Politecnico di Torino

ScuDo
Scuola di Dottorato - Doctoral School
WHAT YOU ARE, TAKES YOU FAR

Doctoral Dissertation
Doctoral Program in Electronics and Telecommunications engineering (34^{th}cycle)

FPGA Acceleration of Domain-specific Kernels via High-Level Synthesis

By

Mohammad Amir Mansoori

******

Supervisor(s):
Prof. Mario R. Casu

Doctoral Examination Committee:
Prof. Christian Pilato , Referee, Politecnico di Milano
Prof. Ioannis Sourdis, Referee, Chalmers University of Technology

Politecnico di Torino
2022

Declaration

I hereby declare that, the contents and organization of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.

Mohammad Amir Mansoori
2022

* This dissertation is presented in partial fulfillment of the requirements for **Ph.D. degree** in the Graduate School of Politecnico di Torino (ScuDo).

I would like to dedicate this thesis to my niece, Gandom, who is 4 years old at this time. I do so to show my endless love to her and to see her happiness and smiley face.

Acknowledgements

And I would like to thank my supervisor, Prof. Mario R. Casu, for his continuous support, guidance, and patience throughout my PhD project. I would also like to acknowledge the support from Prof. Luciano Lavagno for all his invaluable advice. In addition, I acknowledge the financial support provided by the EMERALD project (Grant Agreement No. 764479) and Prof. Francesca Vipiana as the project coordinator.

Abstract

Compute-demanding algorithms in today's applications need to achieve high performance, which is becoming more difficult in general-purpose processors due to the decline of the Moore's law. Domain-specific hardware accelerators can assist general-purpose processors in improving the performance and efficiency while preserving the flexibility. They can accelerate a *domain* of applications rather than a single application making it possible to use the efficient specialized hardware acceleration techniques in a broad range of applications.

In this thesis, we focus on the development of Domain-Specific Accelerators (DSAs) for a broad domain of applications consisting of *biomedical microwave algorithms* and *Machine Learning (ML) techniques*. Although the initial purpose of this research was the development of a biomedical Microwave Imaging (MI) system, The hardware acceleration methods introduced in this thesis are not limited to MI only. We analyzed the recurrent algorithms that are used in these applications to extract their compute-intensive parts that are termed *kernels*. Then we proposed efficient accelerators for these domain-specific kernels to achieve high performance.

The main computational kernels that are considered in this work are *Finite Difference Time Domain (FDTD)*, *Principal Component Analysis (PCA)*, *Support Vector Machine (SVM)*, and *Artificial Neural Networks (ANNs)* including *Multi-Layer Perceptron (MLP)* and *Convolutional Neural Networks (CNNs)*. For each kernel, we proposed highly efficient hardware accelerators to obtain an optimal performance by considering several factors such as processing time, resource usage, and power consumption. The target hardware platform is Field Programmable Gate Arrays (FPGAs) and the hardware design approach is High Level Synthesis (HLS) which is used to convert a software code written in C or C++ into its corresponding hardware description language. Although several FPGA accelerators have already been presented for the above-mentioned kernels, they have some drawbacks and

limitations. Our proposed design methodologies try to address and overcome these limitations.

The proposed hardware accelerator for 3D FDTD considers the impact of polarization currents in dispersive materials, and models the absorbing boundary conditions as Convolutional Perfectly Matched Layers (CPML) in all directions, as opposed to the conventional FDTD accelerators. We use spatial blocking to store a partial block of data while processing the previous block. Local storage of FDTD coefficients and boundary elements, function inlining, and merging the parallel loops are among the other optimization techniques.

The PCA hardware accelerator considered in this work is implemented in FPGA and is designed entirely in HLS. A new block-streaming method is introduced to make the internal PCA computations more efficient. The flexibility of our design allows us to use it for different FPGA targets, with flexible input data dimensions, and it also lets us easily switch from a more accurate floating-point implementation to a higher speed fixed-point solution.

To implement a fast and accurate Support Vector Machine (SVM) classifiers in embedded systems, we propose a flexible FPGA-based SVM accelerator highly optimized through a dataflow architecture. Thanks to HLS and the dataflow method, our design is scalable and can be used for large data dimensions when there is limited on-chip memory. The hardware parallelism is adjustable and can be specified according to the available FPGA resources. The performance of different SVM kernels is evaluated in hardware. In addition, an efficient fixed-point implementation is proposed to improve the speed.

The last computational kernel considered in this thesis is related to the Neural Networks. Although there are some tools available to generate a hardware design from a high level description of the network (like hls4ml), the selection of network parameters and hardware configurations at the same time is not a trivial task. Although several works have recently addressed the problem of performance co-optimization for hardware and network training, most of them considered either a fixed network or a given hardware architecture. In this work, we propose a new framework for joint optimization of network architecture and hardware configurations, which is based on Bayesian Optimization (BO) on top of HLS. We evaluate our methodology on a network optimized for an FPGA target and show the efficiency of the Pareto set obtained by the proposed joint-optimization approach.

Contents

List of Figures **xi**

List of Tables **xvi**

1 Introduction **1**

1.1 Motivation and Problem Statement 3

1.2 Compute-Intensive Kernels 3

1.2.1 3D FDTD 4

1.2.2 PCA using SVD/EVD 5

1.2.3 SVM 6

1.2.4 Neural Networks 6

1.3 Challenges in Designing Domain-Specific Accelerators 7

1.4 Scope and Outline 9

2 Background **12**

2.1 Performance Analysis of Domain-Specific Accelerators 12

2.2 High Level Synthesis for the Design of Hardware Accelerators . . . 14

2.2.1 Design Optimizations in HLS 15

2.2.2 HLS Design Trends in Selected Domains 17

2.3 Applications of Domain-Specific Accelerators 17

2.3.1 Biomedical Microwave Techniques 18

2.3.2 Machine Learning (ML) 21
2.4 Microwave Imaging Algorithms 24
2.4.1 Inverse Scattering Problem 24
2.4.2 Qualitative Imaging 27
2.4.3 Quantitative Imaging 27
2.5 Machine Learning Algorithms 28
2.5.1 Preprocessing 29
2.5.2 Feature Extraction 29
2.5.3 Classification 29

3 FPGA Acceleration of 3D FDTD for Microwave Imaging using HLS 31

3.1 Related Work 34
3.2 FDTD in Microwave Imaging 36
3.2.1 Background 36
3.2.2 Boundary Conditions: CPML 37
3.2.3 FDTD Pseudo-Code 38
3.3 FPGA Design of an FDTD Compute Unit 40
3.3.1 Two Architectures: Large and Small 42
3.3.2 Blocking Method and Merging of J_P Update Equations . . . 44
3.3.3 Loop Merge and Local Storage for Boundaries 46
3.3.4 Loop Pipeline, Function Inline, and Storage for Coefficients 48
3.4 Multi-FPGA Implementation 49
3.5 Results 51
3.5.1 Impact of HLS Optimizations on Performance 52
3.5.2 FDTD Performance on a Single FPGA 54
3.5.3 FDTD Performance on Multiple FPGAs 60
3.6 Discussion 65

3.7 Conclusions 67

4 High Level Design of a Flexible PCA Hardware Accelerator 69

4.1 PCA Algorithm Description 72

4.2 Hyperspectral Imaging 74

4.3 PCA Hardware Accelerator Design 74

4.3.1 Block-Streaming for Covariance Computation 76

4.3.2 High Level Synthesis Optimizations 79

4.3.3 Fixed-Point Design of the Accelerator 84

4.3.4 Hardware Prototype for PCA Accelerator Assessment with the HI Data Set 85

4.4 Results 87

4.4.1 Number of Blocks, Bands, and Pixels 88

4.4.2 Fixed-Point and Floating-Point Comparison 90

4.4.3 Evaluation on Hyperspectral Images 92

4.4.4 Evaluation on Microwave Data 98

4.5 Conclusions 98

5 HLS-based Dataflow Hardware Architecture for Support Vector Machine 100

5.1 SVM Background 102

5.2 Proposed SVM Accelerator 103

5.2.1 Read SVM Inputs 104

5.2.2 Kernel Computation 106

5.2.3 Decision Function 107

5.2.4 Fixed-Point Implementation 107

5.3 Results 107

5.3.1 Microwave Data Set 109

5.4 Conclusions 111

6 Hardware Design and Optimization of Neural Networks and ML Accelerators 112

6.1 Related Work 113

6.2 Multi-objective Framework for Training and Hardware Co-optimization 115

6.2.1 Multi-Objective BO with Constraints (MOBOC) 115

6.2.2 Search Space 116

6.2.3 Function Evaluations 117

6.2.4 Objectives and Constraints Extraction 117

6.2.5 Update Bayesian model 117

6.3 Evaluation on Neural Networks 117

6.3.1 Multi-Layer Perceptron (MLP) 117

6.3.2 Convolutional Neural Networks (CNN) 119

6.3.3 MLP in Microwave Data Set 122

6.4 Conclusions 124

7 Conclusions and Future Work 125

7.1 Conclusions 125

7.1.1 List of Published Papers 128

7.2 Future Work 129

References 132

Appendix A List of Acronyms 145

List of Figures

1.1 Propagation of electromagnetic fields and the impact of variations in electric and magnetic fields. 4

1.2 PCA for dimensionality reduction to remove redundant information. 5

1.3 SVM for binary classification finds the decision boundary with maximum margin between two classes (L2). 6

1.4 Network structure for a) MLP and b) CNN. 7

2.1 Various design metrics in the implementation of a hardware accelerator 14

2.2 Hardware design flow in Vivado 15

2.3 Hardware optimization directives 16

2.4 General diagram of a Microwave Imaging system. 19

2.5 Dielectric properties of breast tissues 20

2.6 Brain stroke detection with MI system containing V antennas and P candidate locations for the stroke positions (r_1 to r_P) 21

2.7 Machine Learning training and inference steps. 22

2.8 Hyper-spectral Images (HI) with C bands. Each band has $R = M \times N$ pixels. 23

2.9 Machine Learning Applications. 23

2.10 Microwave Imaging setup. Inverse scattering is the problem of finding ε and μ from the microwave measurements in the measurement domain. 25

2.11 General diagram of a non-linear image reconstruction iterative algorithm in MI, with the compute-intensive FDTD step. 28

2.12 Three data processing steps in Machine Learning. Note that in DNNs, feature extraction and classification are implemented in different layers of the network. 29

3.1 Boundary regions for *H* field in 3D FDTD. 38

3.2 FDTD CU design in HLS for a single FPGA. 41

3.3 Detailed view of the CU design in Vivado. 42

3.4 Details of the interfaces of the CU for Small and Large designs. . . 43

3.5 Blocking method for FDTD and its difference with a general stencil 45

3.6 Multi-FPGA platform with *F* FPGAs for 3D FDTD acceleration. . . 50

3.7 Accuracy comparison: Acceleware design versus our C++ code. . . 51

3.8 Impact of different HLS optimization methods on the total latency. (numbers on top of the bars show the improvement compared to the original code, and numbers bellow the arrows show the improvement compared to the previous optimization method) 52

3.9 Impact of different HLS optimization methods on the latency of each FDTD function. 53

3.10 Impact of HLS optimization methods on resource usage per SLR. . 54

3.11 The performance of the main FDTD loops in Small and Large design in HLS. 55

3.12 Device view in a) Small and b) Large design after place-and-route (it contains 3 SLRs in the left, middle and right side of the FPGA). . 57

3.13 FDTD execution time for different number of FPGAs, (a) 8 antennas, (b) 24 antennas. 62

3.14 FDTD execution time for 8 FPGAs and different number of antennas, (a) from one antenna up to the maximum number in a single FPGA (3 for Small design), (b) Comparison of the single GPU in this work (GPU3, highly optimized for one antenna) and multi-FPGA design for multiple antennas, (c) more detailed view of multi-FPGA design results. 64

4.1 Architecture of the proposed hardware accelerator for Principal Component Analysis (PCA) in Field-Programmable Gate Arrays (FPGA). 75

4.2 Example of covariance computation with 9 bands and N pixels, $PQ = \Sigma_{i=1}^{N} P_i \times Q_i$, where P, Q are the symbols of bands (α to n). 76

4.3 Example of partitioning of input data into blocks. The total number of bands is $B = 9$ and the block size is $B_{max} = 3$. 77

4.4 Illustration of an example of covariance computation using the block-streaming method with 3 blocks ($B = 9$, $B_{max} = 3$). 77

4.5 Block-streaming method with 4 blocks ($B/B_{max} = 4$). 78

4.6 Order of data storage in the Diagonal and Off-diagonal RAMs inside the Cov unit. 79

4.7 PCA accelerator design in Zedboard. 86

4.8 Impact of block size (B_{max}) on the resource usage and latency for the Virtex7, bands = 48, pixels = 300×300, floating-point design. . 88

4.9 Impact of the number of pixels on the latency for Virtex7, bands=48, $B_{max} = 8$, floating-point design. 89

4.10 Latency and resource usage for Virtex7 with a fixed block size ($B_{max} = 8$), floating-point design. 89

4.11 Resource usage for Zedboard for fixed- and floating-point design, B = 12, pixels = 300×300. 90

4.12 Comparison of the latency of the fixed- and floating-point design for Zedboard, B = 12, pixels = 300×300. 91

4.13 Latency for Zynq7000 with a fixed block size ($B_{max} = 3$), pixels = 300×300. 92

4.14 Resource usage for Zynq7000 with a fixed block size ($B_{max} = 3$), pixels=300×300. 92

4.15 The first 6 principal components of the HI data set. Our PCA accelerator in Zedboard produces the first 3 outputs (PCA1 to PCA3). . . 95

4.16 Energy distribution of the eigenvalues for the Hyperspectral Imaging (HI) data set. 95

4.17 Comparison of different hardware platforms between latency per pixel, power consumption, input size (bands) and energy. 97

4.18 Processing time for PCA compute units in FPGA with Microwave dataset 98

5.1 SVM classification with linear kernel. 102

5.2 Proposed SVM accelerator in HLS. 104

5.3 Impact of the number of FIFO channels (BF) with a total of $4 \times 8 =$ 32 data, (a) $BF = N$, (b) $BF = 2N$, overall latency is reduced. 105

5.4 Manual unrolling for kernel computation. 106

6.1 Optimization of training hyper-parameters and hardware configurations: (a) traditional separate DSE, (b) more efficient joint DSE. DS_1 and DS_2 stand for Design Space of training and hardware design, respectively. 113

6.2 Proposed methodology for training and hardware co-optimization in FPGA devices. 116

6.3 Percentage of training error (float error) and hardware error (fixed-point error) in each BO iteration, (a) proposed joint optimization, (b) separate optimization. 118

6.4 Comparison of (a) prediction time and (b) Pareto fronts. 118

6.5 Pareto-points found by the joint approach, random search, and conventional separate method in the space of prediction error (Hw error) and execution latency (Time). Total number of iterations is 100 for all methods. 121
6.6 Hardware error in our joint method in each BO iteration with (a) linear and (b) exponential error function. 121
6.7 Total points suggested by the joint, separate, and random search methods; note the concentration of the joint method on low errors ($< 10\%$). 122
6.8 Accuracy of MLP during training by MI dataset 123
6.9 Resource usage for MLP in Zynq FPGA 124

List of Tables

1.1 Domain-specific kernels (rows) and their applications (columns). . . 4

3.1 HLS hardware optimization strategies for a FDTD CU. 41

3.2 Description of AXI interfaces. 44

3.3 Resource usage for the Small design: HLS estimation and Vivado implementation results 58

3.4 Resource usage for the Large design: HLS estimation and Vivado implementation results 59

3.5 Performance comparison: CPU = Intel Xeon, GPU1 = Tesla K20C GPU2 = Tesla P40, GPU3 = GPU1 CUDA implementation, and FPGA (UltraScale+). TDP = Thermal Design Power, Energy = TDP$\times$Time. 61

3.6 Performance comparison between our single Small FPGA design and other FPGA implementations. 61

3.7 Dimensions of FDTD coefficients. 63

4.1 Resource usage obtained from HLS for HI data set on Zedboard, bands = 12. 93

4.2 Latency (ms) for Zedboard, HI data set, bands = 12. 93

4.3 Vivado implementation of PCA accelerator on Zedboard for HI data, bands=12, accuracy is compared with MATLAB. 94

4.4 Comparison of our PCA accelerator with other conventional methods. The input data dimensions are set to 30×16 for all designs. 95

4.5 Comparison of the proposed PCA hardware design with other High Level Synthesis (HLS)-based accelerators. The dimensions of a spectral image data set ($640 \times 480 \times 12$) is selected for all of the designs. 96

4.6 Execution time (ms) for the PCA implementation on GPU, Massively Parallel Processing Array (MPPA), and FPGA (our work). 97

5.1 Kernel functions in SVM. 103

5.2 MNIST dataset: performance and resource usage. 108

5.3 Comparison of different SVM kernels. 108

5.4 Comparison of the proposed accelerator with different SVs and same number of features ($N_f = 27$) in the same FPGA (model1 and model2 use different pre-processing methods on training data). . . . 109

5.5 Performance comparison with two manual RTL designs. 109

5.6 Performance analysis of SVM accelerator for medical microwave data set using floating-point data precision ($N_{SV} = 2009, N_{features} = 110, N_{samples} = 900$). 111

5.7 Performance analysis of SVM accelerator for medical microwave data set using fixed-point data precision ($N_{SV} = 2009, N_{features} = 110, N_{samples} = 900$). 111

6.1 Ranges of parameters for the joint training/hardware optimization method. 118

6.2 Search space featuring network architecture and hardware configurations (UF:Unroll Factor) 120

6.3 Pareto points obtained by three methods. 121

6.4 Evaluation of MLP performance in microwave anomaly detection. Fixed-point precision is selected in hardware with total and integer widths of 16 and 10, respectively. 123

Chapter 1

Introduction

Today's increasing demand for compute-intensive applications calls for efficient approaches to achieve the desired performance. Most of the computations in a variety of applications usually take place in general purpose processors, or CPUs. To keep up with the increasing computations in new applications, one could easily wait for the new technology which, according to the Moore's law, could make it possible to run the application faster. However, the end of Moore's law prevents to continue scaling of the performance and efficiency. Therefore, we need to look for other alternatives to speed up the computations in current and future applications. One of the few alternatives is "*Domain-Specific hardware Accelerators (DSAs)* ".

Domain-specific accelerators are subset of hardware accelerators that can be used for a specific domain of applications. There are different applications in which the domain-specific accelerators have been already used with considerable improvement in performance compared to general-purpose computers. These applications include deep learning, bioinformatics, image processing, and many more fields. Designing an accelerator requires considerable effort to acquire an efficient hardware performance, which can be achieved by a combination of the following methodologies: Parallelism, efficient memory systems, specialized operations, and overhead reduction techniques. The reason of recent trend in using DSAs is their potential as one of the few techniques to improve the efficiency and performance of the accelerators even with the decline of Moore's law benefits [1].

DSAs can be used in embedded systems which are computing platforms containing all the hardware and software components embedded in the system to execute an

application-specific program. A high-performance embedded platform very often consists of one or more general purpose processors, together with a number of specialized *hardware accelerators* (DSAs). These can be either on the same electronic board, or integrated in the same System-on-Chip (SoC). One of the popular approaches in recent years to design such accelerators is to use Field Programmable Gate Arrays (FPGAs). Nowadays, advanced SoC FPGAs have a variety of programmable features as well as hardwired functions providing flexibility for the hardware designers. One example is the usage of soft processors or hard processors in these systems. The former uses the internal programmable logic and resources of an FPGA to design a processor, while the latter is a fixed hardware outside the programmable logic of the FPGA and communicates with it via dedicated peripherals in the same SoC. The flexibility comes from large amount of hardware resources available in today's FPGAs, including massive arrays of programmable logic units and their interconnections, large on-chip memories, custom data paths, high speed I/O, and microprocessor cores all co-located on the same chip [2].

Designing a hardware accelerator faces the designers with several challenges. Firstly, design goals and requirements for different applications can vary significantly. In some application such as video encoding and streaming, a high throughput is required while in others accuracy is more important than throughput, such as bio-sensing data acquisition systems. Some applications tend to be inherently computation-limited, in the sense that their performance is determined by the number of computational resources working in parallel, while others tend to be inherently memory-limited, because their performance is bounded by the memory bandwidth. There are finally cases in which depending on the details of the implementation (e.g., on the internal parallelism), an application can transition from a computation- to a memory-bound performance region (this is typical for Deep Learning workloads). Depending on the application, the hardware design goals and technique have to change. Secondly, achieving the optimal performance in hardware accelerators requires a trade off between different design goals, which is clarified in more detail in the next section. Thirdly, in today's high-performance computing applications, one of the important factors is the design and development time, which tends to be very high if the traditional approaches to the hardware design are used. Therefore, it is important to utilize the capabilities of recent design tools to obtain an efficient hardware design, a subject that will be introduced and discussed in more detail in section 2.2.

In the remainder of this chapter, we first introduce the main motivation behind this work and discuss the problem statement. In Section 1.2, the domain-specific computational kernels that are considered in this thesis are briefly introduced. The challenges of designing hardware accelerators for these compute-intensive kernels are described in Section 1.3. Finally, in Section 1.4, an overview of this thesis, with scope and goal, is given.

1.1 Motivation and Problem Statement

The main objective of this thesis is to design efficient hardware accelerators for compute-intensive kernels that are used in two main application areas: *Biomedical Microwave techniques* and *Machine Learning algorithms*. Although the initial motivation of this work was the development of a medical microwave imaging device in the EMERALD project[1], we considered a broader range of applications that is not restricted to Microwave Imaging only and includes Machine Learning methods as well.

There are several algorithms that are used in these application areas which contain computationally-expensive kernels. General purpose processors cannot offer high performance when they are used for these kernels. An alternative is using domain-specific accelerators which provide higher efficiency.

Therefore, the main problem that is addressed in this thesis can be divided into two parts: (1) Analysis of the recurrent algorithms in biomedical microwave imaging and Machine Learning to find the compute-intensive kernels, (2) Designing the domain-specific accelerators for these kernels in an efficient way.

1.2 Compute-Intensive Kernels

In this section, we briefly introduce the compute-intensive kernels inside the algorithms that are used in different application areas. This broad *domain of applications* include "medical microwave techniques" (which contains "Analysis of antennas and electromagnetic wave simulation"), and Machine Learning (which contains "Hyperspectral imaging", "Feature extraction", and "Classification").

[1] www.msca-emerald.eu

Table 1.1 shows the domain-specific kernels considered in this thesis (rows of the table) and their potential application areas (columns of the table). All the kernels in Table 1.1 can be used in the “Biomedical Microwave Techniques” application area. However, we consider a broader domain of applications in which the kernels in this thesis can be used. In the next chapters, the methodologies to design efficient hardware accelerators for these kernels will be presented in order to optimize and enhance the performance of these kernels in FPGAs.

Table 1.1 Domain-specific kernels (rows) and their applications (columns).

	Biomedical Microwave techniques	Antennas analysis	Hyperspectral Imaging	Machine Learning	
				Feature extraction	Classification
FDTD	+	+	-	-	-
SVD/EVD	+	-	-	+	-
PCA	+	-	+	+	-
SVM	+	-	+	-	+
MLP	+	-	+	+	+
CNN	+	-	+	+	+

1.2.1 3D FDTD

Finite Difference Time Domain (FDTD) is a numerical analysis technique to simulate the propagation of electromagnetic fields in different materials. Based on Maxwell equations, any variation in the electric fields will cause a magnetic field and vice versa, as shown in Fig. 1.1.

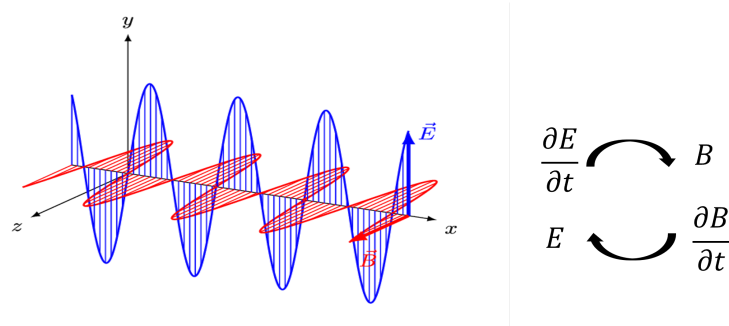


Fig. 1.1 Propagation of electromagnetic fields and the impact of variations in electric and magnetic fields.

As will be stated in Section 2.4.3, FDTD is the critical computational part of one of the non-linear iterative microwave image reconstruction algorithms. Due to the need for higher resolution in MI recent years, three dimensional FDTD has become

more popular. Therefore, hardware acceleration of 3D FDTD can significantly improve the performance of embedded MI systems which use FDTD in their internal algorithms. In Chapter 3 the details of proposed methodology for 3D FDTD hardware accelerator in FPGA is presented.

1.2.2 PCA using SVD/EVD

Principal Component Analysis (PCA) is a feature extraction technique in ML and can be used to remove redundant information in data. An illustrative example is shown in Fig. 1.2 in which the redundant information in the horizontal axis can be removed by transforming original data into its corresponding principal components and ignoring the axis with low variations. PCA is extremely useful when we have large data dimensions and it is difficult to process the entire data, hence reducing the dimensions of data.

As we will see later, PCA consists of several components. These components can be considered as other compute-intensive kernels. For example, Singular Value Decomposition (SVD) and Eigenvalue Decomposition (EVD), can be used not only in PCA, but also in other inverse scattering solutions in MI. Therefore, by hardware implementation of PCA, we can accelerate the execution of several compute-intensive kernels, such as SVD and EVD. We explain the proposed methodology for the hardware acceleration of PCA in Chapter 4.

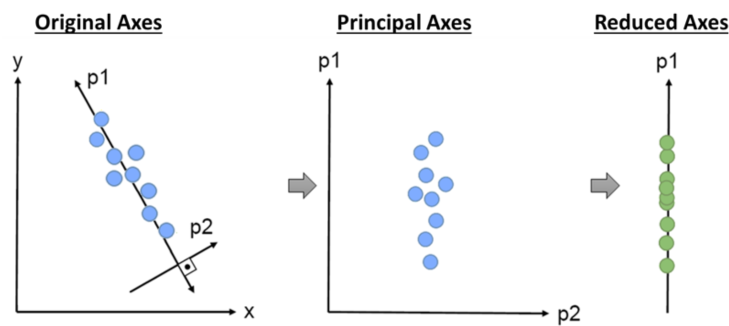


Fig. 1.2 PCA for dimensionality reduction to remove redundant information.

