Diplomarbeit

Sparse Bayesian Learning for Directions of Arrival on an FPGA

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines Diplom-Ingenieurs

von

Herbert Groll, BSc.
Matrikelnummer: 0725518

am
Institute of Telecommunications
Fakultät für Elektrotechnik und Informationstechnik
Technische Universität Wien

unter der Leitung von
Univ.Prof. Ing. Dipl.-Ing. Dr.-Ing. Christoph Mecklenbräuker

Wien, November 2017

Acknowledgements

My deepest gratitude goes to Peter Thorwartl from so-logic electronic consulting for guiding me through the FPGA world, his support during my studies, and providing the necessary tools and hardware to enable this research.

*

To Nina, my family, friends, and colleagues. Without you I would not have survived my studies.

*

Special thanks to Erich Zöchmann for fruitful discussions on the topic, and to Gogo for comments.

~

Abstract

A directions of arrival (DOA) estimator based on sparse Bayesian learning (SBL) is implemented as a fixed-point prototype for an FPGA platform. The prototype is developed mainly using high-level synthesis (HLS) of C++ based model specifications. Prototyping possibilities are explored for incremental verification with well known computing environments. For modeling, the equations of the algorithm are reduced to arithmetic operations considering the signal flow within the iterative structure. The relevant aspects of the used HLS tool, Vivado HLS, concerning fixed-point data types, methods for creating hierarchical designs, and specific modeling techniques are discussed. The prototype is presented in detail. Scheduling of each module is done as soon as possible to make use of the parallel FPGA architecture. Different fixed-point word length assumptions are explained and implementation results are shown in terms of resources, latency, and power consumption estimates.

Finally, a representative DOA source example is simulated and tested with the implemented prototype hardware in the loop. The comparison with a floating-point reference implementation is found to have good agreement with the fixed-point implementation.

Kurzfassung

Ein Schätzer für Richtungserkennung basierend auf sparse Bayesian learning (SBL) wird prototypisch in Festpunktarithmetik auf einer FPGA-Plattform implementiert. Dabei wird gleichzeitig die Methodik der Hardware-Synthese durch Hochsprachen (HLS) angewendet um mittels C++ Programmiersprache eine Spezifikation für ein Modell der digitalen integrierten Schaltung zu erstellen. Diese Abstraktion erlaubt es einerseits, nicht relevanten Implementierungsaufwand zu automatisieren und andererseits, Entwürfe eines Prototyps schnell mittels bekannter computerunterstützte Entwicklungsumgebungen der Numerik zu verifizieren. Die Modellierung des Schätzers erfolgt durch ein Vereinfachen und Umsetzen der Gleichungen des Algorithmus in arithmetische Operationen. Diese werden mit bedacht auf den Signalfluss der digitalen integrierten Schaltung und des iterativen Charakters des Algorithmus entwickelt. Die dabei benötigten Aspekte werden für das verwendete HLS Softwarepaket, Vivado HLS, hervorgehoben. Darunter fallen die Festpunktdatentypen, die Anbindung mehrerer aufgeteilter Module und spezielle Modellierungsarten. Die prototypische Implementierung wird diskutiert. Ein Steuermodul für den Ablauf stellt die nebenläufige Berechnung durch die Module sicher, um von der parallelen FPGA-Architektur zu profitieren. Verschiedene Annahmen bezüglich der benutzten Wortlängen werden untersucht. Danach wird der Ressourcenverbrauch, die Latenz und der erwartete Leistungsverbrauch ermittelt.

Abschließend wird ein representatives Szenario für die Richtungsschätzung simuliert. Der Prototyp des in Festpunktarithmetik implementierten Schätzers am FPGA wird mit einer Referenzimplementierung in Gleitpunktarithmetik verglichen. Die vorgestellte Implementierung erreicht hierbei eine gute Übereinstimmung.

Contents

List of Figures

List of Tables

1. Introduction

1.1. Problem Formulation

A method for estimating the directions of arrival (DOA) of waves is presented in [1].

The signal model assumes a K-sparse signal $\boldsymbol{x}_l \in \mathbb{C}^M$ which is a vector of M dimensions with K nonzero complex source amplitudes x_{lm}, with $m \in [1, \dots M]$. M is the number of points of a grid which defines the DOA of the source amplitudes of K traveling waves where $K \ll M$. Each wave is modeled with narrowband far-field assumption at a fixed frequency ω and velocity of propagation c. Multiple single snapshots of $\boldsymbol{x}_l$ are combined to a matrix $\boldsymbol{X} = [\boldsymbol{x}_1, \dots, \boldsymbol{x}_L] \in \mathbb{C}^{M \times L}$ with L being the number of snapshots. The indices of the K nonzero sources of $\boldsymbol{x}_l$ define a set of active sources, the support of $\boldsymbol{x}_l$ or active set $\mathcal{M}_l = \{m \in \mathbb{N} | x_{ml} \neq 0\} = \{m_1, m_2, \dots, m_K\}$. The model requires the sources to be stationary so that the active set $\mathcal{M}_l = \mathcal{M}$ is constant across all L snapshots. Sources $\boldsymbol{x}_l$ can not be observed directly. Instead a sensor array consisting of N sensors observe a transformation of the complex source amplitudes. A single snapshot of the array data is $\boldsymbol{y}_l \in \mathbb{C}^N$, the multisnapshot array data is then $\boldsymbol{Y} = [\boldsymbol{y}_1, \dots, \boldsymbol{y}_L] \in \mathbb{C}^{N \times L}$. The array data model $\boldsymbol{y}_l = \boldsymbol{A}\boldsymbol{x}_l + \boldsymbol{n}_l$ is a linear transformation of $\boldsymbol{x}_l$ with the transfer matrix $\boldsymbol{A}$ and additive noise $\boldsymbol{n}_l$ due to the sensors. For the multisnapshot array data the model is [1, eq. (1)]

$$\boldsymbol{Y} = \boldsymbol{A}\boldsymbol{X} + \boldsymbol{N} \tag{1.1}$$

The linear transformation with transfer matrix $\boldsymbol{A}$ describes the impinging of multiple wave fronts, each from a different direction θ_m, on the sensor array. Each direction has its column $\boldsymbol{a}_m$ in $\boldsymbol{A} = [\boldsymbol{a}_1, \dots, \boldsymbol{a}_M] \in \mathbb{C}^{N \times M}$ which describes the time delay $\boldsymbol{\tau}_m = [\tau_{m1}, \dots, \tau_{mN}]$ for a wave front of direction θ_m with respect to the sensors. For a uniform linear array (ULA), the element spacing d of sensors is equal. Therefore, $A_{nm} = e^{-j\omega\tau_{nm}} = e^{-j(n-1)\frac{\omega d}{c}\sin\theta_m}$. Each element N_{nl} of the multisnapshot additive noise $\boldsymbol{N} = [\boldsymbol{n}_1, \dots, \boldsymbol{n}_L] \in \mathbb{C}^{N \times L}$ is assumed to be idenpendent and identically distributed (i.i.d.) complex Gaussian, $\mathcal{CN}(0, \sigma^2)$, across sensors and snapshots. The number of sensors N is assumed to be much smaller than the number of DOA positions M. Therefore the transfer matrix $\boldsymbol{A}$ is a non-invertible operator and (1.1) is a underdetermined system of linear equations. Solving the linear problem (1.1) for $N \ll M$ via approximation e.g. with a *Wiener Filter* would result in a high mean squared error (MSE) because it does not consider the sparse source. In [1] the sparse Bayesian learning (SBL) framework of [2] is used to solve the linear problem with multiple measurement vectors.

1.1.1. Bayesian Formulation

The problem of solving the underdetermined system is tackled using Bayesian inference. This involves determining the posterior distribution of the complex source amplitudes $\boldsymbol{X}$ from the likelihood and prior model [1].

A Gaussian likelihood function of $\boldsymbol{Y}|\boldsymbol{X}$ is defined [1, eq. (3)] under the assumption of the additive complex Gaussian noise in (1.1).

$$p(\boldsymbol{Y}|\boldsymbol{X};\sigma^2) = \frac{\exp(-\frac{1}{\sigma^2}\|\boldsymbol{Y}-\boldsymbol{A}\boldsymbol{X}\|_{\mathcal{F}}^2)}{(\pi\sigma^2)^{NL}} \tag{1.2}$$

with variable variance σ^2.

The unknown complex source amplitudes x_{ml} are assumed to be independent both across DOA and snapshots l. Therefore, the prior distribution of x_{ml} is modeled to follow a zero-mean complex Gaussian distribution $\mathcal{CN}(0,\gamma_m)$ with variance γ_m as a prior hyperparameter, which is varying with directional index m but stationary across different snapshots l.

$$p_m(x_{ml};\gamma_m) = \begin{cases} \delta(x_{ml}), & \text{for } \gamma_m = 0 \\ \frac{1}{\pi\gamma_m}\mathsf{e}^{-|x_{ml}|^2/\gamma_m}, & \text{for } \gamma_m > 0 \end{cases} \tag{1.3}$$

Writing (1.3) as a multivariate prior distribution of $\boldsymbol{X}$ leads to [1, eq. (5)]

$$p(\boldsymbol{X};\boldsymbol{\gamma}) = \prod_{l=1}^{L}\mathcal{CN}(\mathbf{0},) \tag{1.4}$$

With $\boldsymbol{\gamma} = [\gamma_1,\ldots,\gamma_M]^T$ being the hyperparameters of the prior distribution and $= \mathsf{E}[\boldsymbol{x}_l\boldsymbol{x}_l^H;\boldsymbol{\gamma}]$ being the covariance matrix of the complex source amplitude vector $\boldsymbol{x}_l$. As the sources x_{ml} are assumed uncorrelated, $\boldsymbol{\Gamma} = \text{diag}(\boldsymbol{\gamma})$. Each hyperparameter $\gamma_m \geq 0$ represents source power at directional index m. From (1.3) follows for $\gamma_m = 0$, $P\{x_{ml} = 0\} = p_m(0;0) = 1$. For $\gamma_m > 0$, an active set $\mathcal{M} = \{m \in \mathbb{N}|\gamma_m > 0\}$ is defined equivalently to the active set of complex source amplitudes. The SBL algorithm in [1] estimates the source power of $\boldsymbol{X}$ by estimating the hyperparameters $\boldsymbol{\Gamma}$.

1.1.2. SBL Algorithm

The algorithm for estimating the DOA of complex Gaussian sources using SBL was presented in [1, Table 1].

```
1: while $\epsilon > \epsilon_{\min}$ and $j < j_{\max}$ do
2:     $j = j+1$, $\boldsymbol{\gamma}^{\text{old}} = \boldsymbol{\gamma}^{\text{new}}$, $\boldsymbol{\gamma} = \boldsymbol{\gamma}^{\text{new}}$
3:     $\boldsymbol{\Sigma}_y = \sigma^2\boldsymbol{I}_N + \boldsymbol{A}\boldsymbol{\Gamma}\boldsymbol{A}^H$
4:     $\gamma_m^{\text{new}}$ = choose [1, eq. (SBL1) or (SBL)]
5:     $\mathcal{M} = \{m \in \mathbb{N} \mid K \text{ largest peaks in} \boldsymbol{\gamma}^{\text{new}}\} = \{m_1,\ldots,m_K\}$
6:     $\boldsymbol{A}_{\mathcal{M}} = [\boldsymbol{a}_{m_1},\ldots,\boldsymbol{a}_{m_K}]$
7:     $(\sigma^2)^{\text{new}} = \frac{1}{N-K}\text{tr}((I_N - \boldsymbol{A}_M\boldsymbol{A}_M^+)\boldsymbol{S}_y)$
```

```
 8:     $\epsilon = \|\boldsymbol{\gamma}^{\text{new}} - \boldsymbol{\gamma}^{\text{old}}\|_1 / \|\boldsymbol{\gamma}^{\text{old}}\|_1$
 9: end while
10: Output: $\mathcal{M}$, $\boldsymbol{\gamma}^{\text{new}}$, $(\sigma^2)^{\text{new}}$
```

The algorithm tries to find a sparse solution for an observed $\boldsymbol{Y}$ by iteratively updating estimated variances γ_m and estimated noise σ^2. The two variants for updating γ_m are [1, eq. (SBL1)]

$$\gamma_m^{\text{new}} = \frac{\gamma_m^{\text{old}}}{\sqrt{L}} \left\| \boldsymbol{Y}^H \boldsymbol{\Sigma}_y^{-1} \boldsymbol{a}_m \right\|_2 / \sqrt{\boldsymbol{a}_m^H \boldsymbol{\Sigma}_y^{-1} \boldsymbol{a}_m}, \tag{SBL1}$$

and [1, eq. (SBL)]

$$\gamma_m^{\text{new}} = \frac{\gamma_m^{\text{old}}}{\sqrt{L}} \left\| \boldsymbol{Y}^H \boldsymbol{\Sigma}_y^{-1} \boldsymbol{a}_m \right\|_2 / \sqrt{\boldsymbol{a}_m^H \boldsymbol{S}_y^{-1} \boldsymbol{a}_m}. \tag{SBL}$$

How to implement the algorithm for an actual hardware realization, was left as an open task. A target platform of interest for such an application is a FPGA, especially because those devices start to be tightly integrated with analog-digital converters (ADCs) nowadays.

1.2. Motivation

The SBL for DOA estimator presented in [1] can be a useful component for real world applications which need localization of wave emitting sources with high spatial or temporal resolution. To our knowledge, the algorithm is only implemented as MATLAB code so far, which limits its use to the processing of offline data. Moreover, an application is tied to a workstation or laptop. An implementation of the estimator suitable for tight integration with measurement equipment could simplify complex measurement systems or even enable its use in new fields of application. Online measurements for the characterization of wireless channels, for example could benefit from the dimension-reduction the SBL DOA estimator has to offer. Important aspects of a prototype are usability, mobility, power consumption, costs, and suitability for real-time processing.

The use of an FPGA platform is an obvious choice for the prototype of a DOA estimator as it provides power efficient parallel processing, allows flexible interfacing with a wide range of communication protocols, and enables rapid prototyping by reconfiguration. For these reasons, FPGA devices are used in nearly all professional measurement equipment today.

The traditional way of FPGA programming is through a low-level hardware description language (HDL), which allows to model the hardware in high detail. The linear algebra of the SBL algorithm [1] is a rather high-level description which needs to be refined towards lower architectural levels. However, this can be a painful and error-prone task with enormous complexity. HLS tools for FPGA allow to specify a task on the algorithmic level and generate a derived low-level description. Moreover, parameters and different implementation variants can be evaluated easily which makes it promising for rapid prototyping.

Power efficiency and cost for the FPGA implementation should be kept moderate. Therefore, a realization in fixed-point arithmetic is preferred over floating-point arithmetic.

1.3. Fields of Interest

This work combines aspects of sparse signal representation, array processing, and high-level synthesis to create a FPGA based platform for DOA estimation. Furthermore, is has the potential for a future use in the analysis of data sets from millimeter wave channel sounders to characterize the wave propagation channel.

2. Sparse Represenation of Signals

2.1. Linear Measurements

A measurement system acquires n measurements as a vector $\boldsymbol{y} \in \mathbb{R}^n$ which are observed from a linear system $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{y}$ where $\boldsymbol{A}$ is a sensing matrix with $\boldsymbol{A} \in \mathbb{R}^{n \times m}$ and $\boldsymbol{x} \in \mathbb{R}^m$ is an unobservable signal. A linear system of that kind is underdetermined if there are more unknowns than equations, i.e. $n < m$. In that case, $\boldsymbol{A}$ has reduced the dimensionality of $\boldsymbol{x}$. If $\boldsymbol{A}$ has full-rank, indefinitely many solutions exist for a higher-dimensional $\boldsymbol{x} \in \mathbb{R}^m$. In order to arrive at a unique solution, additional criteria have to be added. A very popular solution is the minimization of the squared Euclidean norm $\|\boldsymbol{x}\|_2^2$ with respect to the given equation, $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{y}$, which results in the pseudo-inverse $\boldsymbol{A}^T(\boldsymbol{A}\boldsymbol{A}^T) = \boldsymbol{A}^+$ and the unique solution $\widehat{\boldsymbol{x}} = \boldsymbol{A}^+\boldsymbol{y}$. Although this least square solution is very popular in a lot of fields, it is not always a good choice. Especially for sparse $\boldsymbol{x}$, where the least squares method minimizes the energy of $\widehat{\boldsymbol{x}}$ instead of promoting sparseness [3].

2.2. Sparse Representation

If a signal, described by the vector $\boldsymbol{x}$, consists of at most k nonzero elements, i.e. $\|\boldsymbol{x}\|_0 \leq k$ we say it is k-sparse. Most real-world signals are not sparse as such. However, if a sparse representation $\widehat{\boldsymbol{x}}$, which is now k-sparse, can be found to be a good approximation, the signal is said to be compressible, approximately sparse, or relatively sparse [4]. The task for the inverse problem is now to find good approximate sparse solutions with methods[1] that seek sparse solutions and are computationally tractable. One possible method is the use of a Bayesian framework to arrive at sparse solutions for regression analysis known as SBL [6].

2.3. Multiple Measurement Vectors

A measurement system takes linear measurements $\boldsymbol{y}_l$ from the measurement system to obtain a single measurement vector (SMV). The sparse representation of the signal has few non-zero coefficients in $\boldsymbol{x}_l$. A combined matrix $\boldsymbol{X} = [\boldsymbol{x}_1, \boldsymbol{x}_2, \dots, \boldsymbol{x}_L]$ which has constant indices of their non-zero coefficients of each $\boldsymbol{x}_l$ across snapshots, is said to be row-sparse. For row-sparse $\boldsymbol{X}$, multiple snapshots of $\boldsymbol{y}_l$ can be combined into a multiple measurement vector (MMV) $\boldsymbol{Y}$ to benefit from jointly processing the measurements during recovery under some conditions.

