degree p (GBP, FFBP) rutime in s											
					$1024*1024$				$2048*2048$				$4096*4096$			
					(4,4)		(8,2)		(4,4)		(8,2)		(4,4)		(8,2)	
					hw	s↑	hw	s↑	hw	s↑	hw	s↑	hw	s↑	hw	s↑
GBP	1	1	1	1	3.7	1.0	1.9	1.0	14.8	1.0	7.4	1.0	59.3	1.0	29.7	1.0
P1	1	2	8	2	2.0	1.8	1.1	1.7	7.6	1.9	3.9	1.9	29.9	2.0	15.1	2.0
P2	1	4	32	2	1.3	2.8	1.2	1.6	4.1	3.6	2.6	2.8	15.7	3.8	8.5	3.5
P3	1	4	16	1	1.0	3.7	0.7	2.7	3.8	3.9	2.1	3.5	15.0	4.0	7.7	3.9
P4	2	4,2	16,4	2,1	1.0	3.7	0.9	2.1	2.4	6.2	1.7	4.4	8.3	7.1	4.7	6.3
P5	3	2,2,4	4,4,4	1,2,1	0.6	6.2	0.7	2.7	1.3	11.4	1.1	6.7	4.3	13.8	2.6	11.4

runtime of the GBP and FFBP preprocessing step combined. The performance not only depends on the permutation of FFBP parameters, but also on the image size, the raw data size and of course the amount of PEs. For evaluation the discussed Pareto configurations were tested on different image dimensions (squared) on the same raw data set which was used for image result evaluation in Section 6.3. The set has 1408 lines in azimuth with 1508 range samples each. Based on the available resources of the ML605, the Pareto configurations were tested on two different hardware setups of four GBP PEs plus four FFBP PEs (4,4) and eight GBP PEs plus two FFBP PEs (8,2). Table 6.12 lists the combined FFBP runtimes together with the stand alone GBP runtime. The speedup value (s↑) indicates the gain for each configuration when compared to the GBP.

Table 6.12 shows that speedup factors (s↑) between 1.6 and 11.4 for (8,2) and between 1.8 and 13.8 for (4,4) can be reached, depending on image dimensions and FFBP configuration. Just based on numbers, regardless of image quality, this shows the high potential of the FFBP preprocessing stage. Especially for bigger images the reduction of absolute runtime is significant.

Comparing the speedup factors for (4,4) and (8,2) across image dimensions and configurations shows that the bigger the image, the smaller the impact of FFBP processing on runtime. For 1024 × 1024, the speedup factors for (4,4) are usually

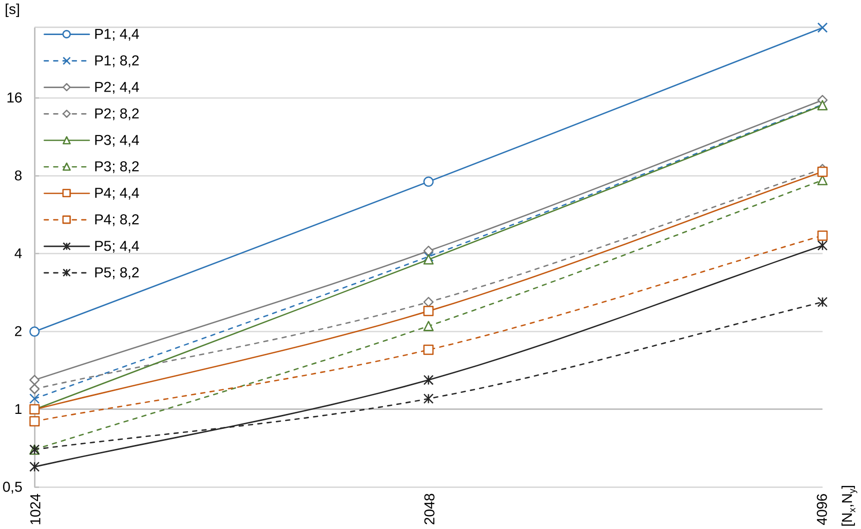


Figure 6.4: Runtime results for different FFBP/GBP PE parallelizaiton degrees p at different Pareto configurations and image dimensions

better than for (8,2) while for 4096 × 4096 they are almost similar. This is because GBP processing mostly depends on the final image size while the FFBP is totally independent of image size and only depends on raw data size which is not scaled in Table 6.12. The impact of small images in combination with high subimage and aperture factorization can also be seen in Fig. 6.4, where both axes are scaled logarithmically instead of linear. While the configurations with more GBP PEs (dashed lines) are clearly faster for bigger images, this changes for smaller images in case of many subimages (P2, P4, P5). For P2 and P4 the starting points of the pairs (dashed and full lines of the same color) for the runtime curves get closer, and for P5 they are even crossed at $[N_x, N_y] = 1024$. For these three configurations and image sizes, the total runtime for the FFBP stage dominates the GBP runtime due to the reduced amount of FFBP PEs. This changes in case the GBP time dominates the process because of large images. It this case, the aperture factorization f_{apt} has a bigger, almost linearly reduction effect on runtime, with only a small overhead depending on the number of subimages. Another factor for FFBP processing is the range line length N_{az}. The longer the processed range line, the more important is a highly parallelized FFBP PE in comparison to the GBP, due to the number of calculations during aperture factorization. This is increased by the amount of generated subimages, as they multiply the amount of required FFBP interpolations.

Since SAR scenarios usually tend to have bigger image dimensions, GBP PEs are more useful. But depending on the scenario and the factorization parameter, it can be useful to not maximize the number of GBP PEs, but rather increase the number of FFBP PEs. Especially if many subimages or iterations need to be performed. Every configuration should be looked at in particular, but the rules of thumb are:

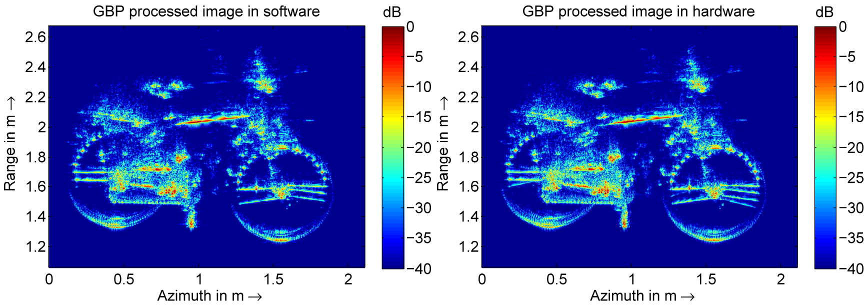
1. the bigger the image dimensions (subimages) the more GBP PEs should be used in comparison to the FFBP
2. the bigger the raw data size in comparison to the image dimensions and/or the more subimages are created, the more FFBP PEs should be used in comparison to the GBP

6.3 Image results

SAR systems provide information in a visual manner that can be directly interpreted by the human eye. To prove that the described system does not alter the provided information significantly, image results of all algorithms need to be compared with reference images. This is necessary for mainly three reasons. First, comparing quality only with metrics is a good indicator, but can also bias the evaluation, wherefore a final visual comparison is required. Second, the used approximations speedup the entire image generation, but also alter the results and therefore need to be counterchecked. Third, the entire processing was converted from floating to fixed-point number format, which limits the dynamic range and thus influences the image result. For comparison, software-generated images with floating-point precision are used as a reference. SAR data for image generation was obtained from a near field FMCW sensor [92] provided by the Ruhr University Bochum (RUB) with the parameters given in Table 5.3.