1.2.3 SVM

Support Vector Machine (SVM) is a powerful classification (and regression) algorithm in ML. SVM for binary classification finds a separating line (or hyper-plane) between data points of two classes which can best separate the classes. As shown in Fig. 1.3, SVM obtains the separating line by maximizing the margin between the classes. Although there are infinite number of separating lines (e.g. L1 to L3 in Fig. 1.3), only one line is optimal (L2) which can be generalized to the new data points that might appear in the future. In Chapter 5, more details about the theory of SVM together with its corresponding hardware accelerator design are explained thoroughly.

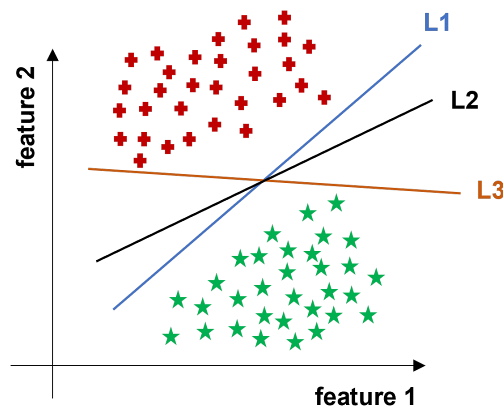


Fig. 1.3 SVM for binary classification finds the decision boundary with maximum margin between two classes (L2).

1.2.4 Neural Networks

Neural Networks (NNs), also called Artificial NNs (ANNs) are a subset of ML and at the heart of Deep Learning (DL) algorithms. Their structure is inspired from the human brain and from the way that biological neurons signal to each other. ANNs comprise of input layers, multiple hidden layers, and output layers, and each layer consists of several compute nodes. These nodes receive and process the inputs and weights from the previous layer to produce the outputs.

There are different types of architectures for NNs. In Multi-Layer Perceptrons (MLPs), network layers are fully connected to each other, meaning that each node in one layer is connected to all the nodes in the next layer, as shown in Fig. 1.4 (a). In

Convolutional Neural Networks (CNNs) each layer consists of multiple filters that are *convolved* with the input data, as shown in Fig. 1.4 (b). Other types of ANNs can be found in [3]. In Chapter 6, more details about hardware acceleration of ANNs will be presented. In addition, we will propose a new framework for co-optimization of training hyper-parameters and hardware configurations to achieve the optimum performance in embedded accelerators design for ML algorithms, including ANNs.

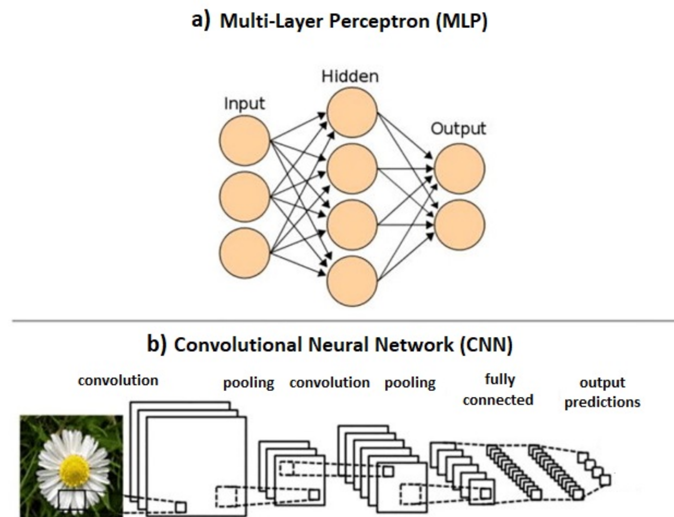


Fig. 1.4 Network structure for a) MLP and b) CNN [4]

1.3 Challenges in Designing Domain-Specific Accelerators

There are several challenges in designing hardware accelerators for the domain-specific kernels introduced in previous section. In the following, we briefly explain these challenges and then, we explain the methodologies that we proposed to overcome these challenges:

1. In **3D FDTD** kernel, one of the main challenges is the **access to a large volume of data** from external memory. Although several methodologies have been proposed to address this issue, they are not effective in Microwave Imaging (MI) where we need to add more details in FDTD equations. Specifically,

previous works did not consider the **impact of polarization current** and dispersive materials in FDTD equations. In addition, the **boundary conditions** used in the majority of the previous works oversimplify the FDTD algorithm. Considering these details in FDTD equations makes the design of the hardware accelerator more challenging which we address in this thesis.

2. In **PCA** kernel, the **large data dimension** and computational complexity of PCA algorithm create several challenges. Due to the large amount of data in PCA computations, most of the previous works could not implement all the computational units in hardware, so they left some essential parts (like covariance computation) to be executed off-line in software. In addition, conventional PCA accelerators use RTL approach for the hardware design which increases the design and development time. **Flexibility** and **efficiency** are other challenges in PCA accelerator design. Supporting different data dimensions and data types is not trivial in a full PCA hardware accelerator design which is considered in this thesis.

3. In **SVM** kernel, the main challenge that is usually ignored in previous works is **scalability**, which means that we can use the same design for larger data dimensions. In addition, most of the previous SVM accelerators only focused on binary classification or simple kernel functions. Multi-class classification in SVM is more challenging because it requires more computations. We addressed these challenges for SVM accelerator design in our thesis.

4. In **ANNs**, the main challenge in designing efficient hardware accelerators is the co-optimization of training hyper-parameters and hardware configurations. Despite the recent efforts in designing co-optimization frameworks, there are still so much opportunities for further enhancement. Specifically, a *truly* **multi-objective optimization** framework in the context of *multiple hardware configurations* has not been fully explored in the domain of HLS-based FPGA acceleration of ANNs. We elaborate further on this issue when introducing the proposed framework.

1.4 Scope and Outline

In this thesis, we focus on the design of domain-specific hardware accelerators to enhance the performance of recurrent algorithms used in a broad domain of applications. Machine Learning and biomedical Microwave Imaging are two main target applications. Although all the proposed accelerators can be used in the biomedical microwave application (as the main target), they have a broader range of applications and are useful in a variety of Machine Learning techniques. The main goal is to use specific hardware accelerators to improve the performance of these recurrent algorithms under various constraints (e.g., latency, power, resource usage). Designing such hardware accelerators requires the analysis of each algorithm and usage of specific hardware optimization techniques to accelerate the execution of the compute-intensive parts. For the hardware design and implementation, we use High Level Synthesis (HLS) and the target hardware device is an FPGA, whose type and size changes depending on the target application. In addition, obtaining the best performance in ML algorithms requires the optimization of training hyper-parameters and hardware configurations, which is also considered in this work.

In the following, the main contributions of this thesis and the methodologies to overcome the above-mentioned challenges are described:

1. Evaluation of widely-used algorithms in a broad domain of applications, including MI and ML.
2. FPGA acceleration of **3D FDTD** for multi-antennas microwave imaging (**Chapter 3**):
    - We proposed a spatial blocking approach to overcome the problem of memory access time.
    - To deal with the challenge of the additional polarization current, we could use HLS capabilities to efficiently process the extra computations by merging the corresponding loops on the polarization currents with the loops on the electric fields update.
    - We presented two versions of our FDTD accelerator: Small and Large designs with different resource usage and number of interfaces to meet memory bandwidth requirements and increase the flexibility.

- For the complex boundary conditions in our FDTD accelerator we merged the parallel loops in the boundary regions and used local memories whenever possible to store the required data. We implemented the computations of boundary equations in HLS by considering *CPML* conditions *in all directions*.
- Our single FPGA accelerator for FDTD could achieve 1.44× lower execution time per antenna compared to the best GPU design of the same algorithm. In our multi-FPGA design with 8 FPGAs and a typical number of 24 antennas, a 11.5x reduction of execution time can be achieved compared to the best GPU design. In addition, our design is more energy efficient than the conventional methods.

3. High level design of a flexible FPGA accelerator for Principal Component Analysis (**PCA**) (**Chapter 4**):

    - We presented an efficient block-streaming methodology to overcome the issue of large data dimensions and memory access time in PCA.
    - In contrast to most of the previous works, the covariance computation is included in our PCA accelerator design for large data dimensions.
    - Our proposed PCA accelerator is flexible because it can be used for different input sizes and FPGA targets.
    - We presented a more accurate floating-point and a faster fixed-point implementation to improve flexibility and efficiency.
    - Compared to a similar FPGA implementation of PCA using VHDL, our HLS design has a 2.3× improvement in the processing time, and a significant reduction of the resource usage. Compared to other HLS-based approaches, our design has a maximum of 2.5× speedup.

4. A new dataflow hardware architecture for Support Vector Machine (**SVM**) (**Chapter 5**):

    - We proposed a scalable hardware accelerator for SVM algorithm in FPGAs that can support different data dimensions while guaranteeing a high throughput.
    - Multi-class classification with recurrent SVM kernels are supported in our proposed accelerator.

- The level of hardware parallelism in our accelerator is adjustable thanks to the HLS-based configurations.
- In addition to the floating-point precision, we presented efficient implementation of fixed-point design for the SVM accelerator.
- A minimum of $10\times$ latency improvement compared to similar HLS-based and $4.4\times$ improvement compared to RTL-based designs can be achieved by our dataflow hardware architecture for SVM accelerator.

5. High level implementation and optimization of conventional **Neural Networks** in FPGAs (**Chapter 6**):

   - We introduced a new framework for the co-optimization of ML training and hardware design.
   - We used Multi-objective Bayesian Optimization on top of High Level Synthesis as a new approach for the co-optimization problem.
   - Instead of using a fixed hardware architecture, our framework supports adjustable HLS-based hardware configurations.
   - The Pareto set achieved with our framework outperforms those obtained with the other methods, with $1.7\times$ and $1.4\times$ improvement in execution time for the minimum error compared to *random* and *separate* methods, respectively.

6. Boosting the performance by using HLS-based hardware optimization techniques
7. Evaluation of each accelerator on a dataset obtained by microwave measurements to assess the feasibility of the proposed solutions for biomedical microwave applications.

In the next chapter, we introduce a background about the design of domain-specific hardware accelerators and their applications. In the following chapters, we explain hardware design methodologies for the domain-specific kernels that were introduced in Sec. 1.2 and compare them with the state-of-the-art related works. In addition, in Chapter 6, the co-optimization methodology for hardware and training is explained. The last chapter is dedicated to the conclusions, future works, and outputs of thesis and the publications.

Chapter 2

Background

2.1 Performance Analysis of Domain-Specific Accelerators

For the best design of an accelerator, the broadest possible range of applications must be covered, which makes it possible to accelerate more than one single application. Domain-specific instructions can be added to programmable processors to provide efficiency while preserving flexibility. In addition, we can build a parallel computer from domain-specific accelerators in order to expand the domain of applications.

To design a domain-specific accelerator, the designer must always find a trade-off between efficiency and generality. If the accelerator is designed for only one specific application, it results in the best possible efficiency, but with a limited range of use. On the other hand, designing a general-purpose processor, although is flexible, results in a weak efficiency. The best approach is to increase the range of applications as much as possible to keep the flexibility and not to lose the efficiency of a single specialized accelerator [1].

In another view, high-performance computing systems with DSAs must be carefully designed to meet stringent requirements. They require not only lots of computations as well as correct functional behaviour, but also must meet several quantifiable and often contradictory design metrics, which must be optimized simultaneously to achieve an efficient implementation. A design metric is a measurable feature of the system's implementation which can be interpreted in several ways. In the

following, different design metrics in the implementation of hardware accelerators are explained.

One of the most important design metrics is *performance* which can be further specified to capture one peculiar aspect of speed. For example, *Execution time* (or *Latency*), and *Throughput* (the rate at which a system can process data and receive new input[1]) are two representations of system performance. Another design metric is *Power* consumption which, if combined with execution time and considered in its *average* form, determines the lifetime of a battery, or when considered in its *peak* form it affects the cooling requirements of a chip. The *size* of computing devices is also important because the physical space and geometry required by the system must be as small as possible to fit on a given area, ranging from the silicon area of single-chip implementation to the form-factor of a board for a multi-chip design. A crucial design metric in the accelerators design is *Cost* which can be divided into *NRE*[2] *cost* (the cost of designing the system, after which any number of units of the system can be manufactured without incurring additional design cost) and *Unit cost* (the cost of manufacturing each copy of the system excluding the NRE cost). Designing a high-performance system usually requires a complex and long design procedure (hence, large NRE) and leads to using a large amount of hardware resources (hence, a high unit cost). *Flexibility*, as another design metric, is defined as the ability to change the functionality of the system without incurring heavy NRE cost. Finally, *Time to market* determines the amount of time required to design and manufacture the system to the point that the system can be sold to customers. Other design metrics can be found in [5].

The above-mentioned metrics are usually in contrast, and improving one of them leads to the degradation of the others. Fig. 2.1 shows an illustration of different design metrics which can be considered as a wheel with different pins. Pushing one pin will cause others to pop out [5]. A hardware designer must be able to optimize these metrics under various constraints.

[1] A more precise definition will be presented in the next chapters.

[2] Non-Recurrent Engineering

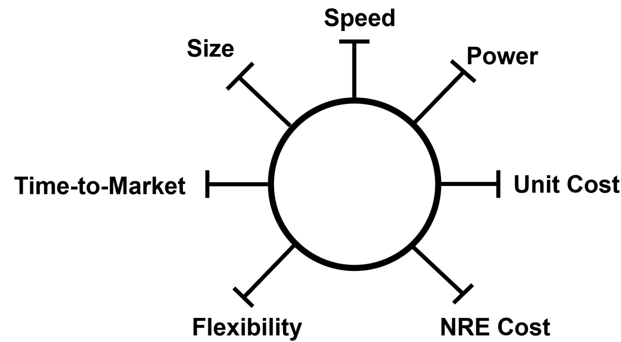


Fig. 2.1 Various design metrics in the implementation of a hardware accelerator

.

2.2 High Level Synthesis for the Design of Hardware Accelerators

To design a hardware accelerator in an FPGA, the traditional approach uses Hardware Description Languages (HDL) like VHDL or Verilog. Although this approach is still the predominant design methodology, it impacts the development time and the design effort. As hardware computing systems become more and more complex, designing an efficient hardware in RTL requires significant effort, which makes it difficult to find the best hardware architecture. In fact, the advances in chip integration capabilities have increased the complexity of embedded systems to such a level that their development time sometimes exceeds even their product lifetime. An alternative solution that is becoming more and more popular in recent years is the High Level Synthesis (HLS) approach. HLS raises the design abstraction level by using software programming languages like C or C++. Through the processes of scheduling, binding, and allocation, HLS converts a software code into its corresponding RTL description. The main advantage of HLS over HDL is that it enables designers to explore the design space more quickly, thus reducing the total development time with a quality of results comparable and often better than RTL design.

To understand the main steps in designing a hardware accelerator with Xinlinx FPGAs based on HLS, we introduce the hardware design flow in Fig. 2.2. Starting from a software code written in C or C++ in Vivado HLS tool, we design our hardware by applying efficient HLS *directives* to optimize our design. In the synthesis step,

the optimized code is compiled and translated into a netlist. In the last step in Vivado HLS, it is possible to run the C/RTL co-simulation to check if the generated RTL is functionally identical to the C++ code. Finally, an RTL IP could be generated that can be used in Vivado tool. The final steps in Vivado are Placement, route, implementation and generattion of the bit-stream.

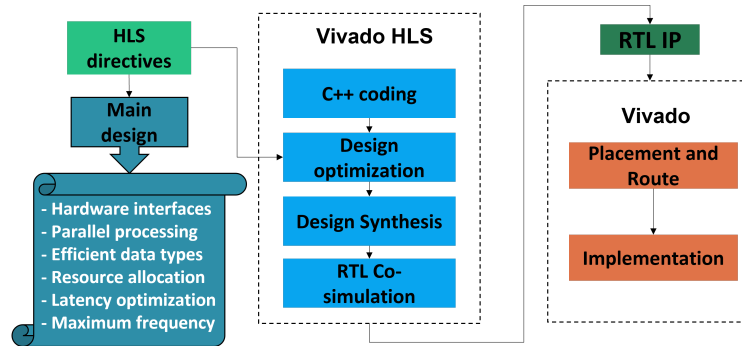


Fig. 2.2 Hardware design flow in Vivado

2.2.1 Design Optimizations in HLS

To introduce the principles behind the HLS-based hardware optimization techniques, a more abstract description of the high-level directives is presented in this part. In Figure 2.3 the most widely used optimization directives are illustrated with their corresponding hardware implementation. These directives include *Loop Pipelining, Loop Unrolling, Array Partitioning,* and *Dataflow.* These directives can be used to reduce the latency, increase the throughput, and make the best use of the hardware resources. Note that latency is the time required to produce the output of a computation starting from when the corresponding input is received. Throughput is the rate at which the outputs are produced (or the inputs are consumed) and is measured as the reciprocal of the time difference between the arrival of two consecutive outputs (or inputs). In the following, these HLS optimization directives are briefly introduced:

- Loop *Pipelining* allows multiple operations of a loop to be executed concurrently on different hardware resources, while those resources are used repeatedly over the various iterations of the loop.
- Loop *Unrolling*, instead, creates multiple instances of the hardware for the loop body, which allows some or all of the loop iterations to occur in parallel.

- By using *Array Partitioning* we can split an array, which is implemented in RTL by default as a single block RAM resource, into multiple smaller arrays that are mapped to multiple block RAMs. This increases the number of memory ports providing more bandwidth to access data.

- The *Dataflow* directive allows multiple functions or loops to operate concurrently. This is achieved by creating channels (FIFOs or Ping-Pong buffers) in the design, which enables the operations in a function or loop to start before the previous function or loop completes all of its operations. The Dataflow directive is mainly used to improve the overall latency and throughput of a design.

- *Arbitrary Precision (AP) data types* make it possible to use efficient number of bits for the data types. As opposed to the software C++ codes in general purpose processors which support fixed number of bits, using AP data types in HLS-based designs can reduce the resource usage and increase the performance.

- *Loop merging* provides the opportunity to execute the multiple loops in parallel which results in reducing the design latency.

- By using *Resource Allocation* directive, it is possible to allocate specific hardware resources for individual computational operations.

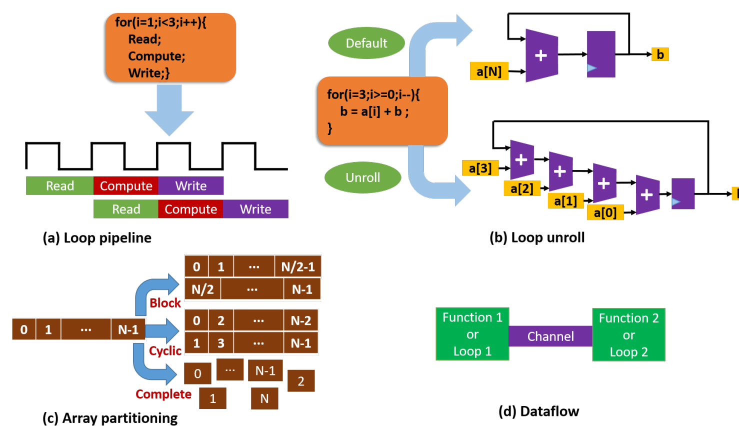


Fig. 2.3 Hardware optimization directives

2.2.2 HLS Design Trends in Selected Domains

Although there are several tools to design HLS-based hardware accelerators that are developed either commercially or by academia [6], we focus on *Vivado HLS*, a commercial tool that is used for Xilinx FPGAs.

There have been a growing interest in recent years in using HLS tools to design domain-specific accelerators. In this subsection, we introduce some of the relevant works as examples that used HLS for their hardware accelerator design. We will highlight the advantages of our accelerators compared to these works in the related chapters. For broader analysis of the literature, including both HLS-based and RTL-based designs, please refer to the corresponding chapter.

In the design of FDTD accelerators, the authors in [7] use an HLS tool called MaxCompiler to optimize their design. However, they used simplified boundary conditions. In [8], memory and power performance of FPGA accelerators for general *stencil* algorithms including FDTD have been investigated by using MaxCompiler tool. For the PCA hardware design, an HLS-based design was introduced in [9]. Schellhorn et al., presented another PCA implementation on FPGA in [10] by using HLS. However, the EVD part could not be implemented in hardware due to the limited available resources. In [11], an HLS design for SVM acceleration is proposed and is extended in [12, 13]. Due to the local storage of SVM coefficients, these works could be tested on small-scale problems. Finally, there are numerous works for the design of ANNs in hardware using HLS, one of which is hls4ml ([14]). However, the efficient co-design of ANNs training and hardware configurations has not been fully explored.

2.3 Applications of Domain-Specific Accelerators

In this section we introduce some of the applications that are relevant for this thesis in which domain-specific accelerators can be used. It is important to understand the main objectives of each application to obtain the specifications and requirements of the accelerators used in these applications. We focus on two research areas as examples of domain-specific applications, which are *biomedical microwave techniques* and *Machine Learning (ML)*. Note that ML is a general research field and has a wide range of applications one of which could be biomedical microwave applications.

2.3.1 Biomedical Microwave Techniques

Microwave Imaging (MI) is a technique to observe the internal structure of an object by using electromagnetic fields at microwave frequencies. It has received considerable attention in medical applications due to the fast diagnosis and high safety for the patient [15]. Unlike other medical imaging modalities, such as CT-scan or X-ray, MI has the advantage of using non-ionizing radiations. This is because microwave radiations are electromagnetic waves at frequencies between $300MHz$ and $300GHz$, which is between radio and infrared frequency ranges. Although ultraviolet, X-ray, and gamma-ray are commonly used in medical imaging, they have much higher frequencies compared to microwaves, which makes them ionizing radiations causing several health risks in the medical imaging devices (as opposed to non-ionizing microwave radiations). Having lower frequencies, microwave radiations have low penetration depth, and it is more challenging to use them for medical diagnosis. Although Magnetic Resonance Imaging (MRI) does not use any radiation either, the MRI instruments are bulky and very expensive and can be used only inside the hospitals. The low-cost, small-scale, portable, and non-invasive characteristics of an MI device make it one of the most promising medical imaging techniques [16].

A Microwave Imaging system uses microwave radiations emitted from a set of antennas arranged in a proper geometry around a given body part. The reflections of these radiations, which are created at the interface between tissues exhibiting different dielectric properties—i.e., a so called *dielectric contrast*—are captured and converted to an output image according to a specific algorithm to highlight, for example, an anomaly within a body part.

The development of an MI device requires collaboration between researchers working on different parts of the device. This is the main goal of the EMERALD project (www.msca-emerald.eu), an MSCA training network funded by the European Commission under the H2020 program. The research work at the basis of this thesis has been done in the framework of this project, in which the various MI techniques used by researchers very often require hardware acceleration, as it will be explained later. Before delving deeper into the various accelerators required by the MI techniques, let us introduce the main components of an MI device.

An MI system consists of several components that are depicted in Fig. 2.4 for the particular case of brain stroke monitoring. A similar figure can be depicted for other

MI applications such as breast cancer detection. The first component of the system is a set of antennas that are arranged around the body part under consideration (i.e. head or breast). These antennas are connected to a switching matrix, which controls which pairs of antennas are active during the measurements. A Vector Network Analyzer (VNA) is connected to the switching matrix and measures the microwave radiations in the form of a scattering matrix. Each element (i, j) of the scattering matrix describes the relation between the wave emitted by the ith antenna and the wave received by the jth antenna. Therefore, if the number of antennas is N, the scattering matrix is a symmetric matrix with $N \times N$ elements showing the relative transferred energy between each pair of antennas. The last component of the device is a processing system that is used to convert the microwave measurements into the image pixels (image reconstruction), or to analyze the microwave measurements to detect anomalies for medical diagnosis (anomaly detection). Precisely the algorithms used in such processing system may require acceleration.

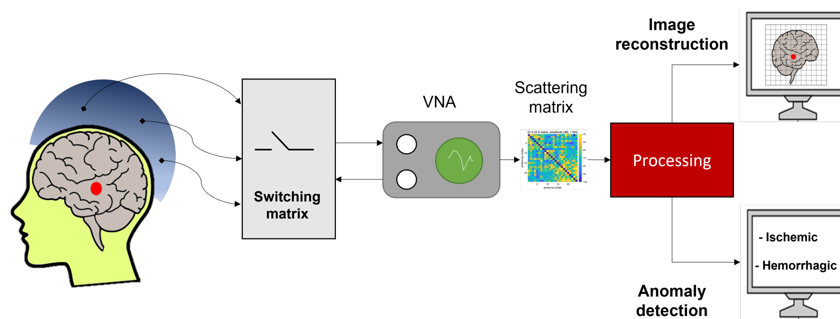


Fig. 2.4 General diagram of a Microwave Imaging system.

When the microwave electromagnetic fields are radiated to the body, the reflected waves are scattered due to the difference between the dielectric properties of the body tissues. This phenomena is termed *scattering*. Related to the scattering phenomena are two heavy computing problems that are implemented in the processing system, especially in iterative algorithms. One is termed *forward scattering* and consists in computing the scattered field via electromagnetic simulations starting from an electromagnetic model of the objects (i.e. the body tissues), which have to be completely known with their dielectric properties. The second one is termed *inverse scattering,* in which the goal is instead to retrieve the properties of an unknown object from the known scattered field. Often forward and inverse scattering are combined in iterative optimization problems in a way that inverse scattering invokes

repeatedly the forward step until convergence, but it is also possible that inverse scattering uses a different technique than forward scattering to obtain the unknown parameters. Medical MI suggests therefore different algorithms to solve the problem of inverse scattering, which will be briefly discussed in Section 2.4.

Microwave Imaging in Breast Cancer Detection

Breast cancer is the most common type of cancer and the leading cause of death in women worldwide [17]. MI is a trending technology in biomedical research community for breast cancer detection. Although mammography is the standard imaging modality in the diagnosis of breast cancer, its ionizing radiations calls for safer imaging techniques. The higher dielectric contrast of the tumor tissues compared to normal breast tissues allows for the breast tumor detection in an MI system, as shown in Fig. 2.5.