[1]Compressed-Sensing is a branch of sparse and redundant representations which is often confused with being the whole field of theory [5].

2.4. Direction of Arrival Estimation on Sensor Arrays

Locating a number of sources is the task of direction finding where the directions of arrival (DOA) of waves is estimated by measuring wavefields with sensor arrays. The waves are restricted to plane waves, i.e. farfield assumption, propagating in a locally homogeneous medium and the sensors of the array are assumed to be isotropic. Sensors are placed on points $\boldsymbol{p}_n \in \mathbb{R}^3$ in space and a signal $y_n(t, \boldsymbol{p}_n)$ is measured independently at each sensor $n \in \{1, 2, \ldots, N\}$. Each plane wave differs in its time of arrival at the various sensors. For a single narrowband wave with wavenumber $\boldsymbol{k} \in \mathbb{R}^3$, where $|\boldsymbol{k}|_2 = {}^{\omega}/_{c} = {}^{2\pi}/_{\lambda}$, the signal with propagation delay at each sensor n is described by $y_n(t, \boldsymbol{p}_n) = \boldsymbol{y}(t)\mathrm{e}^{-j\boldsymbol{k}^T \cdot \boldsymbol{p}_n}$. The signals of all used sensors are stacked to a SMV $\boldsymbol{y}(t, \boldsymbol{P}) = y(t)\boldsymbol{a}(\boldsymbol{k})$ and measured as a single snapshot, where vector $\boldsymbol{a}(\boldsymbol{k})$ is the steering vector[2] incorporating all spatial characteristics of the array [7]. The DOA is estimated by finding the direction of $\boldsymbol{k}$ through $y(t, \boldsymbol{P})$.

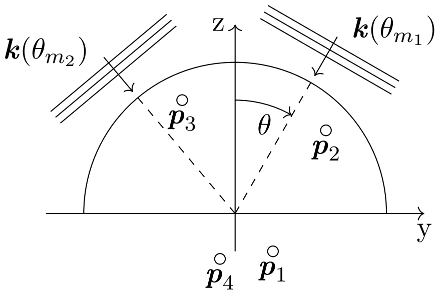


Figure 2.1.: Directions of arrival (DOA) for two plane waves with wavenumber $\boldsymbol{k}_m$ impinging on four sensors given by position $\boldsymbol{p}_n$.

2.4.1. Sparse Representation in DOA

In a 2-dimensional Euclidean space DOA is estimated by finding the polar angle θ of plane waves with corresponding wavenumber $\boldsymbol{k}(\theta)$. Hence, the steering vector is $\boldsymbol{a}(\boldsymbol{k}) = \boldsymbol{a}(\theta)$. All possible directions $\theta \in [\theta_{\min}, \theta_{\max}]$, posed as a finite-dimensional vector $\boldsymbol{\theta} \in \mathbb{R}^M$ with M possible directions, form a collection of steering vectors, the steering matrix $\boldsymbol{A} \in \mathbb{R}^{N \times M}$. The signals of the sensor array are the SMV $\boldsymbol{y}$. A single plane wave impinging on the array with directional angle θ_m, with $m \in \{1, 2, \ldots, M\}$, leads to a $\boldsymbol{y}$ which can be sparsely represented by a vector $x \in \mathbb{C}^M$ with a single non-zero entry. The situation for K plane waves with different directions, impinging on the sensor array is then a linear combination of K single-entry vectors $\boldsymbol{x}_m(t) = x_m(t) \cdot \boldsymbol{e}_m$ or a sparse representation with a vector $\boldsymbol{x}(t)$ which has K non-zero coefficients. The steering matrix $\boldsymbol{A}$ serves as a dictionary of all possible source locations. For fewer sensors (N) in the array than possible source locations (M), the linear system

$$\boldsymbol{y}(t) = \boldsymbol{A}\boldsymbol{x}(t)$$

[2]steering vector or array manifold vector or replica vector

is underdetermined. However, for small K compared to M, sparse recovery techniques can be utilized.

2.4.2. Array SNR

The sensor array adds signals coherently and noise incoherently to improve signal-to-noise ratio (SNR) [7]. Array SNR in decibel for a single snapshot of the sensor array is defined as [1]

$$\mathrm{SNR} = 10 \log_{10} \frac{\mathsf{E}[\|\boldsymbol{A}\boldsymbol{x}_l\|_2^2]}{\mathsf{E}[\|\boldsymbol{n}_l\|_2^2]} \tag{2.1}$$

A important performance measure for the quality of DOA estimation is the root mean squared error (RMSE) of the estimated directions to the real K directions given by the signal test setting and evaluated for J realizations.

$$\text{DOA RMSE} = \sqrt{\frac{1}{K}\frac{1}{J}\sum_{k}^{K}\sum_{j}^{J}|\theta_{k,j} - \hat{\theta}_{k,j}|^2} \tag{2.2}$$

A low DOA RMSE is desirable for good resultion of a localization system.

3. High-Level Synthesis

FPGAs have a compute paradigm which is not deviated from the *von Neumann* architecture. Although they can be programmed to mimic common architectures which might be advantageous during development of those, FPGAs have a flexible interconnection between digital elements and are thus good at using a datapath paradigm for signal processing problems. Instructions don't have to be fetched because they are inherintly defined by the configured datapath. The flexibility allows application-specific architectures. The switching frequencies of an FPGA with only a few 100 MHz is slow compared to those of a GPU or a CPU with several GHz. However, their effectiveness in terms of latency and power consumption on bit manipulation, stream processing, pipelining, and providing tightly placed on-chip memory makes them a preferable technology for a great number of applications.

High-level synthesis (HLS) for FPGAs is the translation of algorithmic or behavioral descriptions to application-specific architectures, which consist of datapath and control unit [8]. A sequence of operations is described on a higher abstract level in a programming language or a subset supported by the HLS tool from which a generated digital system is described at the register-transfer level (RTL) in the form of a hardware description language (HDL).

The HLS process undergoes several phases for a source module.

Analysis Processes the HLS source code analogous to a software compiler to create an intermediate representation. After higher-level optimization, a control flow graph (CFG) is created for assuring strict control dependency due to loop and branching conditions. Between nodes of the CFG are basic blocks with a defined entry point, exit point, and multiple operations in between. A data flow graph (DFG) is created for each block to capture data dependency between operations or possible parallel operations. A combined control/data flow graph (CDFG) from CFG and DFG is now used for actual hardware synthesis.

Scheduling Finds the control steps from the CDFG for each DFG (independently or combined). Memory access, as load-store operations, also need to be scheduled and technology limits have to be obeyed. Different scheduling algorithms are possible to achieve a trade-off between few control steps with high parallel execution of operations, and more control steps with chances of operator sharing[1]. This affects throughput, latency, and needed resources of the module.

[1]Operator sharing is not necessarily the best way to reduce needed hardware resources for FPGAs as additional control logic with multiplexers can exceed the resources from the cut down operators.

Allocation The minimum number of needed operations and storage elements are used to allocate functional units with corresponding hardware resources.

Binding Assigns the functional units to operators and storage elements and adds multiplexers to select them for the scheduled control step.

The control steps for a module are realized by a controller, i.e. a finite-state machine, and added to the resulting register-transfer level (RTL) description.

In order to arrive at an efficient synthesis result, the HLS compiler has to apply a plethora of compiler optimizations [8]. However, some improvements can only be achieved at the algorithmic level by the programmer.

4. HLS Prototyping

Sensor array data is usually sampled by ADCs and processed by a directly connected digital signal processor or FPGA. The algorithms which use the data are usually designed with tools and programming languages at higher levels of abstraction. When prototyping an algorithm, stimuli are generated for the higher-level software model and the results are verified. The software model is iteratively improved until the errors are sufficiently small. The next step is to create a prototype which is compatible with the hardware platform. Likewise, a hardware simulation of the prototype is verified with a testbench, to produce the same output as a reference obtained from a higher-level model, the golden reference. If the output does not match the golden reference, the higher-level model has to be refined for the next iteration. Finding the root cause of an error can be difficult, especially when a higher-level model is not bit-accurate compared to a derived hardware description. It is therefore advantageous to design algorithm prototypes directly with bit-accurate high-level languages which offer automated transformation into hardware descriptions. Furthermore, hardware description language (HDL) produce bit- and cycle-accurate hardware simulations. As this simulates every clock transition, it makes the simulation unnecessary slow for algorithm prototyping. The behavioral modeling with high-level synthesis (HLS) offers a huge reduction in simulation time.

For HLS with Vivado HLS, the C/C++ sources of the design use only a subset of the programming language combined with the headers from Vivado HLS for which the compiler can create synthesizeable code. Vivado HLS allows bit-accurate C simulation of design entries. The *gcc* and *clang* compilers can be used for a behavioral C simulation. A testbench creates stimuli and verifies a design under test (DUT). The testbench for the C simulation does not need to be synthesizeable and can therefore be any piece of code which can call the C design entry of the DUT with necessary arguments. Moreover, the DUT itself can be enhanced by simulation-only code for further verification[1] for prototyping purposes, e.g. to create self-checking modules.

The most simple testbench for a DUT is probably a C program with a *main* function which initializes the arguments to zero, calls the design entry function and quits. Wrappers for other programming languages, scripting languages or inter-process communication with other processes can also be used as long as the required binary format of the design entry arguments can be created. An obvious choice for a device-independent inter-process communication protocol is the ubiquitous TCP/IP stack. A transport control protocol (TCP) and user datagram protocol (UDP) server is created which receives the arguments for a DUT from a remote process, calls the design entry, and sends the results back to the remote process.

[1] A useful library for enhancing C simulation for verification was Armadillo [9], a C++ linear algebra library.

4.1. TCP/IP

When using MATLAB to aid in developing numerical algorithms, the MATLAB MEX interface allows to exchange data between MATLAB and foreign piece of code. This approach is used in [10] to send data from MATLAB to the design entry of a DUT and receive the results. MATLAB also allows to communicate with remote hosts via the TCP/IP protocol to send and receive data. For an algorithm prototype accessible via TCP/IP, MATLAB can create and send complex stimuli to the prototype and evaluate the results received from the prototype utilizing all powerful features included within the MATLAB integrated development environment.

TCP/IP communication can be achieved in MATLAB for TCP with a *tcpclient* object which has *write* and *read* methods. UDP communication is also implemented in MATLAB in several toolboxes, e.g. the DSP System Toolbox[2]. Both protocols can be used as an interface to the prototype. TCP is the more convenient protocol for prototyping if the algorithm can not cope with packet loss, as it is connection-oriented and has flow-control built-in. UDP, on the other hand, can be implemented with a less complex protocol stack and is therefore more easily integrated, e.g. as a full hardware implementation.

Care has to be taken when crossing boundaries of TCP packets, as a connector to the DUT needs to re-align payload of fragmented packets.

4.2. MATLAB / Prototype Interface

The binary format needs to be compatible between MATLAB and the prototype. Common binary formats between Vivado HLS and MATLAB are unsigned integer types (uint8_t, uint16_t, uint32_t, uint64_t), signed integer types (int8_t, int16_t, int32_t, int64_t), and floating point types single- and double-precision floating point refering to the IEEE 754 binary32 and binary64 formats. Other structures need to have a fixed memory layout based on the compatible data types.

4.2.1. MATLAB fi-object to Vivado HLS Arbitrary Precision

MATLAB has a data type for fixed-point number objects (`fi`). This data type is compatible as an exchange format with Vivado HLS's `ap_(u)fixed` data type when used with restrictions. A value V can be used to create a fixed-point numeric object with signed property S, word length W, and fraction length F, with the command `fi(V,S,W,F)`. For real numbers and S = true, `fi(V,true,W,F)` maps to `ap_fixed<W,W-F>`, for real numbers and S = false, `fi(V,false,W,F` maps to `ap_ufixed<W,W-F>`.

For vectors, each fixed-point number with word length W is aligned to the next word length $W_B \in \{8, 16, 32, 64, 128\}$bits in memory. When mapping complex numbers to the `hls::x_complex` container, real and imaginary part are interleaved with real value first in memory.

[2]The DSP System Toolbox provides a UDP sender (*dsp.UDPSender*) and receiver (*dsp.UDPReceiver*) object for bidirectional communication with raw data.

4.3. C-Simulation Platform

A C++ application is implemented as a connector to a higher level testbench for a DUT implemented with HLS. It provides a TCP server to receive stimuli from the remote host and send back results. In [10], input data is received, the design entry of the DUT is called and only upon return, data is sent back to the remote host, which introduces high latency. To be able to stream data continuously, two threads are spawned which act as TCP receiver and TCP sender. Execution of the DUT is carried out in parallel as shown in Figure 4.1. Input data from TCP is passed to an input FIFO of the DUT. Data from the output FIFO of the DUT is sent back while execution can still continue. This enables the remote host to get intermediate results as soon as they are available.

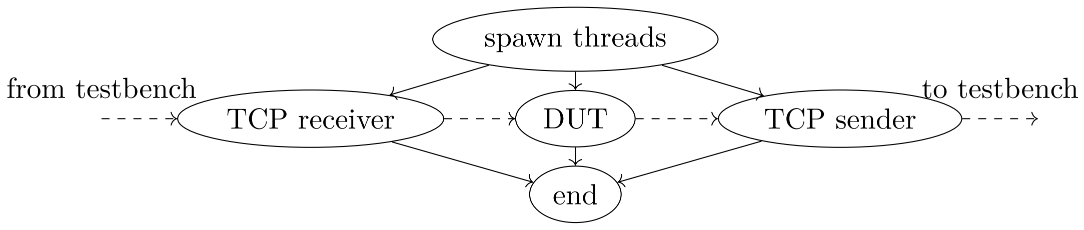


Figure 4.1.: Network adapter for design under test (DUT). The dashed line shows the dataflow. The solid line shows the use of threads to achieve continuous streaming of data.

4.4. Hardware Prototyping Platform

Similar to the C-simulation platform, the synthesized HLS DUT can be evaluated on an FPGA by receiving stimuli and sending back results through a TCP or UDP connector on the hardware. The connector passes input and output data between TCP/UDP and the DUT through FIFOs.

Embedded protocol stacks for TCP and UDP exist which allow the realization of such a connector with little logic resources, cf. [11]. If the prototyping hardware platform has enough resources, a soft-core microcontroller can be added to the FPGA design to make use of free widely available software protocol stacks. In this work a MicroBlaze soft-core microcontroller, an *Ethernet-lite* IP and a DDR3 memory controller are used together with the *lwIP* stack to implement the TCP connector for the DUT. The hardware prototyping platform is shown in Figure 4.2. There is no dependency on using this technology for interfacing the DUT, which means the platform could be substituted with any other TCP/UDP to AXI4-Streaming converter.

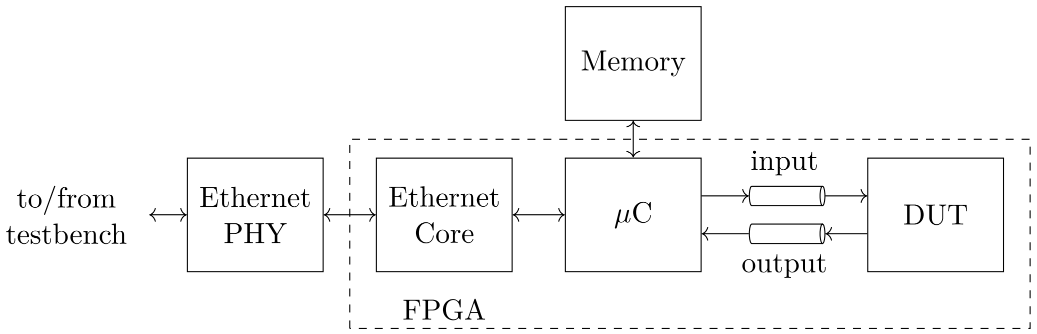


Figure 4.2.: Hardware prototyping platform.

5. Algorithms

This chapter breaks down the algorithm into multiple blocks to simplify and specialize its linear algebra for implementation with HLS. An important insight is the independence of the number of snapshots L. The multisnapshot $\boldsymbol{Y}$ can be substituted by $\boldsymbol{S}_y$ throughout the equations.

5.1. Data sample covariance matrix

The data sample covariance matrix $\boldsymbol{S}_y$ arises from multiple snapshots $\boldsymbol{y}_l$ of an N sensor array. By assuming, that data for snapshot $l = 1$ is available before snapshot $l = 2$ and so on, it is advantageous to fully process each snapshot and discard the data before the next arrives to minimize latency and memory requirements. The sample covariance of the multisnapshot is therefore the sum of its single snapshots.

$$\boldsymbol{S}_y = \frac{\boldsymbol{Y}\boldsymbol{Y}^H}{L} = \boldsymbol{S}_y^H \tag{5.1}$$

$$\boldsymbol{Y}\boldsymbol{Y}^H = \sum_{l=1}^{L} \boldsymbol{y}_l \boldsymbol{y}_l^H \tag{5.2}$$

$$\boldsymbol{y}_l \boldsymbol{y}_l^H = \begin{pmatrix} y_{1,l}y_{1,l}^* & y_{1,l}y_{2,l}^* & \cdots & y_{1,l}y_{N,l}^* \\ y_{2,l}y_{1,l}^* & y_{2,l}y_{2,l}^* & \cdots & y_{2,l}y_{N,l}^* \\ \cdots & \cdots & \cdots & \cdots \\ y_{N,l}y_{1,l}^* & y_{N,l}y_{2,l}^* & \cdots & y_{N,l}y_{N,l}^* \end{pmatrix} \tag{5.3}$$

Moreover, as $\boldsymbol{S}_y$ is Hermitian, only the upper triangle needs to be evaluated and stored.