6.3.1 GBP image results

The GBP is the most accurate SAR algorithm as it does not rely on assumptions like a straight flight path as frequency-domain-based algorithms do. If tracked accurately, all motion deviations can be compensated by the exact geometrical distance calculations of the algorithm. As a consequence, the GBP can be used as a golden reference for image quality. Figure 6.5a shows a SAR FMCW dataset processed in software with the GBP algorithm. The sensor data was obtained while driving a straight path on a linear rail system, passing a laying bike in a distance of $1.2\ m$ from the sensor while being in low elevation. The setup of the SAR system is described in [57]. The system combines high-speed linear movement with the above mentioned Ultra WideBand (UWB) FMCW sensor. The sensor uses a mid-frequency of $80\ GHz$ and can generate highly linear frequency ramps for ramp-lengths between 2 and $16\ ms$ covering a bandwidth of $25.6\ GHz$ per ramp. This allows for a cell resolution of $8 \times 11\ mm$.



(a) Processed in software

(b) Processed in hardware

Figure 6.5: FMCW dataset processed with GBP in software and hardware [53]

Figure 6.5a shows an image with a resolution of 1024×1024 pixel based on a dataset with 1408 range lines with 1504 range samples each. The dynamic range of the image is limited to 40 dB. A bike is clearly recognizable, although not every single feature of the bike is visible. The orientation of the bike to the sensor influences which features are more visible. Figure 6.5b is the same dataset processed with the presented GBP FPGA hardware system. The comparison shows, that neither the implemented CORDIC nor the sinc approximation reduces image quality significantly. This proves, that the simulations and evaluations, which were taken in advance, are very helpful as they allow to implement resource-saving (and therefore faster) implementations, while not reducing image quality. The marginal differences between the two images show that a switch from floating-point to fixed-point number format does not influence dynamic range in a way that image contrast suffers.

6.3.2 FMCW image results

The start-stop-approximation has been described in Section 2.3.3. The impact of violations of the start-stop-approximation has been discussed for many scenarios and algorithms in [56, 100, 101]. In general, a target displacement and a defocussing of the target in range and azimuth is observed. The intensity of these effects depends on the ratio of sensor velocity and FMCW ramp-length. Ribalta [56] has shown, that a modified GBP can almost fully correct these effects on simulated stripmap data. Since this defocussing can decrease image quality significantly, the GBP architecture was modified according to [56] to compensate for this effect. The effect and the results of the correction are shown in Fig. 6.6. To see the effect isolated and not mixed with neighboring reflectors, a single 30 mm corner reflector was used for verification to create a single point target echo. A corner reflector can create a big radar cross-section with comparatively small dimensions. This is possible through the orthogonal arrangement of multiple metal surfaces to each other. A three-way reflection is created that increases the chance of backscattering to the source. During scanning the sensor platform emitted frequency ramps of $14\ ms$ in length. The reference scan was performed in a stop and go mode, so the platform stopped for the time the ramp was emitted. This dataset was used as a reference since the start-stop-approximation is valid and the image is not altered. The second scan was performed with a constant velocity of $60\ cm/s$. The images in Fig. 6.6 are arranged in a way that the first column presents the software-based (floating-point) results, which are considered as a golden reference. The second column shows the hardware processed images for direct comparison. Images in the first row (Figs. 6.6a and 6.6b) were processed with the GBP and show the results of the stop and go scan. As expected both images almost seem identical and show a well-focused point reflector. The images in the second row (Figs. 6.6c and 6.6d) were scanned with a velocity of $60\ cm/s$ during sending/receiving and were processed with the regular GBP algorithm. The effect of motion is clearly visible in the strong defocussing of the point target. This effect increases with ramp-length and velocity and makes the use of the start-stop-approximation questionable for FMCW based SAR in certain scenarios. Processing the $60\ cm/s$ scan with the modified GBP algorithm, which additionally approximates the motion during the transmission, results to Fig. 6.6e and Fig. 6.6f. The significant improvements are clearly visible in the software and hardware version. Defocussing effects in range and azimuth disappear almost completely when compared to the stop and go reference scan of Figs. 6.6a and 6.6b.

These results match with the simulations made in [56]. Considering the software as a golden reference, it can be stated, that the hardware processed results show only minor differences in the main lobe region. In the less important areas around the main lobe stronger differences appear. This is primarily due to the use of different data types, as the software uses floating-point format and the hardware version uses fixed-point format, which limits the dynamic range.

6.3.3 FFBP image results

To prove that the FFBP is not only potent in reducing runtime with relatively low additional resource consumption, hardware processed images need to be compared with the GBP reference. Figure 6.7 shows hardware processed images. Figure 6.7a shows the GBP as a golden reference. The remaining images Fig. 6.7b - Fig. 6.7f show the results of the FFBP configuration points from Table 5.4. These configurations represent corner points of the curve fitted to runtime and the Peak Signal to Noise Ratio (PSNR) value. While runtime reduces when following the Pareto curve to the origin, also PSNR values reduce continuously. This can also be observed in Fig. 6.7b - Fig. 6.7f when the images are compared among each other. While the first two configurations (P1 and P2, Fig. 6.7b and Fig. 6.7c) are almost similar to the GBP reference, P3 (Fig. 6.7d) starts to impair quality visibly. Bigger artifacts start to light up in the lower image sections. This can be explained by the lower amount of subimages of P3 compared to P2. While both configurations only run one iteration and factorize 4 apertures, P2 creates 64 subimages while P3 only creates 16. This reduces the capability to contain the error that is induced by aperture factorization. This effect can be observed at subimage borders. As iterations increase from one to two with P4 (Fig. 6.7e), also aperture factorization increases to eight. It is clearly visible that artifacts start to migrate also to the upper part of the image indicated by the drop of $12\ dB$ compared to P1. This can also not be compensated by increasing the subimage count from 16 to 128 between P3 and P4. The aliasing effect can be explained by undersampling the image as the aperture factorization creates less and fewer range lines. This also explains the ghost images in Fig. 6.7f which shows a considerable loss in quality.