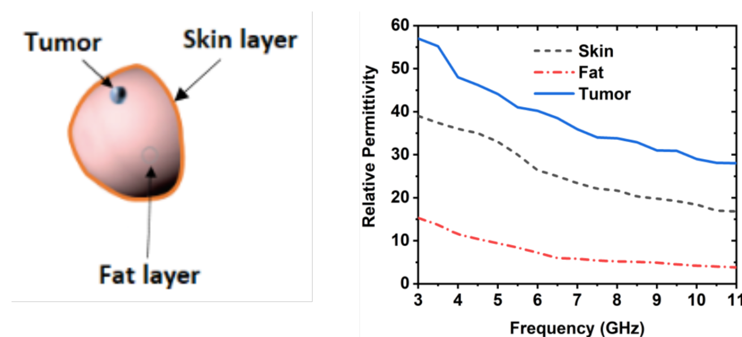


Fig. 2.5 Dielectric properties of breast tissues. Difference in relative permittivity allows for tumor detection in MI ([18]).

There have been several works in recent years concentrating on the development of an MI system in the application of breast cancer diagnosis [19–22, 18, 23].

Microwave Imaging in Brain Stroke Monitoring

Brain stroke is a cerebrovascular disease affecting a large percentage of people worldwide, which can lead to permanent disabilities or even death. Early diagnosis of brain stroke, either ischemic or hemorrhagic, helps in finding the right treatment. Recently, there has been a growing interest in using MI technology for brain stroke monitoring [24–29]. In Fig. 2.6, the setup of a recent MI device is shown, which

consists of a set of antennas that are placed around the head and P positions are considered for the locations of the stroke. The exact location of the stroke can be detected by using an MI inverse scattering solution.

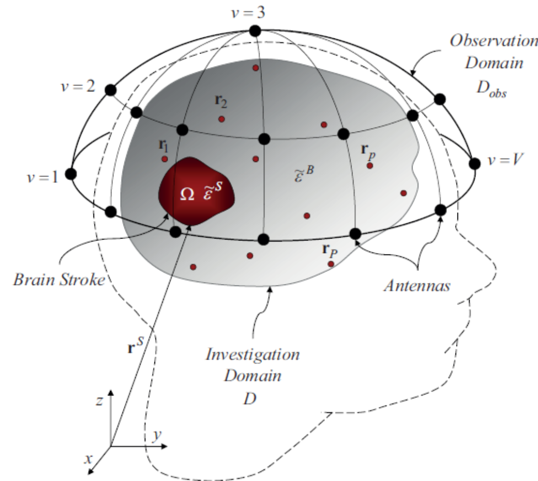


Fig. 2.6 Brain stroke detection with MI system containing V antennas and P candidate locations for the stroke positions (r_1 to r_P) ([29]).

2.3.2 Machine Learning (ML)

Machine Learning algorithms are applicable to a variety of applications including MI. Due to the extensive range of applications of ML, we consider a general framework for ML algorithms which is not limited only to microwave imaging. There are two steps in any ML model that are shown in Fig. 2.7 which shows how the training and inference of an ML model is performed. Any ML model contains a set of training parameters that are obtained during the training step. The training parameters depend on the type of ML approach, including weights and biases for a Neural Network, coefficients for linear regression models, or support vectors for Support Vector Machines (SVMs). Once these training parameters are obtained, they can be used in the inference step to predict the outputs of the model for the new input data. Estimation of the mapping between inputs and outputs are termed classification (for discrete outputs) or regression (for continuous outputs). In Sec. 2.5, different ML models are explained in more detail. Note that using data *pre-processing* and *feature extraction* techniques before the training step leads to higher accuracy and, most of the times, these steps are also required and can be considered part of an ML pipeline.

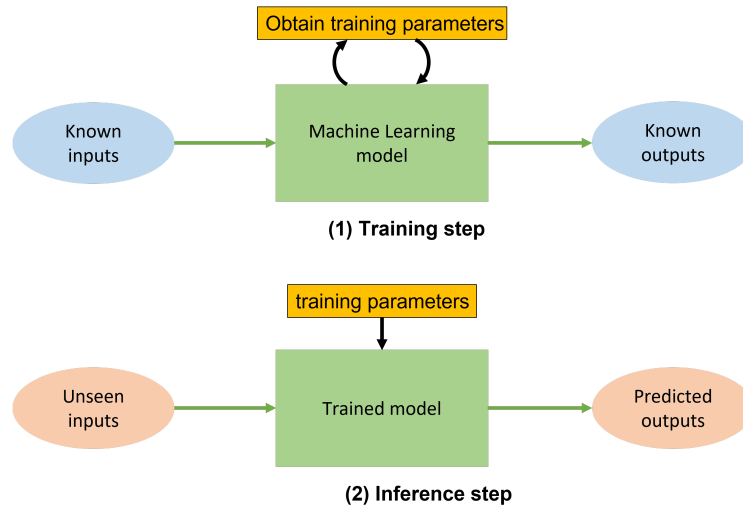


Fig. 2.7 Machine Learning training and inference steps.

Here is an example of an application domain in which feature extraction is required before the actual classification.

Mahine Learning in Hyper-spectral Imaging (HI)

Hyperspectral Imaging (HI) has a broad range of applications, from medical imaging to satellite remote sensing. HI sensors acquire digital images in several narrow electromagnetic spectral bands. This enables the construction of a continuous radiance spectrum for every pixel in the scene. Thus, HI data exploitation helps for example to remotely classify the ground materials-of-interest based on their spectral properties acquired from a satellite, or to classify tumor lesions (i.e., benign vs malign) also based on their electromagnetic radiation in specific spectral bands [30]. HI data are organized in a matrix of size $R \times C$ in which R is the number of pixels in the image and C is the number of spectral bands. A 3D view of the HI data is shown in Fig. 2.8 for better illustration.

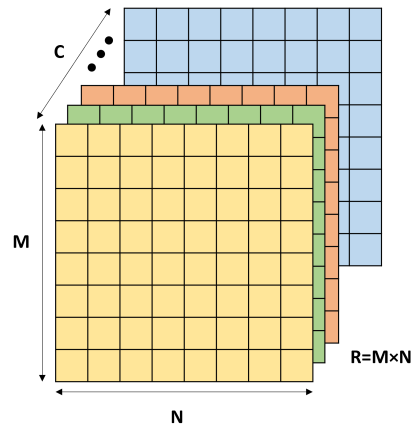


Fig. 2.8 Hyper-spectral Images (HI) with C bands. Each band has $R = M \times N$ pixels.

Due to the large data dimensions in HI, the direct processing of these data via ML is not directly feasible, hence feature extraction techniques are used to remove redundant information in different bands. Note that in our notations, we use interchangeably the terms R (Rows) and pixels, as well as C (Columns) and bands.

Machine Learning in Other Applications

The ML algorithms considered in this thesis can be used in a variety of applications. Due to the wide range of applications of ML, we briefly mention a few of them to highlight the flexibility of our developed methodologies for ML algorithms. In addition to the medical diagnosis, ML can be used in several areas including image, speech, or pattern recognition, object detection, face detection, and Natural Language Processing (NLP), as shown in Fig. 2.9. In [31] and [32] recent trends and research directions of Machine Learning and Deep Learning (DL) are thoroughly described.

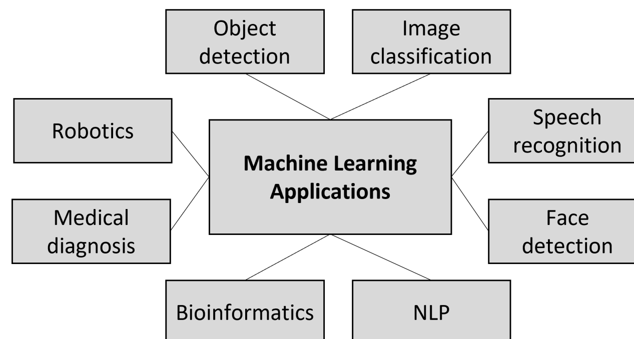


Fig. 2.9 Machine Learning Applications.

ML algorithms in MI have different goals. Some algorithms aim at a qualitative classification, for example by identifying the type of a lesion, its size and its approximate position within a body part. Other algorithms aim instead at a more quantitative information, like the one obtained by solving the inverse scattering problem, which consists in finding the unknown relationship between the system inputs (e.g. microwave measurements) and the desired outputs (e.g. scatterer location and shape, or its exact dielectric values). For this purpose, a training data set is required containing a known set of inputs/outputs which can be obtained for MI applications by making several measurements from different scatterers with different locations and shapes. Once the ML model is trained, it can be used for the new input data to predict the system output.

2.4 Microwave Imaging Algorithms

To understand how an MI system produces the image of the internal structure of the objects, it is required to be familiar with the details of the inverse scattering problem and the algorithms to solve this problem.

2.4.1 Inverse Scattering Problem

The purpose of medical microwave imaging is to determine the electromagnetic properties of an object under test (scatterer) positioned inside an *investigation domain* by means of measurements made with a number of antennas positioned on a *Measurement domain*. Fig. 2.10 illustrates a schematic of an MI imaging setup.

The constitutive parameters of the scatterer are the complex permittivity, ε, and the magnetic permeability, μ. It is worth mentioning that ε_b, μ_b, the parameters of the background (everywhere outside the scatterer), are assumed to be known. In this setting, the electric field at any given position can be expressed as:

$$E^{total} = E^{inc} + E^{scat} \tag{2.1}$$

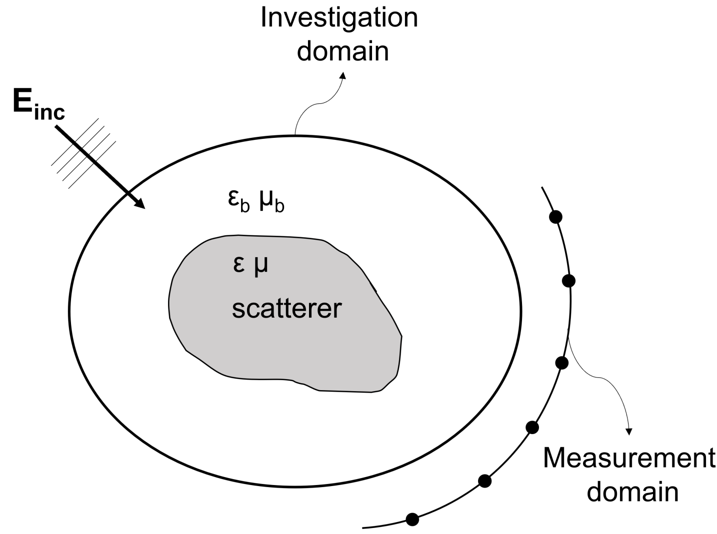


Fig. 2.10 Microwave Imaging setup. Inverse scattering is the problem of finding ε and μ from the microwave measurements in the measurement domain.

in which E^{total} is the total field at the Measurement domain, E^{inc} is the incident field (i.e. the field when there is no scattering object present), and E^{scat} is the scattered field (i.e. the additional field caused by the scatterer). If the contrast between the scatterer in the observation points r and the known background is defined as $\chi(r) = -i\omega(\varepsilon(r) - \varepsilon^b)$, there is a non-linear relationship between E^{total}, E^{scat}, and χ through the *Green*'s function G:

$$E^{scat}(r) = i\omega\mu_0 \int_V G(r,r').E^{total}(r').\chi(r')dv' \tag{2.2}$$

Inverse scattering problem in MI can be defined as obtaining the unknown contrast function χ from the total measurements made by the antennas in the Measurement domain. Due to the non-linear relationship between the scattered field and the contrast function χ, the inverse scattering problem is inherently non-linear. However, with some considerations, it is possible to convert it to a linear problem. The typical approach for this linearization is based on Born approximation. We can write equations 2.1 and 2.2 as the following expression:

$$E^{total}(r) = E^{inc}(r) + i\omega\mu_0 \int_V G(r,r').E^{total}(r').\chi(r')dv' \tag{2.3}$$

As we can see, eq. 2.3 can be written as a recurrent series, that is termed Born series. The first-order approximation of the Born series states that we can approximate the total field (E^{total}) in the right-hand side of eq. 2.3 with the incident field (E^{inc}). This will convert the problem to a linear one reducing the complexity of the solution. This leads to a system of linear equations which can be written as the following term after discretization over a finite spatial basis:

$$E^{scat} = S_{Born}\,\chi \tag{2.4}$$

in which S_{Born} is the linear Born approximation of the scattering operator. All the terms in eq. 2.4 are discrete matrices. Therefore, the general formulation of a linear minimization problem can be used to compute the vector of unknown contrast function (χ):

$$min_{\chi}||E^{scat} - S_{Born}\,\chi|| \tag{2.5}$$

One of the typical approaches to solve the above linear problem is to use Singular Value Decomposition (SVD) of the scattering matrix (S_{Born}) and approximate the unknown contrast by discarding the least significant singular values and associated vectors, which is often called Truncated SVD (TSVD):

$$\chi = \Sigma_{i=1}^{N} \frac{(u_i^* E^{scat})}{\lambda_i} v_i \tag{2.6}$$

in which λ_i, u_i, and v_i are the ith singular value, left singular vector, and right singular vector of S_{Born}, respectively, and are obtained according to the SVD of the scattering matrix:

$$S_{Born} = U\lambda V^* \tag{2.7}$$

Although approximation of the inverse scattering problem with a linear one reduces the accuracy of the solution, it is well suited for the weak scattering objects and low-contrast scenarios. However, for high-contrast objects, more precise methods are required which consider the non-linearity of the inverse problem. Non-linear algorithms for inverse scattering are more accurate, but they are computationally more expensive. These algorithms for solving the inverse scattering problem in MI will be discussed in more detail in the next subsections, which are usually divided into two categories that are *Qualitative* and *Quantitative* algorithms. In Qualitative algorithms, either the shape and location of the scatterer are identified, or a linear approximation of the problem is solved. In Quantitative algorithms, however, the

exact dielectric values of the scatterer are estimated without any approximation. In the following, these two categories are explained in more detail.

2.4.2 Qualitative Imaging

Inverse scattering is inherently a non-linear problem. However, in qualitative imaging, it is approximated as a linear problem. The simplest approach for this linearization is Born approximation. In this category, the presence and shape of the anomaly (i.e. brain stroke) is detected.

Linear Sampling method (LSM) [33]–[34], Factorization method [35], Truncated Singular Value Decomposition (TSVD) [36], Time Reversal (TR) techniques such as multiple signal classification (MUSIC) algorithm [20] and Eigenvalue Decomposition (EVD) of the TR operation [37], Beamforming approaches [38] and several *Radar*-based methods [39] are among these linearized qualitative algorithms. Matrix multiplication is one of the critical parts of these algorithms due to the large data dimensions. Although qualitative methods are not highly accurate, they are well suited for weak scattering objects and low-contrast scenarios. Furthermore, in some algorithms it is possible to obtain quantitative results when such approximation holds.

2.4.3 Quantitative Imaging

In quantitative imaging, the exact values of image pixels are reconstructed by solving the non-linear inverse scattering problem. These methods are more accurate because they consider the non-linearities of the problem. However, they are computationally more intensive.

Some of the non-linear quantitative algorithms are Contrast Source Inversion [40], inexact Newton methods [41], and DBIM-TwIST [42] which belong to the Microwave Tomography deterministic approaches. In addition, stochastic techniques including Simulated annealing and Genetic Algorithm are among the other quantitative MI methodologies [43].

Iterative non-linear image reconstruction

One of the non-linear quantitative approaches in solving MI inverse scattering problem is iterative image reconstruction shown in Fig. 2.11. Starting from an initial guess of the dielectric profile, the *forward solver*, often implemented using the Finite Difference Time Domain (FDTD) approach [44], computes the electromagnetic fields. The output of the forward solver is compared with the actual microwave measurements and, based on the error, the dielectric profile is updated with a specific inverse scattering algorithm. The inversion part is based on DBIM-TwIST algorithm [42] and the forward solver is FDTD algorithm.

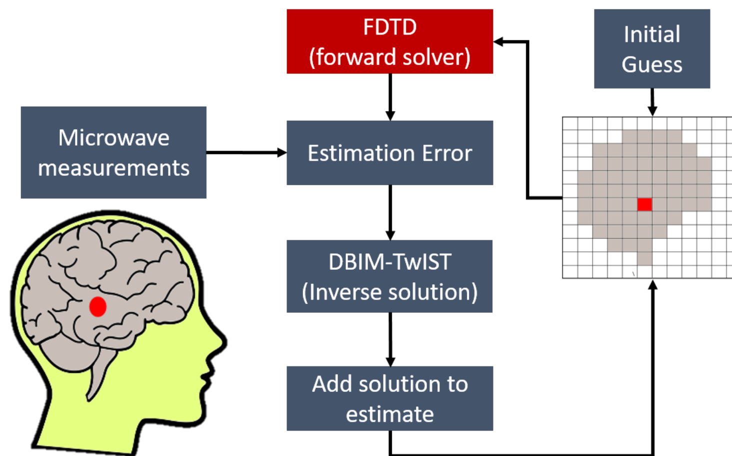


Fig. 2.11 General diagram of a non-linear image reconstruction iterative algorithm in MI, with the compute-intensive FDTD step.

2.5 Machine Learning Algorithms

In this section, the processing steps in Machine Learning approaches are explained in more detail. As shown in Fig. 2.12, there are three main processing steps which are *Pre-processing*, *Feature extraction*, and *Classification* steps. Note that Deep Neural Networks (DNNs) execute the feature extraction and classification steps in the different layers of the network. Section 1.2.4 will cover more explanation about the Neural Networks (NNs). In the following, each processing step in Machine Learning is thoroughly described.

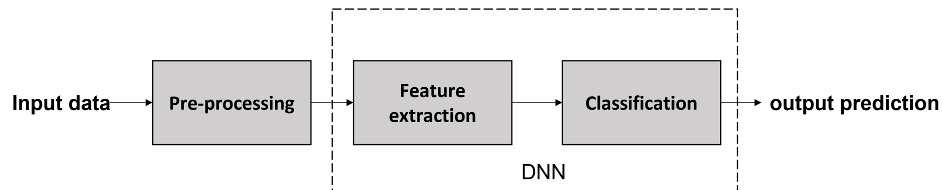


Fig. 2.12 Three data processing steps in Machine Learning. Note that in DNNs, feature extraction and classification are implemented in different layers of the network.

2.5.1 Preprocessing

Data preprocessing is an essential step in Machine Learning which is used to enhance the quality of raw input data which are usually inconsistent, noisy, and inaccurate. Preprocessing helps ML models to better learn and extract information from data. Different data transformation methods can be used in preprocessing step to deal with the imperfect and noisy data. Data normalization, standardization, and scaling can be used for noise treatment. Another method that is used to deal with missing values in data is to replace them with the mean (or median, mode, ...) of the data.

2.5.2 Feature Extraction

The second step in ML is to extract useful information from data. This step can also be used in data dimensionality reduction methods in which the most important information from data is preserved by extracting the most significant features from data. One of the widely used algorithms for feature extraction and dimensionality reduction is Principal Component Analysis (PCA) which is thoroughly explained in Section 1.2.2. Other examples of feature extraction techniques are Neural Networks, such as Auto-encoders, or convolutional layers in DNNs.

2.5.3 Classification

The last step in ML is the classification of input features (if the output is continuous, it is termed “regression”). Classification algorithms use input features in training data to predict the probability that the data will fall into one of the predefined categories. Different algorithms can be used for classification in ML. Support Vector Machine (SVM) is one of the popular classification techniques that will be covered in

section 5.1. Other examples of classification algorithms are Decision Trees, Random Forests, and K-Nearest Neighbours (KNN). In addition, DNNs are widely used for both feature extraction and classification.

Chapter 3

FPGA Acceleration of 3D FDTD for Microwave Imaging using HLS

Microwave Imaging (MI) uses microwaves emitted and captured by several antennas to create an image of the inner dielectric profile of an object. It has attracted attention among biomedical researchers due to its low-cost, non-ionizing and non-invasive characteristics. Medical diagnosis in MI is based on the contrast between the dielectric properties of normal and anomalous tissues [45].

MI solves the electromagnetic *inverse-scattering* problem, which is inherently non-linear and makes the reconstruction a challenging task. Although approximate linear methods have been used [39][20][46], they have limited accuracy especially when the object is highly heterogeneous. More accurate, non-linear approaches solve the inverse problem by updating the dielectric estimation iteratively, which results in a high execution time. The high execution time comes from FDTD, for which several hardware accelerators have been proposed, as covered in Sec. 3.1. Although a GPU implementation is a natural choice, depending on the complexity of the problem—primarily number of elements in the volume and number of antennas—a 3D image reconstruction can still take hours to finish. This motivated us to design an alternative hardware accelerator for the MI algorithm developed by Kosmas et al. [42].

The MI algorithm was originally coded in MATLAB with the forward part accelerated by a Tesla GPU using an efficient commercial FDTD software library, *Acceleware* [47]. This code keeps the GPU fully busy by efficiently parallelizing

the execution on the number of elements in the 3D volume. Since there is no room left to further parallelize on the number of antennas, the GPU code of the FDTD is executed sequentially for each of them. In MI systems with tens of antennas [46], leveraging instead the antenna parallelism could be the key to reducing the overall execution time from hours to minutes.

Although one obvious solution is to parallelize the execution on many GPUs, this is impractical for various reasons, including cost, form factor, and overall power consumption. Instead, an implementation on a *Multi-FPGA platform* can offer an equivalent performance at a fraction of cost and power, not to mention the much more manageable size and weight. A GPU implementation, however, can still outperform FPGA platforms with limited capacity.

For these reasons, we developed a 3D FDTD accelerator using a high-level approach, so that the code can be both implemented in FPGA using a High-Level Synthesis (HLS) flow, or easily changed to be executed in a GPU. This is possible because current FPGA design flows accept portable C/C++ high-level descriptions, which are enriched with specific directives, termed *pragmas*, for generating the desired Register-Transfer Level (RTL) code for FPGA hardware implementation. This high-level approach allows to explore the design space by changing the pragmas in a more efficient way compared to RTL design. In [48], a comprehensive analysis and the implications of using several HLS optimization transformations (including the HLS pragmas) have been presented for High-Performance Computing applications. In [49] and [50], the efficiency and performance of HLS-based design space exploration are explored. In [51], a fast HLS simulator is introduced to accelerate the hardware simulation process, and in [52] and [53], new methodologies are proposed for the optimum selection of HLS directives.

One of the design challenges for FDTD is the optimization of the memory access to a large amount of data, which is complicated by the relatively low amount of on-chip memory. While many previous publications considered a less challenging 2D FDTD, we focus instead on a full 3D implementation. To cope with the memory access issues in 3D FDTD, the few previous works on the subject use specific blocking methods to read blocks of data from the external memory, which is an approach that we also use in this work.

However, previous 3D FDTD works use simplified Periodic Boundary Conditions (PBC), which is not an accurate approach for some MI problems, but simplifies

the hardware design. Improving the accuracy calls for more appropriate, but more difficult to implement in hardware, boundary conditions like Convolutional Perfectly Matched Layer (CPML), which is used in few works and in a limited way. Finally, previous works on 3D FDTD do not consider *dispersive* materials, which leads to more complex equations with dependencies that result in less straightforward parallelization. To the best of our knowledge, we are the first to propose a full-fledged 3D FDTD in FPGA that implements both CPML boundary conditions in all directions and uses an exact model for dispersive materials. We propose two possible FPGA implementations that use a different amount of on-chip memory, which creates a trade off between performance and resource usage.

In summary, the following is the list of our contributions:

- We propose the first FPGA accelerator for 3D FDTD integrated in an MI algorithm for medical applications.
- This is the first 3D FDTD accelerator to fully model dispersive materials, which makes the FPGA design more challenging.
- The CPML boundary conditions for 3D FDTD are used for all directions in contrast to previous accelerators designed with a high level approach that either do not consider CPML or consider periodic structures with CPML conditions only for one direction.
- Two hardware architectures with different characteristics are proposed and their pros and cons are analyzed.
- The entire hardware is designed using a High Level Synthesis (HLS) tool and several specific hardware optimization methods are used to design an efficient hardware.
- Both single- and multi-FPGA platforms are analyzed that can be used to accelerate FDTD with multiple antennas.
- The GPU implementation derived from our 3D FDTD code has a comparable performance with a commercial GPU implementation.

In the remainder of this chapter, we discuss the related work in Sec. 3.1 and the principles of FDTD for MI in Sec. 3.2, present the FPGA hardware accelerator in

Sec. 3.3 and related results in Sec. 4.4. In Sec. 3.6 we discuss the challenges of the design and an overview of the solutions, and finally, we conclude in Sec. 4.5.

3.1 Related Work

Several FDTD accelerators proposed in the literature are based on GPUs. For instance, in [54] an implementation based on CUDA associates each thread to a cell in the FDTD grid and obtains the same accuracy of the CPU design with a speed-up ratio proportional to the grid size. Other GPU-accelerated versions of FDTD are proposed in [55][56][57].

High power consumption of GPUs draws attention to FPGA implementations. In [58] an FPGA accelerator for 2D FDTD is designed using OpenCL for two hardware platforms. The authors apply several OpenCL pragmas to create deeply pipelined loops. However, they do not consider the impact of boundary conditions. The FDTD hardware accelerator in [7] uses a HLS tool called MaxCompiler developed by Maxeler technologies. In their work, the authors investigate different boundary conditions, including PBC and CPML. However, their design can be used only for periodic structures where one can model the entire simulation space by a single periodic cell. In this cell, CPML conditions are applied only to the top and bottom boundaries, and PBC conditions are used for the other four boundaries. In contrast, we consider CPML in all directions as required by the MI application, at the cost of a much more complex design.

Takei et al. present an OpenCL-based design for 2D FDTD on FPGA [59]. To reduce the global memory access, they used an overlapped tiling method that can locally store small blocks of data. Despite lower power consumption compared to GPU, the processing time could not be reduced for large grid sizes. Waidyasooriya et al. in [60] extend the work in [59] to 3D FDTD by pipelining multiple FDTD iterations. Although they achieve better performance than CPU- or GPU-based designs, they only consider periodic structures for the boundary conditions. In addition, they simplify the FDTD update equations by ignoring the polarization current, hence reducing the required memory bandwidth. Recently, in [61] an FPGA design for 3D FDTD that considers CPML boundary conditions has been proposed, although the authors do not consider the impact of dispersive materials and polarization currents. In addition, they use Verilog to design their hardware at

RTL, which increases the design and development time compared to the HLS-based design and makes less efficient the design space exploration.