5.2. Data covariance matrix

The data covariance matrix $\boldsymbol{\Sigma}_y$ is defined as [1, eq. 13]

$$\boldsymbol{\Sigma}_y = (\sigma^2)^{\text{new}} \boldsymbol{I}_N + \boldsymbol{A}\boldsymbol{\Gamma}\boldsymbol{A}^H \tag{5.4}$$

where $\boldsymbol{\Gamma} = \operatorname{diag}(\boldsymbol{\gamma}^{\text{new}})$ and $\boldsymbol{\gamma}^{\text{new}} = [\gamma_1^{\text{new}}, \ldots, \gamma_M^{\text{new}}]^T$. Each $\gamma_m^{\text{new}} \in \boldsymbol{\gamma}^{\text{new}}$ is updated by the SBL algorithm [1, eq. (SBL1) and (SBL)] in a sequential way, i.e. first γ_1^{new}, then

γ_2^{new} up to γ_M^{new}. Therefore, equation (5.4) can be formulated as

$$[\boldsymbol{\Sigma}_y]_{ij} = \sum_{m=1}^{M} [\boldsymbol{a}_m \boldsymbol{a}_m^H]_{ij} \gamma_m^{\text{new}} \qquad i \neq j \tag{5.5}$$

$$[\boldsymbol{\Sigma}_y]_{ii} = \sum_{m=1}^{M} [\boldsymbol{a}_m \boldsymbol{a}_m^H]_{ii} \gamma_m^{\text{new}} + (\sigma^2)^{\text{new}} \tag{5.6}$$

where only the lower triangle or upper triangle and the diagonal have to be calculated as $\boldsymbol{\Sigma}_y$ is Hermitian. The noise power estimate $(\sigma^2)^{\text{new}}$ is evaluated after the last value of γ_m^{new} is available, i.e. γ_M^{new}. Thus, it is added to the diagonal as a last step.

5.3. Inverse data covariance matrix

The inverse of the data covariance matrix $\boldsymbol{\Sigma}_y$ is needed in calculation of [1, eq. (SBL1) and (SBL)]. If a matrix inverse operation is used in compound with a matrix or vector product the inverse is not explicitly needed.[1] Instead a linear system $\boldsymbol{Ax} = \boldsymbol{b}$ needs to be solved for $\boldsymbol{x}$ only. $\boldsymbol{\Sigma}_y$ as defined in (5.4) is derived from the expectation of (1.1). It is therefore the sum of uncorrelated noise $\sigma^2\boldsymbol{I}$ and $\mathsf{E}[\boldsymbol{A}\boldsymbol{x}_l\boldsymbol{x}_l^H\boldsymbol{A}^H]$. The first is clearly a positive definite matrix, the second is a Gram matrix[2] with elements $\boldsymbol{a}_i\boldsymbol{\Gamma}\boldsymbol{a}_j^H = \boldsymbol{a}_i\boldsymbol{\Gamma}^{-1/2}\left(\boldsymbol{a}_j\boldsymbol{\Gamma}^{-1/2}\right)^H$ for $j, i = [1, \ldots, M]$. As a Gram matrix is always positive semidefinite [12, Theorem 3.1], the sum of a positive definite and a positive semidefinite matrix is also positive definite if each one and the sum is bigger than $\boldsymbol{0}$, cf. [12, Definition 3.1]. Therefore, $\boldsymbol{\Sigma}_y$ is positive definite if $\sigma^2 > 0$ and $\boldsymbol{x}^H\boldsymbol{A}\boldsymbol{\Gamma}\boldsymbol{A}^H\boldsymbol{x} \geq \boldsymbol{0}$ for all $\boldsymbol{x} \neq \boldsymbol{0}$. We can do a Cholesky factorization of $\boldsymbol{\Sigma}_y = \boldsymbol{L}_{\boldsymbol{\Sigma}}\boldsymbol{L}_{\boldsymbol{\Sigma}}^H$ and make use of the lower triangular Cholesky factor $\boldsymbol{L}_{\boldsymbol{\Sigma}}$ where

$$\boldsymbol{\Sigma}_y^{-1} = \boldsymbol{L}_{\boldsymbol{\Sigma}}^{-H}\boldsymbol{L}_{\boldsymbol{\Sigma}}^{-1} \tag{5.7}$$

is needed.

5.4. SBL1 and SBL

The common numerator of the two different equations for updating γ_m^{new}, [1, eq. (SBL1) and (SBL)] is

$$\frac{\|\boldsymbol{Y}^H\boldsymbol{\Sigma}_y^{-1}\boldsymbol{a}_m\|_2}{\sqrt{L}} = \sqrt{\boldsymbol{b}_m^H\boldsymbol{S}_y\boldsymbol{b}_m}. \tag{5.8}$$

Due to the Hermitian matrix $\boldsymbol{S}_y$, the implemented arithmetic operations accesses only the upper triangle of $\boldsymbol{S}_y$, cf. appendix A.1. The vector $\boldsymbol{b}_m = \boldsymbol{\Sigma}_y^{-1}\boldsymbol{a}_m$ is calculated with (5.7) by first solving

$$\boldsymbol{L}_{\boldsymbol{\Sigma}}\tilde{\boldsymbol{a}}_m = \boldsymbol{a}_m \tag{5.9}$$

[1] In MATLAB editor, a tooltip immediately notifies the user about the performance gain of using `x=A\B` instead of `x=inv(A)b` when solving $\boldsymbol{Ax} = \boldsymbol{b}$ for $\boldsymbol{x}$.

[2] The names Grammian matrix, Gramian matrix or Gram matrix have all been found in literature.

for $\tilde{\boldsymbol{a}}_m$ with forward substitution and, second, solving

$$\boldsymbol{L}_{\boldsymbol{\Sigma}}^H \boldsymbol{b}_m = \tilde{\boldsymbol{a}}_m \tag{5.10}$$

for $\boldsymbol{b}_m$ with back substitution. For [1, eq. (SBL1)], the intermediate result $\tilde{\boldsymbol{a}}_m$ can be used to calculate $\boldsymbol{a}_m^H \boldsymbol{\Sigma}_y^{-1} \boldsymbol{a}_m$ for every index m and iteration as

$$\boldsymbol{a}_m^H \boldsymbol{\Sigma}_y^{-1} \boldsymbol{a}_m = \boldsymbol{a}_m^H \boldsymbol{L}_{\boldsymbol{\Sigma}}^{-H} \boldsymbol{L}_{\boldsymbol{\Sigma}}^{-1} \boldsymbol{a}_m = \tilde{\boldsymbol{a}}_m^H \tilde{\boldsymbol{a}}_m = \|\tilde{\boldsymbol{a}}_m\|_2^2. \tag{5.11}$$

For [1, eq. (SBL)], the term $\boldsymbol{a}_m^H \boldsymbol{S}_y^{-1} \boldsymbol{a}_m$ needs to be calculated for every index m only once, for all iterations of the multisnapshot. The Cholesky factorization of $\boldsymbol{S}_y = \boldsymbol{R}_{\boldsymbol{S}_y}^H \boldsymbol{R}_{\boldsymbol{S}_y}$ helps again to reduce calculation complexity.

$$\boldsymbol{a}_m^H \boldsymbol{S}_y^{-1} \boldsymbol{a}_m = \boldsymbol{a}_m^H \boldsymbol{R}_{\boldsymbol{S}_y}^{-1} \boldsymbol{R}_{\boldsymbol{S}_y}^{-H} \boldsymbol{a}_m = \left(\boldsymbol{R}_{\boldsymbol{S}_y}^{-H} \boldsymbol{a}_m\right)^H \left(\boldsymbol{R}_{\boldsymbol{S}_y}^{-H} \boldsymbol{a}_m\right) = \|\boldsymbol{R}_{\boldsymbol{S}_y}^{-H} \boldsymbol{a}_m\|_2^2 \tag{5.12}$$

5.5. Active set

Constructing the active set $\mathcal{M}$, thus finding the K peaks in $\boldsymbol{\gamma}^{\text{new}}$, is a filtering and a selection problem. The filtering makes sure to identify neighbouring points as being a peak or no peak. A trivial peak detecting filter algorithm is

```
if $\gamma_{m-1}^{\text{new}} < \gamma_m^{\text{new}} \leq \gamma_{m+1}^{\text{new}}$ then
    $\gamma_m^{\text{new}}$ is a peak, $\gamma_m^{\text{filter}} = \gamma_m^{\text{new}}$
else
    $\gamma_m^{\text{new}}$ is no peak, $\gamma_m^{\text{filter}} = 0$
end if
```

With limited precision in calculating $\boldsymbol{\gamma}^{\text{new}}$, small bumps near a bigger peak would falsely be identified as several peaks and could prevent smaller peaks from being selected. Therefore, additional pre-filtering is used with a moving averaging window FIR filter. The division in averaging can be skipped, because ultimatively we are just interested in the index m of the peak. With a window size of 3, improved results could be achieved. For an input $x_\gamma[m]$, the output $y_\gamma[m-1]$ is

$$y_\gamma[m-1] = \sum_{i=0}^{2} x_\gamma[m-i] \qquad x_\gamma[m] = \begin{cases} \gamma_1^{\text{new}}, & \text{for } m \leq 1 \\ \gamma_m^{\text{new}}, & \text{for } 1 < m < M \\ \gamma_M^{\text{new}}, & \text{for } m \geq M \end{cases} \tag{5.13}$$

After filtering and identifying of the peaks, the largest peaks have to be selected. The selection could be done in a parallel way by, e.g. a sorting network [13] with minimal total number of cycles. However, the time between calculating the result of γ_m^{new} and $\gamma_{m+1}^{\text{new}}$ is assumed to be larger than linearly comparing one value γ_m^{new} with previous $\mathcal{M}_{\gamma_{m-1}}^{\text{new}}$, the set of K largest peaks up to index $m-1$. Therefore, the following linear selection, used iteratively M times, is found to be sufficiently fast.

```
1: for all $m \in [1, \ldots, M]$ do
```

```
2:     find $\min(\mathcal{M}_{\gamma_{m-1}^{\text{new}}})$
3:     if $\gamma_m^{\text{new}} > \min(\mathcal{M}_{\gamma_{m-1}^{\text{new}}})$ then
4:         replace $\min(\mathcal{M}_{\gamma_{m-1}^{\text{new}}})$ with $\gamma_m^{\text{new}}$
5:         replace $\arg\min(\mathcal{M}_{\gamma_{m-1}^{\text{new}}})$ with $m$
6:     end if
7: end for
```

This adds only minimal delay after the last value, i.e. γ_M^{new}, was calculated, because the smallest element, i.e. $\min(\mathcal{M}_{\gamma_{M-1}^{\text{new}}})$, is already selected in advance.

5.6. Noise Variance

The noise variance estimation, as hyperparameter σ^2, is calculated in [1, eq. (27)] using the estimated active steering vectors, combined in $\boldsymbol{A}_{\mathcal{M}}$, and the sample covariance matrix $\boldsymbol{S}_y$.

$$(\sigma^2)^{\text{new}} = \frac{1}{N-K}\text{tr}((I_N - \boldsymbol{A}_M \boldsymbol{A}_M^+)\boldsymbol{S}_y) \tag{5.14}$$

The term $\frac{1}{N-K}$ is simply a multiplication with a constant and $\boldsymbol{A}_M^+$ is the pseudo-inverse of $\boldsymbol{A}_M$.

5.6.1. Projection Matrix

A projection matrix $\boldsymbol{P} = \boldsymbol{A}_M \boldsymbol{A}_M^+$ is defined in [1, eq. (25)] which can be decomposed using Cholesky decomposition of $\boldsymbol{A}_{\mathcal{M}}^H \boldsymbol{A}_{\mathcal{M}} = \boldsymbol{R}_{\mathcal{M}}^H \boldsymbol{R}_{\mathcal{M}}$ with the Cholesky factor $\boldsymbol{R}_{\mathcal{M}}$ in its upper triangle matrix version, cf. appendix A.2.

$$\boldsymbol{P} = \boldsymbol{A}_{\mathcal{M}}(\boldsymbol{R}_{\mathcal{M}}^H \boldsymbol{R}_{\mathcal{M}})^{-1} \boldsymbol{A}_{\mathcal{M}}^H = \boldsymbol{Q}\boldsymbol{Q}^H \tag{5.15}$$

with

$$\boldsymbol{R}_{\mathcal{M}}^H \boldsymbol{Q}^H = \boldsymbol{A}_{\mathcal{M}}^H \tag{5.16}$$

The matrix $\boldsymbol{Q}$ can be retrieved by solving (5.16), which is of dimension $N \times K$. Alternatively, a different matrix $\boldsymbol{Q}$ can be derived by a QR-decomposition of $\boldsymbol{A}_{\mathcal{M}} = \boldsymbol{Q}\boldsymbol{R}_{\mathcal{M}}$ (cf. thin QR Factorization [14, Theorem 5.2.3]). However, Vivado HLS does only include a floating point version of the QR-decomposition so we continue using the Cholesky factorization for the decomposition.

5.6.2. Trace

For the evaluation of the trace

$$\text{tr}((\boldsymbol{I}_N - \boldsymbol{P})\boldsymbol{S}_y) = \text{tr}(\boldsymbol{S}_y) - \text{tr}(\boldsymbol{P}\boldsymbol{S}_y),$$

calculating $\text{tr}(\boldsymbol{S}_y)$ involves just summing up the diagonal elements $[\boldsymbol{S}_y]_{nn}$, which can be done once for a matrix $\boldsymbol{S}_y$. The other term can be calculated with (5.16) as

$$\text{tr}(\boldsymbol{P}\boldsymbol{S}_y) = \sum_{k=1}^{K} \boldsymbol{q}_k^H \boldsymbol{S}_y \boldsymbol{q}_k, \tag{5.17}$$

with the same underlying arithmetic operations but without the square root as in (5.8),cf. appendix A.3.

5.7. Convergence rate

The convergence rate ϵ is defined as [1, eq. 25]

$$\epsilon = {\|\boldsymbol{\gamma}^{\text{new}} - \boldsymbol{\gamma}^{\text{old}}\|_1}/{\|\boldsymbol{\gamma}^{\text{old}}\|_1} \tag{5.18}$$

As each value γ_m^{new} is calculated in series, a iterative formulation is useful. Furthermore, as we only need to compare ϵ with a minimum convergence rate ϵ_{min}, we can reformulate the problem to elimitate a costly divide operation.

$$\begin{aligned} \Delta\gamma_M = \|\boldsymbol{\gamma}^{\text{new}} - \boldsymbol{\gamma}^{\text{old}}\|_1 = \sum_m^M |\gamma_m^{\text{new}} - \gamma_m^{\text{old}}| \\ \Delta\gamma_m = \Delta\gamma_{m-1} + |\gamma_m^{\text{new}} - \gamma_m^{\text{old}}| \\ \Delta\gamma_0 = 0 \end{aligned} \tag{5.19}$$

The convergence can be evaluated when the last value, i.e. γ_M^{new}, is calculated.

$$\begin{aligned} \epsilon > \epsilon_{\text{min}} \\ \|\boldsymbol{\gamma}^{\text{new}} - \boldsymbol{\gamma}^{\text{old}}\|_1 > \epsilon_{\text{min}} \cdot \|\boldsymbol{\gamma}^{\text{old}}\|_1 \\ \Delta\gamma_M > \epsilon_{\text{min}} \cdot \|\boldsymbol{\gamma}^{\text{old}}\|_1 \end{aligned} \tag{5.20}$$

6. Vivado HLS

The algorithms developed in chapter 5 are implemented using *Xilinx Vivado HLS*, a high-level synthesis (HLS) tool targeting Xilinx FPGAs based on the HLS tool AutoPilot. It allows to create a hardware specification based on the C programming language[1] which can be transformed into a specification on the RTL level by the HDLs, VHDL and Verilog. Thus, development can focus on the algorithmic level.

In this chapter, the utilized fixed-point data types are analyzed and abstracted for linear algebra. Ways of splitting a large Vivado HLS project into multiple standalone projects are explored. Finally, other useful implementation techniques are discussed.

6.1. Arbitrary Precision Data Type

Fixed-point operations are typically faster, need less hardware resources, and consume less power than floating-point operations which lead to cheaper devices. The HLS block for e.g. calculating the unscaled sample covariance matrix $\boldsymbol{Y}\boldsymbol{Y}^H$ without further architectural optimization takes 14 DSP Slices compared to 4 DSP Slices and needs at least 1.25 more cycles when half-precision floating-point is used instead of a 16 bit fixed-point data type. Therefore, fixed-point implementations can lead to more efficient digital systems.

Binary fixed-point numbers, when used in software, are stored in registers and memory with a data width which is suitable for the underlying processor architecture, e.g. usually 8 bit, 16 bit, 32 bit, 64 bit, or a multiple of that. Vivado HLS defines arbitrary precision data types, which are bit-accurate and therefore a better hardware description of a binary fixed-point number for digital design. Those data types store a number in its two's complement representation at adjustable data width. The arbitrary precision data type is further subdivided into integers `ap_(u)int<W>` and fractional type for binary fixed-point numbers `ap_(u)fixed<W,I>`, each with a signed and an unsigned variant [15]. The fractional binary fixed-point data types `ap_(u)fixed<W,I>` are of special interest to our application as they allow us to describe numbers for a given word length of W bits to have a given integer length of I bits and a fractional length of $W - I$ bits. This is basically a scaled version of an integer with the same word length W, scaled by a factor of $2^{-(W-I)}$, which means there is no difference in a storage point of view. The scaling factor is a power of 2 which, for binary numbers stored in two's complement, can be achieved efficiently by bit-shifting. However, it is often advantageous to have scaling implicitly defined by the data type, especially when writing digital signal processing (DSP) algorithms with distinct data types of different scale. In Vivado HLS, a lot of

[1]C, C++, and SystemC can be used.

arithmetic operations are defined for the arbitrary precision data type which respect different word length and integer length of the operands.