Overall, the image quality corresponds with the reduction in runtime for each configuration. However, all configurations reach a quality that allows identifying the bike clearly. The first FFBP configuration, which can be calculated almost twice as fast as the GBP, is almost identical in quality. This shows the potential to save time with a small reduction in the quality of the image.

6.4 Evaluation and discussion of results

While time-domain-based SAR algorithms are usually not seen as a real-time option, the resource, performance and image results show that even the highly complex GBP algorithm can be considered under certain constraints as real-time capable. This is mainly possible by exploiting all options for parallelization of the algorithm in a streaming design. Modular designs allow for scaling the architecture in terms of size and shows that it is possible to exhaust every FPGA to a high degree. Nevertheless, FPGAs need to be chosen based on their resource diversification as not every ratio suits every configuration when a maximum utilization is an objective. Even mid-sized FPGAs can host multiple parallel processing elements. The upgrade for motion during ramp transmission for FMCW processing only adds a marginal resource amount to the GBP design and will therefore only alter the possible degree of parallelization in border cases. Also, the FFBP architecture can be implemented with much fewer resources then the GBP and can be implemented in parallel to the GBP without limiting its parallelization dramatically. Instead, the main memory consumption is more critical for FFBP usage, as big raw data sets, paired with configurations that create many subimages, can exceed the memory limits by far. Interlacing both modules in a tree structure helps to limit memory usage. As interlacing keeps both steps active at the same time, the module configuration must be adjusted to the available bus limitations to not cause too many stalls in any PE. All in all, the high resource availability of modern FPGAs in combination with a sophisticated design, allows for the implementation of time-domain-based algorithms despite their high degree of complexity.

Runtime results of the GBP indicate an almost reversed ratio between processing time and resources, which comes close to optimum efficiency and shows that almost linear speedup is possible. The average throughput of one PPP per processing element and clock cycle, already enable mid-sized FPGAs to process smaller images in matters of seconds which can take several minutes on a GPP. Similar to the resource consumption of the FMCW architecture, the difference in processing delay compared to the GBP architecture is marginal. Basically, only the pipeline delay is increased, which has almost no effect on average throughput. For the FFBP a clear statement is less simple, as processing time does not scale linearly with resource utilization. Efficiency changes with the SAR scenario and the FFBP configurations because a given ratio of processing elements might not fight to the scenario. But since SAR setups are usually in the range of multiple thousands of samples/pixels, resources should be used mostly for the GBP PEs, as the high degree of complexity dominates processing time. A part of the GBP resources should be used for FFBP implementation. The resource and performance results show that the FFBP implementation can reduce processing time between 50 and 90 percent with a resource increase of roughly 15 percent.

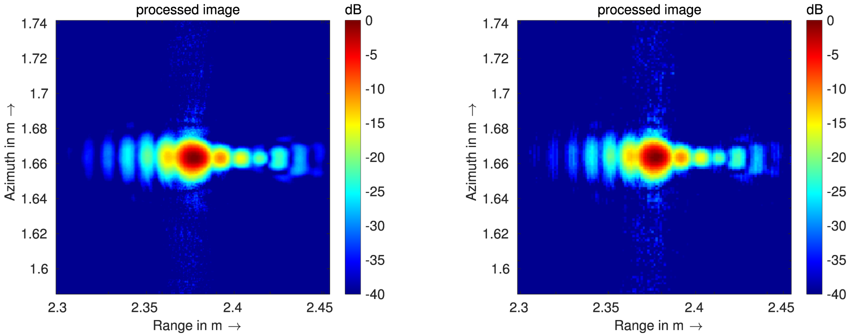
The image results of the GBP, including all approximation and the switch from floating-point to fixed-point number representation, show almost no change in the image when compared to an ideal software reference. This is interesting, as the used near field data set requires a high dynamic range, therefore also other data set should not alter this result. The use of FMCW data can cause major image quality reduction in case of long ramps and a high sensor velocity. The FMCW architecture improves image quality significantly and can compensate for the effects under such conditions almost perfectly. The FFBP evaluation is less clear and must be assessed together with the resource and runtime results. While runtime reduction by 50 percent is possible with almost no loss in image quality (depending on the SAR data), a dramatic reduction of up to 90 percent in runtime can cause clearly visible artifacts in the image. As the error intensity for the different configurations varies, based on the scenario parameters of the data set, this must be evaluated individually. As the modules are not fixed to a certain configuration, this does not influence the implementation itself. This is why the FFBP implementation allows for a significant gain in performance with a controllable loss in quality. The fact that the FFBP can be controlled in complexity allows to max out image quality for each runtime constraint.

Besides image quality and performance, the power dissipation must be as low as possible for mobile environments. Power consumption measurements of the design were conducted for the Xilinx Virtex 6 XC6VLX240T processed in $40\ nm$ on the ML605 development platform. Since semiconductors manufactured in a smaller process usually show lower power dissipation, a Xilinx Virtex Ultrascale+ VU9P FPGA processed in $16\ nm$ on the VCU118 development is evaluated is well. The average power consumption of both FPGAs is based on the Xilinx power analyzer tool. The estimated numbers for the Virtex 6 are backed up with a real power measurement of the ML605. The tool analyzes the entire design and estimates power consumption. For the Virtex 6, the average consumption is estimated at $6.5\ W$. Compared to the real measured value of $4.4\ W$, the estimated value seems to be a conservative estimation. The $4.4\ W$ are the difference between a full reset and full busy status of the GBP design. Although the real measured value is lower, the higher estimated power consumption value is used for comparison. For the Virtex Ultrascale+ the tools estimates an average power consumption of $18\ W$. Based on the ratio between real and estimated value for the ML605, the real values are expected to be slightly smaller than estimated by the tool for the Virtex Ultrascale+.

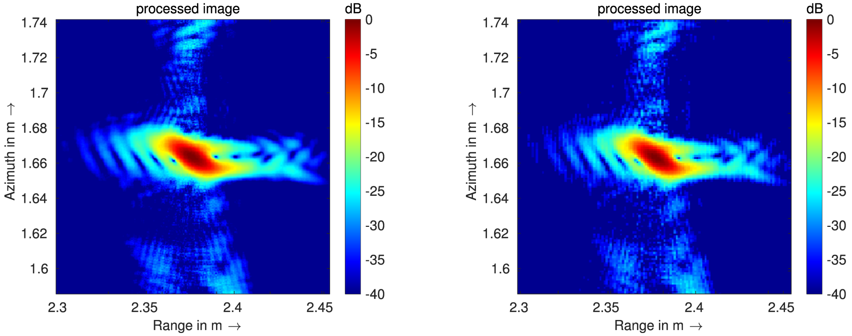
To compare the efficiency of the two FPGAs, runtime is put into relation to the average power consumption. The GBP runtime for 8 PEs on a Virtex 6 for a 4096×4096 pixel image with a dataset of 1408 pulses results to $29.7\ s$ (see Table 6.12). This value can be extrapolated to the Virtex Ultrascale+ as Fig. 6.1 shows that 64 PEs would fit on the FPGA. Runtime would reduce by factor 8 to $3.7\ s$. If assumed that performance scales linear with power consumption, a speedup factor of $18\ W/6.5\ W = 2\ 8$ would be expected. The factor between these two values represent the increase of power efficiency between the two FPGAs and results to $8/2.8 = 2.9$.