FDTD can be seen as a *Stencil* computation. In stencils, the elements of a multi-dimensional grid are updated iteratively based on the neighbouring cells using a fixed pattern. The main bottleneck in both GPU and FPGA designs for stencil computation is the data transfer time between global and local memories. The common approach to alleviate this problem is to use *spatial* or *temporal* blocking. In the former, a spatial block of data is stored in on-chip memory to reduce the access time, and in the latter, different time steps are pipelined for further parallelization. Regarding stencil acceleration in FPGAs, there have been extensive research works in recent years. In [62], Waidyasooriya et al. extended their previous FPGA accelerator to a general stencil computation by increasing the degree of parallelism. In addition to pipelining multiple iterations, they could compute multiple grid cells in parallel. However, they did not report results for 3D FDTD with complex boundary conditions like CPML. In [63], another FPGA design for 3D stencils using OpenCL uses a combination of spatial and temporal blocking methods. In [64], [8], memory and power performance of FPGA accelerators for general stencils have been investigated. Other successful designs for stencil acceleration have been presented in [65] and [66]. Although some of the above works have considered FDTD as a benchmark for stencil computation, they analyzed simplistic scenarios that cannot be adapted to the special requirements of FDTD as used in MI. For example, simple boundary conditions like Dirichlet [64][8][66] or PBC [63] cannot be used in MI. *Polybench*, a benchmark suite used in some stencil accelerators, like [65], does not include a full 3D FDTD as only the Transverse Electric (TE) mode is considered (other directions of the fields are ignored). In addition, not modeling dispersive materials as done in [62] can improve the hardware acceleration (e.g., with deeper pipelining on time iterations), but cannot be done in MI applications.

3.2 FDTD in Microwave Imaging

3.2.1 Background

The main equations in FDTD for MI are the time-domain Maxwell equations for dispersive and lossy materials:

$$\nabla \times H = \frac{\partial D}{\partial t} + \sigma_e E + J_S \tag{3.1}$$

$$D(\omega) = \varepsilon(\omega)E(\omega) \tag{3.2}$$

$$-\nabla \times E = \frac{\partial B}{\partial t} + \sigma_m H + M_S \tag{3.3}$$

$$B(\omega) = \mu(\omega)H(\omega) \tag{3.4}$$

H and E are magnetic and electric fields; B and D are magnetic and electric flux densities; M_S and J_S are magnetic and electric current densities (zero in the following); σ_m and σ_e are magnetic and electric conductivity; μ is magnetic permeability and ε is electric permittivity.

Most biological materials in MI have permeability close to that of free space, $\mu = \mu_0$, the permeability of free space. Thus, $B = \mu_0 H$ and (3.3)-(3.4) can be merged into one equation ($-\nabla \times E = \mu_0 \frac{\partial H}{\partial t} + \sigma_m H + M_S$). On the contrary, the frequency dependency of ε in dispersive materials must be modeled, typically with the Debye model that we also use here. Taking this into account, Eqn. (3.2) can be written as: $D = \varepsilon_0 E + P$, in which ε_0 is the free-space permittivity and P is the *polarization* vector that is proportional to E. Polarization current, J_P, is the time derivative of P: $J_P = \frac{\partial P}{\partial t}$.

FDTD solves the equations above based on finite difference approximations. The 3D volume is divided into cuboids called Yee cells [67] in such a way that each magnetic field is surrounded by four electric fields and vice versa. This results in two main *update* equations for electric and magnetic fields, respectively. Compared to the simpler case of non-dispersive materials, however, an additional variable accounting for the polarization current (J_P) appears in the update equation for the electric field only. The FDTD algorithm for a Debye model repeats the following two steps at each time step [44]:

1. update magnetic fields;
2. for each electric field component:
    (a) update electric fields and store them in a new variable E';
    (b) update polarization currents based on the new and old values of electric fields;
    (c) update the old fields with the new ones ($E = E'$).

3.2.2 Boundary Conditions: CPML

FDTD is used to obtain the propagation of the electromagnetic waves in the simulation space. Due to the finite size of this space, the propagation must be terminated in the "Boundary Regions". Therefore, the update equations in these regions must be modified by adding proper boundary conditions. Dirichlet conditions consider that the fields in the boundaries are zero, while PBC considers the fields to be repeated after a fixed number of cells. These conditions will create unwanted reflections from the boundary regions towards the inner simulation space. CPML is a more complex boundary condition that eliminates these reflections by letting the propagation be absorbed in the boundary region.

To better understand the following description of the FDTD code and the role played by the boundary conditions the exemplifications in Fig. 3.1 are helpful. The figure shows all the cells and identifies two boundary planes for each *direction* (x, y, z) using a different coloring. The figure refers to the magnetic field and shows that, for instance, the components H_x and H_z must be updated in the two planes of the y direction (called *front* and *back faces* in the following), while the components H_y and H_z must be updated in the two planes of the x direction (*left* and *right faces*), and so on. A similar figure can be used for the electric field.

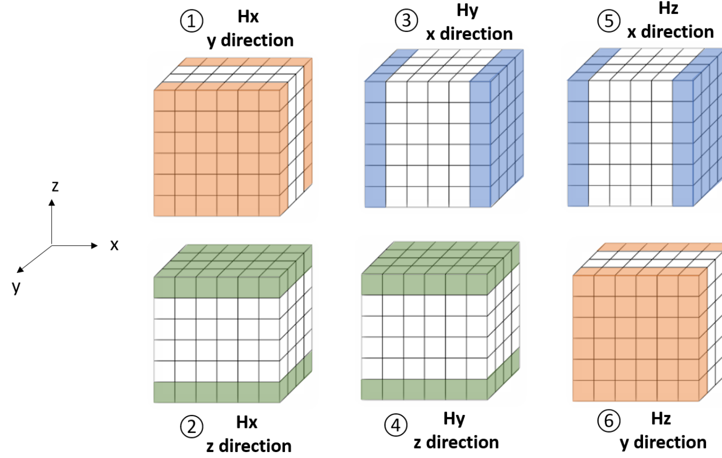


Fig. 3.1 Boundary regions for H field in 3D FDTD.

3.2.3 FDTD Pseudo-Code

Each FDTD update equation can be written in a general form like the following equation for H_x:

$$H_x = a_{hx}H_x + b_{hy}(E_y^{+x} - E_y^c) + b_{hz}(E_z^{+x} - E_z^c) + d_{hx}(\Psi_{Hxy} + \Psi_{Hxz}). \quad (3.5)$$

E^c and E^{+x} are the electric field of the central cell being calculated and of the next cell, respectively; Ψ_{Hxy}, Ψ_{Hxz} are used in the boundaries only and can be considered to be zero in the main cells. All the other terms are constant with a spatial dependency. When computing the cells in the boundary layers in y direction, Ψ_{Hxz} is zero and when updating the cells in the boundaries of z direction, Ψ_{Hxy} is zero. In the overlay of boundary cells in y and z directions, both terms are present. Therefore, the update equation for H_x can be divided into different regions including the main cells, boundary cells of y direction (front and back) and boundary cells of z direction (top and bottom). Separate loops must be considered for each region to obtain the final output.

These separate loops are described by the FDTD pseudo-code in Alg. 1. Notice the difference between *Update H* and *Update E* equations due to the polarization currents $J_{P\{x,y,z\}}$. E^c, H^c are the electric and magnetic fields of the central cell being calculated; (E^{+x}, E^{+y}, E^{+z}) and (H^{-x}, H^{-y}, H^{-z}) are the electric and magnetic fields

Algorithm 1: 3D FDTD Pseudo-code

for $s \in$ *Antennas* **do** `// Loop over antennas`

  for $c \in$ *Domain Cells* $\cup$ *Boundary Cells* **do** `// Initialize at t=0`

    $E_{\{x,y,z\}}(s,c,t=0)=0, H_{\{x,y,z\}}(s,c,t=0)=0$

  `// From now on (s,c,t) omitted for readability`

  for $t=1$ *to* $T_{\max}$ **do** `// Loop over time steps`

    `// Update H: loop over all cells`

    for $c \in$ *Domain Cells* $\cup$ *Boundary Cells* **do**

      $H_x = a_{hx}H_x + b_{hy}(E_y^{+x}-E_y^c)+b_{hz}(E_z^{+x}-E_z^c)$

      $H_y = a_{hy}H_y + b_{hx}(E_x^{+y}-E_x^c)+b_{hz}(E_z^{+y}-E_z^c)$

      $H_z = a_{hz}H_z + b_{hx}(E_x^{+z}-E_x^c)+b_{hy}(E_y^{+z}-E_y^c)$

    `// Update H boundary: loop over boundary cells`

    for $c \in$ *Boundary Cells of y direction, front face* **do**

      $\Psi_{Hx\mathbf{y}} = c_{hy1}\Psi_{Hx\mathbf{y}} + c_{hy2}(E_{\mathbf{z}}^{+y}-E_{\mathbf{z}}^c),\ H_x \mathrel{+}= d_{hx}\Psi_{Hx\mathbf{y}}$

      $\Psi_{Hz\mathbf{y}} = c_{hy1}\Psi_{Hz\mathbf{y}} + c_{hy2}(E_{\mathbf{x}}^{+y}-E_{\mathbf{x}}^c),\ H_z \mathrel{+}= d_{hz}\Psi_{Hz\mathbf{y}}$

    for $c \in$ *Boundary Cells of y direction, back face* **do**

      `// see footnote`[a]

    for $c \in$ *Boundary Cells of x direction, left face* **do**

      $\Psi_{Hy\mathbf{x}} = c_{hx1}\Psi_{Hy\mathbf{x}} + c_{hx2}(E_{\mathbf{z}}^{+x}-E_{\mathbf{z}}^c),\ H_y \mathrel{+}= d_{hy}\Psi_{Hy\mathbf{x}}$

      $\Psi_{Hz\mathbf{x}} = c_{hx1}\Psi_{Hz\mathbf{x}} + c_{hx2}(E_{\mathbf{y}}^{+x}-E_{\mathbf{y}}^c),\ H_z \mathrel{+}= d_{hz}\Psi_{Hz\mathbf{x}}$

    for $c \in$ *Boundary Cells of x direction, right face* **do**

      `// see footnote`[a]

    for $c \in$ *Boundary Cells of z direction, top face* **do**

      $\Psi_{Hx\mathbf{z}} = c_{hz1}\Psi_{Hx\mathbf{z}} + c_{hz2}(E_{\mathbf{y}}^{+z}-E_{\mathbf{y}}^c),\ H_x \mathrel{+}= d_{hx}\Psi_{Hx\mathbf{z}}$

      $\Psi_{Hy\mathbf{z}} = c_{hz1}\Psi_{Hy\mathbf{z}} + c_{hz2}(E_{\mathbf{x}}^{+z}-E_{\mathbf{x}}^c),\ H_y \mathrel{+}= d_{hy}\Psi_{Hy\mathbf{z}}$

    for $c \in$ *Boundary Cells of z direction, bottom face* **do**

      `// see footnote`[a]

    `// Update E: loop over all cells`

    for $c \in$ *Domain Cells* $\cup$ *Boundary Cells* **do**

      $E_x' = a_{ex}E_x + b_{ey}(H_y^c-H_y^{-x})+b_{ez}(H_z^c-H_z^{-x})$

      $E_y' = a_{ey}E_y + b_{ex}(H_x^c-H_x^{-y})+b_{ez}(H_z^c-H_z^{-y})$

      $E_z' = a_{ez}E_z + b_{ex}(H_x^c-H_x^{-z})+b_{ey}(H_y^c-H_y^{-z})$

      $E_x' \mathrel{+}= c_P J_{Px}, E_y' \mathrel{+}= c_P J_{Py}, E_z' \mathrel{+}= c_P J_{Pz}$

    `// Update E boundary: loop over boundary cells`

    for $c \in$ *Boundary Cells of y direction, front face* **do**

      $\Psi_{Ex\mathbf{y}} = c_{ey1}\Psi_{Ex\mathbf{y}} + c_{ey2}(H_{\mathbf{z}}^c-H_{\mathbf{z}}^{-y}),\ E_x' \mathrel{+}= d_{ex}\Psi_{Ex\mathbf{y}}$

      $\Psi_{Ez\mathbf{y}} = c_{ey1}\Psi_{Ez\mathbf{y}} + c_{ey2}(H_{\mathbf{x}}^c-H_{\mathbf{x}}^{-y}),\ E_z' \mathrel{+}= d_{ez}\Psi_{Ez\mathbf{y}}$

    for $c \in$ *Boundary Cells of y direction, back face* **do**

      `// see footnote`[b]

    for $c \in$ *Boundary Cells of x direction, left face* **do**

      $\Psi_{Ey\mathbf{x}} = c_{ex1}\Psi_{Ey\mathbf{x}} + c_{ex2}(H_{\mathbf{z}}^c-H_{\mathbf{z}}^{-x}),\ E_y' \mathrel{+}= d_{ey}\Psi_{Ey\mathbf{x}}$

      $\Psi_{Ez\mathbf{x}} = c_{ex1}\Psi_{Ez\mathbf{x}} + c_{ex2}(H_{\mathbf{y}}^c-H_{\mathbf{y}}^{-x}),\ E_z' \mathrel{+}= d_{ez}\Psi_{Ez\mathbf{x}}$

    for $c \in$ *Boundary Cells of x direction, right face* **do**

      `// see footnote`[b]

    for $c \in$ *Boundary Cells of z direction, top face* **do**

      $\Psi_{Ex\mathbf{z}} = c_{ez1}\Psi_{Ex\mathbf{z}} + c_{ez2}(H_{\mathbf{y}}^c-H_{\mathbf{y}}^{-z}),\ E_x' \mathrel{+}= d_{ex}\Psi_{Ex\mathbf{z}}$

      $\Psi_{Ey\mathbf{z}} = c_{ez1}\Psi_{Ey\mathbf{z}} + c_{ez2}(H_{\mathbf{x}}^c-H_{\mathbf{x}}^{-z}),\ E_y' \mathrel{+}= d_{ey}\Psi_{Ey\mathbf{z}}$

    for $c \in$ *Boundary Cells of z direction, bottom face* **do**

      `// see footnote`[b]

    `// Update` $\mathbf{J_P}$`: loop over all cells`

    for $c \in$ *Domain Cells* $\cup$ *Boundary Cells* **do**

      $J_{P\{x,y,z\}} = s_P J_{P\{x,y,z\}} + Q_P(E'_{\{x,y,z\}} - E_{\{x,y,z\}})$

      $E_{\{x,y,z\}} = E'_{\{x,y,z\}}$

    for $c \in$ *Antenna Cells* **do** `// Add antenna source signals:`

      $E_z(s,c,t) = \text{Source}(s,c,t)$

[a] Same equations of previous loop with new $\hat{\Psi}_H$ and $\hat{c}_h$ variables

[b] Same equations of previous loop with new $\hat{\Psi}_E$ and $\hat{c}_e$ variables

of the next and the previous cell, respectively, in (x, y, z) directions. Except variables $H, E, J_P, \Psi_H, \Psi_E$, all the other terms are constant with a spatial dependency, which requires a significant amount of memory. For more details on FDTD please refer to [44]. Note that the dependency on time step (t) and antenna index (s) in Alg. 1 is omitted whenever needed to improve readability. In addition, even though the antenna source signals can be added in any direction, in our design the antennas emit an electric field in the z direction.

As Alg. 1 clearly shows, the outer loop can be easily *unrolled* and assigned to different parallel Compute Units (CUs), each in charge of one antenna. In the next two sections, we focus first on the design and optimization of a single CU and its implementation on one FPGA, then we focus on the multi-FPGA implementation of a multi-CU system.

3.3 FPGA Design of an FDTD Compute Unit

We validated our initial C++ design in terms of accuracy against the Acceleware commercial code. Both codes use 32-bit Floating-Point (FP) data for all the variables in Alg. 1. Note that computing precision is critical in MI iterative algorithms, as the errors tend to accumulate and lead to inexact solutions. This is why fixed-point data type, which would certainly lead to higher computation speed, cannot be used in this case.

Although we easily converted our initial code to RTL for FPGA implementation using HLS tools[1], the estimated performance (latency in number of clock cycles times the estimated clock period) was worse than the GPU one in Acceleware. To enhance the performance, we adopted various hardware optimization strategies, which consisted in the use of specific HLS *pragmas* and some modifications to the original C++ code. Although the code can be easily adapted to different FPGAs from different vendors, we focused our optimizations and experiments on a Xilinx target. Therefore, from now on, we often refer to *Vivado HLS* and *Vivado* as the tools for HLS development and implementation, respectively. It is important to note that the design goal in our hardware accelerator is to minimize the total latency of the FDTD computation, because this reduces the overall execution time of the

[1] Both the development tools and the advanced target FPGAs nowadays support the synthesis of FP arithmetic to hardware.

MI iterative algorithm. This can be achieved by applying the HLS optimization strategies described in the following.

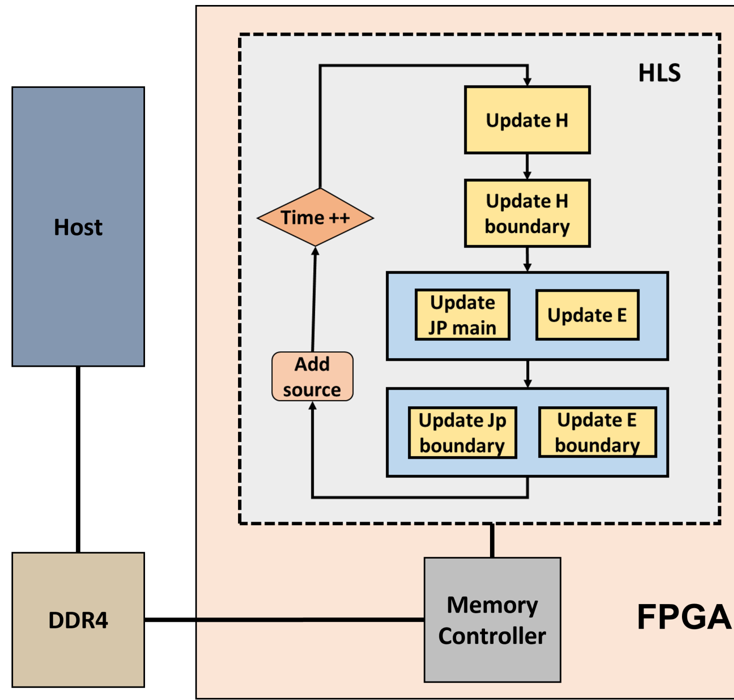


Fig. 3.2 FDTD CU design in HLS for a single FPGA.

Fig. 3.2 presents a schematic representation of the HLS code for a single CU, in which each block is a function that corresponds to an update equation in Alg. 1. The J_P equations, which are separate in Alg. 1, are merged with the Update E and E boundary blocks in Fig. 3.2 to avoid rereading the E fields from the external memory. Table 3.1 summarizes the HLS optimization techniques used in the hardware design and the functions in which they are used, as explained thoroughly in the next subsections. The *top-level function* denotes the function that contains the loop over the time steps in Alg. 1.

Table 3.1 HLS hardware optimization strategies for a FDTD CU.

Method	**functions**
Blocking	Update E and Update H
Merge JP	Update E and E boundary
Loop merge	Update E boundary and H boundary
Storage for Boundaries	Update H boundary and/or Update E boundary
Storage for constant coefficients	top-level function
Loop pipeline	all
function inline	all

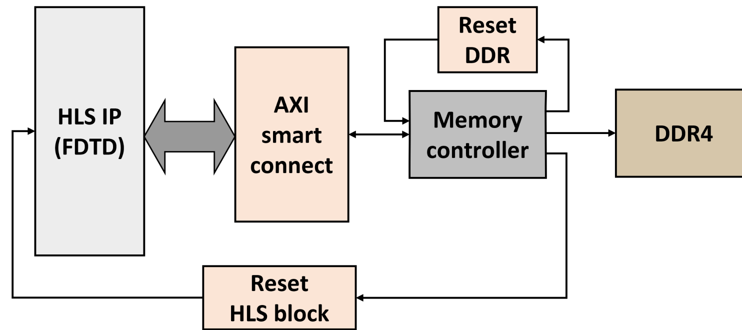


Fig. 3.3 Detailed view of the CU design in Vivado.

Vivado HLS synthesizes the C++ code and generates the RTL code of an IP block that is integrated in the Vivado implementation flow with the additional blocks in Fig. 3.3. These include a memory controller, two reset blocks, and an AXI interconnect block that connects to the memory controller the IP interfaces, all compliant with the AXI standard

3.3.1 Two Architectures: Large and Small

For a more flexible design, we propose two hardware architectures, termed *Large* and *Small,* which use a different number of AXI I/O interfaces and a different amount of local on-chip memory for storing magnetic or electric fields in the boundary regions. In the large design more computing resources and more AXI interfaces are used, which results both in a lower number of CUs that can be implemented in a single FPGA but also in a lower latency per CU compared to the Small design in which less resources and interfaces are used. Hence, depending on the number of CUs over which a designer wishes to parallelize the computation and the number of available FPGAs, either the Large or the Small design is the preferred choice in terms of execution time, as shown later.

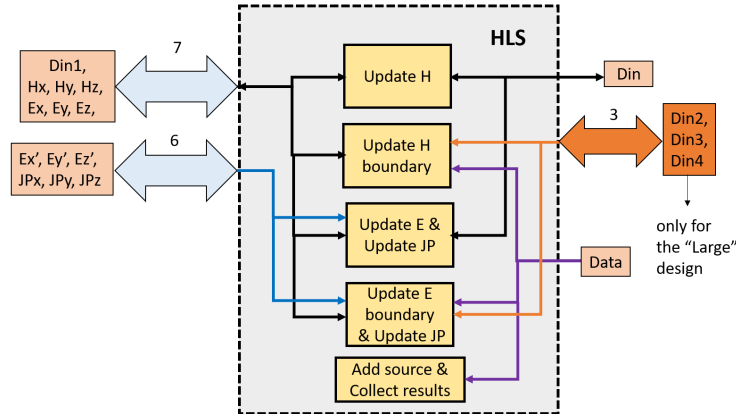


Fig. 3.4 Details of the interfaces of the CU for Small and Large designs.

To avoid performance bottlenecks in the Large design, 18 AXI interfaces are needed, as shown in Fig. 3.4, to guarantee a concurrent access to all the data arrays needed during the computation. Note that the aggregate bandwidth is still compatible with the external DRAM specifications, as we show later. Since in our target FPGA each memory controller can handle only up to 16 ports, one CU needs two memory controllers, which is possible in large FPGAs with multiple external DRAM memory banks.

In the Small design, on the other hand, we reduced the number of AXI ports to 15 by removing three AXI ports (Din2, Din3, Din4) (orange box in Fig.3.4) and using a shared port (Din1) instead. This reduction of AXI interfaces makes it possible to reduce the resource usage as shown in Sec. 3.3.3 and Sec. 3.5.2. As a result, the data in the Small design is routed to some of the blocks in Fig. 3.4 via shared interfaces, at the cost of a lower performance. However, by using lower resources, we can fit more CUs into a single large FPGA with multiple memory controllers, each handling one single CU, hence increasing the overall throughput per single FPGA.

Table 3.2 shows how the variables in Alg. 1 are mapped to the AXI ports according to the design version. Note that Din2-4 only exist in the Large design. In Tab 3.2, b_h_xy is the combination of b_{hx} and b_{hy} in one single array, and similarly b_e_xy for b_{ex} and b_{ey}. Ψ_{E*} and Ψ_{H*} associated to port Din1 refer to all the variables of that type in Alg. 1.

In the following, we explain the optimization methods listed in Table 3.1.

Table 3.2 Description of AXI interfaces.

AXI ports	Design	Variables (refer to Alg. 1)
Data	Small/Large	c_{h1}, c_{h2}, c_{e1}, c_{e2}, Qp, $source$
Din	Small/Large	b_{h}_xy, b_{e}_xy
Din1	Small	Ψ_{E*}, Ψ_{H*}, b_{hz},b_{ez}
	Large	Ψ_{E*}, b_{hz},b_{ez}, Ψ_{Hxy}, Ψ_{Hyx}, Ψ_{Hxz}
Din2	Large	Ψ_{Hzy}, Ψ_{Hzx}, Ψ_{Hyz}
Din3	Large	$\hat{\Psi}_{Hxy}$, $\hat{\Psi}_{Hyx}$, $\hat{\Psi}_{Hxz}$
Din4	Large	$\hat{\Psi}_{Hzy}$, $\hat{\Psi}_{Hzx}$, $\hat{\Psi}_{Hyz}$
Ex, Ey, Ez	Small/Large	E_x, E_y, E_z
Ex′, Ey′, Ez′	Small/Large	E'_x, E'_y, E'_z
Hx, Hy, Hz	Small/Large	H_x, H_y, H_z
JPx, JPy, JPz	Small/Large	J_{Px}, J_{Py}, J_{Pz}

3.3.2 Blocking Method and Merging of J_P Update Equations

To optimize the memory access, we use a method similar to the spatial blocking used in stencils. Fig. 3.5(a) shows the general approach using shift registers as local memory in a 2D stencil with dimensions X and Y. The stencil moves from left to right until it reaches the end of a row and moves one row downward. To compute the current cell C, the four surrounding cells (N,E,S,W) are needed. As the stencil moves, a new cell is written in the head of a shift register and used immediately as "new" S cell, while the "old" N cell is removed from the tail of the shift register. The shift register holds the last $2X+1$ cells from the grid (darker orange cells).

There are significant differences between a generic 2D stencil and a 3D FDTD. As shown in Fig. 3.5(b), updating the magnetic field in a cell requires the electric field in the current and *next* positions, while updating the electric field requires the magnetic field in the current and *previous* positions. Therefore, instead of the 5-cell stencil pattern, we need 3 cells for H update and 3 cells for E update. For a 2D FDTD, this would require two shift registers (for H and E) each of size $X+1$, but for a 3D FDTD, the size becomes $XZ+1$.

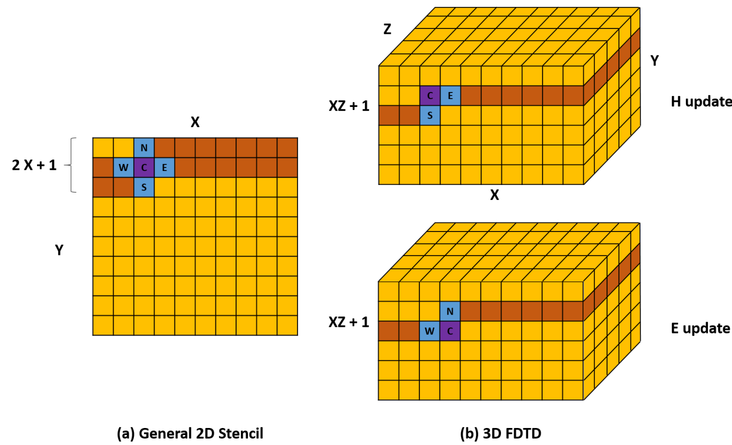


Fig. 3.5 Blocking method for FDTD and its difference with a general stencil

In addition, since in 3D FDTD the three components of E and H fields need each a separate shift register, the number of resources in FPGA is severely impacted. Therefore, instead of using shift registers with Flip-Flops or Look-Up Tables (LUTs), we use BRAMs to locally store the fields. Since BRAMs are dual-port SRAMs, however, it is not possible to read more than two values per clock cycle. Moreover, we cannot partition the arrays to overcome the two-port limitation, because the accessed elements are not always in the same partition. Therefore, we replicate the BRAMs multiple times to simultaneously access all the required cells. As we will see in Sec. 4.4, to balance the resource utilization it is possible, by means of the HLS resource allocation directive, to replace the BRAMs with LUTs arranged as distributed memories.