The following set of numbers can be represented by `ap_ufixed<W,I>`

$$u \cdot 2^{-(W-I)}, \quad u \in \{x \in \mathbb{N} \mid 0 \leq x \leq 2^W - 1\} \tag{6.1}$$

and for `ap_fixed<W,I>`

$$i \cdot 2^{-(W-I)}, \quad i \in \{x \in \mathbb{N} \mid -2^{W-1} \leq x \leq 2^{W-1} - 1\}. \tag{6.2}$$

This representation is analogous to the `sfixed` and `ufixed` types[2] of the fixed-point package [16, fixed_pkg.vhdl] added in VHDL 2008.

Precision Traits

The result of arithmetic operations with fixed-point numbers may need a higher word length or precision to prevent errors due to overflows, underflows and rounding. A multiplication of $c = a \cdot b = 3 \cdot 7 = 21$, with ($a = 3$, $W_a = 2$, $I_a = 2$) and ($b = 7$, $W_b = 3$, $I_a = 3$), needs at least a word length of $W_c = W_a + W_b$ with an integer length of $I_c = I_a + I_b$, for example. Manual bit width accounting would be a cumbersome and error-prone way of specifying the widths of each variable. Instead, a C++ template programming technique called *traits* [17] helps with determining the widths of operation results. Traits for the arbitrary precision data types are provided with the type `hls::x_traits<`*`type1, type2`*`>::`*`operator`*`_T` from the header files in the Vivado HLS library package.

```
typedef ap_ufixed<2,2> a_t;
typedef ap_ufixed<3,3> b_t;
typedef typename hls::x_traits<a_t,b_t>::MULT_T c_t;

a_t a = 3;
b_t b = 7;
c_t c;

c = a * b; // 21
```

The implicit scaling also allows to set $I > W$ or $I \leq 0$ which can be seen in the following example.

```
typedef ap_ufixed<2,5> d_t;
typedef ap_ufixed<2,-2> e_t;
typedef typename hls::x_traits<d_t,e_t>::MULT_T f_t;

d_t d = 16;
```

[2]Instead of word length W and integer length I, the bit position of most significant bit (MSB) and least significant bit (LSB) relative to the decimal point is specified for VHDL 2008 types `sfixed` and `ufixed`.

```
e_t e = e_t("0b0.0001"); // 1/(2^4)
f_t f;

f = d * e; // f = 1, ap_ufixed<2+2,5+(-2)>
```

The precision traits permit a convinient way to get the resulting widths for one operation. For operations like $b = \|\boldsymbol{a}\|_1 = \sum\_a_i \in \mathcal{A}|a_i|$, the final bit-width respecting the dimension of $\boldsymbol{a}$ can be retrieved with the trait `hls::x_traits_d<type, dimension>::ACCUM_T`. If an algorithm includes a feedback, exact computation without error is not possible anymore for unbounded input because the data width can not grow indefinitely. In that case, the precision of results has to be reduced which can also be done with the help of traits. Custom modify traits have been implemented to change the integer or fraction part of the fixed-point data type, e.g. if we know the result of operation $y = a \cdot x$ with resulting data type `ap_ufixed<W,I>`, with $W = W_a + W_x$ and $I = I_a + I_x$, to be bounded to integer width I_y, we can remove $(I_a + I_x) - I_y$ unnecessary integer bits by using `hls::x_mod_traits<ap_ufixed<W,I>,` I_y`>::INT_SET_T` to modify the data type to $W - (I_a + I_x) + I_y$ and I_y.

6.1.1. Quantization Adjustment

A reduction of precision in the fraction part is a quantization which is subject to rounding. The rounding behavior can be modified for each specialized data type. The default rounding mode for `ap_(u)fixed` is rounding by truncation, `AP_TRN`, which does not need extra hardware resources. The recommended mode for numerical calculations is unbiased rounding [18], e.g. the round to even mode, `AP_RND_CONV`, which is used in the Cholesky factorization `hls::cholesky`.

6.1.2. Overflow Modes

If the result of an operation needs a larger data width than available, overflows occur. Different overflow modes can be used for each specialized data type which define how the result will be modified. The default overflow mode for `ap_(u)fixed` is wrap around, `AP_WRAP`. For an unsigned data type, adding 1 to the highest number wraps around to the smallest number, subtracting 1 from the smallest number wraps around to the highest number. For a signed data type, adding 1 to the highest number wraps around to the most negative number and vice versa.

Another very useful overflow mode is saturation, `AP_SAT`. When a result exceeds the highest representable number, it saturates at the highest number. Saturation is also used for Cholesky factorization. Furthermore, the assignment of γ_{new} and σ^2 in each iteration uses saturation to prevent overflows.

6.2. Linear Algebra Library

Vivado HLS offers a linear algebra library [15] for basic matrix computation. Although the library is of limited utility when using the `ap_(u)fixed` data type, we still use its class for complex numbers, `hls::x_complex`, the matrix multiplication function, `hls::matrix_multiply`, and a Cholesky factorization, `hls::cholesky`. Moreover, it is a good starting point for developing our own algorithms based on the techniques used in the library, especially when dealing with precision traits.

6.2.1. Complex class

The complex class `hls::x_complex<`*`type`*`>` is a container for complex number with a real and imaginary part of a common data type. Basic operations on the complex number and precision traits are also defined next to the complex class. The `ap_fixed<W,I>` data type can be used as a specialization of the complex class. However, care has to be taken when the defined operators are used, as they can not always handle binary operations with different data types. To deal with this deficiency, operands can be casted to the precision of the result before performing the operation, or a two-step operation can be used by first loading the first operand and then executing an assignment operation, e.g. multiplication assignment $*=$.

```
typedef hls::x_complex<ap_fixed<3,3> > a_t;
typedef hls::x_complex<ap_fixed<4,4> > b_t;
typedef typename hls::x_traits<a_t,b_t>::MULT_T c_t;

a_t a = a_t(3,-4);
b_t b = b_t(-8, -7);
c_t c, cast_a, cast_b;

c = a * b; // does not compile

c = a;
c *= b; // c = 4+53i

c = b;
c *= a; // c = 4+53i

cast_a = a; cast_b = b;

c = cast_a * cast_b; // c = 4+53i, ap_fixed<3+4+1,3+4+1>
```

The three correct expressions are equal and produce the same hardware results.

6.2.2. Matrix Multiplication

The linear algebra library uses two-dimensional C-style arrays of a data type to represent matrices. Matrices $\boldsymbol{A}_{M\times N}$ and $\boldsymbol{B}_{N\times O}$, where each element is a complex fixed-point number, can be defined as

```
typedef hls::x_complex<ap_fixed<WA,IA> > a_t;
typedef hls::x_complex<ap_fixed<WB,IB> > b_t;
a_t A[M][N];
b_t B[N][O];
```

where each row $\boldsymbol{a}_m$ is an array of length N. The function template `hls::matrix_multiply` can be used for matrix multiplication of a matrix $\boldsymbol{A}$ with a matrix $\boldsymbol{B}$. Similar to the multiplication of two scalars, we need to specify a data type for the result. A trait is used again to derive the resulting type, which depends on the type of $\boldsymbol{A}$, type of $\boldsymbol{B}$ and their dimensions. A transposition type is also specified for each matrix to allow normal, transpose and conjugate transpose access of the input elements. The trait provided in the library, `hls::matrix_multiply_traits`, accepts only a single input type. Therefore, the input precision has to be equal or greater than both input precisions to prevent overflow or quantization errors. A extended trait, `hls::matrix_multiply_ap_traits`, and a extended multiply function, `hls::matrix_multiply_ap`, is implemented, which respects different input types. The following example determines the data type for the elements in the resulting matrix $\boldsymbol{C} = \boldsymbol{AB}$.

```
typedef hls::matrix_multiply_ap_traits<hls::NoTranspose,
     hls::NoTranspose, M, N, N, O, a_t, b_t> mult_config_t;
typedef typename mult_config::ACCUM_T c_t;

c_t C[M][O];
```

Furthermore, the container class `MArray2d_hls` for a matrix with traits `MArray2d_hls_traits` is implemented, which keeps track of precision and dimension of a matrix. It allows to derive the data type more comfortable. The following example defines the matrix containers, their data type and finally calculates the matrix multiplication.

```
typedef MArray2d_hls<a_t, M, N> Matrix_A_t
typedef MArray2d_hls<b_t, N, O> Matrix_B_t;
typedef typename MArray2d_hls_traits<Matrix_A_t,
     Matrix_B_t>::MULT_NN_T Matrix_C_t;

Matrix_A_t A;
Matrix_B_t B;
Matrix_C_t C;

MArray2d_hls_mult_nn<Matrix_A_t, Matrix_B_t,
     Matrix_C_t>::mult(A, B, C);
```

6.2.3. Cholesky Factorization

The Cholesky factorization `hls::cholesky` from the linear algebra library is used to get the Cholesky factor $\boldsymbol{L}$ of a positive definite matrix $\boldsymbol{A} = \boldsymbol{L}\boldsymbol{L}^H$. It uses the complex version of [19, eq. (2.9.4) and (2.9.5)]

$$L_{ii} = \left(A_{ii} - \sum_{k=1}^{i} L_{ik} L_{ik}^* \right)^{1/2} \tag{6.3}$$

$$L_{ji} = \frac{1}{L_{ii}} \left(A_{ij} - \sum_{k=1}^{i} L_{ik} L_{jk}^* \right), \quad j = i+1, i+2, \ldots, N \tag{6.4}$$

where the reciprocal diagonal $1/L_{ii}$ is stored as intermediate result. A modification of `hls::cholesky` allows us to use $1/L_{ii}$ outside of the function. This is beneficial when we use the Cholesky factor $\boldsymbol{L}$ to solve the equation $\boldsymbol{L}\boldsymbol{y} = \boldsymbol{b}$ by forward substitution or $\boldsymbol{L}^H\boldsymbol{x} = \boldsymbol{y}$ by back substitution, which allows us to replace a time consuming division with a multiplication by $1/L_{ii}$. It is harder to set a precision when using (6.3). From [14, Section 4.2.6] $L_{ji}^2 \leq \sum_{k=1}^{i} L_{ik}^2 = A_{ii}$ follows if we keep the real diagonal of $\boldsymbol{A}$ in $[-1, 1]$ then real and imaginary parts $\boldsymbol{L}$ will also stay in $[-1, 1]$. Therefore, only 2 integer bits are needed. For fraction length, things are not so easy as can be seen in [20]. We just use a sufficiently wide fixed length, which leaves room for optimizations.

6.2.4. Forward Substitution

Based on a simple forward substitution algorithm $y_i = \left(b_i - \sum_{k=0}^{i-1} L_{ik} \boldsymbol{y}_k \right) / L_{ii}$, a HLS function is implemented, which makes use of the diagonal reciprocals $1/L_{ii}$ provided as input next to lower triangular $\boldsymbol{L}$ and $\boldsymbol{b}$.

6.2.5. Back Substitution

Based on a simple backsubstitution algorithm $x_i = \left(y_i - \sum_{k=i+1}^{N} R_{ik} x_k \right) / R_{ii}$, a HLS function is implemented, which makes use of the diagonal reciprocals $1/R_{ii}$ provided as input next to upper triangular $\boldsymbol{R}$ and $\boldsymbol{b}$.

6.2.6. Square Root

The square root operator, used in the Cholesky Factorization and in [1, eq. (SBL) and (SBL1)], is taken from `hls_sqrt.h`. Is uses a simple digit-by-digit algorithm to derive the square root. This leaves room for speed optimizations by using square root approximation algorithms, not further investigated in this work.

6.3. Hierarchical Design

Ultimatively, Vivado HLS transforms C/C++/SystemC design entries into Verilog/VHDL entities. It is possible to create a hierarchical HLS project with several structural modules inside. However, some techniques are hard or inefficient to model within one HLS project alone. Another reason for breaking down the problem into several HLS modules is the reduction of complexity, to reduce compilation time for single modules, and to avoid running out of memory on the workstation due to complexity scaling problems of the tool or other limits.

6.3.1. Interfaces

One module's output is the input of another module. Which interface mode is possible for a module's single port depends on the argument type, i.e. scalar, array, or reference. Some of the techniques used in this work are mentioned below.

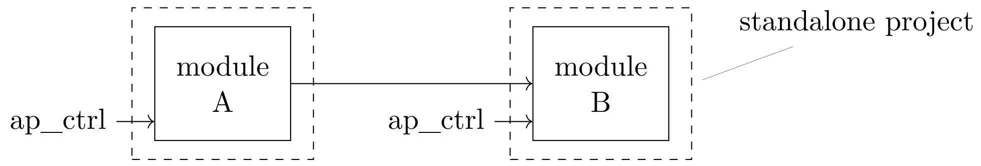


Figure 6.1.: Two standalone Vivado HLS projects

Output Reference

Module *A* can use an reference (or pointer) to set a value at an output port `a_out`. Without handshake interface (*ap_none* or *ap_stable*), the data port is valid at latest when the control signal of module *A* signals the end of operation. The default behaviour for an output reference port [15] is a one-way handshake (*ap_vld*)), i.e. a valid pulse, that the data is safe to read by module *B*. This works only if module *B* is scheduled prior to the valid pulse otherwise the pulse is never registered. A one-way handshake allows to distribute the data to one or more than one module at a time but lacks the possibility to synchronize the producer module with the consumer. An optional register can be used on the port to hold data valid until the next valid pulse arrives. Sometimes a two-way (*ap_hs*) handshake interface is the better choice when only two modules need to exchange data or synchronize. An additional acknowledges signal (*ap_ack*) completes the data transaction for a single value, i.e. producer and consumer are synchronized. The following example shows design entries of a module *A* with an output port with two-way handshake and a corresponding module *B* with the same data type and two-way handshake on an input port.

```
void module_a(a_t & a_out)
{
#pragma HLS INTERFACE ap_hs port=a_out
```

```
  a_out = do_generate_result_a();
}
```

```
void module_b(a_t a_in)
{
#pragma HLS INTERFACE ap_hs port=a_in
 do_use_result_a(a_in);
}
```

The `a_out` and `a_in` port of module *A* and *B* need to be connected in the RTL flow, e.g. by using a Vivado IP Integrator Block Design.

Return Value

The HLS module *B* can use the return value of another HLS module *A* as input when model *A* uses *ap_ctrl_hs* or *ap_ctrl_chain* as its block interface and the two modules are scheduled correctly. The return value of *A* is available as the signal `ap_return` when the control signal `ap_done` goes high.

ap_ctrl_hs For *ap_ctrl_hs*, `ap_return` becomes invalid when `ap_done` goes low, i.e. the next clock cycle, unless a register is used on the return port.

```
result_a_t module_a(...)
{
  #pragma HLS INTERFACE register port=return
  return do_generate_result_a(...);
}
```

This ensures that `ap_return` stays valid until the next rising-edge of `ap_done`.

```
void module_b(result_a_t result_a)
{
  #pragma HLS INTERFACE ap_vld register port=result_a
  do_use_result_a(result_a);
}
```

A module *B* can use the return value of *A* starting with the rising edge of `ap_done` from *A* until the next rising edge.

ap_ctrl_chain The *ap_ctrl_chain* block interface is the recommended way of chaining HLS modules [15]. When it is used, an additional `ap_continue` signal allows to postpone further operation of the block as soon as the `ap_done` signal goes high. It will stay high and `ap_return` is held valid.

```
result_a_t module_a(...)
{
  #pragma HLS INTERFACE ap_ctrl_chain port=return
  return do_generate_result_a(...);
}
```

```
void module_b(result_a_t result_a)
{
  #pragma HLS INTERFACE ap_hs register port=result_a
  do_use_result_a(result_a);
}
```

Memory

If a larger block of data is shared between two HLS modules, a dual-port memory can be used for communication. Usually, but not necessarily, one module is the producer and the other the consumer of the data. Each module needs a suitable memory interface, the memory itself needs to be instantiated, and the interfaces have to be connected. When using a Vivado IP Integrator Block Design, the block RAM (BRAM) interface is a natural choice which permits to connect the HLS modules with a memory as two edges between three nodes [15]. In Vivado HLS, a port of a module can be set as memory interface if it corresponds to an array argument of the design entry function. By knowing type and size of the array, memory address calculation is implicitly done with the array index. This also works if the array is two-dimensional or nested in a container, e.g. the `MArray2d` container for matrices. The following example reads from a BRAM through input argument `A_in` and read-writes to a different BRAM through output argument `L_out`.

```
void decompose(Matrix_A_t & A_in, Matrix_L_t & L_out )
  #pragma HLS interface bram port=A_in
  #pragma HLS resource variable=A_in core=ROM_1P
  #pragma HLS interface bram port=L_out
  #pragma HLS resource variable=L_out core=RAM_1P

  do_decompose(A_in,L_out);
}
```

The *bram* interface is set via an HLS pragma directive on each port. HLS is free to decide if it uses one or two ports of a single external BRAM. To be able to exchange data between two modules, the *resource* of the external BRAM has to be constrained to a single-ported interface, e.g. *ROM_1P* or *RAM_1P*.