In order to put the runtime results in relation to other architectures, the FPGA based GBP runtime is used for comparison to the non dedicated and dedicated GBP architectures mentioned in Section 3.1 and Section 3.2. This requires to measure the GBP performance in PPP/s. For the same image size and dataset size as in Table 6.12, $4096 \times 4096 \times 1408 = 23.6G$ projections (PPP) are required. This results to $23.6G/29.7s = 795\frac{MPPP}{s}$ for the Virtex 6 and to $23.6G/3.7s = 6.4\frac{GPPP}{s}$ for the Virtex Ultrascale+.

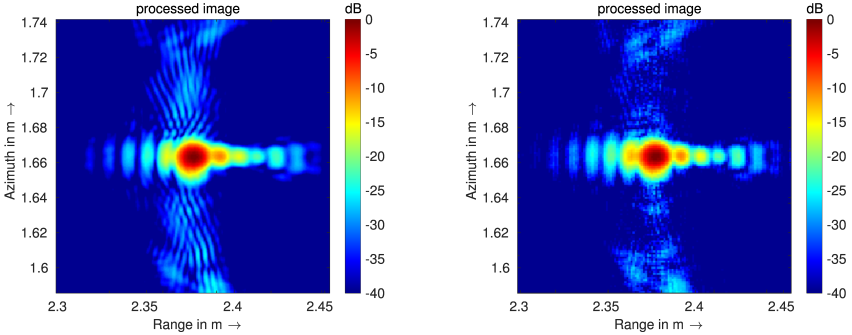
The mentioned non dedicated architectures reach values of $35\frac{GPPP}{s}$ at $605\ W$ [43], $23\frac{MPPP}{s}$ at $11\ W$ [59], $9\frac{MPPP}{s}$ at $10\ W$ [44], $6.2\frac{GPPP}{s}$ at $238\ W$ [45] and $543\frac{MPPP}{s}$ at $9\ W$ [60]. Those numbers point out that FPGAs are much more economical. The embedded Graphic Processing Unit (GPU) shows high potential though. And considering that hand-optimized FPGA designs take significantly longer to implement when compared to non dedicated implementations, FPGAs are not always preferable. The mentioned dedicated architectures reach values of $46\frac{MPPP}{s}$ at $10\ W$ [47], $2.1\frac{GPPP}{s}$ at $27\ W$ [49] and $2.9\frac{GPPP}{s}$ at $15\ W$ [48]. These results back up the high potential of FPGAs and show that regarding energy consumption, no hard constraint can be called for a mobile environment. But of course only architectures with the highest possible PPP/Ws ratio are attractive for implementation. Across all presented values, only FPGA implementations cross the border of 100 PPP/Ws or more. The results show, that time-domain processing is an additional option for processing and especially important for processing under motion of the platform. In combination with the FFBP any real-time constraint can be met, in case image quality is a variable parameter.



(a) Stop and go mode software GBP (b) Stop and go mode hardware GBP

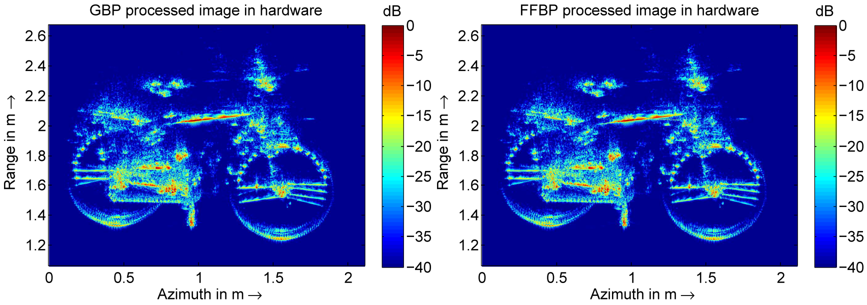


(c) 60 cm/s software GBP (d) 60 cm/s hardware GBP

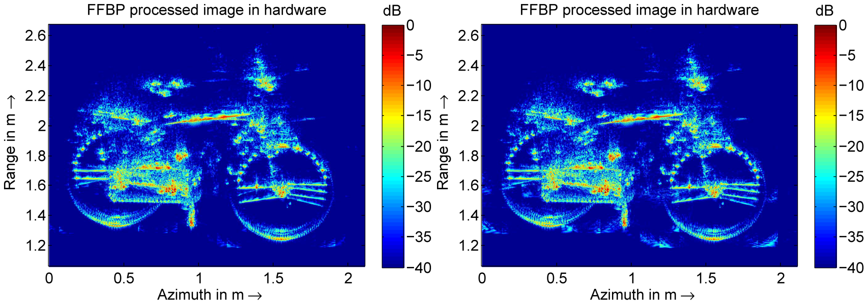


(e) 60 cm/s software corrected GBP (f) 60 cm/s hardware corrected GBP

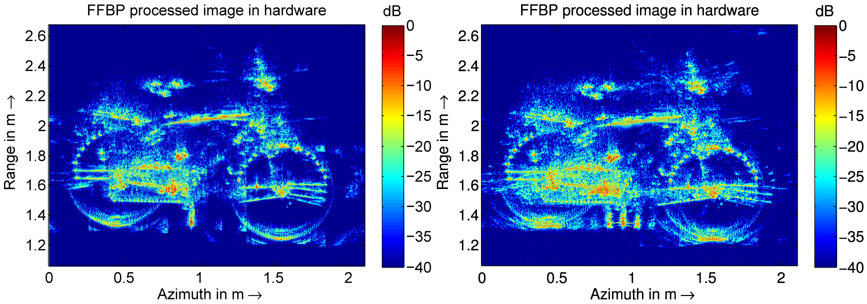
Figure 6.6: FMCW dataset with single corner reflector, recorded with and without motion during scanning, processed in hardware and software for comparison with the corrected GBP algorithm [54]



(a) GBP hardware reference (b) FFBP P1 configuration



(c) FFBP P2 configuration (d) FFBP P3 configuration



(e) FFBP P4 configuration (f) FFBP P5 configuration

Figure 6.7: FMCW dataset processed in hardware with GBP as reference and different FFBP configurations according to Pareto corner points [58]