This local memory for blocking is represented by the *Hram* variable in Alg. 2. Note that, after a round of initialization with an initial plane in the z direction, the memory gets filled with a new plane while the computation happens on the previous plane. The concurrency of memory access to the new plane and computation on the old plane is a key factor to obtain a high computing throughput and so a low execution latency. Another local memory used only for the boundary field cells is *BF* in Alg. 2, which is described in detail in Sec. 3.3.3.

Another optimization consists in merging the loop that updates J_P with the other loops for E update and E-boundary update, previously shown as three separate loops in Alg. 1. For this purpose, we split the J_P update equations in two parts for the main

Algorithm 2: Pseudo-code for Update E with Blocking and JP merge.

```
Update E:
s = 0; size = (X − 1)(Y − 1) + 1;
for k=Z-1;k>1;k-- do // Process XY planes in Z direction
    for Domain Cells ∪ Boundary Cells in plane k do
        if Boundary Cells and Use local storage for H then
            BF = H;// Local storage in BRAM
        Blocking:
        // Store new XY plane in memory
        Hram_{x,y,z}[s++] = H_{x,y,z}[k];
        #pragma resource Hram RAM_2P_LUTRAM
        if s = size then // Hram=full
            s = 0;
        if (k < Z − 1) then // ignore first plane (Z-1)
            // Process old XY plane (while reading new)
            H^c_{x,y,z} = Hram_{x,y,z}[s];
            H^{-y}_{x,z} = Hram_{x,z}[(s + X − 1)%size];
            H^{-z}_{x,y} = Hram_{x,y}[(s + size − 1)%size];
            H^{-x}_{y,z} = Hram_{y,z}[(s + 1)%size];
            Main Update E (for plane k + 1):
            E'_x = UpdateE(E_x, H^c_y, H^{-x}_y, H^c_z, H^{-x}_z, J_Px);
            E'_y = UpdateE(E_y, H^c_x, H^{-y}_x, H^c_z, H^{-y}_z, J_Py);
            E'_z = UpdateE(E_z, H^c_x, H^{-z}_x, H^c_y, H^{-z}_y, J_Pz);
            Merge JP update (for plane k + 1):
            if Domain Cells then
                J_P{x,y,z} = s_P J_P{x,y,z} + Q_P(E'_{x,y,z} − E_{x,y,z});
                E_{x,y,z} = E'_{x,y,z};
```

domain and boundary cells. In Alg. 2 the part of the main cells is updated with the *if* condition in the last few lines.

For space reasons, we do not show the pseudo-code for Update H, which uses the same blocking method of Alg. 2.

3.3.3 Loop Merge and Local Storage for Boundaries

A key strategy to improve the FDTD performance is to move the accesses of the many array variables from the external DRAM to on-chip SRAMs. Due to the large number of cells, however, even for the largest FPGAs this strategy is applicable only to a subset of the variables. For this reason, we use on-chip memory only for the boundary elements, when possible. This is shown in Algs. 2-3 with local storage *BF* enabled or disabled with an *if* conditional statement.

The loops for the update equations for the six boundary regions in Fig. 3.1, which are separated in Alg. 1, can be merged in pairs according to the coloring scheme in

Algorithm 3: Pseudo-code for Update H-boundary with merging boundary loops and local storage.

Update H:
{...
if *Boundary Cells* **and** *Use local storage for E* **then**
  $BF = E$;`// Local storage in BRAM`
...}
Update H-boundary:
if *NOT use local storage for E* **then**
  Change BF to E
#pragma loop merge
for *Boundary cells in y direction, front face* **do**
  $\Psi_{Hx\mathbf{y}} = c_{hy1}\Psi_{Hx\mathbf{y}} + c_{hy2}(BF_{\mathbf{z}}^{+y} - BF_{\mathbf{z}}^{c}),\ H_x \mathrel{+}= d_{hx}\Psi_{Hx\mathbf{y}};$
  $\Psi_{Hz\mathbf{y}} = c_{hy1}\Psi_{Hz\mathbf{y}} + c_{hy2}(BF_{\mathbf{x}}^{+y} - BF_{\mathbf{x}}^{c}),\ H_z \mathrel{+}= d_{hz}\Psi_{Hz\mathbf{y}};$
for *Boundary cells in y direction, back face* **do**
  $\hat{\Psi}_{Hx\mathbf{y}} = \hat{c}_{hy1}\hat{\Psi}_{Hx\mathbf{y}} + \hat{c}_{hy2}(BF_{\mathbf{z}}^{+y} - BF_{\mathbf{z}}^{c}),\ H_x \mathrel{+}= d_{hx}\hat{\Psi}_{Hx\mathbf{y}};$
  $\hat{\Psi}_{Hz\mathbf{y}} = \hat{c}_{hy1}\hat{\Psi}_{Hz\mathbf{y}} + \hat{c}_{hy2}(BF_{\mathbf{x}}^{+y} - BF_{\mathbf{x}}^{c}),\ H_z \mathrel{+}= d_{hz}\hat{\Psi}_{Hz\mathbf{y}};$
`// 2 (×2) other loops for x and z directions`

Fig. 3.1. Alg. 3 shows how the loops in Update H-boundary are merged by using the *loop merge* HLS pragma. Note that the complete CPML boundary conditions require 6 loops in total for all the directions for each field (E or H), while in previous works, the simplified CPML boundary conditions require only 2 loops as they are used only for one direction.

Despite the loop merging in the H-boundary update, the problem of accessing the Ψ_{H*} arrays from the external memory several times remains intact. To maximize the execution speed of the merged loops, the arrays need to be accessed four times in parallel, so the four AXI ports Din1-4 almost entirely dedicated to them shown in Table 3.2 and in Fig. 3.4 (3 dedicated ports, Din2-4, and one shared port, Din1).

While the Large design leverages the simultaneous access through these ports, the Small one eliminates ports Din2-4. As discussed later in Sec. 3.5.2, in the Small design there is no benefit in using the local BRAMs to store the electric fields ($BF = E$), since the performance is limited by the serialized access to all the Ψ_{H*} arrays through one shared AXI port.

The loop merging for the Update E-boundary is shown in Alg. 4. Here we merged the part of the JP update equations related to the boundary cells, while the part related to the main cells is merged with the E update, as shown in Alg. 2.

Algorithm 4: Pseudo-code for Update E-boundary with merging boundary loops and J_P merge.

Update E:
{...
if *Boundary Cells* **and** *Use local storage for H* **then**
    $BF = H$;`// Local storage in BRAM`
...}
Update E-boundary:
if *NOT use local storage for H* **then**
    Change BF to H;
#pragma loop merge
for *Boundary cells in y direction, front face* **do**
    $\Psi_{Exy} = c_{ey1}\Psi_{Exy} + c_{ey2}(BF_{\mathbf{z}}^{c} - BF_{\mathbf{z}}^{-y}),\ E'_x \mathrel{+}= d_{ex}\Psi_{Exy}$;
    $\Psi_{Ezy} = c_{ey1}\Psi_{Ezy} + c_{ey2}(BF_{\mathbf{x}}^{c} - BF_{\mathbf{x}}^{-y}),\ E'_z \mathrel{+}= d_{ez}\Psi_{Ezy}$;
    Merge JP update, front face:
    $J_{P\{x,z\}} = s_P J_{P\{x,z\}} + Q_P(E'_{\{x,z\}} - E_{\{x,z\}})$;
    $E_{\{x,z\}} = E'_{\{x,z\}}$;
for *Boundary cells in y direction, back face* **do**
    $\hat{\Psi}_{Exy} = \hat{c}_{ey1}\hat{\Psi}_{Exy} + \hat{c}_{ey2}(BF_{\mathbf{z}}^{c} - BF_{\mathbf{z}}^{-y}),\ E'_x \mathrel{+}= d_{ex}\hat{\Psi}_{Exy}$;
    $\hat{\Psi}_{Ezy} = \hat{c}_{ey1}\hat{\Psi}_{Ezy} + \hat{c}_{ey2}(BF_{\mathbf{x}}^{c} - BF_{\mathbf{x}}^{-y}),\ E'_z \mathrel{+}= d_{ez}\hat{\Psi}_{Ezy}$;
    Merge JP update, back face:
    $J_{P\{x,z\}} = s_P J_{P\{x,z\}} + Q_P(E'_{\{x,z\}} - E_{\{x,z\}})$;
    $E_{\{x,z\}} = E'_{\{x,z\}}$;
`// 2(×2) other loops for x and z directions`

3.3.4 Loop Pipeline, Function Inline, and Storage for Coefficients

The last optimization strategies consist in setting directives to a) inline all the functions and b) pipeline the innermost loop in nested loops like those over all the cells in three dimensions. Both strategies significantly improve the Initiation Interval (II) of the loops, which is the distance in clock cycles between the starting of two consecutive iterations and corresponds to the inverse of the loop throughput, as discussed thoroughly in Sec. 4.4. A perfectly pipelined loop starts a new computation every clock cycle (II=1). In practice, dependencies between iterations prevent to obtain this goal for every loop. As for function inlining, it allows for further resource sharing when this does not impact performance and allows for optimization across function hierarchies. Alg. 5 shows how the pragmas associated with these strategies are used in the update functions.

Finally, we locally store constant coefficients $a_{e\{x,y,z\}}$ and $d_{e\{x,y,z\}}$ in Alg. 1 in URAMs, which are large memories available in high-capacity Xilinx FPGAs. Since the coefficient values vary over the entire domain, they need a large amount of storage, but one AXI port is enough to load them during the initialization. By using

Algorithm 5: HLS pragmas of loop pipelining and function inlining in 3D FDTD algorithm.

```
Update functions:
#pragma inline// Function is inlined
// Loop over all cells in (x,y,z) dimensions
for k=1 to Z do
    for j=1 to Y do
        for i=1 to X do // Innermost loop on x is pipelined
            #pragma pipeline
            ...
```

both BRAMs and URAMs, we achieve a high utilization of local FPGA storage as shown in Sec. 4.4.

3.4 Multi-FPGA Implementation

The possibility of unrolling the outermost loop in Alg. 1 and let multiple CUs work in parallel on different antennas, is hindered by the limited resources available in one FPGA. For example, the FPGA used in our experiments supports up to 3 CUs in the Small design and 2 CUs in the Large one. In particular, this FPGA contains three so-called Super Logic Regions (SLRs) and our fastest design uses one CU per SLR. Although we could place more CUs, the advantage of parallelism is countered by a) the slower memory access caused by the AXI ports sharing, and b) the slower clock frequencies caused by the routing congestion. The only chance to improve performance is to use multiple FPGAs.

We emphasize that the computations in each FPGA and for each antenna are independent and there is no need for data sharing between them. This straightforward parallelism simplifies the deployment over commercially available and energy-optimized multi-FPGA platforms. This is in contrast with the use of multiple GPUs, e.g. in High-Performance Computing (HPC) clusters, which are more expensive and less energy efficient for similar performance. To show this contrast and to demonstrate the advantages of using FPGAs over GPUs, we proposed the deployment of our FDTD accelerator on a Multi-FPGA platform.

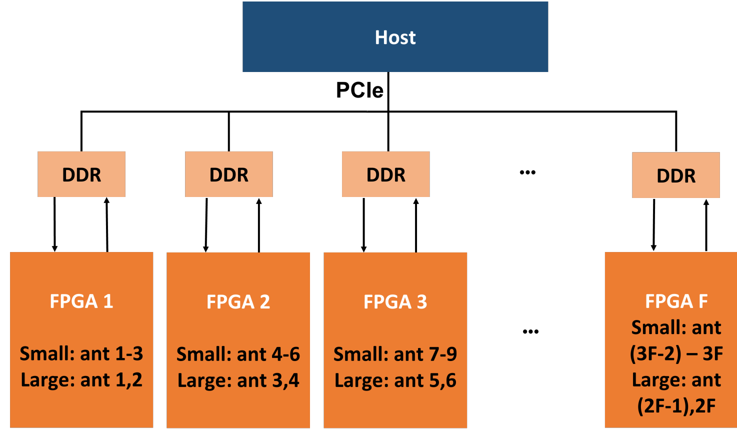


Fig. 3.6 Multi-FPGA platform with F FPGAs for 3D FDTD acceleration.

Multi-FPGA platforms have become easily accessible, like for example the Amazon AWS EC2 F1 instances. The AWS platform has eight Xilinx UltraScale+ FPGAs, each connected to a multi-bank local DDR DRAM. The FPGAs are connected via the PCIexpress (PCIe) bus to an x86 host CPU, as shown in Fig. 3.6. The figure also shows how multiple CUs working on $3F$ or $2F$ antennas, depending on the design version with either 15 or 18 AXI ports, can be allocated to F FPGAs.

The host code in the CPU coordinates the execution of the CUs in the FPGAs similarly to how the host code controls execution of multiple threads in a GPU. Initially the host transfers the inputs required by FDTD to all the local DDR memories. As we show in Sec. 4.4, the overhead for this initial PCIe transaction is negligible compared to the computing time.

The total execution time depends on the number of FPGAs, the number of antennas, and the design version. With A antennas and F FPGAs, depending on the maximum supported number of antennas n_A in each FPGA (2 in the Large version or 3 in the Small one) the relation between the total execution time (T_{tot}) and the time for each antenna (T_{ant}) is:

$$T_{tot} = \left\lceil \frac{A}{n_A \cdot F} \right\rceil \times T_{ant} \tag{3.6}$$

By using the AWS platform with 8 FPGAs, T_{tot} will be equal to T_{ant} for up to $8 \times 3 = 24$ or $8 \times 2 = 16$ antennas in the Small and Large designs, respectively. For a larger number of antennas, the time will scale with factor $\lceil \frac{A}{24} \rceil$ in the Small design

and $\lceil \frac{A}{16} \rceil$ in the Large one. Note, however, that T_{ant} is different for the two cases, as discussed in the next section.

3.5 Results

Although the code that we developed is portable, we performed our experiments on a specific Xilinx FPGA target, the Virtex UltraScale+ used in the Amazon EC2 F1 instance (vu9p-flgb2104-2-i). This FPGA consists of 3 Super Logic Regions (SLRs) positioned at the left, middle, and right side of FPGA. It also contains 4 DDR4 memory interfaces with each interface accessing a 16 GiB memory. The middle SLR contains 2 memory interfaces while the left and right SLRs contain one interface each. Before reporting the performance results obtained on this FPGA, we briefly discuss the accuracy of the C++ code in comparison to the Acceleware code.

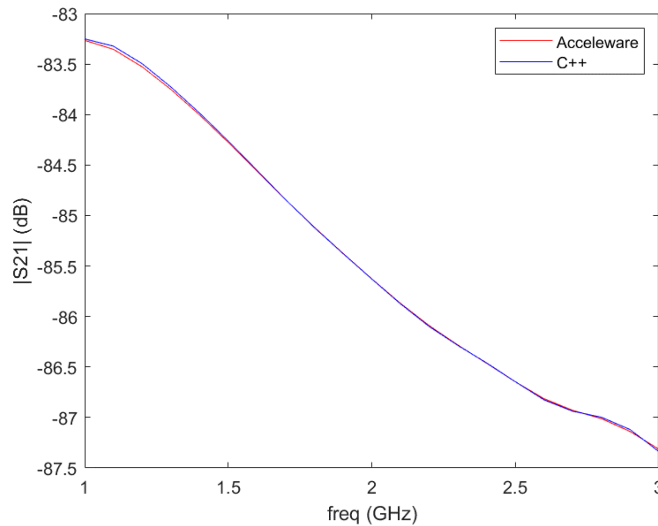


Fig. 3.7 Accuracy comparison: Acceleware design versus our C++ code.

To perform the accuracy check, we simulated a grid of $50 \times 50 \times 50$ main cells with a boundary region of 10 cells on each side. Therefore, in total, the simulation space has $70 \times 70 \times 70 = 343000$ cells. The total number of time steps was 1000. Fig. 3.7 reports the magnitude of the S_{21} scattering parameter related to the transmission between an antenna source and an observation point in the simulation space as a function of frequency. This is a typical information used in the DBIM-

TwIST MI algorithm and is obtained from the FDTD forward solver. The curves show an almost perfect overlapping between what Acceleware and our synthesizable C++ code obtain, with a Mean Absolute Percentage Error (MAPE) of 0.01%.

For what concerns the execution time, this is the product of the overall execution latency, in clock cycles, and the clock period. The latency is minimized at a high level with a proper design space exploration, which we could perform thanks to the flexibility of HLS coding and the features of Vivado HLS (2019.1 version). During the HLS phase, we aimed to keep the clock frequency target high enough so that the resulting performance would be competitive with the GPU design, but not too high in order to avoid issues at the implementation stage, specially during the routing phase. Determining the proper clock target and the most appropriate strategies for the implementation required a few iterations between the high-level abstract design in Vivado HLS and the low-level physical design in Vivado.

In the following we explain the impact of HLS-based optimization methods on the hardware performance. After that, we describe the design procedures and the results obtained first on a single FPGA and then on the multi-FPGA platform. We also report a comparison between the FPGA design and three GPU designs: the first one was developed in Matlab and tested on an NVIDIA Tesla K20c, the second one was developed using Acceleware and tested on a Tesla P40 GPU, and the third one is our design obtained from the same C++ code that runs on the FPGA and tested also on the Tesla K20c GPU. For the comparison we used the same simulation space of the accuracy check discussed above.

3.5.1 Impact of HLS Optimizations on Performance

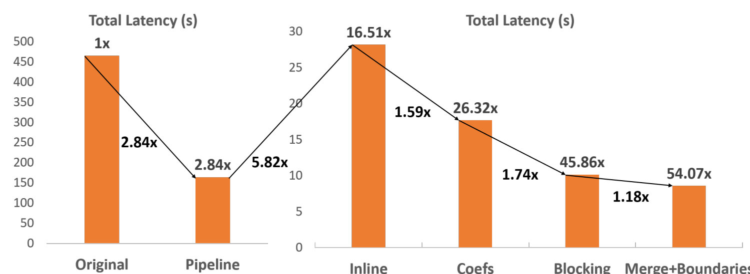


Fig. 3.8 Impact of different HLS optimization methods on the total latency. (numbers on top of the bars show the improvement compared to the original code, and numbers bellow the arrows show the improvement compared to the previous optimization method)

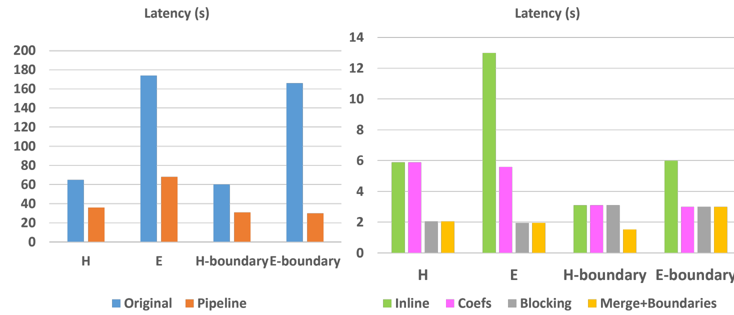


Fig. 3.9 Impact of different HLS optimization methods on the latency of each FDTD function.

The HLS optimization techniques described in Sec. 3.3 improve the performance of our FDTD hardware accelerator. It is important to note that different optimization strategies are applied step-by-step and the designer must have enough knowledge about the algorithm bottlenecks to find the best HLS optimizations. Each of these changes the performance and also the bottlenecks, hence the designer must monitor and analyze the performance change while exploring the accelerator design space using HLS. Here, the impact of each HLS optimization on the hardware performance is explored in more detail. Starting from the original code without any HLS directives, we add each optimization method incrementally and measure the performance of the FDTD Compute Unit (CU) in terms of latency and resource usage. The results in Fig. 3.8 show that each directive contributes to reducing the latency until the minimum latency of 8.6 s is obtained when all the directives are applied. (The figure is split in two histograms with different scales for better readability.) Fig. 3.9 shows the latency of each FDTD function and their relation with HLS directives. Each directive operates on one or multiple functions: *pipeline* and *inline* directives operate on all functions, storage for coefficients (*Coefs*) is applied to Update E and E-boundary functions, *Blocking* is applied to Update H and E, and loop *Merge* + storage for *Boundaries* is applied to H-boundary function[2].

[2]It can be applied to E-boundaries as well, but it complicates routing in the implementation step in our target FPGA.

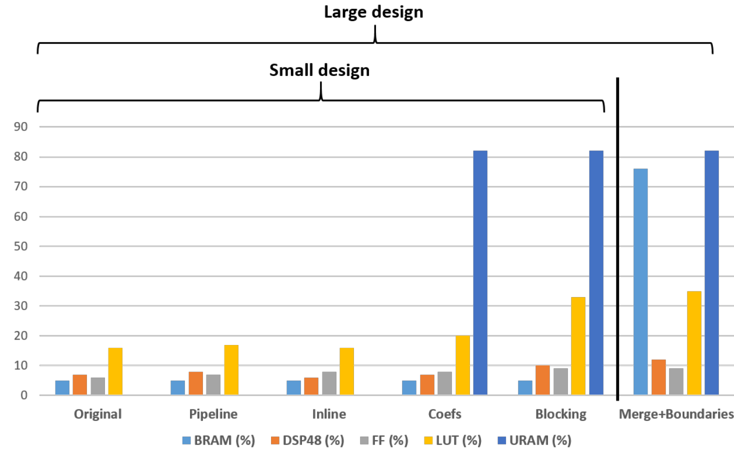


Fig. 3.10 Impact of HLS optimization methods on resource usage per SLR.

The HLS estimation of the resource usage for the accelerator with respect to different HLS directives is shown in Fig. 3.10. Note that the Small design uses all the HLS directives except for loop merging and storage for the boundaries. The high URAM usage is due to the storage of constant coefficients (Sec. 3.3.4). The Blocking method (Sec. 3.3.2) increases the LUT usage and the last directive in the Large design (*Merge+Boundaries*) increases the BRAM usage. Although DSPs and FFs are not fully utilized, the advantage of using a large FPGA with high number of resources is the higher availability of URAMs. Compared to the design without URAMs (*Inline* directive in Fig. 3.10), the optimized Small design including URAMs (after *Blocking* method) obtains 2.8× improvement in the total latency (Fig. 3.8).

3.5.2 FDTD Performance on a Single FPGA

To optimize the clock frequency, we used specific Vivado *strategies* for both logic synthesis and implementation. For synthesis we use the *Flow_PerfOptimized_High* strategy, which sets the tool options to maximize timing performance and gives less importance to minimizing resource usage (e.g., no resource sharing, FSM extraction forced to one-hot, no LUT combining, etc.). For the implementation of the Large design we used the *Performance_HighUtilSLRs* strategy, which aims to maximize the utilization of an SLR; for the Small one we used the *ExtraTiming_Opt* strategy, which gives priority chiefly to meeting timing constraints. For both designs, we obtain a maximum clock frequency of 167 MHz.

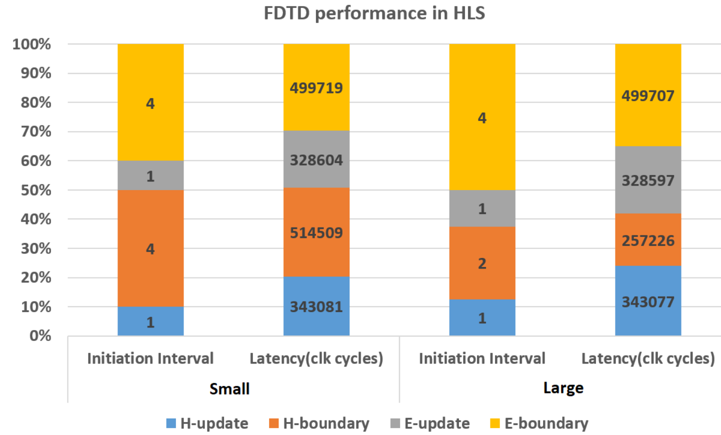


Fig. 3.11 The performance of the main FDTD loops in Small and Large design in HLS.

For the latency optimization, Fig. 3.11 shows II and latencies of the loops present in the various FDTD building blocks, for both the Small and Large designs. It is observed that for E Update and H Update we can achieve the minimum II of 1 thanks to the blocking method described in Sec. 3.3.2 and the loop pipelining described in Sec. 3.3.4.

Optimizing the II of the boundary loops is a much more complicated task. First of all, many more variables need to be accessed from memory, as clear from the comparison between Algs. 3-4 and Alg. 2. Note that merging the loops on two boundary faces in the same direction, while beneficial for the sharing of some logic, does not reduce the number of accesses to variables defined over two physically distinct areas of the domain. Note also that further increasing the number of AXI ports is not possible because of the mentioned limitations of memory controllers and the complications arising from accessing multiple controllers in different SLRs. The only viable option is to store the E and H variables in local on-chip memories, while using four separate ports for Ψ_E and Ψ_H variables (the corresponding ports between E and H can be shared). Even with this solution, the best achievable II for the boundary loops with the selected number of AXI ports is 2, as determined by the simultaneous access from the same AXI port to variables defined in two distinct boundary faces.

Unfortunately, II=2 is not achievable for both E and H boundaries at the same time. In particular, sharing the same BRAMs for both boundary fields complicates routing, leading to timing failures in the implementation. An alternative is to use

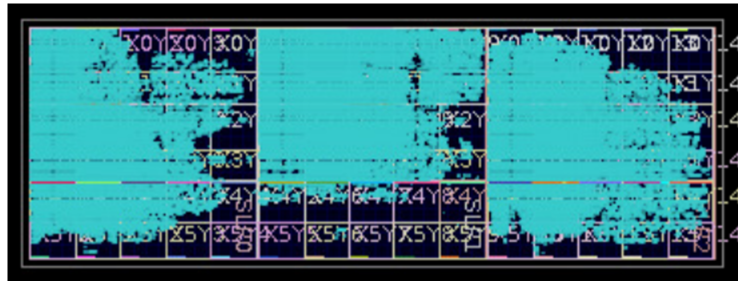
separate BRAMs, but the resource usage exceeds the available BRAMs in each SLR region (48% in total, i.e. about one and a half out of three SLRs available). As a result, one CU gets placed across two SLRs, and the routing phase in Vivado ends with a large negative slack because of the large routing delay of the many wires that cross the SLRs.