Stream

Streaming data through a processing pipeline is a very efficient way of using HLS for FPGAs. A stream is unidirectional and has therefore a strict producer or consumer role. In Vivado HLS, a port can have a stream interface if access on an argument of the design entry function is sequential, exclusively-read or exclusively-write, and each value is accessed in sequence conversly to a random access memory (RAM). If those conditions are not met, either by mistake of the user or the tool, a special access container, `hls::stream`, can be used to enforce compatibility. Vivado and Vivado HLS have a good support for the AXI4-Streaming (*axis*) protocol with all kind of FIFOs, converters, and switches to use it as a streaming interconnect between various modules in a system. By using the *axis* interface on a port, two modules can be connected by a single edge between two nodes in a Vivado IP Integrator Block Design [15][3]. The following example reads from two streams and writes its sum and difference to two different output streams. It shows two different ways of using input and output arguments compatible for streaming.

```
void sum_diff(a_t a, hls::stream<b_t> & stream_b,
    hls::stream<c_t> & stream_sum, d_t & diff)
{
#pragma HLS interface axis port=arr_a
#pragma HLS interface axis port=stream_b
#pragma HLS interface axis port=stream_sum
#pragma HLS interface axis port=arr_diff

  a_t internal_a; b_t internal_b;

  internal_a = a;
  internal_b = stream_b.read();

  diff = (internal_a - internal_b);
  stream_sum.write(internal_a + internal_b);
}
```

If we need to read a single value from a stream twice, the condition for a streaming interface would be violated. Instead, we can use a stored version of the stream, e.g. implemented with a shift register, and read each value of the stream only once for storing it. If the samples are processed in bursts, reading and writing scalars does not correctly model the behavior. Therefore, arrays or pointers are used instead of the scalars.

```
void multirate_sum_diff(a_t arr_a[BURST_SIZE],
    hls::stream<b_t> & stream_b,
    hls::stream<c_t> & stream_sum, d_t *arr_diff)
{
#pragma HLS interface axis port=arr_a
```

[3]The *ap_fifo* interface is the HLS native streaming type and more lightweight than *axis*. The *axis* interface allows greater interoperability between modules. Arbitrary precision data types are sign-extended to the next byte [15], for example, which makes it easy to combine HLS modules with other *AXI4* infrastructure or when accessing data with a processor.

```
#pragma HLS interface axis port=stream_b
#pragma HLS interface axis port=stream_sum
#pragma HLS interface axis depth=BURST_SIZE port=arr_diff

  for (unsigned int idx=0; idx < BURST_SIZE; idx++) {
    a_t internal_a; b_t internal_b;

    internal_a = arr_a[idx];
    internal_b = stream_b.read();

    arr_diff[idx] = (internal_a - internal_b);
    stream_sum.write(internal_a + internal_b);
  }
}
```

Each function call in the example reads now a burst of values from the input streams. The array `arr_a` can be treated as a stream because each element is read in sequential order and only once. The streams modeled with `hls::stream` use the same argument type as in the previous single value example. Care has to be taken when using pointers because they can be used for dereferencing a scalar or an array which results in different behavior. Moreover, the depth of the `arr_diff` port can not be determined automatically and has to be specified, otherwise RTL simulation can stall[4].

6.4. Dataflow Optimization

The dataflow optimization analyzes statements in a scope and arranges them as different blocks in a pipeline according to the data dependency. The optimization works best if each block produces data on n ports for n connected consumer blocks. A corresponding DFG should show unidirectional data flow without feedback loops. Dataflow optimization is activated by placing directives at the desired scope.

```
#pragma HLS dataflow
```

It is sometimes cumbersome to get some trivial algorithms working with the dataflow optimization enabled. Loading a value from a memory A or a memory B into a register R depending on a flag, for example, is interpreted as an unsupported conditional execution. However, loading register R_A from A, R_B from B, and conditionally assigning R from R_A or R_B is permitted.

[4]It is better to model the port as `hls::stream` or as reference to an array of fixed size, i.e. `d_t (& arr_diff)[BURST_SIZE]`

6.5. ROM

If elements of an array are read-only, then the array can be implemented as a lookup-table. The array can be initialized by a classic C-style array-initializer or a separate function. The following example specializes and instantiates a sine lookup-table of length 16 and type `a_t` from a generic `rom` template.

```
template<typename T, int N>
struct rom {
  T values[N];
  static rom<T,N> init_sine(void) {
    rom<T,N> tmp;
    for (int i=0; i<N; i++) {
      double theta = ((double)i*2*M_PI)/((double)N);
      tmp.values[i] = T(sin(theta));
    }
    return tmp;
  }
};

rom<a_t, 16> sine_lut = rom<a_t, 16>::init_sine();
```

The initializer `init_sine()` is executed during the transformation step of synthesis, therefore no calculation is done in hardware. Care has to be taken when the initializer is too complex, as the ROM generation process can silently fail. However, a hardware co-simulation with a proper testbench quickly reveals a deviation from software simulation. In that case, the initializer could be simplified, for example with an intermediate step. First, a separate program generates header files or coefficient files which are already parameterized. Second, a simple initializer uses the generated ROM header files or reads the values from the coefficient file.

6.6. Reset

HLS functions can have static variables which keep their values across multiple function calls. The same is true for global variables. Static and global variables are initialized as the FPGA is programmed. Additionally, they can be reinitialized together with the inherent control logic of the function when the *reset* directive is added for a variable [15]. The control logic itself is generated with reset capability and can be set to a defined state by asserting the *ap_rst* signal. Reset polarity can be active high or low. It is desirable to explicitly set the reset polarity of the HLS project so that it is equal across multiple HLS blocks[5].

[5]The default behavior is to use active high reset. Using any *AXI* interface on any port sets reset to active low.

6.7. Pipeline Optimization

Sequential operations on an input value need one operation to finish before the next operation can be executed. If this sequence is executed again for a different input, i.e. an input independent of the previous output, then it is possible to pipeline the operations. This permits each operation to start as soon as an input is available and reduces the time the sequence can process new data, also known as *initiation interval (II)*. The hardware resources for each operation need to exist in parallel which is a natural thing when using an FPGA. HLS distinguishes between two types of pipeline optimizations, *task pipelining* and *loop pipelining* [15], with the same underlying principle. Both optimize for a specified or low II to improve the total throughput of a task (function) or loop. For a stream of data, the total latency can be reduced, whereas the latency for a single output value in the scope of the pipeline stays nearly the same[6].

6.8. Scheduling

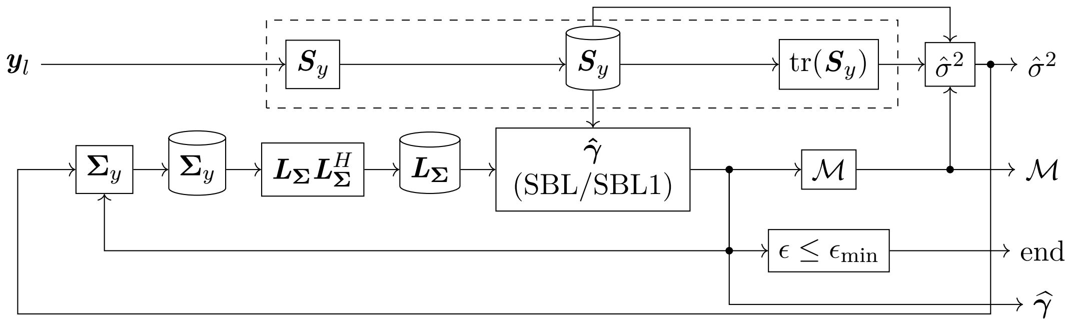
Several independent operations can be scheduled to run in parallel. This is done automatically if it is obvious for the HLS tool. For the example in 6.3.1 HLS automatically schedules reading from ports a and b on the first, calculation of $a + b$ and $a - b$ on the second, and writing to ports *sum* and *diff* on the last clock cycle.

[6]However, pipeline optimization of a loop or a function on a lower level can improve latency of a single output value at the upper level.

7. Implementation

7.1. Overview

The algorithm is implemented, consisting of submodules created with Xilinx Vivado HLS 2017.2, and combined into a complete prototype with connecting memory blocks and a custom scheduler using Xilinx Vivado 2017.2. Figure 7.1 shows the dataflow of the SBL algorithm with memory elements between HLS blocks. The FPGA design is completed with an TCP connector for a full hardware prototyping platform as shown in 4.2.



$\boldsymbol{y}_l$
$\boldsymbol{S}_y$
$\boldsymbol{S}_y$
$\mathrm{tr}(\boldsymbol{S}_y)$
$\hat{\sigma}^2$
$\hat{\sigma}^2$
$\boldsymbol{\Sigma}_y$
$\boldsymbol{\Sigma}_y$
$\boldsymbol{L}_{\boldsymbol{\Sigma}}\boldsymbol{L}_{\boldsymbol{\Sigma}}^H$
$\boldsymbol{L}_{\boldsymbol{\Sigma}}$
$\hat{\boldsymbol{\gamma}}$ (SBL/SBL1)
$\mathcal{M}$
$\mathcal{M}$
$\epsilon \leq \epsilon_{\min}$
end
$\hat{\boldsymbol{\gamma}}$

Figure 7.1.: Dataflow including memories between HLS blocks. Input: $\boldsymbol{y}_l$, outputs: noise variance $\hat{\sigma}^2$, active set $\mathcal{M}$, variance $\hat{\boldsymbol{\gamma}}$, convergence criteria $\epsilon \leq \epsilon_{\min}$. The blocks in the dashed rectangle are evaluated only once for a single multiple measurement vector (MMV).

7.2. Simplifications

The number of snapshots (L) is a power of two $L = 2^{L_b}$. As L appears only as a divisor $y = x/L$, this permits a division of a fixed point value x to be transformed into a bit-shift by L_b positions and $y = x \cdot 2^{-L_b}$. If L_b is known during compilation it is a mere reinterpretation of the bit vector with no extra operation involved[1]. Number of sensors (N) and directions of arrival (M) are a power of 2 for architectural optimization reasons. This can be advantageous, e.g. for efficient memory usage due to address layout of memory blocks. The transfer matrix $\boldsymbol{A}$ is pre-calculated during compilation

[1]Changing L only effects $\boldsymbol{S}_y$. It would be possible to allow a dynamic number of snapshots by setting L to a maximum value $L = 2^{L_{b,\max}}$ with $L_b \in \{1, 2, \dots, L_{b,\max}\}$ and put a configurable bit-shifter on the read port of the $\boldsymbol{S}_y$ memory.

Parameter	Name	Value
M	DOA	512
N	sensors	16
L	snapshots	64
K	sources	3
f_{clk}	clock frequency	150 MHz

Table 7.1.: Configuration of the FPGA implementation.

and implemented as a ROM in hardware. It is instantiated for each HLS module which uses $\boldsymbol{A}$ to avoid the need of bus arbitration and to minimize the access time.

7.3. Configuration

The FPGA prototype uses a configuration which is derived from the example given in [1] and modified to fulfill the previous described simplifications.

The input to the system are quantized 16 bit complex values sampled from a sensor array.

7.4. SBL Scheduler

The building blocks of the implemented system for SBL for DOA estimation have independent control signals which need to be asserted in the correct order for the intended application.

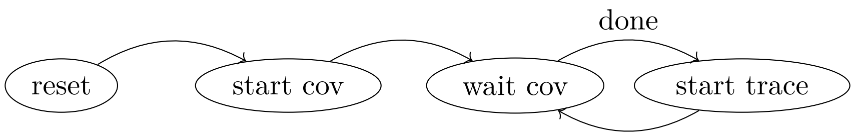
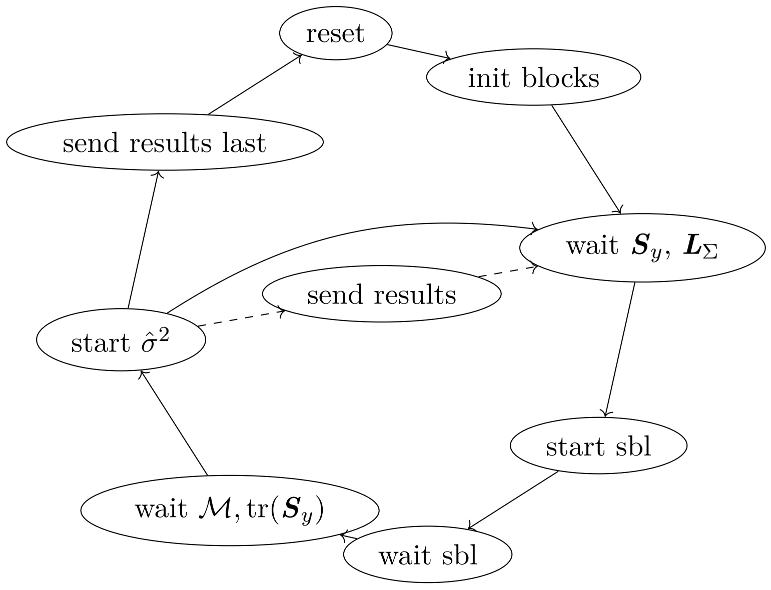


Figure 7.2.: System finite-state machine

Figure 7.2 shows the finite-state machine (FSM), which schedules calculation of the covariance matrix $\boldsymbol{S}_y$ first, waits for the results to be *done*, schedules calculation of intermediate results which are valid for a multisnapshot, i.e. trace $\text{tr}(\boldsymbol{S}_y)$ (SBL1 and SBL) and Cholesky factorization of $\boldsymbol{S}_y$ (SBL). The memory for $\boldsymbol{S}_y$ is organized as a ping-pong buffer to allow pipelined calculation of the covariance matrix for the next multi-snapshot.

Each multisnapshot needs multiple iterations for calculating the estimates. A second FSM, figure 7.3, takes care of the correct control flow between the components of the SBL algorithm [1, Table 1]. A block can be scheduled as soon as its preconditions are met, i.e. data is available.

Figure 7.4 shows the relationship between schedule times of the blocks.



reset
init blocks
send results last
wait $\boldsymbol{S}_y$, $\boldsymbol{L}_\Sigma$
send results
start $\hat{\sigma}^2$
start sbl
wait $\mathcal{M}, \mathrm{tr}(\boldsymbol{S}_y)$
wait sbl

Figure 7.3.: Block finite-state machine.

7.5. Bit-width

A hardware implementation needs to have known word lengths during compilation for each input, output, memory and register. This is necessary in order to allocate and bind hardware resources for each storage element and operation. As described in 6.1, most operations using fixed-point numbers lead to a higher word length of the result compared to the word lengths of the operands. The concatenation of operations would quickly lead to unfeasible high resource requirements. Certain trade-offs concerning precision have to be made in order to meet resource constraints.

The fixed-point implementation of the SBL DOA estimator assumes input data from the sensor array $\boldsymbol{Y}$ as 16 bit word length complex, for real and imaginary part each, and 2 bit integer length[2] to allow signed values in $[-1, 1]$. The calculated outputs are $\hat{\boldsymbol{\gamma}}$ with 48 bit word length and 18 bit integer length, $\hat{\sigma}^2$ with 32 bit word length and 2 bit integer length, and three indices of the identified peaks as 16 bit unsigned integer.

The transfer matrix $\boldsymbol{A}$ is usually normalized by $1/\sqrt{M}$ to simplify analysis due to $|\boldsymbol{a}|_2 = 1$ [3]. The normalization leads to a reduction of the integer length, hence for $\boldsymbol{A}$ we use a 16 bit word length and $(2 - \log_2 M/2)$bit integer length.

$\boldsymbol{\Sigma}_y$ is set to the same precision as $\hat{\sigma}^2$ due to $\boldsymbol{A\Gamma A}^H$ with absolute value of its elements $|\boldsymbol{a}_l^H \boldsymbol{\Gamma} \boldsymbol{a}_i| \leq \|\boldsymbol{a}_l\|_2 \|\boldsymbol{\Gamma} \boldsymbol{a}_i\|_2 \leq \max_m \hat{\gamma}_m$. However, $\|\boldsymbol{\Gamma} \boldsymbol{a}_i\|_2 \leq 1$ is expected as $\boldsymbol{\Gamma}$ is sparse.

The Cholesky factors $\boldsymbol{L}_\Sigma$, $\boldsymbol{R}_{\boldsymbol{S}_y}$, and $\boldsymbol{R}_\mathcal{M}$ need a 2 bit integer length as described in 6.2.3, a 48 bit word length was used for $\boldsymbol{L}_\Sigma$, $\boldsymbol{R}_{\boldsymbol{S}_y}$, 32 bit $\boldsymbol{R}_\mathcal{M}$, which was sufficient to get good results.