7 Summary

Synthetic Aperture Radar (SAR) is a constantly developing technology, which is capable of generating high resolution aerial imagery. SAR systems emerged to a standard technology for surface cartography, surveillance, disaster control, height map generation and many more. Since SAR sensors are active, they do not depend on daylight and can penetrate certain materials basing on the chosen wavelength, e.g. clouds. Signal processing is required to extract an image from indirect observations of an area. To direct the platform along a region of interest, the images should be available in real-time. This can be challenging, as SAR sensor systems acquire data with a very high rate, while the signal processing is also complex. The processing can be performed either in frequency- or time-domain. It is common to perform processing in the frequency-domain, as the complexity of $\mathcal{O}(N^2 logN)$ is better suited for real-time processing than algorithms in the time-domain, which usually reside in the complexity class of $\mathcal{O}(N^3)$. Since SAR systems are getting smaller while allowing for higher resolution, they are also used in small Unmanned Aerial Vehicles (UAVs). In contrast to bigger platforms, UAVs are more likely to be affected by atmospheric conditions due to a lower speed and weight. While frequency-domain algorithms are less complex, they also rely on a straight trajectory, otherwise, reduced image quality is the result. This requirement might be compromised for UAVs. Different motion compensation algorithms are capable of correcting deviations but only in small ranges. Time-domain-based algorithms allow for arbitrary trajectories and deviation in motion. Since UAVs are limited in dimension and energy consumption, the processing system can not be scaled to match the higher processing demand of such algorithms. Due to the noise characteristic, SAR data cannot be compressed to send data to ground stations for processing. Therefore, energy-efficient on board real-time processing is required in the time-domain, in case of heavy deviations from a linear trajectory or in case of non-linear trajectories.

This work implements a time-domain-based, energy-efficient SAR processing system. A study on architectures in Chapter 3 based on Digital Signal Processors (DSPs), Graphic Processing Units (GPUs) and multi-/manycore architectures reveals, that such platforms (beside one exception) either lack in energy efficiency or processing power. To tackle the limitations in system dimensions, energy budget limitation, and processing complexity, the developed system is based on a Field Programmable Gate Array (FPGA) architecture. In contrast to other architectures, FPGAs can be tailored to exactly match the required task, wherefore no energy for not utilized resources is wasted.

The time-domain-based Global Backprojection (GBP) algorithm is implemented to ensure maximum image quality at arbitrary trajectories. The principle of the chosen algorithms is explained in Chapter 2. To allow for real-time processing in time-domain, also for bigger image dimensions, the Fast Factorized Backprojection (FFBP) is implemented. The FFBP algorithm is an accelerated version of the GBP which allows for a trade-off between image quality and processing time and resides in the complexity class of $\mathcal{O}(N^2 logN)$. The latest development in SAR systems, is Frequency Modulated Continuous Wave (FMCW) Radio Detection And Ranging (Radar). In contrast to pulsed Radar sensors, FMCW sensors allow for high resolution images at comparatively low transmit power. Due to the limited transmit power, the sensors are limited in range. For UAVs this is not a critical limitation as they operate at low altitudes. But the FMCW principle violates the so-called start-stop-approximation of standard processing algorithms. To compensate for this violation, the FPGA architecture is adapted to fully compensate any flaws introduced by the nature of FMCW data.

A premise for efficient dedicated hardware design is a proper analysis of the algorithms and their inherent limitations. This is important, as a maximum in performance requires a constant data throughput so that no resource is left unused. To implement a streaming architecture, Chapter 4 covers a full analysis of both algorithms. This includes the listing of the mathematical steps to understand which steps are often used and might offer the chance for an approximation to speed up processing. The second step covers the analysis of data-, flow- and control-dependencies to maximize instruction parallelism in the architecture. The third step is the analysis of streaming capabilities to prevent stalls and speedup the entire system. This includes memory access and caching strategies, the optimization for the design of multiple parallel Processing Elements (PEs) and the optimization of the pipeline for all PEs to shorten the critical path and reach a maximum in clock frequency.

The theoretical considerations are mapped into detailed descriptions of the specific PEs in Chapter 5. To ensure for a fast design, the core operations are substituted (if possible) with resource-saving and streaming friendly approximations. The methods for approximation are considered and analyzed in terms of fitness for the given task. All building blocks for the GBP, FFBP and FMCW designs are described in detail including the control structure around the system. Considerations are driven by the requirement to combine and fulfill all given constraints at the best possible rate while reducing processing time to a minimum by implementing a highly efficient architecture that is utilizing the maximum of available resource.

Since designing and implementing FPGA architectures is time costly, a framework is used to port the designs easily between different FPGAs platforms. This is possible by using system abstraction layers. The platform-independent design is encapsulated in a subsystem layer, platform-specific modules are located in the interchangeable main system layer. Like this, modifications in the subsystem will automatically affect all platforms. To ensures that the design can make full use of future technology and to keep up with the ongoing development of SAR sensors, all designs are scalable by instancing more PEs. Like this, a design can utilize the maximum of resources on each FPGA.

As a proof of concept the design was mapped onto a Xilinx Virtex 6 FPGA and tested on a Commercial Of The Shelf (COTS) ML605 development board to get results for resource utilization, the runtime for different algorithm configurations, power consumption values and image results on approximated fixed-point hardware. In Chapter 6 these results are presented and discussed. To ensure that the image results can match with implementations without approximations or precision reduction, everything is compared to software results as a golden reference.

It can be concluded, that the current increase in resources on FPGAs allows for real-time processing of the GBP for mid-size images. The results on the image quality of the hardware implementation show no significant reduction in quality. The real-time constraint depends of course on the SAR system and platform specifications. In case higher resolutions are required, the FFBP can be used to trade-off runtime against image quality. System dimensions and power consumption allow for a setup in a mobile UAV scenario, while the FPGA design can always be ported to a more potent platform and can be scaled to max out the available resources to gain the maximum possible speedup.

Bibliography

[1] B. A. Campell, *Radar Remote Sensing of Planetray Surfaces*. The Press Syndicate ot the University of Cambridge, 2002.

[2] J.-R. Kim, J. Sumantyo, and S.-Y. Lin, "Preliminary study for the long wavelength planetary SAR sensor design and applications," in *EGU General Assembly Conference Abstracts*, ser. EGU General Assembly Conference Abstracts, vol. 17, Apr. 2015, p. 8446.

[3] D. G. Blumberg, "High Resolution X- Band SAR Imagery for Precise Agriculture and Crop Monitoring," in *Science and Applications of SAR Polarimetry and Polarimetric Interferometry*, ser. ESA Special Publication, vol. 644, Mar. 2007, p. 49.

[4] S. Plank, A. Twele, and S. Martinis, "Landslide mapping in vegetated areas using change detection based on optical and polarimetric sar data," in *Earth Observations for Geohazards*, 2016.