Nevertheless, by using the local memory either in E or H boundary loops, we respect the BRAM limits in one SLR without increasing the routing complexity. In the Large design, we apply it to H-boundary. This requires to have four separate ports for Ψ_{H*} (Din1-4 in Fig. 3.4 and Table 3.2 for the Large design). Therefore, as shown in Fig. 3.11, in the Large design the II for H-boundary is 2, while for E-boundary is 4. In the Small design, however, both E- and H- boundary loops obtain II=4 as the local memory is not used: eliminating ports Din2-4 already degrades II from 2 to 4, hence making local memory totally ineffective. Using less BRAMS, however, makes room for more CUs in a single FPGA, hence balancing the longer latency of each CU with higher parallelism.

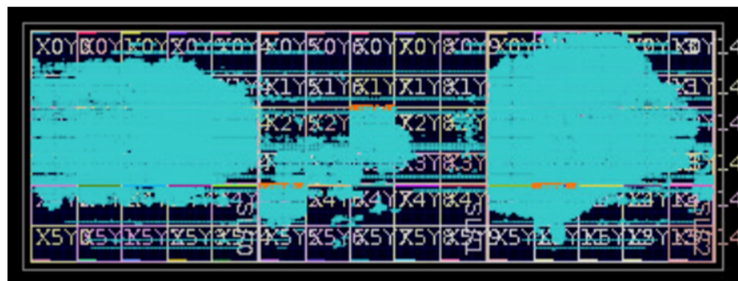
The clock frequency estimated by HLS for both design versions is 170 MHz, very close to the final 167-MHz frequency obtained after implementation. Critical paths are related to the data transfer from AXI ports to local memory (BRAM, LUT-based, or URAM). This is because memory blocks are scattered in the SLRs, thus causing significant routing delays.

The estimated execution time for the Small and Large designs is 10.14 s and 8.6 s, respectively, and is determined by the computation time and not by the DDR memory access. This is because the maximum bandwidth of 19.2 GB/s of each DDR4 memory bank in the AWS F1 instance is always greater than the peak required bandwidth given by simultaneous *writes* and *reads*, each of which consumes a bandwidth of $4\text{B} \times 167\,\text{MHz} = 0.667\,\text{GB/s}$. In the Small design, each CU uses one Memory Controller (MC) connected to a single memory bank for at most 15 reads and 4 writes (Alg. 4), thus the peak bandwidth is $(15+4) \times 0.667 = 12.7\,\text{GB/s}$, which is less than 19.2 GB/s. In the Large design, each CU has 16 ports mapped to one MC (MC_1) and 2 ports to another MC (MC_2). With 16 reads and 4 writes in MC_1, and 2 reads and 1 write in MC_2, the peak bandwidth is $20 \times 0.667 = 13.36\,\text{GB/s}$ for MC_1 and $3 \times 0.667 = 2\,\text{GB/s}$ for MC_2, both less than 19.2 GB/s. When more CUs are mapped to one FPGA, it is not an issue either, as shown in the following.

Single FPGA Small Design with 3 CUs



a) Small design (3 antennas)



b) Large design (2 antennas)

Fig. 3.12 Device view in a) Small and b) Large design after place-and-route (it contains 3 SLRs in the left, middle and right side of the FPGA).

Since the Small design fits in one SLR and the FPGA consists of three SLRs, up to three CUs working in parallel on three antennas can be instantiated as shown in Fig. 3.12(a). Since each CU is connected to a separate MC and so to a separate memory bank, the DDR4 bandwidth limit is not exceeded.

Table 3.3 shows both the resource usage estimation obtained with Vivado HLS and the actual values after place-and-route in Vivado for the Small design. The HLS estimation includes only the FDTD block, while Vivado results include all the blocks in Fig. 3.3. The percentage for the HLS estimation in Table 3.3 refers to one SLR. For the Vivado results, it refers instead to the entire FPGA with three CUs, each using one single SLR. Table 3.3 shows a high resource utilization of URAMs. This is because we use them to store large constant arrays (a_{ei}, b_{ei}) in the update E function (see Sec. 3.3.3).

Table 3.3 Resource usage for the Small design: HLS estimation and Vivado implementation results

		BRAM (%)	DSP48 (%)	FF (%)	LUT (%)	URAM (%)
HLS: 1 CU (One SLR)		4	11	10	35	82
Vivado: 3 CUs	FDTD	3.47	17.46	15.25	34.09	82
	Smart Connect	0	0	2.95	4.29	0
	DDR Ctrl	3.54	0.1	1.05	1.93	0
	Total	7	17.6	19.29	40	82

Single FPGA Large Design with 2 CUs

One CU in Large design consumes more BRAMs than the total number of BRAMs in one SLR. Therefore, a maximum of two CUs can be mapped onto one FPGA with three SLRs, as shown in Fig. 3.12(b). Two CUs share three MCs (MC_1,MC_2 and MC_3) while still not exceeding the DDR4 bandwidth. MC_1 and MC_2 are used for 2×16 ports and MC_3 is used for 2×2 ports. The peak bandwidth for both MC_1 and MC_2 is 13.36 GB/s (calculated as for the single CU). Regarding MC_3, the maximum number of reads and writes in 4 ports is 8 in the worst case leading to a peak bandwidth of $8 \times 0.667 = 5.3$ GB/s, again less 19.2 GB/s.

The resource usage is shown in Table 3.4. The high BRAM usage is due to the local storage of the electric fields for the boundary region (BF, see Sec. 3.3.3). The percentage in the HLS estimation is for one SLR while the percentage in Vivado is for the entire FPGA. Note how two CUs use more than 2/3 of the total BRAM resources, hence making it impossible to map three CUs like in the Small design. For the blocking method described in Sec. 3.3.2 we cannot map $Hram$ in Alg. 2 to BRAMs, otherwise the design gets too congested (90% BRAM usage) and there is a significant penalty in clock frequency. For this reason we use LUTs as distributed RAM, using the LUT resource allocation pragma shown in Alg. 2.

As shown in Table 3.3 and Table 3.4, the DSP usage in both designs is low. This shows that our implementation is not compute-bound. The limiting factors in our design are related to the access to the internal on-chip memories and the number of AXI ports. We do not exceed the memory bandwidth by carefully selecting the number of AXI ports. Further increasing the number of ports (to the maximum supported ports that is $16 \times 4 = 64$), while still compatible with bandwidth, is not possible in practice due to the complexities in routing stage and accessing multiple SLRs. The application is memory-bound with respect to the access to on-chip memory. It is constrained by the internal resource limitations. To improve the

performance, we need either more local storage, or a greater number of AXI ports. For the former we are limited by the internal resources, and for the latter we are restricted by the complexities arising from the routing stage.

Table 3.4 Resource usage for the Large design: HLS estimation and Vivado implementation results

		BRAM (%)	DSP48 (%)	FF (%)	LUT (%)	URAM (%)
HLS: 1 CU (One SLR)		76	13	10	35	82
Vivado: 2 CUs	FDTD	73.7%	12.7	11.4	23.2	55
	Smart Connect	0	0	0.5	3.56	0
	DDR Ctrl	3.54	0.1	1.05	1.92	0
	Total	77.24	12.83	13.25	28.69	55

Comparison Between FPGA, GPU, and CPU

Table 6.3 reports performance and power consumption for the FDTD accelerator on a CPU, three GPU designs (including the one derived from the code developed in this work), and FPGA. To allow a proper performance comparison between accelerators with different capacities in terms of parallel processed antennas, we chose the *time per antenna* as performance metric (total time divided by the number of parallel processed antennas). Since we could not measure the actual power consumed by the FPGA, the power consumption for FPGA reported in the third column of the table is the total consumed on-chip power obtained by the post-route report from the *Vivado Power Analysis* tool. This value can be therefore considered only a crude approximation of the actual power consumed by the device. The corresponding power consumption for CPU and GPU could not be measured either, so for a fair qualitative comparison, we compared the maximum Thermal Design Power (TDP) for our UltraScale+ FPGA with CPU and GPU designs in the second to last column of the table. In addition, we reported the maximum energy consumption in the last column (obtained from the TDP values and the execution time). It is observed that the maximum TDP for our FPGA target is between the CPU and GPU designs, but thanks to the lower execution time it would result in a more energy efficient implementation, should the actual power consumption scale more or less in the same way in the three designs from TDP to actual power values. The results confirm that FPGAs can be more energy efficient than GPUs and CPUs, and show that FPGAs can be competitive with GPUs at scientific computation. Somewhat surprisingly, our GPU design (GPU3) is also 14% faster than the Acceleware GPU code. Compared to

MATLAB implementation in GPU1, the CUDA implementation in this work (GPU3) is about 3 times faster. This is because CUDA is the native programming method for NVIDIA GPUs. As for the comparison between the two FPGA designs, the higher parallelism of the Small design results in a better performance, albeit at a higher power cost.

Table 3.6 shows a comparison between the performance of our proposed FPGA design for 3D FDTD with other FPGA implementations. It is observed that none of the previous works considered the impact of polarization currents which creates additional computations in each cell in the simulation space. In addition, the boundary conditions in our design is CPML in all directions that is not considered in works [8] and [7]. The performance can be measured in Mcells/s by dividing the total number of cells for all the time steps by the total processing time $((Max.Ant. \times T_{max} \times Total_Cells)/Time(s))$. It can be converted to GFLOP/s by multiplying the performance in MCells/s to the number of operations in each cell. As shown in Table 3.6, although the performance in Mcells/s is not high in this work, the performance in GFLOP/s outperforms other implementations. This is because of the polarization currents in FDTD equations that create additional operations in each cell. Note that in [61], the peak performance was reported that is obtained without full CPMLs and is not a fair comparison with our work, so we computed the performance of [61] in MCells/s when there are CPML boundaries in all directions. In addition, our design can operate in higher clock frequency than previous methods.

3.5.3 FDTD Performance on Multiple FPGAs

Increasing the number of FPGAs can improve the FDTD performance. To assess such improvement we measured the execution time for a fixed number of antennas with different number of FPGAs. Fig. 3.13 shows the FDTD execution time for 8 and 24 antennas by varying the number of FPGAs from 1 to 10. It is observed that the reduction of the execution time depends on the number of antennas and FPGAs as well as the design version (Small or Large). Note from the curves in Fig. 3.13 that when the number of available FPGAs is sufficient to process all the antennas in parallel, increasing the number of FPGAs beyond that number does not further improve the performance. For example, in the Amazon EC2 F1 instance with 8 FPGAs, the Small and Large designs can process up to $A_{\max} = 3 \times 8 = 24$ and $A_{\max} = 2 \times 8 = 16$ antennas in parallel, respectively, in a fixed time equal to T_{ant} in

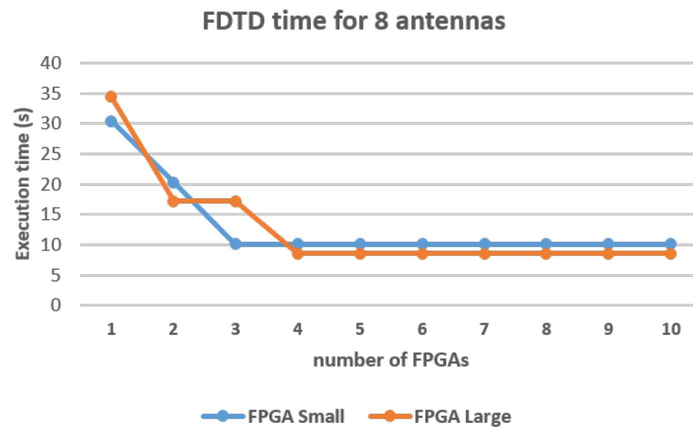
Table 3.5 Performance comparison: CPU = Intel Xeon, GPU1 = Tesla K20C GPU2 = Tesla P40, GPU3 = GPU1 CUDA implementation, and FPGA (UltraScale+). TDP = Thermal Design Power, Energy = TDP$\times$Time.

Hardware :release date	**Max. ant.**	**Time per ant. (s)**	**On-chip Power (W)**	**Max. TDP (W)**	**Max. Energy (J)**
CPU : 2017	1	24.97	-	105	2621
GPU1 (Matlab) : 2012	1	12.06	-	225	2713
GPU2 (Acceleware) : 2016	1	5.64	-	250	1410
GPU3 (this work, CUDA) : 2012	1	4.88	-	225	1098
UltraScale+ (This work, Small design) : 2016	3	3.38	16.24	128	432
UltraScale+ (This work, Large design) : 2016	2	4.3	14.5	128	550

Table 3.6 Performance comparison between our single Small FPGA design and other FPGA implementations.

	Work [8]	**Work [7]**	**Work [61]**	**This work Small Design**
Boundary Conditions	Dirichlet (zero)	CPML (limited)	CPML (Full)	CPML (Full)
Polarization	No	No	No	Yes
Language	MaxCompiler (HLS)	MaxCompiler (HLS)	Verilog	Vivado HLS (C++)
Mcells/s	325	100	336	101
GFLOP/s	**11.7**	**5.7**	**29.2**	**34.2**
Freq (MHz)	**100**	**100**	**100**	**167**

Eqn. (3.6). Whenever the number of antennas to process exceeds $A_{\max}$, the time will increase according to Eqn. (3.6) with $F = 8$. Note how the best solution in Fig. 3.13, either Small or Large, depends on the number of antennas and the number of FPGAs, due to the interplay between the different number of CUs per FPGA and the different value of T_{ant}.



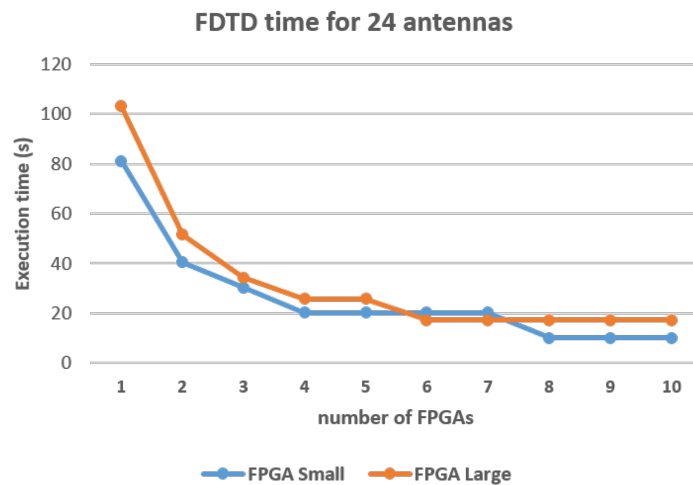


Fig. 3.13 FDTD execution time for different number of FPGAs, (a) 8 antennas, (b) 24 antennas.

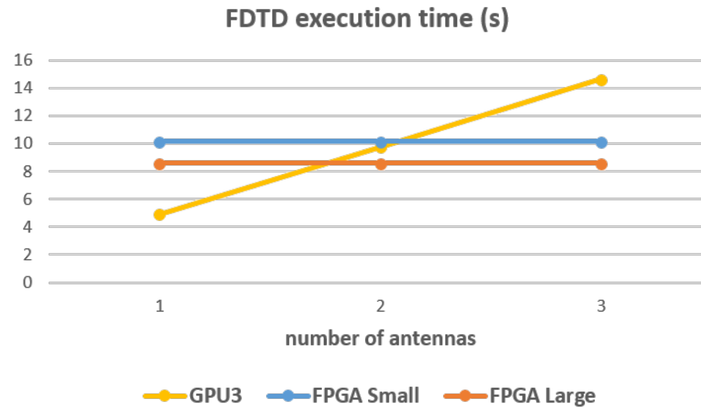
While the computation time remains constant for up to $A_{\max}$ antennas, the time required to transfer all the coefficients from the host to the DDR memories via the PCIe bus grows proportionally to the number of FPGAs because of the inevitable

data duplication [68]. (As the initial values of E and H fields are zero, there is no need to take them into consideration.) To account for this overhead, we analyzed the maximum data transfer time. Table 3.7 shows the type and size of the coefficients used in Alg. 1 that need to be transferred. In Table 3.7, N is the size of the 3D FDTD simulation space including the boundary regions ($70 \times 70 \times 70$ in our experiments), and n_b is the size of the boundary in each side (10 in our case). Each coefficient, except Q_p, s_p, c_P, is a 3-dimensional vector. For example, b_h represents coefficients (b_{hx}, b_{hy}, b_{hz}).

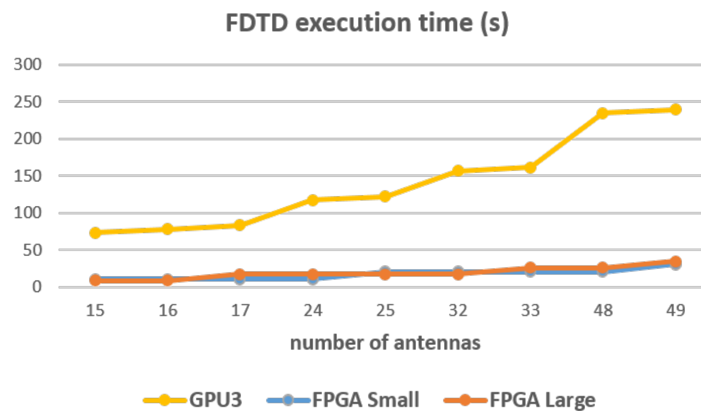
Table 3.7 Dimensions of FDTD coefficients.

Coefficient	**type**	**size**	**Coefficient**	**type**	**size**
$a_h, d_h, c_p, s_p,$	scalar	1	$\mathbf{b_h}, \mathbf{a_e}, \mathbf{b_e}, \mathbf{d_e}$	vector	$3N$
$c_{h1,2}, c_{e1,e2}$	vector	n_b	$\mathbf{Q_P}$	vector	N

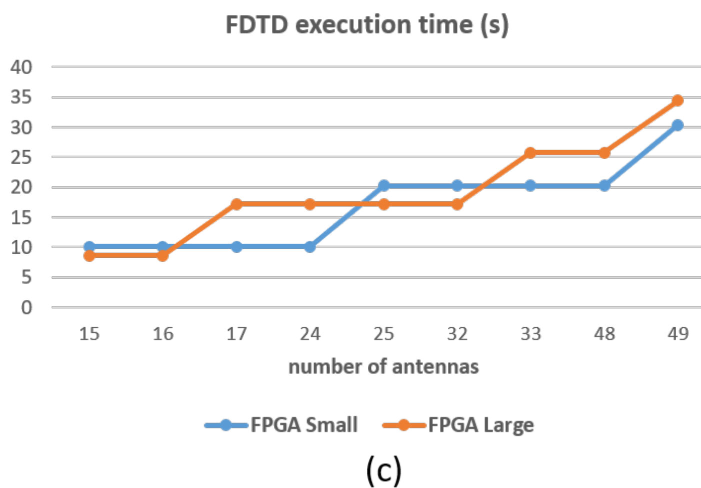
The FDTD coefficients with the largest size in Table 3.7 are highlighted in bold. These coefficients are 3-dimensional vectors, except for Q_p, which is a 1-dimensional vector. Other coefficients have small size and do not affect the transfer time significantly. Therefore, the data to be transferred via PCIe amounts to $(4 \times 3 + 1) \times N = 13 \times N$ floating-point constants. By considering the maximum PCIe data transfer rate in the AWS F1 instance (12 GB/s), we can obtain the maximum data transfer time for each antenna, which is $T_{PCIe} = 1.4$ ms. By comparing T_{PCIe} with the total FDTD execution time for each antenna (8.6 s in the best case), we can see the the data transfer time is negligible.



(a)



(b)



(c)

Fig. 3.14 FDTD execution time for 8 FPGAs and different number of antennas, (a) from one antenna up to the maximum number in a single FPGA (3 for Small design), (b) Comparison of the single GPU in this work (GPU3, highly optimized for one antenna) and multi-FPGA design for multiple antennas, (c) more detailed view of multi-FPGA design results.

Fig. 3.14 shows the execution time for multiple antennas accelerated by 8 FPGAs and compares it with the best GPU results (Tesla k20c in this work, GPU3). With the exception of the cases in which one or two antennas are processed, the FPGA designs show always a higher performance. For many antennas to process, the multi-FPGA accelerator can significantly reduce the execution time. It should be noted that when the number of antennas is a multiple of 16 or 24, there is an increase in the execution time as predicted by Eqn. (3.6).

Finally, we report system-level performance results for the MI reconstruction algorithm. With 24 antennas, on the Tesla P40 the execution time of the Acceleware FDTD code was 135.4 s per iteration, while the inversion part, executed by the host, is negligible (0.07 s). In our Small Multi-FPGA design, the FDTD takes instead 10.14 s, which corresponds to a speed-up of more than 13x. Compared to the GPU design in this work (GPU3) the speed-up is 11.5x.

3.6 Discussion

In this section, we would like to summarize the main characteristics of our design of an FDTD accelerator, the main challenges faced during the design, and the solutions to these challenges by using HLS. First of all, it is important to note that although it is possible to take any synthesizable code written in C, C++, OpenCL, or systemC, and accelerate it in hardware by using HLS, the performance highly depends on the algorithm and it is not always possible to obtain the desired performance using HLS. It depends on the algorithm, degree of parallelization, hardware optimizations, target FPGA device, available resources, and achievable clock frequency.

Secondly, regarding the characteristics of our design, it should be noted that for our GPU design, all the resources are used to optimize the performance for one antenna. As the GPU resources are already fully used, there is no space left for further parallelization. On the contrary, in our FPGA design, we leveraged the multi-antenna parallelism to reduce the overall execution time. This is highly beneficial in Microwave Imaging systems due to the large number of required antennas. This parallelism on the number of antennas in addition to the number of cells in the 3D volume is one of the key characteristics of our FPGA design for FDTD algorithm. Other characteristics of our design are related to the design methodology that we adopted for the acceleration using HLS. Due to the iterative

nature of FDTD algorithm, it is possible to parallelize the computations in the volume cells by "unrolling" and "pipelining" the loops. Merging the loops, local storage of boundaries and constant coefficients, and using a blocking strategy to reduce the memory access time are among other characteristics that could optimize the hardware acceleration.

Thirdly, we describe the challenges of the design and their solutions in this work. One of the main challenges in this application (3D FDTD) is the access to a high volume of data from external memory. This high memory bandwidth requirement becomes more challenging when we use the more advanced FDTD computations, related to the polarization currents (which has not been considered in previous works). To overcome this issue in this work, we used HLS capabilities to:

- efficiently process the extra computations on polarization currents by merging JP currents loops in Update E and E-boundary. Table 6.3 shows the higher GFLOPs for this work that is related to these extra computations.
- define high number of AXI ports to meet bandwidth requirement (Fig. 3.4, 15 or 18 ports)
- create a spatial blocking approach to reduce the data transfer time between external and local memories.

Another challenge is related to the computations in the boundary regions in 3D volume. Due to the access pattern in these regions, the parallelization of boundary computations is more challenging. By using HLS, we could:

- Merge the parallel loops in the boundary regions for parallel computations
- use local memories for boundary regions whenever possible to store the required data

There are some other challenges related to the hardware "implementation" stage when using more storage resources and higher number of ports. These configurations must be carefully selected in order to avoid routing failures in the implementation step. We proposed two hardware architectures (Large and Small) with different hardware configurations for a more flexible design, both of which are implementable in FPGA. The combination of these strategies with the usage of other HLS-based

optimizations described in this chapter makes it possible for our FPGA design to have a comparable performance to the GPU design and other HLS-based approaches.

Finally, there were several other HLS works tackling these problems for FDTD (and Stencils) acceleration in FPGA. Most of them focused on solving the issue of memory access time by presenting different blocking methods to process blocks of data from the 3D volume in multiple iterations. These methods include the combination of "spatial" and "temporal" blocking. Specifically, the OpenCL-based designs proposed in [60], [62], and [63] are among the successful works in the HLS domain for these blocking strategies. The problem of complex boundary conditions is still an open issue in hardware accelerators designed by HLS and the related works simplified the boundary conditions in favor of more parallelization. Using these methods in previous works could reduce the total processing time. However, ignoring the impact of polarization currents makes them ineffective in medical Microwave Imaging applications. The complex data dependencies created by the polarization currents call for a more straightforward approach to tackle the issues of memory transfer time, boundary conditions, and polarization currents at the same time, that is what we focused on in this work.

3.7 Conclusions

In this chapter, we proposed a multi-FPGA hardware accelerator for 3D FDTD to be used in MI for medical applications. It is designed entirely in HLS making it possible to use several hardware optimization methods to obtain the best performance. The distinctive features of FDTD in this work are the modeling of polarization currents in dispersive materials, and the use of CPML boundary conditions in all directions. The combination of these features add extra complexity to the hardware design, but the HLS optimizations, including loop merge, blocking, pipelining and local memory storage, results in an efficient accelerator that is comparable with GPU or CPU-based design. **Compared to the best GPU design** of the same FDTD algorithm, **a single FPGA** can achieve **1.44x lower execution time per antenna**. Our FPGA design is **more energy efficient** than CPU or GPU-based designs, and the maximum power for the FPGA design is still lower than the GPUs. In addition, the **multi-FPGA design outperforms the other accelerators** by processing multiple antennas in parallel.

For a typical number of 24 antennas, **11.5x reduction of execution time can be achieved compared to the best GPU design.**

Chapter 4

High Level Design of a Flexible PCA Hardware Accelerator

Principal Component Analysis (Principal Component Analysis (PCA)) is a widely-used method for reducing dimensionality. It extracts from a set of observations the Principal Components (PCs) that correspond to the maximum variations in the data. By projecting the data on an orthogonal basis of vectors corresponding to the first few PCs obtained with the analysis and removing the other PCs, the data dimensions can be reduced without a significant loss of information. Therefore, PCA can be used in various applications when there is redundant information in the data. For example, in Microwave Imaging (MI), PCA is useful before image reconstruction to reduce data dimensions in microwave measurements [69–71]. In a recent work [72], PCA is used as a feature extraction step prior to tumor classification in MI-based breast cancer detection.