[2] 2 bit integer length is used because otherwise +1 would not be representable with 1 bit integer length in `ap_fixed<W,1>`, cf. section 6.1

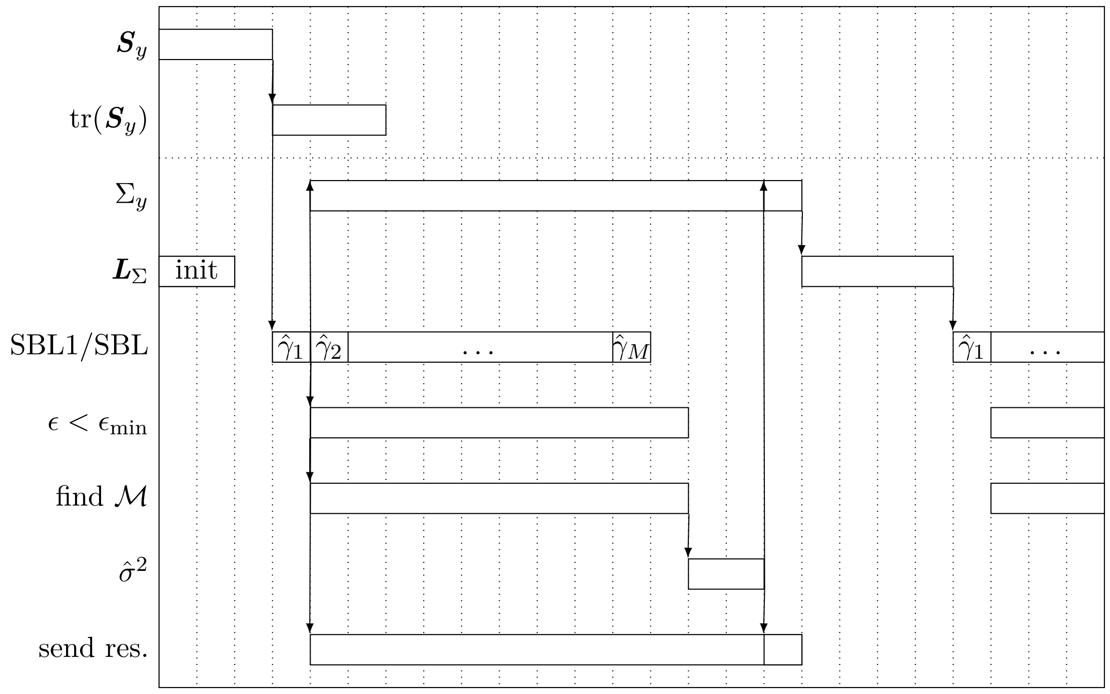


Figure 7.4.: Concurrency of HLS blocks of DOA estimator.

The sample covariance matrix $\boldsymbol{S}_y$ is derived from the multisnapshot $\boldsymbol{Y}$ which is in the interval $[-1, 1]$. It is therefore sufficient to reduce the integer length to 2 bit.

The result of forward substitution $\tilde{\boldsymbol{a}}_m$ was set to 32 bit word length with 4 bit integer length. However, the result of the subsequent back substitution $\boldsymbol{b}_m$ needed a wider integer length as numerical problems during simulation have shown. $\boldsymbol{b}_m$ was set to 48 bit word length and 20 bit integer length.

7.6. FPGA Resources

The Xilinx KC705 evaluation board is used for the implementation of the SBL DOA estimator. It features a *Xilinx 7 Series* Kintex-7 XC7K325T FPGA and peripheral components, of which only an ethernet PHY and a DDR3 SO-DIMM are utilized for interfacing the prototype. There is no strict dependence on the KC705. Therefore, it can be easily replaced by any other off-the-shelf board with a comparable FPGA, ethernet connection and memory. The most important resources of the FPGA for DSP algorithms are dedicated blocks of multipliers with additional specialized logic called *DSP Slices*, or *DSP48E1 Slices* for Xilinx 7 Series FPGAs. The XC7K325T has 840 DSP48E1 Slices.

Table 7.2 shows the used resources of each HLS block.

Figure 7.5 shows the FPGA resource occupation for the SBL1 variant with highlighted HLS blocks. Blocks with a high multiplier count distribute themselves along the *DSP*

	Slice LUTs	Slice Registers	DSP Slices	BRAM (18Kb)
$\boldsymbol{S}_y$	266	289	4	
$\boldsymbol{L}_\Sigma$	4259	4765	104	12
ϵ	497	753	3	
filter	516	552		
$\mathcal{M}$	682	796		
γ (SBL1)	7232	8285	284	16
γ (SBL)	13970	14624	476	32
σ^2	6878	7669	164	20
$\boldsymbol{\Sigma}_y$	438	979	22	16
$\mathrm{tr}(\boldsymbol{S}_y)$	54	94		
Mem_γ				4
Mem_{S_y}				24
Mem_{Σ_y}				4
Mem_{L_Σ}				14
total SBL1	20822	24182	581	110
total SBL	27560	30521	773	126
total available	203800	407600	840	890

Table 7.2.: FPGA resources of individual HLS blocks

slices columns present on the chip. The high DSP utilization of the $\hat{\gamma}$ calculating block (SBL1) is distributed over a large chip area. The memory elements used to exchange random access data between HLS blocks are not shown. They occupy BRAM resources, a on-chip memory with dual ports.

As HLS enables easier exploration of the design space, we modify the parameters M and N for the resource intensive SBL1 block to analyze an impact on the hardware resource count. Figure 7.6b shows the resources for several design parameter pairs (M, N). Occupation of look-up tables (LUTs) and registers remain at a similar level. BRAM usage is increasing for greater M and N due to larger ROMs for the steering matrix $\boldsymbol{A}$ and higher memory usage for intermediate results. The number of DSP slices is not shown but does not change at all.

7.7. Latency

The latency of a block depends on the number of control step (CS), each CS's latency, and how loops are placed to repeat some of the steps. HLS generates C synthesis reports with minimum and maximum latency of a block and each sub-module. The difference between minimum and maximum arises from the fact that loops can have variable bounds which lead to a variable number of iterations. Even tough some nested loops have fixed bounds, e.g. iterating over the lower triangle elements of a two dimensional array, sometimes they

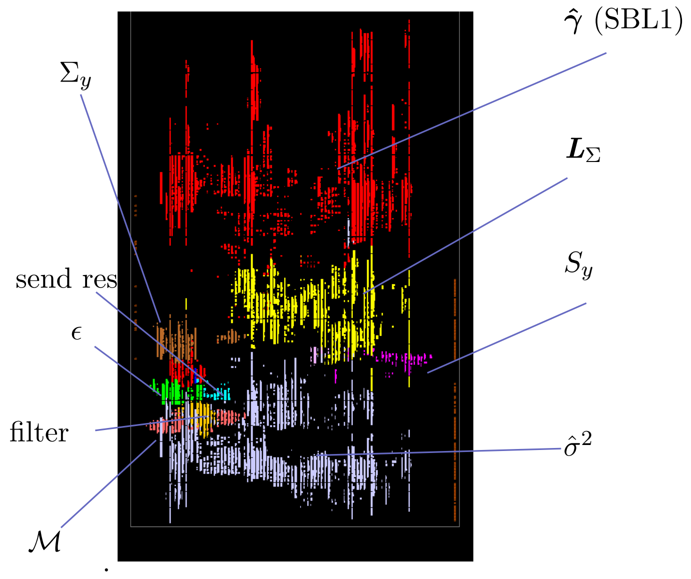


Figure 7.5.: FPGA device occupation

cannot be resolved by the HLS tool as such[3]. Therefore it is more convenient to take latency results from an RTL simulation or hardware co-simulation. It is important to note that the latency of each block also depends on the target clock frequency. This is because HLS is scheduling and pipelining the operations of a block depending on a given timing constraint, i.e. the clock period, to meet the technology limit. Table 7.3 shows the total or added latency of each HLS block in clock cycles.

Some blocks are scheduled together with other blocks. Hence, those don't contribute directly to the total latency of a single iteration but only add a small latency to the total latency as shown in Figure 7.4. During the first iteration of a multisnapshot, the blocks SBL1 and SBL make use of pre-calculated $\boldsymbol{L}_\Sigma$ from initial values of $\hat{\sigma}^2 = \sigma_0^2$ and $\hat{\boldsymbol{\gamma}} = \boldsymbol{\gamma}_0$. Thus the latency of the first iteration is smaller than those of the following iterations of the same multisnapshot.

Latency differentiation between first iteration, following iteration, SBL1 and SBL can be neglected due to the small differences compared to the high number of cycles needed in this implementation. At 150 MHz, approximately 43 iterations/s are possible compared to 24 iterations/s for the floating-point reference implementation in MATLAB and 30 iterations/min for the HLS C simulation[4] using a *Intel Xeon CPU E5-2690 v3* at 2.60 GHz. The bottleneck is clearly the slow calculation of 512 different $\hat{\gamma}_m$ in each iteration j, which has great potential for improving performance. For hard real-time requirements on the input of the system, a maximum number of iterations j_{max} has to be considered which constrains the rate $R_{\boldsymbol{Y}}$ of producing a new sample covariance

[3]It was not possible for Vivado HLS 2017.2 to calculate the exact latency of iterating over all rows in a loop 1 and over all columns $\leq$ row in nested loop 2. For small loops, an indexing ROM can help to convert 2 dimensional deterministic indexing to 1 dimensional linear indexing.

[4]Only a single CPU thread is used for C simulation.

operation	SBL1		SBL	
	cycles	%	cycles	%
$\boldsymbol{S}_y$	62 145	1.8	62 145	1.8
$\mathrm{tr}(\boldsymbol{S}_y)$	35	<1	35	<1
$\boldsymbol{L}_\Sigma$	8731	<1	8731	<1
$\hat{\gamma}$	3 484 161	99.6	3 471 870 (4 433 042)	99.6
find $\mathcal{M}$	$\hat{\gamma}_M$ + 49	<1	$\hat{\gamma}_M$ + 49	<1
$\epsilon < \epsilon_{min}$	$\hat{\gamma}_M$ + 3	<1	$\hat{\gamma}_M$ + 3	<1
$\hat{\sigma}^2$	5987	<1	5987	<1
$\boldsymbol{\Sigma}_{\boldsymbol{y}}$	$\hat{\sigma}^2$ + 33	<1	$\hat{\sigma}^2$ + 33	<1
send res.	$\hat{\sigma}^2$ + 3	<1	$\hat{\sigma}^2$ + 3	<1
1. iteration	3 552 342		4 501 223	
iteration n_{iter}	3 498 961	100	3 486 670	100

Table 7.3.: Latency of each HLS block.

matrix $\boldsymbol{S}_y$.

$$R_{\boldsymbol{Y},\max}(j_{\max}) \leq \frac{1}{T_{j_{\max}}} = \frac{f_{\text{clk}}}{n_{\text{iter}} \cdot j_{max}}$$

The calculation of $\boldsymbol{S}_y$ reduces data from $N \times L$ to $N(N+1)/2$ which is process parallel to the iterations of the SBL algorithm. Therefore, by increasing the number of snapshots L of the MMV, the maximum rate of the SMV, $R_{\boldsymbol{y},\max}$, is also increased.

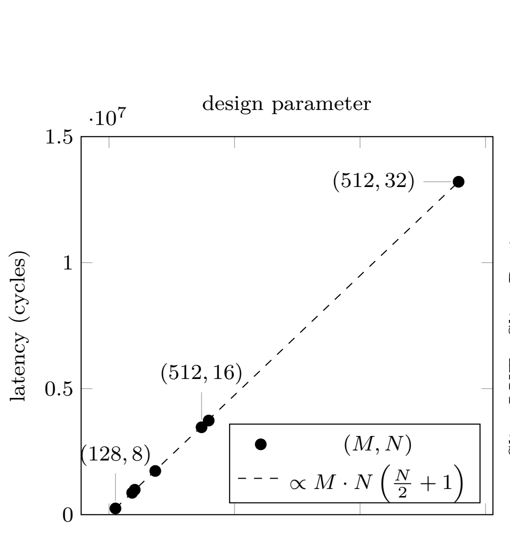
$$R_{\boldsymbol{y},\max}(j_{\max}) = L \cdot R_{\boldsymbol{Y},\max}(j_{\max}) \leq \frac{L \cdot f_{\text{clk}}}{n_{\text{iter}} \cdot j_{max}}$$

For $j_{max} = 100$ and $L = 64$, the maximum rate $R_{\boldsymbol{y},\max}(100) = 27.4$ SMV per second.

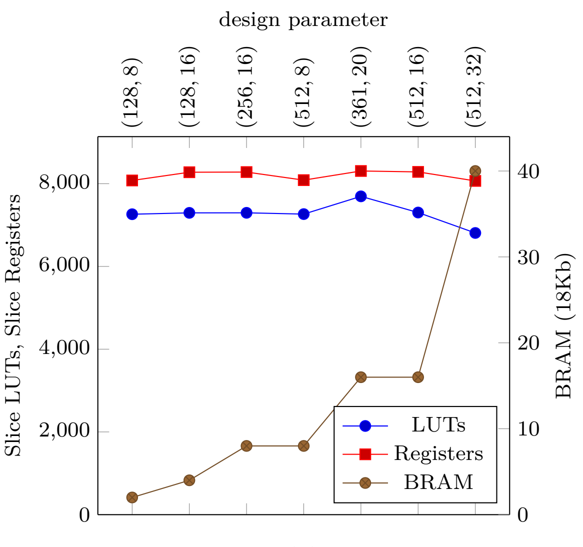
Figure 7.6a shows the impact in latency for the HLS block SBL1 when changing the design parameters M and N. For this implementation, all pairs (M, N) are approximately on a line where $n_{\text{SBL1}} \propto M \cdot N\,(N/2 + 1)$. This relation indicates sequential processing within the module. Clearly, the loop for calculating the M different $\hat{\gamma}_m$ could be processed (partially) in parallel to reduce latency. Furthermore, finer pipeline optimization could be applied within the module to reduce the II between $\hat{\gamma}_m$ and $\hat{\gamma}_{m+1}$. However, this has to be done with a trade-off between latency and resources. Finally, further reduction in latency of a single $\hat{\gamma}_m$ could be achieved by reducing precision through approximations or smaller word length.

7.8. Power Estimation

The implementation of the algorithm is estimated to consume less than 1.5 W when operated at a clock frequency of 150 MHz on the Kintex-7 XC7K325T FPGA, according



(a)



(b)

Figure 7.6.: (a) Latency and (b) resources for the SBL1 block depending on design parameter (M, N). The number of DSP slices stays at 284.

to the Xilinx Vivado Power Report.

8. Simulation Results

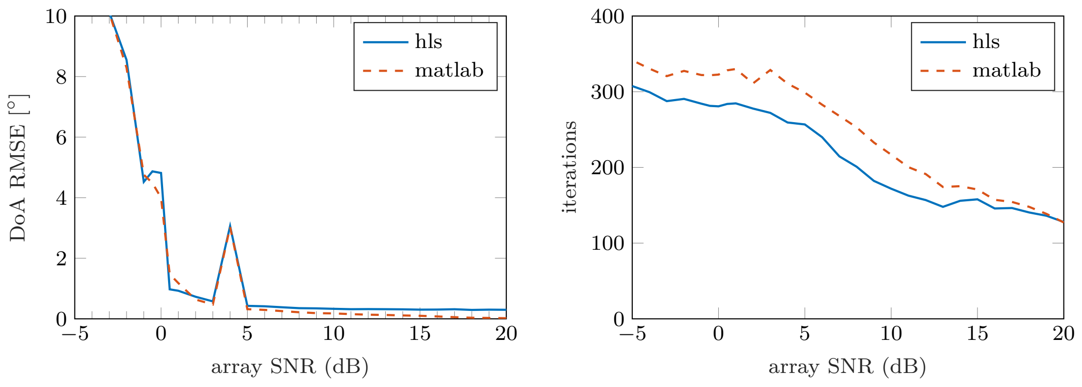
The fixed-point prototype of the algorithm [1] implemented with Vivado HLS, cf. Chapter 7, is compared with a reference floating-point implementation in MATLAB. A Xilinx Kintex 7 FPGA board is configured with the prototype and connected as DUT to the MATLAB environment via TCP/IP as shown in Figure 4.2. Although a C simulation of the prototype leads to identical results in terms of accuracy to the implemented design on an FPGA, the FPGA design is much faster, thus more suitable for long running measurements.

8.1. Settings

Based on the example scenario of [1], $K = 3$ independent sources are placed on an angular grid in the interval $\theta \in [-90, 90]^\circ$ with $M = 512$ different DOA positions. The sources, with magnitudes 12 dB, 22 dB, and 20 dB, are placed on the nearest grid-point with respect to the angles $-3°$, $2°$, and $75°$. All three sources together are observed on an equally spaced sensor array of $N = 16$ sensors with an array SNR $= 20 \log_{10} {}^{\|\boldsymbol{A}\boldsymbol{x}_l\|_2}/_{n_\text{rel}}$ where n_rel is the relative noise level of the additive i.i.d. complex Gaussian noise. Multiple snapshots $L = 64$ are concatenated to a single multisnapshot $\boldsymbol{Y}$ which serves as the input argument to the system. The sources are stationary across snapshots. The transfer matrix $\boldsymbol{A}$, with elements $\boldsymbol{a}_{n,m} = \frac{1}{\sqrt{M}} \exp\left(-j(n-1)\pi \sin\left({}^{\pi(m-1)}/_{(M-1)} - {}^{\pi}/_{2}\right)\right)$, is known in advance to the system and does not change between snapshots. Initial values for the algorithm are $\sigma_0^2 = 0.1$ and $\boldsymbol{\gamma}_0 = \boldsymbol{1}$. As another difference to [1], $\boldsymbol{Y}$ is scaled so that $\text{Re}(\boldsymbol{Y}) \in [-1, 1]$ and $\text{Im}(\boldsymbol{Y}) \in [-1, 1]$ according to the used data format of the implementation. This scaling is applied equally to the fixed-point implementation in Vivado HLS and the floating-point reference implementation in MATLAB. This has implications to the initial values of $\hat{\boldsymbol{\gamma}}$ and $\hat{\sigma}^2$ which are not further investigated in this work.