[5] D. Cerutti-Maori, J. Klare, A. Brenner, and J. H. G. Ender, "Wide-area traffic monitoring with the sar/gmti system pamir," *IEEE Trans. Geosci. Remote Sens.*, vol. 46, no. 10, pp. 3019–3030, Oct. 2008.

[6] J. K. Tennant and T. Coyne, "Star-3i interferometric synthetic aperture radar (insar): Some lessons learned on the road to commercialzation," in *4th International Airborne Remote Sensing Conference and Exhibition*, Intermap Technologies, Calgary, Alberta, Canada, 1999.

[7] C. Huelsmeyer, "Verfahren, um entfernte metallische gegenstaende mittels elektrischer wellen einem beobachter zu melden," Nov. 21 1905, dE Patent 165,546. [Online]. Available: https://www.google.com/patents/DE165546C?cl=de

[8] S. Jin, N. Haghighipour, and W. Ip, *Planetary Exploration and Science: Recent Results and Advances*, ser. Springer Geophysics. Springer Berlin Heidelberg, 2016. [Online]. Available: https://books.google.de/books?id=xphgvgAACAAJ

[9] H. Klausing, *Radar mit realer und synthetischer Apertur: Konzeption und Realisierung*. Oldenbourg Wissensch.Vlg, 2000. [Online]. Available: http://books.google.de/books?id=iLt4AAAACAAJ

[10] D. Gabor, Ed., *A New Microscopic Principle*, Research Laboratory, British Thomson-Houston Co., Ltd., Rugby, 1949.

[11] M. Soumekh, *Synthetic Aperture Radar Signal Processing: with MATLAB Algorithms*. John Wiley & Sons, 1999.

[12] S. Johnston, *Holographic Visions: A History of New Science*, ser. Spencer, H.; Herbert Spencer lectures. OUP Oxford, 2006. [Online]. Available: https://books.google.al/books?id=3hQRP6kj6vIC

[13] C. A. Wiley, Ed., *Synthetic Aperture Radars - A Paradig for Technology Evolution*, 1985.

[14] W. A, "Pulsed doppler radar methods and apparatus," Jul. 20 1965, uS Patent 3,196,436. [Online]. Available: https://www.google.com/patents/US3196436

[15] L. J. C. W. E. Vivian and E. N. Leith, "Report of project michigan: A doppler technique for obtaining very fine angular resolution from a side-looking airborne radar," University of Michigan Willow Run Laboratory, Tech. Rep., 1964.

[16] W. M. Brown, G. G. Houser, and R. E. Jenkins, "Synthetic aperture processing with limited storage and presumming," *IEEE Transactions on Aerospace and Electronic Systems*, vol. AES-9, no. 2, pp. 166–176, March 1973.

[17] J. C. Kirk, "A discussion of digital processing in synthetic aperture radar," *IEEE Transactions on Aerospace and Electronic Systems*, vol. AES-11, no. 3, pp. 326–337, May 1975.

[18] W. J. van de Lindt, "Digital technique for generating synthetic aperture radar images," *IBM Journal of Research and Development*, vol. 21, no. 5, pp. 415–432, Sept 1977.

[19] C. Wu, Ed., *A digital system to produce imagery from SAR data*, Oct. 1976.

[20] I. G. Cumming and F. H. Wong, *Digital Processing of Synthetic Aperture Radar Data: Algorithms and Implementation*. Artech House Inc., 2005.

[21] J. R. Bennett and I. G. Cumming, Eds., *A Digital Processor for the Production of Seasat Synthetic Aperture Radar Imagery*, Symposium on Machine Processing of Remotely Sensed Data. The Laboratory for Applications of Remote Sensing, 1978.

[22] *Digital processing of SEASAT SAR data*, vol. 4, Washington, DC, Apr. 1979.

[23] F. Rocca, "Synthetic Aperture Radar: a New Application for Wave Equation Techniques," *Stanford Exploration Project SEP-56*, pp. 167–189, 1987.

[24] I. Cumming, F. Wong, and K. Raney, "A sar processing algorithm with no interpolation," in *Geoscience and Remote Sensing Symposium, 1992. IGARSS '92. International*, vol. 1, May 1992, pp. 376–379.

[25] J. Radon, "Über die bestimmung von funktionen durch ihre integralwerte längs gewisser mannigfaltigkeiten.[on the determination of functions from their integrals along certain manifolds]," *SBLeipzig*, vol. 29, p. 69, 1917.

[26] R. N. Bracewell, "Strip integration in radio astronomy," in *Australian Journal of Physics*, vol. 9, 1956, p. 198.

[27] W. H. Oldendorf, "Isolated flying spot detection of radiodensity dis-continuities-displaying the internal structural pattern of a complex object," *IRE Transactions on Bio-Medical Electronics*, vol. 8, no. 1, pp. 68–72, Jan 1961.

[28] D. E. Kuhl and R. Q. Edwards, "Image separation radioisotope scanning," *Radiology*, vol. 80, no. 4, pp. 653–662, 1963. [Online]. Available: http://dx.doi.org/10.1148/80.4.653

[29] A. M. Cormack, "Representation of a Function by Its Line Integrals, with Some Radiological Applications," *Journal of Applied Physics*, vol. 34, pp. 2722–2727, Sep. 1963.

[30] R. N. Bracewell and A. C. Riddle, "Inversion of fan-beam scans in radio astronomy," *Astronomical Journal*, vol. 150, no. 2, pp. 427–434+, 1967.

[31] A. Lakshminarayanan, "Reconstruction from divergent ray data," Department for Computer Sience Technology at State University of New York at Buffalo, Technical Report TR-92, 1975.

[32] T. F. Budinger and G. G.T., "Image processing for 2-d and 3-d reconstructions from projproject: Theory and practices in medicine and the physical sciences," Stanford University, Tech. Rep., 1975.

[33] D. C. Munson, J. D. O'Brien, and W. K. Jenkins, "A tomographic formulation of spotlight-mode synthetic aperture radar," *Proceedings of the IEEE*, vol. 71, no. 8, pp. 917–925, Aug 1983.

[34] J. A. Fawcett, "Inversion of n-dimensional spherical averages," *SIAM Journal on Applied Mathematics*, vol. 45, no. 2, pp. 336–341, 1985. [Online]. Available: https://doi.org/10.1137/0145018

[35] H. Hellsten and L. E. Andersson, "An inverse method for the processing of synthetic aperture radar data," *Inverse Problems*, vol. 3, no. 1, p. 111, 1987. [Online]. Available: http://stacks.iop.org/0266-5611/3/i=1/a=013

[36] L.-E. Andersson, "On the determination of a function from spherical averages," in *SIAM Journal on Mathematical Analysis. Vol 19*, 1988.