PCA involves various computing steps consisting of complex arithmetic operations, which result in a high computational cost and so a high execution time when implementing PCA in software. To tackle this problem, hardware acceleration is often used as an effective solution that helps reduce the total execution time and enhance the overall performance of PCA.

The main computing steps of PCA, which will be described thoroughly in the next section, include the computation of an often large covariance matrix, which stresses the I/O of hardware systems, and the computation of the singular values of the matrix. Since the covariance matrix is symmetric, the singular values are also

the eigenvalues and can be computed either with Eigenvalue Decomposition (EVD) or with Singular Value Decomposition (SVD): the more appropriate algorithm to implement in hardware is chosen depending on the application and the performance requirements. Other important steps in PCA are the data normalization, which requires to compute the data mean, and the projection of the data on the selected PCs.

In recent years, numerous hardware accelerators have been proposed that implement either PCA in its entirety or some of its building blocks. For example, in [73] different hardware implementations of EVD were compared and analyzed on CPU, GPU, and Field-Programmable Gate Arrays (FPGA), and it was shown that FPGA implementations offer the best computational performance, while the GPU ones require less design effort. A new FPGA architecture for EVD computation of polynomial matrices was presented in [74], in which the authors show how the Xilinx System Generator tool can be used to increase the design efficiency compared to traditional RTL manual coding. Leveraging a higher abstraction level to improve the design efficiency is also our goal, which we pursue using the High-Level Synthesis (HLS) design approach, as we discuss later, as opposed to VHDL- or Verilog-based RTL coding. Wang and Zambreno [75] introduce a floating-point FPGA design of SVD based on the Hestenes-Jacobi algorithm. Other hardware accelerators for EVD and SVD were proposed in [76–79].

In [80] an embedded hardware was designed in FPGA using VHDL for the computation of Mean and Covariance matrices as two components of PCA. Fernandez et al. [81] presented a manual RTL design of PCA for Hyperspectral Imaging (HI) in a Virtex7 FPGA. The Covariance and Mean computations could not be implemented in hardware due to the high resource utilization. Das et al. [82] designed an FPGA implementation of PCA in a network intrusion detection system, in which the training phase (i.e., computing the PCs) was done offline and only the mapping phase (i.e., the projection of the data on the PC base) in the online section was accelerated in hardware. Our goal is instead to provide a complete PCA implementation, which can be easily adapted to the available FPGA resources thanks to the design flexibility enabled by the HLS approach.

Recently, some FPGA accelerators have been introduced that managed to implement a complete PCA algorithm. In [83] such an accelerator was designed in a Virtex7 FPGA using VHDL, but it is applicable only to relatively small matrix

dimensions. Two block memories were used for the internal matrix multiplication to store the rows and columns of the matrices involved in the multiplication, which resulted in a high resource usage. Thanks to our design approach, instead, we are able to implement a complete PCA accelerator for large matrices even with few FPGA resources.

FPGAs are not the only possible target for PCA acceleration. In [84], all the PCA components were implemented on two different hardware platforms, a GPU and a Massively Parallel Processing Array (MPPA). Hyperspectral images with different dimensions were used as test inputs to evaluate the hardware performance. It is well known, however, that these kinds of hardware accelerators are not as energy-efficient as FPGAs. Therefore we do not consider them, especially because our target computing devices are embedded systems in which FPGAs can provide an efficient way for hardware acceleration.

Recently, HLS-based accelerators have been proposed for PCA. In [9], a design based on HLS was introduced for a gas identification system and implemented on a Zynq SoC. Schellhorn et al., presented in [10] another PCA implementation on FPGA using HLS for the application of spectral image analysis, in which the EVD part could not be implemented in hardware due to the limited resources. In [85], we presented an FPGA accelerator by using HLS to design the SVD and projection building blocks of PCA. Although it could be used for relatively large matrix dimensions, the other resource-demanding building blocks (especially covariance computation) were not included in that design. In a preliminary version of this work [86], we proposed another HLS-based PCA accelerator to be used with flexible data dimensions and precision, but limited in terms of hardware target to a low-cost Zynq SoC and without the support for block-streaming needed to handle large data matrices and covariance matrices, which instead we show in this work.

The PCA hardware accelerators proposed in the literature have some drawbacks. The manual RTL approach used to design the majority of them is one of the disadvantages, which leads to an increase in the total development time. Most of the previous works, including some of the HLS-based designs, could not implement an entire PCA algorithm including all the computational units. Other implementations could not offer a flexible and efficient design with a high computational performance that could be used for different data sizes.

In this work we close the gap in the state of the art and propose an efficient FPGA hardware accelerator that has the following characteristics:

- The PCA algorithm is implemented in FPGA in its entirety.
- It uses a new block-streaming method for the internal covariance computation.
- It is flexible because it is entirely designed in HLS and can be used for different input sizes and FPGA targets.
- It can easily switch between floating-point and fixed-point implementation, again thanks to the HLS approach.
- It can be easily deployed on various FPGA-based boards, which we prove by targeting both a Zynq7000 and a Virtex7 in their respective development boards.

The rest of this chapter is organized as follows. At first, in Section 4.1 the PCA algorithm is described with the details of its processing units. We briefly describe in Section 4.2 the application of PCA to Hyperspectral Imaging (HI), which we use as a test case to report our experimental results and to compare them with previously reported results. The proposed PCA hardware accelerator and the block-streaming method is presented in Section 4.3 together with the HLS optimization techniques. The implementation results and the comparisons with other works are reported in Section 4.4. Finally, the conclusions are drawn in Section 4.5.

4.1 PCA Algorithm Description

Let X be an array of size $R \times C$ in which R (*Rows*) is the number of data samples and C (*Columns*) is the main dimension in which there is redundant information. PCA receives X and produces a lower-dimensionality array Y of size $R \times L$ with $L < C$ through the steps shown in Algorithm 6.

In the first step, the mean values of each column of the input data are computed and stored in matrix M for data normalization. The second step is the computation of the covariance of the normalized data, which is one of the most computationally expensive steps of PCA due to the large number of multiplications and additions. After

computing the eigenvalues or singular values (and the corresponding eigen/singular vectors) of the covariance matrix by using EVD or SVD in the third step, they are sorted in descending order and the first L eigen/singular vectors are selected as Principal Components in the fourth step. The selection of PCs is based on the cumulative energy of eigen/singular values. After computing the total energy (E) as in the following equation,

$$E = \sum_{i=1}^{C} \sigma_i, \tag{4.1}$$

where σ_i is the energy of the ith eigen/singular value, the first L components are selected in such a way that their cumulative energy is no less than a predetermined fraction of total energy, the threshold T (%), as follows:

$$100 \times \frac{\sum_{i=1}^{L} \sigma_i}{E} \geq T. \tag{4.2}$$

Finally, in the last step, the normalized input is projected into the principal components space to reduce the redundant information.

Algorithm 6: PCA algorithm

Step 1- Mean computation: /* $M_{R\times C}$ is the matrix representation of vector $Mean_{1\times C}$ */

$[Mean]_{1\times C} = \frac{1}{R}\sum_{i=1}^{R}[X_i]_{1\times C}$ /* $[X]_i$ is the ith row of the input matrix $X_{R\times C}$ */

$M_{R\times C} : [M_i]_{1\times C} = [Mean]_{1\times C}, i = 1,2,...,R$ /* $[M_i]$ is the ith row of matrix M*/

Step 2- Covariance calculation:

$[COV]_{C\times C} = \frac{1}{R-1}(X-M)^T \times (X-M)$

Step 3- EVD/SVD of covariance:

$COV = U\Sigma U^T$

Step 4- Sort and selection:

$\Sigma^s, U^s = Sort(\Sigma, U)$

$[PC]_{C\times L} = Select(\Sigma^s, U^s)$

Step 5- Projection:

$[Y]_{R\times L} = (X-M)\times PC$

4.2 Hyperspectral Imaging

Although we are interested in the application of PCA to Microwave Imaging (MI), to the best of our knowledge there is no hardware accelerator for PCA in the literature that is specifically aimed at such application. In order to compare our proposed hardware with state-of-the-art PCA accelerators, we had to select another application for which an RTL- or HLS-based hardware design was available. Therefore, we selected the Hyperspectral Imaging (HI) application.

As described in Chapter 2, Hyperspectral Imaging (HI) data are provided in multiple spectral bands, and each pixel of the image consists of several bands. Thus, HI data exploitation helps to remotely identify the ground materials-of-interest based on their spectral properties [87]. Usually, there is redundant information in different bands that can be removed with PCA. In the following notations, we use interchangeably the terms $R(Rows)$ and pixels, as well as $C(Columns)$ and bands.

In general, we assume that HI data can be represented as a matrix with R rows (pixels) and C columns (spectral bands), and there are redundant information in the columns (bands). For Microwave Imaging, the $R \times C$ matrix could represent data gathered in C different frequencies in the microwave spectrum by R antennas, or a reconstructed image of the scattered electromagnetic field of R pixels also at C frequencies.

4.3 PCA Hardware Accelerator Design

Figure 4.1 shows the architecture of the PCA accelerator and its main components. Since the accelerator is developed in HLS using C++, the overall architecture correspond to a C++ function and its components correspond to subfunctions. At the highest level, there are two subfunctions named *Dispatcher* and *PCA core*. The Dispatcher reads the input data stored in an external DDR memory and sends them to the PCA core through the connecting FIFOs. The connection with FIFOs is also described in HLS with proper code pragmas.

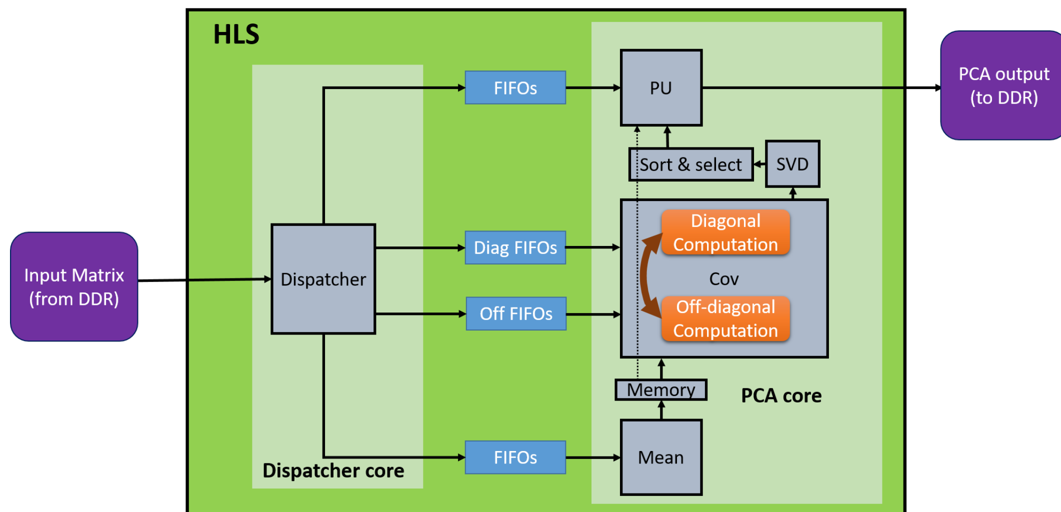


Fig. 4.1 Architecture of the proposed hardware accelerator for Principal Component Analysis (PCA) in Field-Programmable Gate Arrays (FPGA).

The PCA core contains different processing units. The first is the *Mean* unit, which computes the mean vector corresponding to the mean value of each column of the input matrix. This vector is stored in an internal memory that is used by the next processing units, *Cov* and *PU*, for data centering. The Cov unit uses the new block-streaming method for computing the covariance matrix, which will be explained thoroughly in the following subsection. It reads the required data from two sets of FIFOs corresponding to diagonal and off-diagonal computation. Then the *SVD* unit computes the singular values of the covariance matrix, and the *Sort and Select* unit sorts them and retains the first components. Finally, the *Projection* unit reads the input data again and computes the multiplication between the centered data and the sorted and selected PCs to produce the final PCA output data, which are written back to the external DDR memory.

The computational cost of PCA depends on the input dimensions. When the main dimension from which the redundant information must be removed (columns or bands in HI) is lower than the other dimension (rows or pixels in HI), the PCA performance is mainly limited by the computation of the covariance matrix, due to the large number of multiplications and additions that are proportional to the number of pixels. Indeed, in the covariance computation all the pixels in one band must be multiplied and accumulated with the corresponding pixels in all of the bands. This is illustrated in Figure 4.2 where the bands are specified by the letters α to n and pixels are indicated by the indices 1 to N. The result of the covariance computation, which

is the output of the Cov unit, is a matrix of size (*bands* $\times$ *bands*) that becomes the input of the SVD unit. In HI applications, in which it is true that *pixels* $\gg$ *bands*, the covariance computation is the major limitation of the whole PCA design, hence its acceleration can drastically enhance the overall performance.

Multiple parallel streaming FIFOs are needed to match the parallelism of the accelerated Cov unit. The number of FIFOs is determined based on the input data dimensions and the maximum bandwidth of the external memory. Streaming more data at the same time through the FIFOs enables a degree of parallelism that is matched to the number of columns of the input matrix. It is important to note that all of the hardware components are described in a single HLS top-level function, which simplifies the addition of different flexibility options to the design such as variable data dimensions, block sizes, and number of FIFOs.

The complexity of the Cov unit compared to other parts raises the importance of the block-streaming method in covariance computation as this method allows using the same design for higher data dimensions or improving the efficiency in low-cost embedded systems with fewer hardware resources.

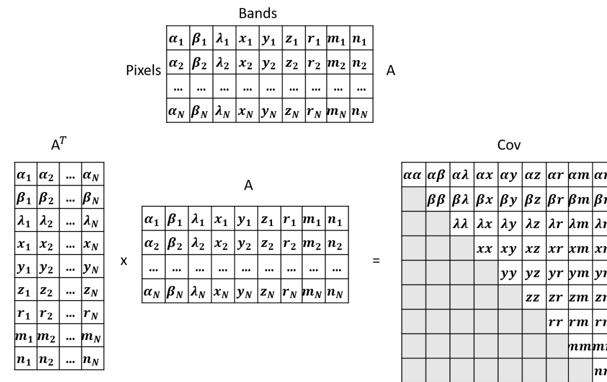


Fig. 4.2 Example of covariance computation with 9 bands and N pixels, $PQ = \Sigma_{i=1}^{N} P_i \times Q_i$, where P, Q are the symbols of bands (α to n).

4.3.1 Block-Streaming for Covariance Computation

The block-streaming method is helpful whenever there is a limitation in the maximum size of input data that can be stored in an internal memory in the Cov unit. Therefore, instead of streaming the whole data one time, we stream "blocks" of data several times through the connecting FIFOs. There are two internal memories inside the

Cov unit each of which can store a maximum number of bands (B_{max}) for each pixel. These memories are used in the diagonal and off-diagonal computations, so we call them “Diag” and “Off-diag” RAMs, respectively. The input data is partitioned into several blocks along the main dimension (bands) with a maximum dimension of B_{max} (block size). Each block of data is streamed through the two sets of FIFOs located between the Dispatcher and Cov unit (Diag and Off-diag FIFOs) in a specific order, and after the partial calculation of all the elements of the covariance matrix for one pixel, the data blocks for the next pixels will be streamed and the partial results accumulated together to obtain the final covariance matrix.

To better understand the block-streaming method, we provide two examples in Figures 4.3–4.6.

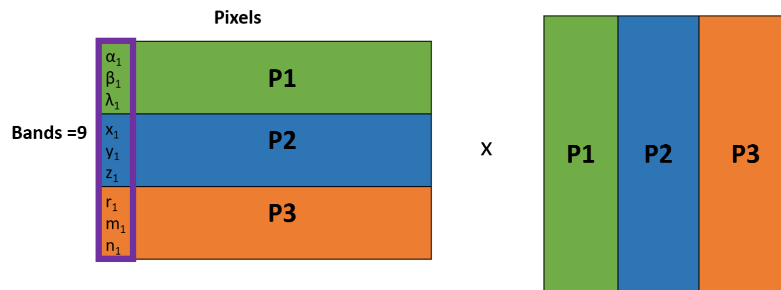


Fig. 4.3 Example of partitioning of input data into blocks. The total number of bands is $B = 9$ and the block size is $B_{max} = 3$.

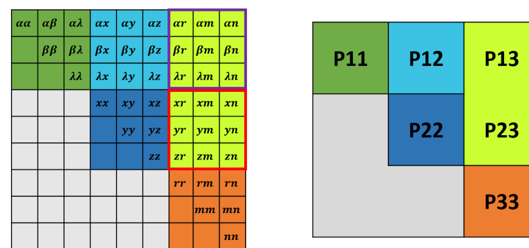


Fig. 4.4 Illustration of an example of covariance computation using the block-streaming method with 3 blocks ($B = 9$, $B_{max} = 3$).

The first example is illustrated in Figure 4.3 in which the total number of bands is $B = 9$ and the block size is $B_{max} = 3$. Therefore, we have 3 blocks of data that are specified in figure as P1 to P3. The Block-streaming method consists of the following steps that can be realized from Figure 4.4:

1. **Diagonal computation:**

The 3 blocks of data (P1 to P3) for the first pixel are streamed in Diag FIFOs one by one and after storage in the Diag RAM, the diagonal elements P11, P22, and P33 are computed.

2. **Off-diagonal computation of the last block:**
   (a) Keep the last block (P3) in the Diag RAM.
   (b) Stream the first block (P1) into Off-Diag FIFOs, store it in Off-Diag RAM, and compute $P13 = P1 \times P3$.
   (c) Stream the second block (P2) into Off-Diag FIFOs, store it in Off-Diag RAM, and compute $P23 = P2 \times P3$.

3. **Off-diagonal computation of the preceding blocks:**
   (a) Update the Diag RAM by the last values of Off-Diag RAM (P2).
   (b) Stream the first block (P1) into Off-Diag FIFOs, store it in Off-Diag RAM, and compute $P12 = P1 \times P2$.

4. **Stream Pixels:** Steps 1 to 3 are repeated for the next pixels and the results are accumulated to obtain the final covariance matrix.

The second example is illustrated in Figure 4.5 in which the number of blocks is 4 and after the diagonal computation (in green color) there are 3 steps for off-diagonal computations that are indicated in 3 different colors. Figure 4.6 shows the order of data storage in the Diag and Off-diag RAMs. After the 7th and 9th steps, the Diag RAM is updated by the last value of Off-Diag RAM (P3 and P2).

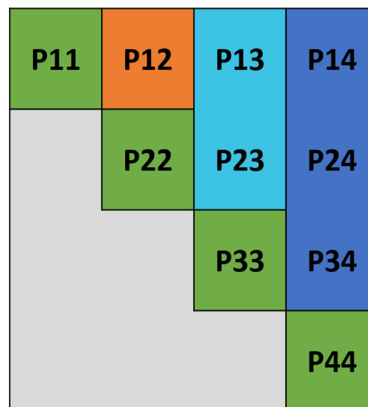


Fig. 4.5 Block-streaming method with 4 blocks ($B/B_{max} = 4$).

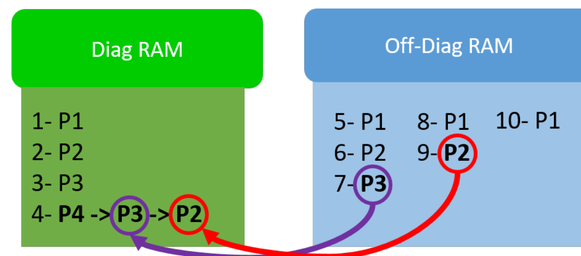


Fig. 4.6 Order of data storage in the Diagonal and Off-diagonal RAMs inside the Cov unit.

4.3.2 High Level Synthesis Optimizations

Tool Independent Optimization Directives and Their Hardware Implementation

The PCA hardware accelerator is designed in C++ using Vivado HLS, the HLS development tool for Xilinx FPGAs. The HLS optimization directives can be applied easily in HLS by using their corresponding code *pragmas*. HLS enables us to specify the hardware interfaces as well as the level of parallelism and pipelined execution and specific hardware resource allocation thanks to the addition of code pragmas. By exploring different combinations of the optimization directives, it is possible to determine relatively easily the best configuration in terms of latency and resource usage. Therefore, several interface and hardware optimization directives have been applied in the HLS synthesis tool, as explained below.

Input Interfaces

The main input interface associated to the Dispatcher input and the output interface associated to the PU output consist of AXI master ports, whose number and parallelism are adapted to the FPGA target. For the Zynq of the Zedboard, four AXI ports (with a fixed width of 64 bits) are connected to the Dispatcher input in such a way to fully utilize the available memory bandwidth. In the Virtex7 of the VC709 board we can use instead only one AXI port with a much larger bit-level parallelism. The output interface for both boards is one AXI master port that is responsible for writing the output to the memory. Other interfaces are specified as internal FIFOs between the Dispatcher and the PCA core. As shown in Figure 4.1, four sets of FIFOs send the streaming data (containing a data block of bands) from the Dispatcher to the corresponding processing units in the PCA core. Mean and

Projection units receive two sets of FIFOs and Cov unit receives another two. Each set of FIFOs is partitioned by B_{max}, the size of a data block, so that there are B_{max} FIFOs in each set.

These FIFOs are automatically generated by applying the Vivado HLS *Dataflow* directive for the connection between the Dispatcher and the PCA core. This directive lets the two functions execute concurrently and their synchronization is made possible by the FIFO channels automatically inserted between them.

Code Description and Hardware Optimizations

In this part the code description for each PCA component is presented. To optimize the hardware of each PCA unit, we analyzed the impact of different HLS optimizations on the overall performance. Specifically, we considered the impact of loop pipelining, loop unrolling, and array partitioning on latency and resource consumption. The best HLS optimizations are selected in such a way that the overall latency is minimized by utilizing as many resources as required.

The Mean unit computes the mean values of all the pixels in each band. The code snippet for the Mean unit is presented in Algorithm 7 and consists of two loops on the rows and columns to accumulate the pixel values and another loop for the final division by the number of rows.

Algorithm 7: Mean computation

```
mean_row_loop:
for r=0 to R do
    #pragma HLS PIPELINE
    mean_col_loop:
    for c=0 to C do
        a_mean[c] = Din_Mean[r][c];
        tmp_mean[c]+ = a_mean[c];
Divide_loop:
for c=0;c<C;c++ do
    a_mean[c] = tmp_mean[c]/R;
```

The best HLS optimization for the Mean unit is to *pipeline* the outer loop (line #pragma HLS PIPELINE in Algorithm 7), which reduces the Initiation Interval (II),

i.e., the index of performance that corresponds to the minimum time interval (in clock cycles) between two consecutive executions of the loop (ideally, II = 1). In addition, the memory arrays *a_mean* and *tmp_mean* are partitioned by B_{max} (not shown in the code snippet) to have access to multiple memory locations at the same time, which is required for the loop pipelining to be effective, otherwise the II will increase due to the latency needed to access a single, non-partitioned memory.

The Cov unit uses the block-streaming method to compute the covariance matrix. Its pseudo code is presented in Algorithm 8. The HLS optimizations include loop pipelining, unrolling, and the arrays full partitioning. In Algorithm 8 full indexing is not displayed to make the code more readable and only the relation between the variables and the indexes is shown by the parentheses. For example, DiagFIFO(r,b) indicates that the indexes of variable DiagFIFO are proportional to (r,b). The standard Cov computation is adopted from [86] and is used for diagonal covariance computations. The write function in the last line writes the diagonal and off-diagonal elements of covariance matrix from variables CovDiag and CovOff to the corresponding locations in CovOut. As shown in Algorithm 8, there are two pipeline directives that are applied to the loops on the diagonal and off-diagonal blocks, respectively. The memory arrays need to be fully partitioned, which is required to unroll the inner loops. As described before, a thorough analysis of different possible optimizations was performed to find out a trade-off between resource usage and latency.

Algorithm 8: Cov computation, block-streaming

```
for r=0 to R do /* Stream Pixels */
    for b=0 to NB do /* Diagonal Computations, NB = B/Bmax */
        #pragma HLS PIPELINE
        DiagRAM = DiagFIFO(r,b) - a_mean(b);
        /* Start standard Cov computation [86] */
        for c1=0 to B_max do
            for c2=c1 to B_max do
                .../* indexing */
                CovDiag(b,Index) = DiagRAM[c1] * DiagRAM[c2];
        /* Finish standard Cov computation */
    for ct=1 to NB do /* Off-Diagonal computations */
        for b=0 to NB-ct do
            #pragma HLS PIPELINE
            if Step3(a) then /*refer to section 4.1, the four steps of
              block-streaming method*/
                DiagRAM = OffRAM;
            OffRAM = OffFIFO(r,b) - a_mean(b);
            CovOff(b,ct)+ = OffRAM * DiagRAM;
/* Write to the final Cov matrix */
CovOut = write(CovDiag,CovOff);
```

The next processing unit is the EVD of the covariance matrix. For real and symmetric matrices (like the covariance matrix) EVD is equal to SVD and both methods can be used. For SVD computation of floating-point matrices, there is a built-in function in Vivado HLS that is efficient especially for large dimensions. On the other hand, a fixed-point implementation of EVD is highly beneficial for embedded systems due to the lower resource usage (or better latency) compared to the floating-point version of the same design.

For these reasons, in this work we propose two versions of the PCA accelerator. The first one is the floating-point version, which uses the built-in HLS function for SVD computation. The second one is the fixed-point version, which uses the general two-sided Jacobi method for EVD computation [88, 89]. The details of this method for computing EVD is shown in Algorithm 9. As we will show in the results section,

there is no need to add any HLS optimization directives to the fixed-point EVD (for our application) because the overall latency of the PCA core, which includes EVD, is lower than the data transfer time, so there is no benefit in consuming any further resources to optimize the EVD hardware. We do not report the code of the floating-point version as it uses the built-in Vivado HLS function for SVD [1].