8.2. Results for SBL1

A Monte Carlo simulation with $J = 100$ realizations is carried out for the SBL1 variant. The root mean squared error (RMSE) of the DOA in Figure 8.1a shows good spatial resolution for array SNR ≥ 1 dB for the fixed-point prototype implemented with Vivado HLS with little difference to the floating-point implementation in MATLAB. The RMSE surge at 1 dB for this configuration is at higher SNR compared to -1 dB in [1, Fig. 1c] due to differences in the settings. The mean number of iterations for reaching the stop



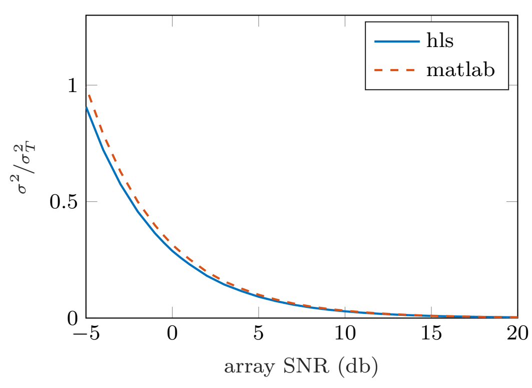
(a) (b)

Figure 8.1.: (a) RMSE of DOA and (b) mean iterations at stop criteria $\epsilon \leq \epsilon_{\text{min}}$ depending on SNR

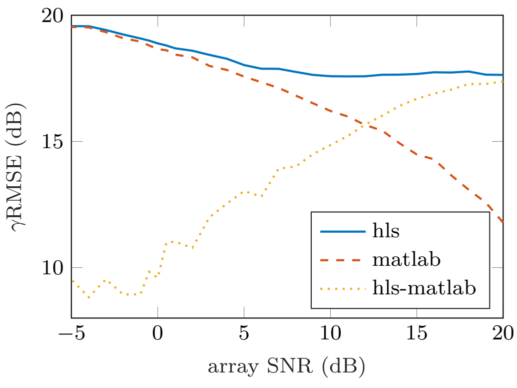
criterion $\epsilon \leq \epsilon_{\text{min}} = 0.001$ is shown in Figure 8.1b. ϵ_{min} is reached with fewer iterations the higher the array SNR, likewise for HLS and MATLAB.

The estimated noise power $\hat{\sigma}^2$, as shown in Figure 8.2a, is underestimated as already described in [1]. Figure 8.2b shows the RMSE of estimated $\hat{\gamma}$ for the fixed-point and floating-point implementation. The deviation of the fixed-point implementation compared to the floating-point reference is higher for higher array SNR.

Figure 8.3 shows the spatial spectrum of estimated $\hat{\gamma}$ for different array SNR. The three sources are well resolved by the detected peaks in $\hat{\gamma}$ for array SNR above 1 dB. Below 1 dB, falsely detected peaks become more likely and lead to a higher DOA RMSE.



(a)



(b)

Figure 8.2.: (a) Estimated $\hat{\sigma}^2$ and (b) RMSE of estimated $\hat{\gamma}$ at stop criteria $\epsilon \leq \epsilon_{\text{min}}$ depending on SNR. RMSE between HLS and MATLAB implementation is shown as dotted line.

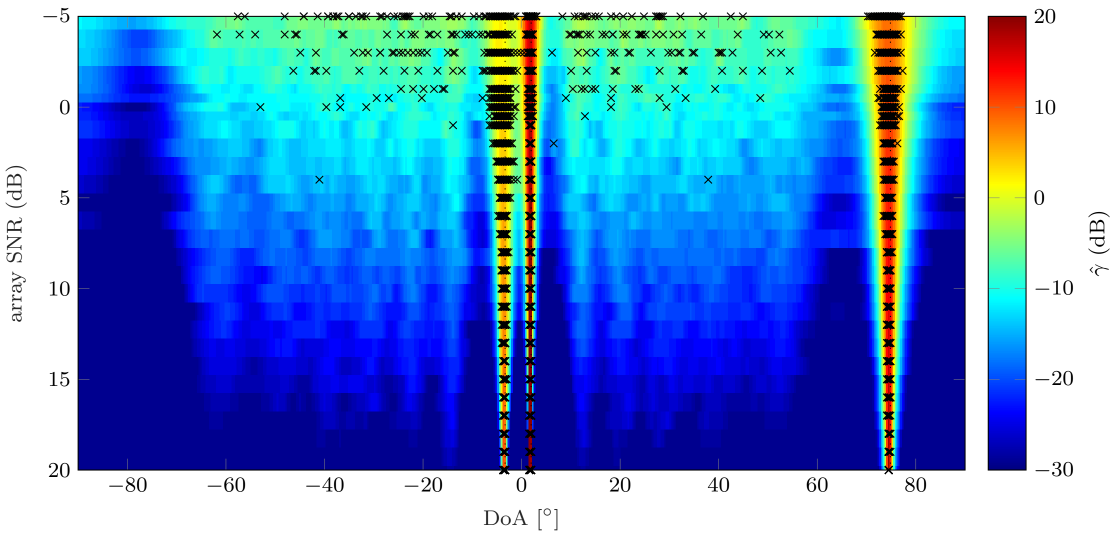


Figure 8.3.: DOA estimation of the sources. The detected peaks are marked. The dashed lines show the sources of the signal model.

8.2.1. Moving 3rd source

The same setting as before is used again with sources at $-3°$ and $2°$ but the 3rd source is now swept over the range of $\theta \in [-90°, 90°]$. Here, $J = 10$ realizations are carried out.

Figure 8.6 shows the region where the 3rd source is close to the other 2 sources at different array SNR. Near both ends of the spatial spectrum, the sensor array loses the ability to resolve sources, as shown in Figure 8.4 for $-90°$. Therefore, the 3rd source needs to keep distanced from the ends of the spatial spectrum, i.e. $-90°$ and $90°$, and from the sources at $-3°$ and $2°$. Figure 8.5 shows the limits for the DOA estimator to resolve all three sources correctly up to a DOA RMSE of $\leq 1°$.

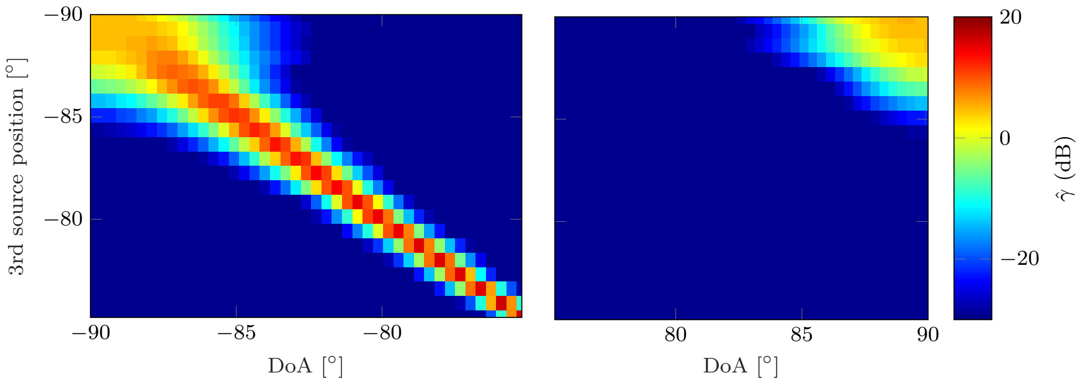


Figure 8.4.: Estimated $\hat{\gamma}$ for 3rd source positioned near one end of spatial spectrum.

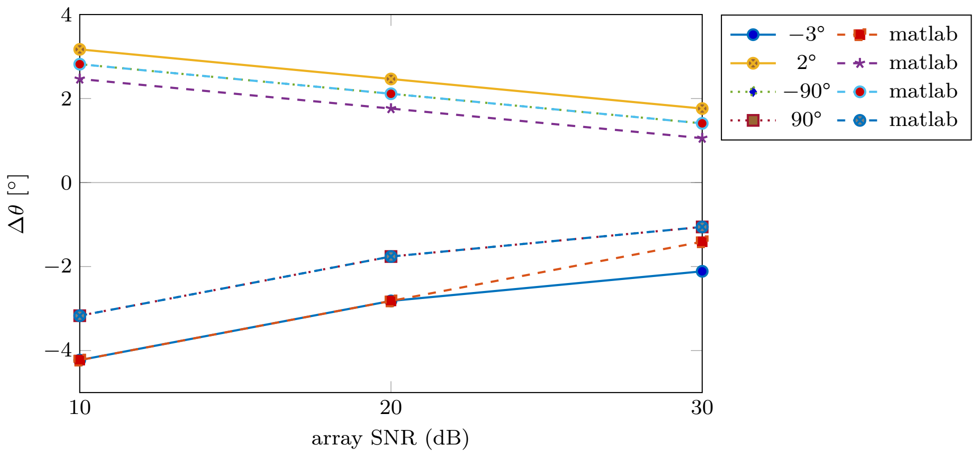
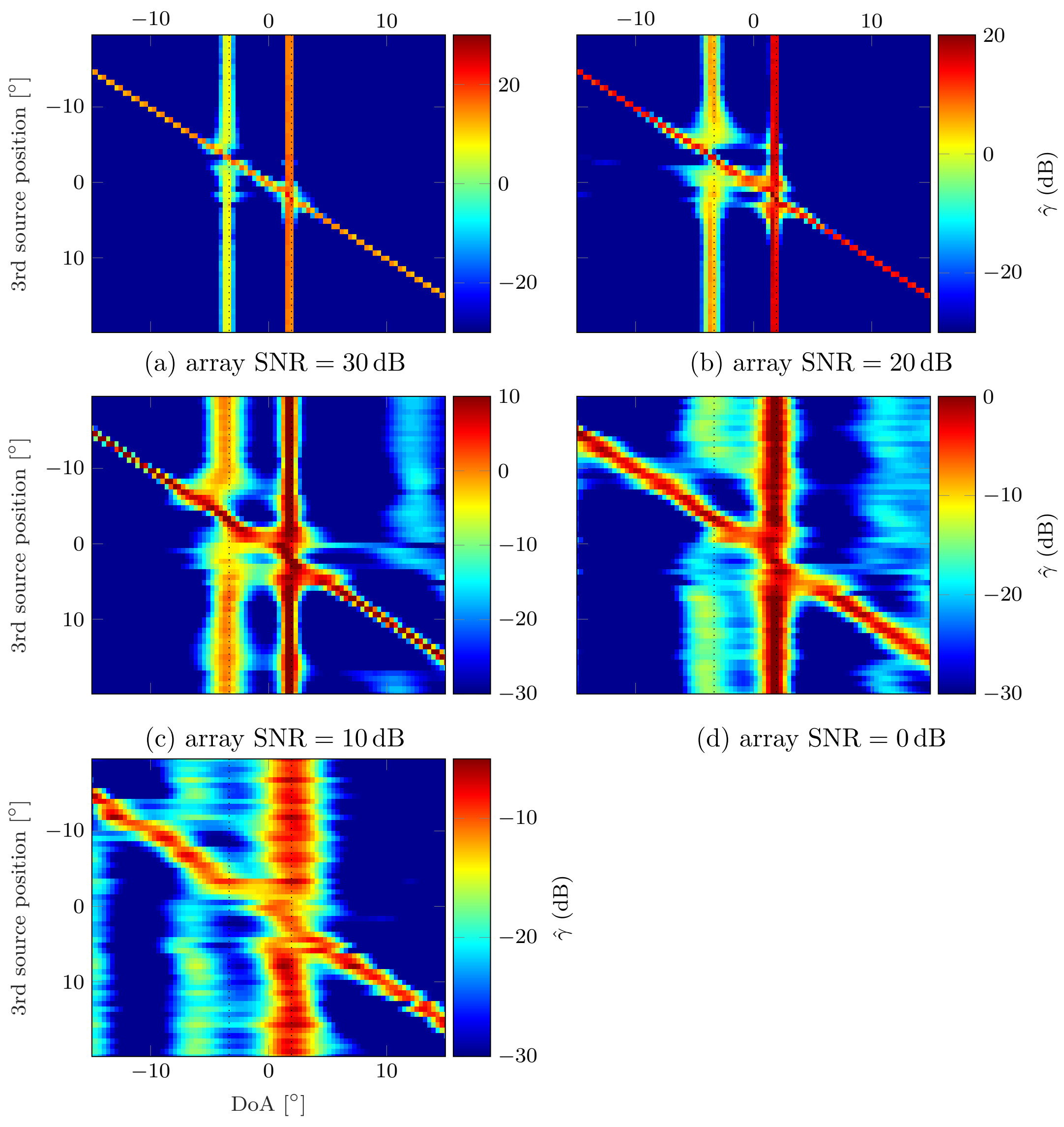


Figure 8.5.: Minimum relative angle between 3rd source and $-90°/90°/-3°/2°$ for DOA RMSE $\leq 1°$



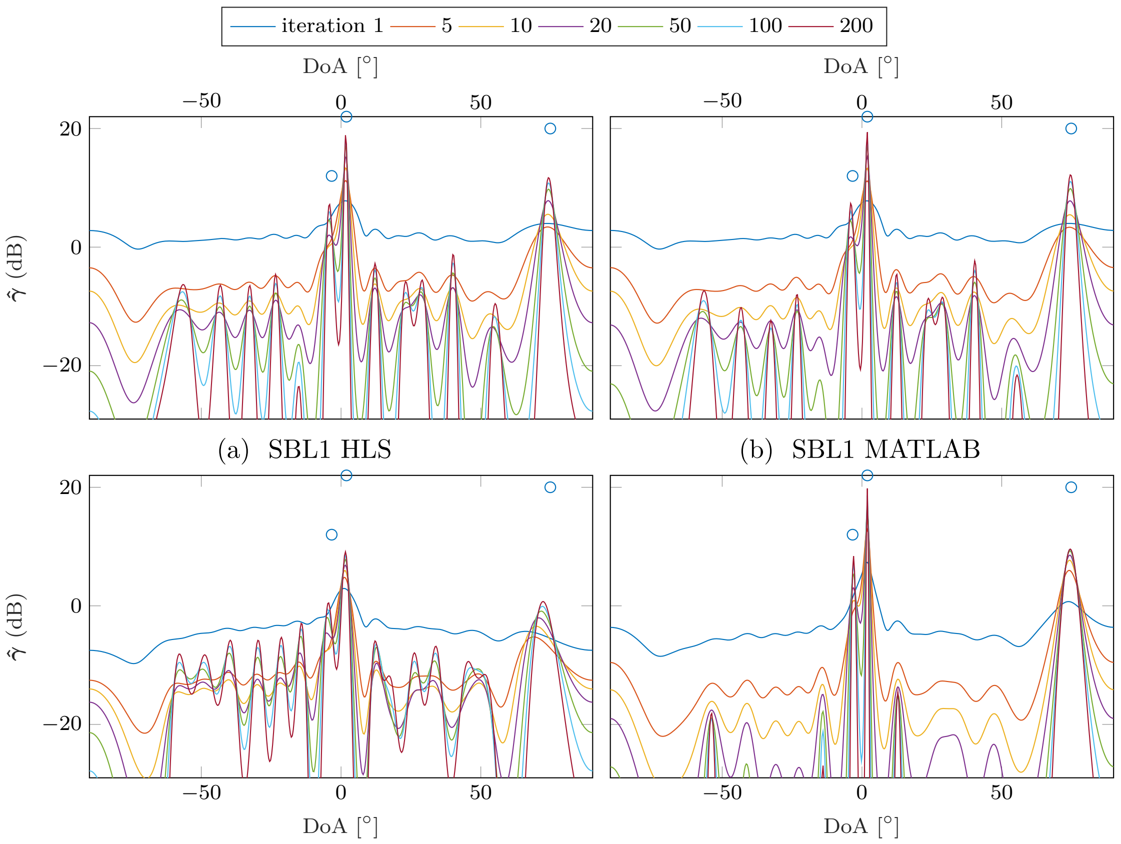
(e) array SNR = −5 dB

Figure 8.6.: Estimated $\hat{\gamma}$ of SBL for different array SNR. 1st and 2nd source are fixed, dotted lines. Position of 3rd source is varied.

8.3. Convergence

The convergence of the estimated parameter $\hat{\gamma}$ for array SNR of 2 dB is shown in Figure 8.7. Sparse solutions, i.e. peaks in $\hat{\gamma}$, are promoted by the SBL algorithm. The peaks in the fixed-point implementation with HLS of SBL and SBL1 remain at a higher level compared to the floating-point with MATLAB. This could be due to the limited dynamic range of fixed-point numbers. Furthermore, SBL shows a higher rate of change for the first iterations compared to SBL1.

The convergence rate ϵ in Figure 8.8 shows the relative change of $\hat{\gamma}$ for each iteration. The convergence for the fixed-point implementations of SBL and SBL1 approximately follows the floating-point implementations.



(a) SBL1 HLS (b) SBL1 MATLAB

(c) SBL HLS (d) SBL MATLAB

Figure 8.7.: Convergence of parameter $\hat{\gamma}$ with SBL1 and SBL implemented with HLS and MATLAB. The source positions are marked with (○). Array SNR = 2 dB.