[37] A. F. Yegulalp, "Fast backprojection algorithm for synthetic aperture radar," in *In Proceedings 1999 IEEE Radar Conference*, 1999.

[38] O. Seger, M. Herberthson, and H. Hellsten, "Real time sar processing of low frequency ultra wide band radar data," in *Proc. of EUSAR '98 - European Conference on Synthetic Aperture Radar*, May 1998, pp. 489–492.

[39] J. McCorkle and M. Rofheart, "An order n2 log(n) backprojector algorithm for focusing wide-angle wide-bandwidth arbitrary-motion synthetic aperture radar," in *In SPIE AeroSense Conference,*, 1996.

[40] S. Nilsson and L. E. Andersson, "Application of fast back-projection techniques for some inverse problems of synthetic aperture radar," pp. 62–72, 1998. [Online]. Available: http://dx.doi.org/10.1117/12.321872

[41] S. Basu and Y. Bresler, "O(n2log2n) filtered backprojection reconstruction algorithm for tomography," *Image Processing, IEEE Transactions on*, vol. 9, no. 10, pp. 1760 –1773, oct 2000.

[42] L. M. H. Ulander, H. Hellsten, and G. Stenstroem, "Synthetic-aperture radar processing using fast factorised backprojection," in *EUSAR*, 2000.

[43] J. Park, P. T. P. Tang, M. Smelyanskiy, D. Kim, and T. Benson, "Efficient backprojection-based synthetic aperture radar computation with many-core processors," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 28:1–28:11. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389034

[44] J. Zhao, Y. Yuan, J. Wang, and Y. Jin, "Research on bp imaging algorithm parallelization using c6678 dsps," in *IET International Radar Conference 2015*, Oct 2015, pp. 1–4.

[45] W. Chapman, S. Ranka, S. Sahni, M. Schmalz, U. Majumder, L. Moore, and B. Elton, "Parallel processing techniques for the processing of synthetic aperture radar data on gpus," in *2011 IEEE International Symposium on Signal Processing and Information Technology (ISSPIT)*, Dec 2011, pp. 573–580.

[46] P. Schleuniger and S. Karlsson, *Tinuso: A processor architecture for a multi-core hardware simulation platform*, 2010, vol. 3.

[47] P. Schleuniger, A. Kusk, J. Dall, and S. Karlsson, "Synthetic aperture radar data processing on an fpga multi-core system," in *International conference on Architecture of Computing Systems ARCS*, international conference on Architecture of Computing Systems.

[48] W. Chapman, S. Ranka, S. Sahni, M. Schmalz, and U. K. Majumder, "Parallel processing techniques for the processing of synthetic aperture radar data on fpgas," in *The 10th IEEE International Symposium on Signal Processing and Information Technology*, Dec 2010, pp. 17–22.

[49] D. Pritsker, "Efficient global back-projection on an fpga," in *2015 IEEE Radar Conference (RadarCon)*, May 2015, pp. 0204–0209.

[50] T. Avery and G. Berlin, *Fundamentals of Remote Sensing and Airphoto Interpretation*, ser. Prentice Hall series in geographic information science. Macmillan, 1992. [Online]. Available: https://books.google.de/books?id=5WYZAQAAIAAJ

[51] R. Bracewell, *The Two-Dimensional Convolution Theorem*. Boston, MA: Springer US, 2003, pp. 204–221. [Online]. Available: https://doi.org/10.1007/978-1-4419-8963-5_6

[52] L. A. Gorham and L. J. Moore, "Sar image formation toolbox for matlab," in *Proc. of SPIE*, vol. 7699, no. 1. SPIE, 2010.

[53] F. Cholewa, M. Pfitzner, C. Fahnemann, P. Pirsch, and H. Blume, "Synthetic aperture radar with backprojection: A scalable, platform independent architecture for exhaustive fpga resource utilization," in *2014 International Radar Conference*, Oct 2014, pp. 1–5.

[54] F. Cholewa, M. Wielage, P. Pirsch, and H. Blume, "An fpga architecture for velocity independent backprojection in fmcw-based sar systems," 2016.

[55] R. L. Freeman, *Telecommunication System Engineering*, 3rd ed. New York, NY, USA: John Wiley & Sons, Inc., 1996.

[56] A. Ribalta, "Time-domain reconstruction algorithms for fmcw-sar," in *IEEE Geosience and Remote Sesning Letters*, vol. 8, 2011.

[57] M. Wielage, F. Cholewa, P. Pirsch, and H. Blume, "Experimental violation of the start-stop-approximation using a holistic rail-based uwb fmcw-sar system," in *EUSAR 2016; 11th European Conference on Synthetic Aperture Radar*, 2016.

[58] F. Cholewa, M. Wielage, P. Pirsch, and H. Blume, "Synthetic aperture radar with fast factorized backprojection: A scalable, platform independent architecture for exhaustive fpga resource utilization," in *2017 International Radar Conference*, 2017.

[59] M. Wielage, F. Cholewa, P. Pirsch, and H. Blume, "Parallelization strategies for fast factorized backprojection sar on embedded multi-core architectures," in *International Conference on Microwaves, Communications, Antennas and Electronic Systems*, 2017.

[60] 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)*, Nov 2017, pp. 351–357.

[61] Intel. Specification of intel xeon e5-2670 processor. [Online]. Available: https://ark.intel.com/products/64595/Intel-Xeon-Processor-E5-2670-20M-Cache-2_60-GHz-8_00-GTs-Intel-QPI

[62] ——. Specification of intel xeon phi 5120d coprozessor. [Online]. Available: https://ark.intel.com/de/products/75801/Intel-Xeon-Phi-Coprocessor-5120D-8GB-1_053-GHz-60-core

[63] Hardkernel. Specification of odroid-xu4 multicore platform. [Online]. Available: http://www.hardkernel.com/main/products/prdt_info.php

[64] Samsung. Specification of samsung exynos 5 octa 5422 processor. [Online]. Available: http://www.samsung.com/semiconductor/minisite/exynos/products/mobileprocessor/exynos-5-octa-5422/

[65] Arm. Specification of arm cortex-a15 processor. [Online]. Available: https://developer.arm.com/products/processors/cortex-a/cortex-a15

[66] ——. Specification of arm cortex-a7 processor. [Online]. Available: https://developer.arm.com/products/processors/cortex-a/cortex-a7

[67] T. Instruments. Specification of tms320c6678 dsp. [Online]. Available: http://www.ti.com/product/TMS320C6678

[68] ——. Specification of tms320c66x dsp. [Online]. Available: http://www.ti.com/lit/an/sprt580a/sprt580a.pdf

[69] nVidia. Specification of nvidia jetson tx2. [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems-dev-kits-modules/

[70] Arm. Specification of arm cortex-a57 processor. [Online]. Available: https://developer.arm.com/products/processors/cortex-a/cortex-a57

[71] A. Hast and L. Johansson, "Fast factorized back-projection in an fpga," Master's thesis, 2006.

[72] J. V. Oldfield and R. C. Dorf, *Field-programmable Gate Arrays: Reconfigurable Logic for Rapid Prototyping and Implementation of Digital Systems*. New York, NY, USA: Wiley-Interscience, 1994.

[73] M. Pfitzner, F. Cholewa, P. Pirsch, and H. Blume, "A flexible hardware architecture for real-time airborne wavenumber domain sar processing," *EUSAR, 9th European Conference on Synthetic Aperture Radar*, pp. 28 –31, april 2012.

[74] D. A. Patterson and J. L. Hennessy, *Computer Architecture: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1990.

[75] A. J. Bernstein, "Analysis of programs for parallel processing," *IEEE Transactions on Electronic Computers*, vol. EC-15, no. 5, pp. 757–763, Oct 1966.

[76] M. Leeser, S. Coric, E. Miller, H. Yu, and M. Trepanier, "Parallel-beam backprojection: An fpga implementation optimized for medical imaging," *The Journal of VLSI Signal Processing*, vol. 39, 2005.

[77] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Trans. Comput.*, vol. 21, no. 9, pp. 948–960, Sep. 1972. [Online]. Available: http://dx.doi.org/10.1109/TC.1972.5009071

[78] L. Papula, *Mathematik für Ingenieure und Naturwissenschaftler Band 1: Ein Lehr- und Arbeitsbuch für das Grundstudium*, ser. Viewegs Fachbücher der Technik. Vieweg+Teubner Verlag, 2011. [Online]. Available: http://books.google.de/books?id=vTuCSeEal6AC

[79] D. Fowler and E. Robson, "Square root approximations in old babylonian mathematics: Ybc 7289 in context," *Historia Mathematica*, vol. 25, no. 4, pp. 366 – 378, 1998. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0315086098922091

[80] H. S. White, "Mathematics in india by kim plofker," *The Mathematical Intelligencer*, vol. 32, no. 2, pp. 68–70, Jun 2010. [Online]. Available: https://doi.org/10.1007/s00283-009-9115-1

[81] B. Tirtha and V. Agrawala, *Vedic Mathematics: Or, Sixteen Simple Mathematical Formulae from the Vedas (for One-line Answers to All Mathematical Problems)*, ser. Hindu Vishvavidyalaya Nepal Rajya Sanskrit series. Motilal Banarsidass, 1970. [Online]. Available: https://books.google.de/books?id=iE0gAQAAIAAJ

[82] R. E. Goldschmidt, "Applications of division by convergence," 1964.

[83] *Taylor expansions and applications*. Milano: Springer Milan, 2008, pp. 223–255. [Online]. Available: https://doi.org/10.1007/978-88-470-0876-2_7

[84] J. Mason and D. Handscomb, *Chebyshev Polynomials*. CRC Press, 2002. [Online]. Available: https://books.google.de/books?id=8FHf0P3to0UC

[85] G. Baker, G. Baker, G. Baker, P. Graves-Morris, S. Baker, C. U. Press, and G. Rota, *Pade Approximants: Encyclopedia of Mathematics and It's Applications, Vol. 59 George A. Baker, Jr., Peter Graves-Morris*, ser. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 1996. [Online]. Available: https://books.google.de/books?id=Kf2e2uzBZhoC

[86] J. E. Volder, "The cordic trigonometric computing technique," *Electronic Computers, IRE Transactions on*, vol. EC-8, 1959.

[87] J. S. Walther, "A unified algorithm for elementary functions," in *Proceedings of the May 18-20, 1971, Spring Joint Computer Conference*, ser. AFIPS '71 (Spring). New York, NY, USA: ACM, 1971, pp. 379–385. [Online]. Available: http://doi.acm.org/10.1145/1478786.1478840

[88] P. Meher, J. Valls, T.-B. Juang, K. Sridharan, and K. Maharatna, "50 years of cordic: Algorithms, architectures, and applications," *Circuits and Systems I: Regular Papers, IEEE Transactions on*, vol. 56, 2009.

[89] M. Pfitzner, F. Cholewa, P. Pirsch, and H. Blume, "Close-to-hardware error analysis for real-time wavenumber domain processsing," in *2012 International Radar Conference*, 2012.

[90] M. Pfitzner, "Fpga-basierte hardware-architektur fuer die echtzeit-sar-bilddatengenerierung mit integrierter motion compensation informationstechnik," Ph.D. dissertation, 2015.

[91] R. G. Lyons, *Understanding Digital Signal Processing*, 1st ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1996.

[92] N. Pohl, T. Jaeschke, and M. Vogt, "Ultra high resolution sar imaging using an 80 ghz fmcw-radar with 25 ghz bandwidth," *EUSAR, 9th European Conference on Synthetic Aperture Radar*, 2012.

[93] D. T. Luc, *Pareto Optimality*. New York, NY: Springer New York, 2008, pp. 481–515. [Online]. Available: https://doi.org/10.1007/978-0-387-77247-9_18

[94] M. Kock, S. Hesselbarth, and H. Blume, "Hardware accelerated design space exploration framework for communication systems," in *SDR-WInnComm*, 2013.

[95] Xilinx. Specification of ml605 emulation platform. [Online]. Available: http://www.xilinx.com/ml605

[96] [Online]. Available: http://beecube.com/

[97] Xilinx. Specification of vc707 emulation platform. [Online]. Available: http://www.xilinx.com/vc707

[98] M. Pfitzner, S. Langemeyer, P. Pirsch, and H. Blume, "A flexible real-time sar processing platform for high resolution airborne image generation," in *2011 International Radar Conference*, vol. 1, oct. 2011, pp. 26 –29.

[99] X. Corp., *Virtex-6 FPGA Data Sheet*, Xilinx Corp., 2014. [Online]. Available: https://www.xilinx.com/support/documentation/data_sheets/ds152.pdf

[100] S. V. Tsynko, "On the use of start-stop approximation for spaceborne sar imaging," in *SIAM Journal on Imaging Sciences*, vol. 2, no. 2, 2009.

[101] A. Ribalta, "Omega-k algorithm without the stop-and-go approximation for high resolution sar image reconstruction," in *Geoscience and Remote Sensing Symposium (IGARSS)*, 2012.

Wissenschaftlicher Werdegang

10/2002-09/2007	**Studium der Informatik** Universität Lüneburg (Diplom)
10/2007-09/2010	**Studium der Informatik** Universität Hannover (Master)
04/2011-06/2018	**Wissenschaftlicher Mitarbeiter** Institut für Mikroelektronische Systeme Fachgebiet Architekturen und Systeme an der Leibniz Universität Hannover