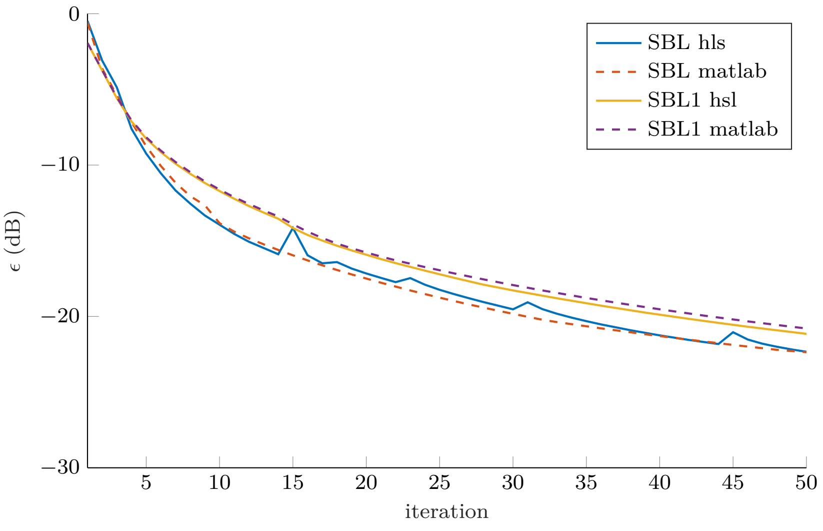


Figure 8.8.: Convergence of ϵ towards $\epsilon_{\text{min}} = -30\,\text{dB}$. Comparison between SBL1 and SBL implemented with HLS and MATLAB. Array SNR $= 2\,\text{dB}$

9. Conclusion

The sparse Bayesian learning algorithm for directions of arrival estimation of [1] is suitable for a fixed-point implementation on an FPGA. Several signal processing steps benefit from the datapath paradigm and parallelism inherent to the architecture of FPGAs. The presented prototype reaches good agreement with the reference floating-point implementation in terms of detection quality, measured in DOA root mean squared error (RMSE), and convergence rate at nearly twice the speed.

Developing the prototype on the algorithmic level using HLS reduces time and effort of transforming the algorithm to a synthesizable hardware description. The division into multiple independent HLS blocks decreases complexity further and increases testability. The high-level modeling facilitates easier modification of parameters for design space exploration.

Using fixed-point throughout the processing chain is feasible for SBL, although for some cases it is difficult to set appropriate word and integer length. Verification with high-level languages is essential for evaluating results. The HLS prototype can easily be enhanced with verification frameworks for C simulation, or a connector can be attached to the DUT and perform platform independent verification.

Outlook The latency introduced by each iteration of the algorithm is still quite high. Some arithmetic operations with large latency could possibly be replaced by approximated versions, e.g. $1/x$ or $1/\sqrt{x}$. Moreover, pipeline optimization needs to be reviewed for some modules to improve dataflow.

To reduce costs, resource sharing can be explored to save hardware resources and target smaller FPGA devices. Although this can be achieved by HLS at the operator level, for bigger functional units it must be considered on the algorithmic level. However, variable word length of fixed-point data types often prevent reuse of operators. The word lengths of the fixed-point data types can be reviewed for more detailed error analysis to reduce resources.

Furthermore, optimizations for special sensor arrays arrangements can be considered to get more compact descriptions of the transfer matrix $\boldsymbol{A}$ and save hardware resources.

In the future, this implementation will be tested further on real-world data acquired by a 60 GHz channel sounder built at the Institute of Telecommunications.

A. Appendix

A.1. SBL1 and SBL – Numerator

The common numerator, mentioned in section 5.4, for updating γ_m^{new} can be developed as follows

$$\frac{\| \boldsymbol{Y}^H \boldsymbol{\Sigma}_y^{-1} \boldsymbol{a}_m \|_2}{\sqrt{L}} = \sqrt{\frac{(\boldsymbol{Y}^H \boldsymbol{\Sigma}_y^{-1} \boldsymbol{a}_m)^H \boldsymbol{Y}^H \boldsymbol{\Sigma}_y^{-1} \boldsymbol{a}_m}{L}} = \sqrt{\frac{(\boldsymbol{\Sigma}_y^{-1} \boldsymbol{a}_m)^H \boldsymbol{Y} \boldsymbol{Y}^H \boldsymbol{\Sigma}_y^{-1} \boldsymbol{a}_m}{L}}$$
$$= \sqrt{\frac{\boldsymbol{b}_m^H \boldsymbol{Y} \boldsymbol{Y}^H \boldsymbol{b}_m}{L}} = \sqrt{\boldsymbol{b}_m^H \boldsymbol{S}_y \boldsymbol{b}_m}.$$

By expanding the inner product $\boldsymbol{b}_m^H (\boldsymbol{S}_y \boldsymbol{b}_m)$ further we get

$$\begin{aligned}
\boldsymbol{b}_m^H \boldsymbol{S}_y \boldsymbol{b}_m = & \begin{pmatrix} b_{m,1}^* & b_{m,2}^* & \cdots & b_{m,N}^* \end{pmatrix} \boldsymbol{S}_y \begin{pmatrix} b_{m,1} \\ b_{m,2} \\ \cdots \\ b_{m,N} \end{pmatrix} \\
= & b_{m,1} b_{m,1}^* [\boldsymbol{S}_y]_{11} + b_{m,1} b_{m,2}^* [\boldsymbol{S}_y]_{12}^* + b_{m,1} b_{m,3}^* [\boldsymbol{S}_y]_{13}^* + \cdots + b_{m,1} b_{m,N}^* [\boldsymbol{S}_y]_{1N}^* \\
& + b_{m,2} b_{m,1}^* [\boldsymbol{S}_y]_{12} + b_{m,2} b_{m,2}^* [\boldsymbol{S}_y]_{22} + b_{m,2} b_{m,3}^* [\boldsymbol{S}_y]_{23}^* + \cdots + b_{m,1} b_{m,N}^* [\boldsymbol{S}_y]_{3N}^* \\
& + \dots \\
& + b_{m,N} b_{m,1}^* [\boldsymbol{S}_y]_{1N} + b_{m,N} b_{m,2}^* [\boldsymbol{S}_y]_{2N} + \cdots + b_{m,N} b_{m,N}^* [\boldsymbol{S}_y]_{NN} \\
= & \sum_{n=1}^{N} \left[b_{m,n} b_{m,n}^* [\boldsymbol{S}_y]_{nn} + \sum_{i=n+1}^{N} \left(b_{m,n} b_{m,i}^* [\boldsymbol{S}_y]_{ni}^* + b_{m,i} b_{m,n}^* [\boldsymbol{S}_y]_{ni} \right) \right] \\
= & \sum_{n=1}^{N} \left[b_{m,n} b_{m,n}^* [\boldsymbol{S}_y]_{nn} + 2 \sum_{i=n+1}^{N} \operatorname{Re} \left(b_{m,i} b_{m,n}^* [\boldsymbol{S}_y]_{ni} \right) \right]
\end{aligned}$$

Further simplifications can be achived with

$$b_{m,n} b_{m,n}^* \operatorname{Re}([\boldsymbol{S}_y]_{nn}) = [\boldsymbol{S}_y]_{nn} \left(\operatorname{Re}(b_{m,n})^2 + \operatorname{Im}(b_{m,n})^2 \right),$$

with real diagonal elements $[\boldsymbol{S}_y]_{nn} = \mathrm{Re}([\boldsymbol{S}_y]_{nn})$, and expansion of

$$\begin{aligned}\mathrm{Re}\left(b_{m,i}b_{m,n}^*[\boldsymbol{S}_y]_{ni}\right) =&\mathrm{Re}([\boldsymbol{S}_y]_{ni})\mathrm{Re}(b_{m,i}b_{m,n}^*) - \mathrm{Im}([\boldsymbol{S}_y]_{ni})\mathrm{Im}(b_{m,i}b_{m,n}^*)\\ =&\mathrm{Re}([\boldsymbol{S}_y]_{ni})\left[\mathrm{Re}(b_{m,n})\mathrm{Re}(b_{m,i}) + \mathrm{Im}(b_{m,n})\mathrm{Im}(b_{m,i})\right]\\ &+ \mathrm{Im}([\boldsymbol{S}_y]_{ni})\left[\mathrm{Im}(b_{m,n})\mathrm{Re}(b_{m,i}) - \mathrm{Re}(b_{m,n})\mathrm{Im}(b_{m,i})\right]\end{aligned}$$

Finally, $\boldsymbol{b}_m^H\boldsymbol{S}_y\boldsymbol{b}_m$ is calculated by

$$\begin{aligned}\boldsymbol{b}_m^H\boldsymbol{S}_y\boldsymbol{b}_m =&\sum_{n=1}^{N}\mathrm{Re}([\boldsymbol{S}_y]_{nn})\left(\mathrm{Re}(b_{m,n})^2 + \mathrm{Im}(b_{m,n})^2\right)\\ &+ 2\sum_{n=1}^{N}\sum_{i=n+1}^{N}\mathrm{Re}([\boldsymbol{S}_y]_{ni})\left[\mathrm{Re}(b_{m,n})\mathrm{Re}(b_{m,i}) + \mathrm{Im}(b_{m,n})\mathrm{Im}(b_{m,i})\right]\\ &+ 2\sum_{n=1}^{N}\sum_{i=n+1}^{N}\mathrm{Im}([\boldsymbol{S}_y]_{ni})\left[\mathrm{Im}(b_{m,n})\mathrm{Re}(b_{m,i}) - \mathrm{Re}(b_{m,n})\mathrm{Im}(b_{m,i})\right]\end{aligned} \tag{A.1}$$

where only the upper triangle of $\boldsymbol{S}_y$ is needed.

A.2. Projection Matrix Decomposition

The projection matrix $\boldsymbol{P}$ from (5.15) with $\boldsymbol{A}_{\mathcal{M}} = \boldsymbol{Q}\boldsymbol{R}_{\mathcal{M}}$ is

$$\begin{aligned}\boldsymbol{P} &= \boldsymbol{A}_{\mathcal{M}}\boldsymbol{A}_{\mathcal{M}}^{+} = \boldsymbol{A}_{\mathcal{M}}(\boldsymbol{A}_{\mathcal{M}}^H\boldsymbol{A}_{\mathcal{M}})^{-1}\boldsymbol{A}_{\mathcal{M}}^H\\ &= \boldsymbol{A}_{\mathcal{M}}(\boldsymbol{R}_{\mathcal{M}}^H\boldsymbol{R}_{\mathcal{M}})^{-1}\boldsymbol{A}_{\mathcal{M}}^H\\ &= \boldsymbol{A}_{\mathcal{M}}\boldsymbol{R}_{\mathcal{M}}^{-1}\boldsymbol{R}_{\mathcal{M}}^{-H}\boldsymbol{A}_{\mathcal{M}}^H\\ &= (\boldsymbol{A}_{\mathcal{M}}\boldsymbol{R}_{\mathcal{M}}^{-1})(\boldsymbol{A}_{\mathcal{M}}\boldsymbol{R}_{\mathcal{M}}^{-1})^H\\ &= \boldsymbol{Q}\boldsymbol{Q}^H\end{aligned} \tag{A.2}$$

A.3. Trace

Calculating the trace in (5.14) is done by

$$\mathrm{tr}((\boldsymbol{I}_N - \boldsymbol{P})\boldsymbol{S}_y) = \mathrm{tr}(\boldsymbol{S}_y - \boldsymbol{P}\boldsymbol{S}_y).$$

The trace is a linear function, hence $\mathrm{tr}(\boldsymbol{B}+\boldsymbol{C}) = \mathrm{tr}(\boldsymbol{B}) + \mathrm{tr}(\boldsymbol{C})$. If both matrices $\boldsymbol{BC}$ and $\boldsymbol{CB}$ exist, $\mathrm{tr}(\boldsymbol{BC}) = \mathrm{tr}(\boldsymbol{CB})$ holds. Therefore,

$$\begin{aligned}
\mathrm{tr}(\boldsymbol{S}_y - \boldsymbol{P}\boldsymbol{S}_y) &= \mathrm{tr}(\boldsymbol{S}_y) - \mathrm{tr}(\boldsymbol{P}\boldsymbol{S}_y) \\
&= \mathrm{tr}(\boldsymbol{S}_y) - \mathrm{tr}(\boldsymbol{Q}\boldsymbol{Q}^H\boldsymbol{S}_y) \\
&= \mathrm{tr}(\boldsymbol{S}_y) - \mathrm{tr}(\boldsymbol{Q}^H\boldsymbol{S}_y\boldsymbol{Q}) \\
&= \mathrm{tr}(\boldsymbol{S}_y) - \sum_{k=1}^{K} \boldsymbol{q}_k^H\boldsymbol{S}_y\boldsymbol{q}_k.
\end{aligned}$$

The second term can be calculated by using the same approach as for (A.1) and extend for multiple columns.

$$\begin{aligned}
\sum_{k=1}^{K} \boldsymbol{q}_k^H\boldsymbol{S}_y\boldsymbol{q}_k = \sum_{n=1}^{N} &\mathrm{Re}([\boldsymbol{S}_y]_{nn}) \left(\sum_{k=1}^{K}\mathrm{Re}(q_{nk})^2 + \sum_{k=1}^{K}\mathrm{Im}(q_{nk})^2\right) \\
&+ 2\sum_{n=1}^{N}\sum_{i=n+1}^{N}\left[\mathrm{Re}([\boldsymbol{S}_y]_{ni})\sum_{k=1}^{K}\mathrm{Re}(q_{ik}q_{nk}^*)\right] \\
&+ 2\sum_{n=1}^{N}\sum_{i=n+1}^{N}\left[\mathrm{Im}([\boldsymbol{S}_y]_{ni})\sum_{k=1}^{K}\mathrm{Im}(q_{ik}q_{nk}^*)\right]
\end{aligned} \tag{A.3}$$

with

$$\begin{aligned}
\sum_{k=1}^{K}\mathrm{Re}(q_{ik}q_{nk}^*) &= \sum_{k=1}^{K}\mathrm{Re}(q_{ik})\mathrm{Re}(q_{nk}) + \mathrm{Im}(q_{ik})\mathrm{Im}(q_{nk}) \\
\sum_{k=1}^{K}\mathrm{Im}(q_{ik}q_{nk}^*) &= \sum_{k=1}^{K}\mathrm{Im}(q_{ik})\mathrm{Re}(q_{nk}) - \mathrm{Re}(q_{ik})\mathrm{Im}(q_{nk})
\end{aligned}$$

Acronyms

ADC analog-digital converter.

BRAM block RAM.

CFG control flow graph.

CDFG control/data flow graph.

CS control step.

DFG data flow graph.

DOA directions of arrival.

DSP digital signal processing.

DUT design under test.

FPGA field-programmable gate array.

FSM finite-state machine.

HDL hardware description language.

HLS high-level synthesis.

II initiation interval.

i.i.d. idenpendent and identically distributed.

MMV multiple measurement vector.

MSB most significant bit.

MSE mean squared error.

LSB least significant bit.

LUT look-up table.

RAM random access memory.

RMSE root mean squared error.

RTL register-transfer level.

SBL	sparse Bayesian learning.
SMV	single measurement vector.
SNR	signal-to-noise ratio.
UDP	user datagram protocol.
ULA	uniform linear array.
TCP	transport control protocol.

Bibliography

[1] Peter Gerstoft, Christoph F. Mecklenbräuker, A. Xenaki, and S. Nannuru. Multisnapshot sparse bayesian learning for doa. *IEEE Signal Processing Letters*, 23(10):1469–1473, Oct 2016.

[2] D. P. Wipf and B. D. Rao. An empirical bayesian strategy for solving the simultaneous sparse approximation problem. *IEEE Transactions on Signal Processing*, 55(7):3704–3716, July 2007.

[3] Angeliki Xenaki, Peter Gerstoft, and Klaus Mosegaard. Compressive beamforming. *The Journal of the Acoustical Society of America*, 136(1):260–271, 2014.

[4] Y.C. Eldar and G. Kutyniok. *Compressed Sensing: Theory and Applications*. Compressed Sensing: Theory and Applications. Cambridge University Press, 2012.

[5] M. Elad. *Sparse and Redundant Representations: From Theory to Applications in Signal and Image Processing*. Springer New York, 2010.

[6] Michael E. Tipping and Alex Smola. Sparse bayesian learning and the relevance vector machine, 2001.

[7] Harry L. van Trees. *Detection, estimation, and modulation theory : 4. Optimum array processing*. Wiley, Hoboken, NJ [u.a.], 2002.

[8] D. Koch, F. Hannig, and D. Ziener. *FPGAs for Software Programmers*. Springer International Publishing, 2016.

[9] Conrad Sanderson. Armadillo: An open source c++ linear algebra library for fast prototyping and computationally intensive experiments. Technical report, 2010.

[10] A. Suardi, E. C. Kerrigan, and G. A. Constantinides. Fast fpga prototyping toolbox for embedded optimization. In *2015 European Control Conference (ECC)*, pages 2589–2594, July 2015.

[11] P. Födisch, B. Lange, J. Sandmann, A. Büchner, W. Enghardt, and P. Kaever. A synchronous Gigabit Ethernet protocol stack for high-throughput UDP/IP applications. *Journal of Instrumentation*, 11:P01010, January 2016.

[12] T.K. Moon and W.C. Stirling. *Mathematical Methods and Algorithms for Signal Processing*. Prentice Hall, 2000.

[13] Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 2nd edition, 1998.

[14] Gene H. Golub and Charles F. Van Loan. *Matrix computations*. Johns Hopkins studies in the mathematical sciences. Johns Hopkins Univ. Press, Baltimore, Md., 4. ed. edition, 2013.

[15] Xilinx. *Vivado Design Suite User Guide: High-Level Synthesis*, June 2017. UG902 (v2017.2).

[16] IEEE Computer Society. Ieee standard vhdl language reference manual. *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)*, pages c1–626, Jan 2009.

[17] Todd Veldhuizen. Techniques for scientific c++. Technical Report 542, Indiana University Computer Science, 2000.

[18] Donald E. Knuth. *The Art of Computer Programming, Volume II: Seminumerical Algorithms*. Addison-Wesley, 3rd edition, 1998.

[19] W.H. Press. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge University Press, 2007.

[20] M. Martel, A. Najahi, and G. Revy. Toward the synthesis of fixed-point code for matrix inversion based on cholesky decomposition. In *Proceedings of the 2014 Conference on Design and Architectures for Signal and Image Processing*, pages 1–8, Oct 2014.

Declaration of Authorship