Algorithm 9: EVD computation, Two-sided Jacobi method

```
/*Initialize the eigenvector matrix V and the maximum iterations
  max = bands*/
V = I; /* I is the identity matrix */
for l=1 to max do
    for all pairs i<j do
        /* Compute the Jacobi rotation which diagonalizes
          [H_ii H_ij; H_ji H_jj] = [a c; c b] */ τ = (b − a)/(2 ∗ c);
        t = sign(τ)/(|τ| + sqrt(1 + τ^2));
        cs = 1/sqrt(1 + t^2); sn = cs ∗ t;
        /* update the 2 × 2 submatrix */
        H_ii = a − c ∗ t;
        H_jj = b + c ∗ t;
        H_ij = H_ji = 0;
        /* update the rest of rows and columns i and j */
        for k=1 to bands except i and j do
            tmp = H_ik;
            H_ik = cs ∗ tmp − sn ∗ H_jk;
            H_jk = sn ∗ tmp + cs ∗ H_jk;
            H_ki = H_ik; H_kj = H_jk;
        /* update the eigenvector matrix V /*
        for k=1 to bands do
            tmp = V_ki;
            V_ki = cs ∗ tmp − sn ∗ V_kj;
            V_kj = sn ∗ tmp + cs ∗ V_kj;
```

[1] Vivado Design Suite User Guide: High-Level Synthesis, UG902 (v2019.1), Xilinx, San Jose, CA, 2019 https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug902-vivado-high-level-synthesis.pdf

The last processing unit is the Projection Unit (PU), which computes the multiplication between the centered data and the principal components. Algorithm 10 presents the code snippet for the PU. Similar to the Mean unit, we applied some optimizations to this code. The second loop is pipelined and, as a consequence, all the inner loops are unrolled. In addition, the memory arrays involved in the multiplication must be partitioned. For more information on the hardware optimizations for Mean and PU and their impact on the latency and resource usage please refer to [86].

Algorithm 10: Projection computation

```
for r=0 to R do
    for c1=0 to L do
        #pragma HLS PIPELINE
        tmp = 0;
        for n=0 to C do
            .../* Index control */
            Din_Nrml[n] = Din_PU[r][n] - a_mean[n];
        for c2=0 to C do
            tmp+ = (Din_Nrml[c2] * PC[c2][c1];
        Data_Transformed[r][c1] = tmp;
```

4.3.3 Fixed-Point Design of the Accelerator

There are many considerations when selecting the best numerical representation (floating- or fixed- point) in digital hardware design. Floating-point arithmetic is more suited for applications requiring high accuracy and high dynamic range. Fixed-point arithmetic is more suited for low power embedded systems with higher computational speed and fewer hardware resources. In some applications, we need not only speed, but also high accuracy. To fulfill these requirements, we can use a fixed-point design with a larger bit-width. This increases the resource usage to obtain a higher accuracy, but results in a higher speed thanks to the low-latency fixed-point operations. Therefore, depending on the requirements, it is possible to select either a high-accuracy low-speed floating-point, a low-accuracy high-speed fixed-point, or a middle-accuracy high-speed fixed-point design. Available resources in the target hardware determine which design or data representation is implementable on the device. In high-accuracy high-speed applications, we can use the fixed-point design

with a high resource usage (even more than the floating-point) to minimize the execution time.

To design the fixed-point version of the PCA accelerator, the computations in all of the processing units must be in fixed-point. The total Word Length (WL) and Integer Word Length (IWL) must be determined for every variable. The range of input data and the data dimensions affect the word lengths in fixed-point variables, so the fixed-point design may change depending on the data set.

For our HI data set with 12 bands, we used the MATLAB fixed-point converter to optimize the word lengths. In HLS we selected the closest word lengths to the MATLAB ones because some HLS functions do not accept all the fixed-point representations (for example in the fixed-point square root function, IWL must be lower than WL).

The performance of EVD/SVD depends only on the number of bands. As we will see in the next section, the latency of our EVD design in HLS is significantly higher than the HLS built-in SVD function for floating-point inputs. One possible countermeasure is to use the SVD as the only floating-point component in a fixed-point design to obtain better latency, by adding proper interfaces for data-type conversion. However, when the data transfer time (i.e., the Dispatcher latency) is higher than the PCA core latency, the fixed-point EVD is more efficient because of the lower resource usage, which is the case for a small number of bands.

4.3.4 Hardware Prototype for PCA Accelerator Assessment with the HI Data Set

The proposed hardware design is adaptable to different FPGA targets and its performance will be evaluated in the results section in particular for two test hardware devices. In this subsection, as an example of system-level implementation using our flexible accelerator we introduce its implementation on a low-cost Zynq SoC mounted on the Zedboard development board. We used this implementation to evaluate our PCA accelerator on the HI data set. The details are illustrated in Figure 4.7. The input data set is stored in an SD card. The Zynq Processing System (PS) reads the input data from the SD card and writes them to the DDR3 memory. The Zynq FPGA reads the data from the DDR3 memory by using two High Performance (HP) ports (Zynq SoC devices internally provide four HP interface ports that connect

the Programmable Logic (PL) to the Processing System (PS)). As the HI data consist of images in 12 bands and each pixel has an 8-bit data width, to match the processing parallelism we need an I/O parallelism of 12 × 8 = 96 bits to read all the bands at once. Therefore, we use two 64-bit HP ports for the input interface. After the PCA computation in the accelerator, the output is written back to the DDR3 memory through the first HP port by using an *AXI Smart Connect* IP block (AXI smart connect IP block connects one or more AXI memory-mapped master devices to one or more memory-mapped slave devices). Finally, the Zynq PS reads the PCA outputs from the DDR3 and writes them to the SD card.

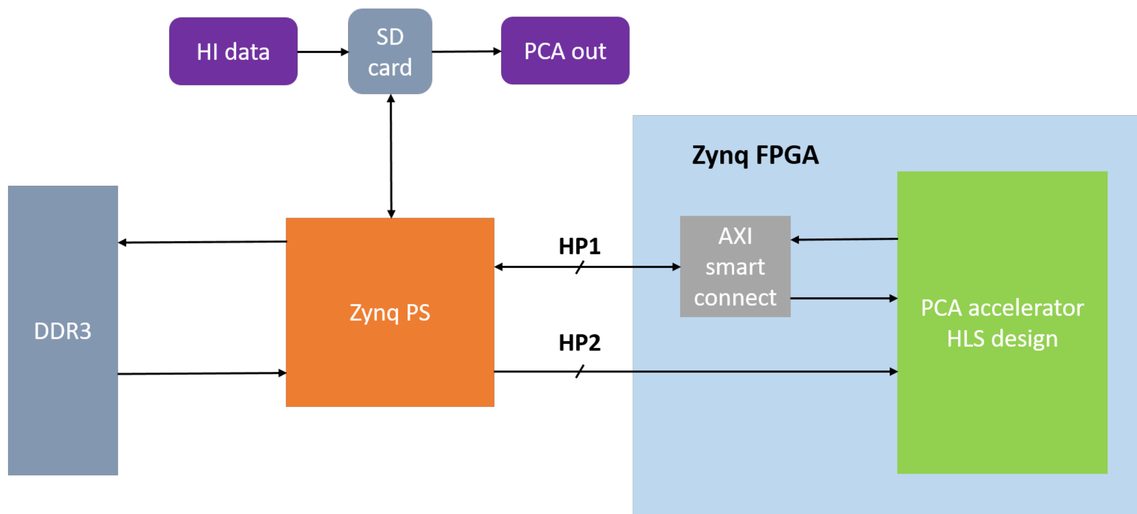


Fig. 4.7 PCA accelerator design in Zedboard.

For other applications or different FPGA targets, the connection of the PCA accelerator to the I/O system or to a processor is easily adapted thanks to the flexibility enabled by the HLS approach. It is important to note that in many designs the hardware might be so fast that its performance becomes limited by the memory speed. To avoid this problem, it is necessary that the consumption rate of the hardware that fetches the data from the DDR memory is matched to the speed of that memory. In particular, in our design, by considering the maximum bandwidth B_{DDR} (Gb/s) of the external DDR memory and the clock frequency F (GHz) of the Dispatcher unit, we can obtain the maximum bit-width as $bw_{max} = B_{DDR}/F$ for the PCA accelerator input connected to the Dispatcher input. Depending on the input data set and the number of bands, we can obtain the maximum required I/O parallelism. For example, if the number of bands is B and the data width of each pixel is DW bits, we need a maximum of $bw = B \times DW$ to read one pixel for all the

bands at once in one clock cycle. The final input bit-width of the Dispatcher (bw_{Disp}) is selected in such a way that we do not exceed the memory bandwidth. Therefore, if $bw \leq bw_{max}$, then $bw_{Disp} = bw$. Otherwise, we have to fix the Dispatcher input bit-width to $bw_{Disp} = bw_{max}$ (note that the Dispatcher input is an AXI port and its data width must be a power of 2 and a multiple of 8. In addition, some FPGA targets, like the Zedboard, can read data from memory using separate AXI ports (HP ports)). It should be noted that all the above-mentioned conditions can be easily described in HLS using a set of variables and C++ macros that are set at the design time. In order to map the design into a new FPGA target, the only required change is to adjust the pre-defined variables based on the hardware device.

4.4 Results

The proposed PCA accelerator is implemented using Vivado HLS 2019.1. To demonstrate the flexibility of the proposed method, we did the experiments on two Xilinx FPGA devices and their development boards, the relatively small Zynq7000 (XC7z020clg484-1) on a Zedboard and the large Virtex7 (XC7vx690tffg1761-2) on a VC709 board. The results are evaluated for different numbers of bands, blocks and pixels. In addition, for the smaller FPGA with limited resources, we report a comparison between the fixed-point and the floating-point versions of the accelerator. Finally, the HI data set is used to evaluate the performance of the PCA accelerator in the Zynq device. Accuracy, execution time, and power consumption are also measured for both floating- and fixed-point design. Note that we define the execution time or latency as the period of time between reading the first input data from the external memory by the Dispatcher and writing the last PCA output to the memory by the Projection unit.

In the following subsections the impact of input dimensions (bands and pixels), size of the blocks (B_{max} in the block-streaming method), and data format (floating- or fixed-point) on the resource usage and latency is evaluated for the two hardware devices.

4.4.1 Number of Blocks, Bands, and Pixels

To show the efficiency of the trade-off between latency and resources enabled by the block-streaming method, different numbers of bands and blocks are considered. In the first experiment with the floating-point version on the Virtex7, we consider the total number of bands set at 48 and the size of the block (B_{max}) as a parameter that changes from 4 to 16. Figure 4.8 shows that, as expected, by using a larger block the total latency decreases in exchange for an increase in the resource usage. The latency for different parts of the accelerator is shown with different colors. The most time-consuming parts are Cov and Dispatcher (Dispatcher latency is the time for data transfer through the FIFOs). By increasing the block size (when $B_{max} = 16$) we can reduce the latency of Cov computation, so that the only limitation becomes the Dispatcher latency. It should be noted that the PCA core and the Dispatcher work concurrently, which reduces the overall latency of the entire design.

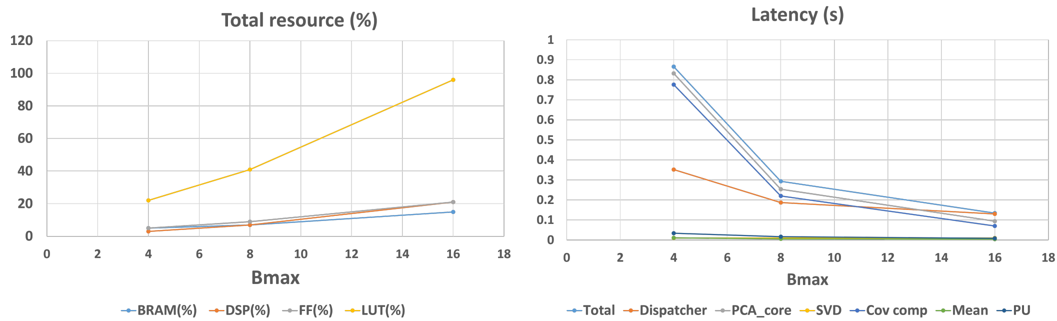


Fig. 4.8 Impact of block size (B_{max}) on the resource usage and latency for the Virtex7, bands = 48, pixels = 300 × 300, floating-point design.

Increasing the number of pixels changes the latency of the design as the time to stream the whole data increases. Figure 4.9 shows the latency of different parts of the design when changing the total number of pixels. The resource consumption remains constant and does not change with the number of pixels. As expected, the latency of SVD is also constant because it depends on the number of bands, not on the number of pixels. For the other parts, the latency increases almost proportionally.

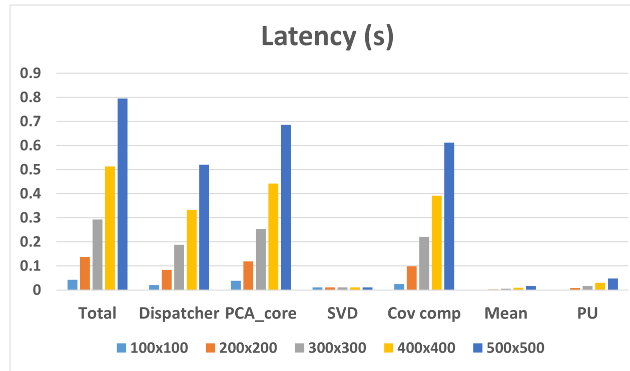


Fig. 4.9 Impact of the number of pixels on the latency for Virtex7, bands=48, $B_{max} = 8$, floating-point design.

In the next experiment the block size is fixed to $B_{max} = 8$ and the total number of bands is variable. The resource usage in the Virtex7 for the floating-point version of the PCA core without the SVD part (PCA-SVD), and the latency for different bands with a fixed B_{max} are shown in Figure 4.10. The number of pixels in this case is 300×300. The number of bands has a direct impact on the generated SVD hardware, so the resource usage of SVD unit is excluded from the total resources to obtain a better estimate of the performance of the block-streaming method.

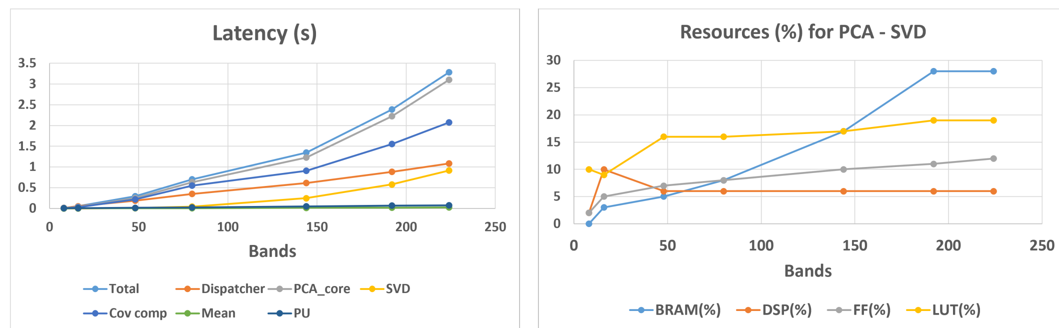


Fig. 4.10 Latency and resource usage for Virtex7 with a fixed block size ($B_{max} = 8$), floating-point design.

As shown in Figure 4.10, the latency increases with the number of bands because the computational time depends on the main dimension. From the resource usage, it is evident that the FFs, DSPs, and LUTs are almost constant (except for a slight increase due to other components of PCA core like Mean and PU). The number of BRAMs, however, increases because in the HLS design there are other two memory arrays in addition to Diagonal and Off-diagonal RAMs to store the temporary values

of the computations (CovDiag and CovOff in Algorithm 8) and the dimensions of these arrays depend on the ratio between the total bands and B_{max}. Still, up to a very large number of bands, the total resource usage of the most critical component is well below 30%.

4.4.2 Fixed-Point and Floating-Point Comparison

The fixed-point design of the PCA accelerator is evaluated on the Zynq7000 for different numbers of bands and blocks and is compared with the floating-point design. We first obtained the word lengths using the MATLAB fixed-point converter and then used the nearest possible word lengths in the HLS design.

The total resource usage for the fixed- and floating-point design is shown in Figure 4.11 for a fixed number of bands ($B = 12$). In the floating-point design (histogram on the left side of Figure 4.11), the maximum block size is $B_{max} = 3$ because the LUTs required for larger block values exceed the LUTs available. For the fixed-point design (histogram on the right side of Figure 4.11), however, the block size can be up to 4. The comparison of the floating- and fixed-point designs for the same block size ($B_{max} = 3$) shows that there is a reduction in the resource usage for the fixed-point design except for the DSP usage. This is because to obtain a similar accuracy the fixed-point representation requires a larger bit-width. As a consequence, the HLS design requires more DSPs to implement the same operations in fixed-point.

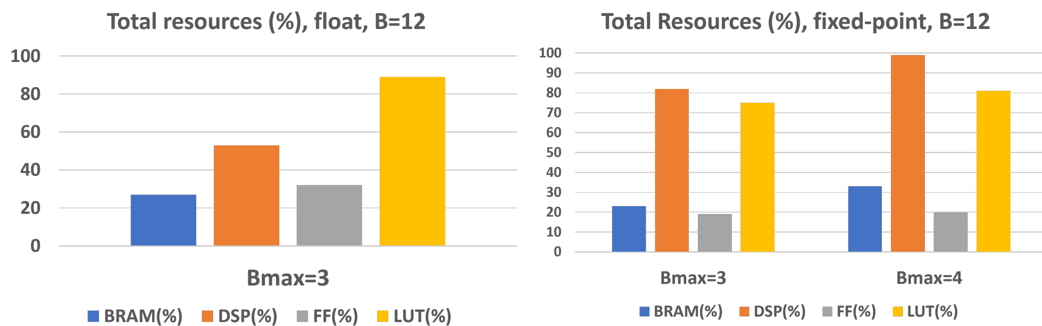


Fig. 4.11 Resource usage for Zedboard for fixed- and floating-point design, $B = 12$, pixels = 300×300.

The larger amount of resources needed by fixed-point design is counterbalanced by the lower latency, as shown in Figure 4.12 for $B = 12$ and $B_{max} = 3, 4$. The

fixed-point design has a lower latency at the same block size and even less latency when using a larger block ($B_{max} = 4$). This is because the latency of fixed-point operations is lower than the floating-point ones. For example, the fixed-point adder has a latency of 1 clock cycle, while the floating-point adder has a latency of 5 clock cycles.

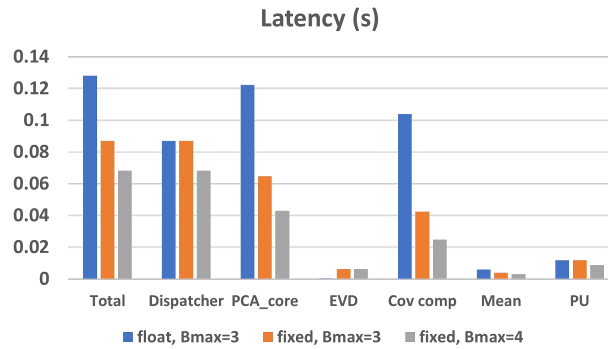


Fig. 4.12 Comparison of the latency of the fixed- and floating-point design for Zedboard, B = 12, pixels = 300 × 300.

Figures 4.13 and 4.14 illustrate the total latency of the PCA accelerator and its resource usage for different numbers of bands for a fixed block size ($B_{max} = 3$). As shown in Figure 4.13, the latency of the floating-point design is limited by the PCA core function, whereas in the fixed-point design the Dispatcher latency is the main limitation. This is because the PCA core and the Dispatcher operate concurrently, as noted before, and therefore the total latency is basically the maximum between the two latencies, which may change depending on the implementation details (in this case floating- versus fixed-point data representation). The comparison of the resource usage in Figure 4.14 shows that except for an increase in the DSP usage, other resources are reduced in the fixed-point design. As explained before, the increase in DSP usage is due to the larger bit-width needed for the fixed-point data representation.

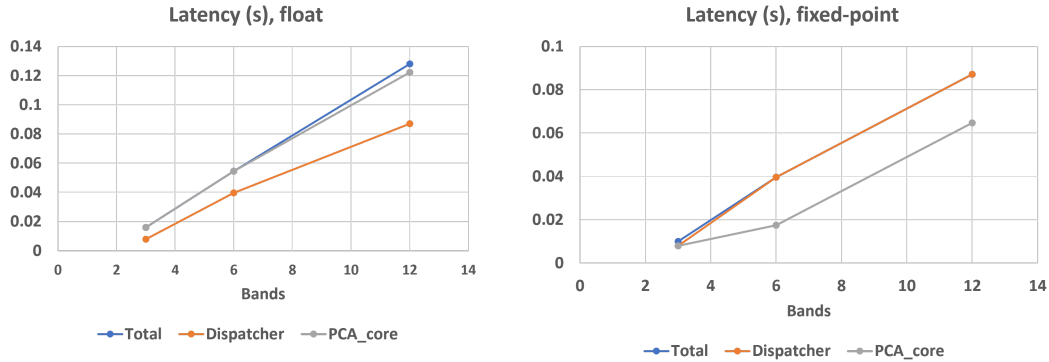


Fig. 4.13 Latency for Zynq7000 with a fixed block size ($B_{max} = 3$), pixels = 300 × 300.

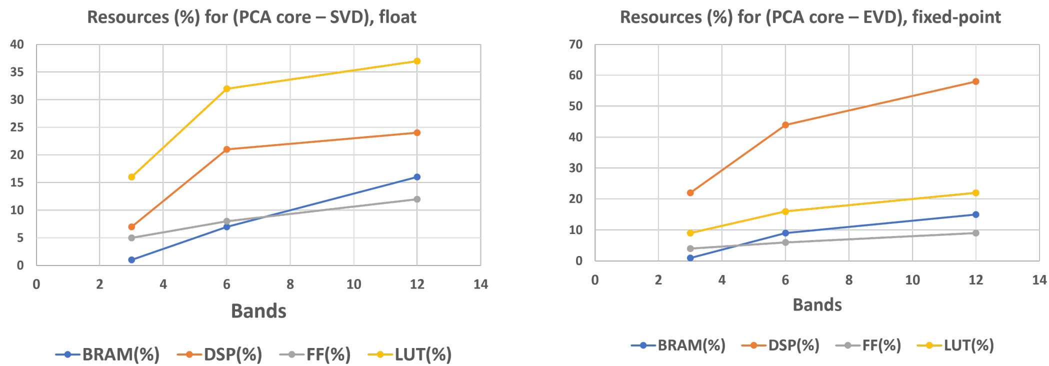


Fig. 4.14 Resource usage for Zynq7000 with a fixed block size ($B_{max} = 3$), pixels=300 × 300.

4.4.3 Evaluation on Hyperspectral Images

The hyperspectral image data set is obtained from Purdue Research Foundation and is available online [2]. It shows a portion of southern Tippecanoe county, Indiana, and comprises 12 bands each of which corresponds to an image of 949 × 220 pixels. We will show that by using PCA, most of the information in the 12 bands is redundant and could be obtained from the first 3 principal components.

The PCA accelerator for this data set is evaluated on the Zynq7000 of the Zedboard for different possible block sizes ($B_{max} = 3,4$). The HLS estimation of the resource usage for the floating- and fixed-point design is indicated in Table 4.1. For the floating-point design, the maximum block size is $B_{max} = 3$. In fixed-point design,

[2] https://engineering.purdue.edu/~biehl/MultiSpec/hyperspectral.html

however, we can use a larger block size ($B_{max} = 4$), which leads to the increase in the resource usage.

Table 4.1 Resource usage obtained from HLS for HI data set on Zedboard, bands = 12.

	BRAM (%)	**DSP (%)**	**FF (%)**	**LUT (%)**
floating-point (B_{max} = 3)	27	57	33	92
fixed-point (B_{max} = 3)	23	82	20	75
fixed-point (B_{max} = 4)	33	99	20	81

Table 4.2 shows the latency of different components of the design. According to Table 4.2, the fixed-point minimum latency is about half of the floating-point latency. in addition, the fixed-point EVD latency is about 15 times larger than the floating-point SVD latency. However, this does not affect the total latency because the Dispatcher latency in the fixed-point design is higher than the PCA core latency. Therefore, due to the concurrency between Dispatcher and PCA core, the total latency is limited by the Dispatcher. The resource usage for EVD is lower than SVD, so by using the fixed-point EVD we can improve the overall performance because the resources saved by EVD can be used in the rest of the design for more parallelism leading to a lower total latency.

Table 4.2 Latency (ms) for Zedboard, HI data set, bands = 12.

	Total	**Dispatcher**	**PCA_core**	**SVD/EVD**	**Cov**	**Mean**	**PU**
floating-point (B_{max} = 3)	296.6981	202.0993	282.9185	0.399916	241.1409	13.77959	27.55981
fixed-point (B_{max} = 3)	202.0993	202.0993	141.7858	6.267679	98.75294	9.18632	27.55913
fixed-point (B_{max} = 4)	158.4643	158.4643	91.26137	6.267679	57.4145	6.88974	20.6694

The PCA accelerator resource usage and power consumption in the target hardware are measured by the Vivado software and are shown in Table 4.3. In addition, the accuracy of the PCA output from FPGA is compared with the MATLAB output by using the Mean Square Error (MSE) metric. MATLAB uses double precision, whereas our FPGA design uses single-precision floating-point as well as fixed-point computations. Although the accuracy of our fixed-point design is reduced, its MSE is still negligible. In contrast, the latency for the fixed-point improves by a factor of 1.8, which shows the efficiency of the fixed-point design.

Table 4.3 Vivado implementation of PCA accelerator on Zedboard for HI data, bands=12, accuracy is compared with MATLAB.

	BRAM	DSP	FF	LUT	Power (W)	Accuracy (MSE)
floating-point (B_{max} = 3)	75 (27%)	124 (57%)	29971 (28%)	27090 (51%)	2.266	2.08×10^{-7}
fixed-point (B_{max} = 4)	92 (33%)	218 (99%)	18288 (17%)	20801 (40%)	2.376	1.1×10^{-3}

The PCA output images generated by the FPGA are visually the same as the MATLAB output. Figure 4.15 represents the first six outputs of PCA applied to the hyperspectral images data set. The FPGA produces the first 3 principal components that are indicated in the figure as PCA1 to PCA3. As shown in the energy distribution in Figure 4.16, the first 3 principal components contain almost the entire energy of the input data. The first 3 PCs in Figure 4.15 correspond to these 3 components. It is evident from Figure 4.15 that the PCs after the third component do not contain enough information in contrast with the first 3 PCs.

The flexibility of our PCA accelerator allows us to compare it with other state-of-the-art references presenting PCA acceleration with different dimensions and target devices. Table 4.4 represents the resource usage, frequency and execution time for our FPGA design compared with two other references [83, 9]. The input data for all of the references contain 30 rows (pixels) and 16 columns (bands in our design). The first reference uses an HLS approach to design its PCA accelerator on a Zynq FPGA, and the second one uses a manual RTL design in VHDL on a Virtex7 FPGA target. Our HLS design on the same Virtex7 FPGA uses fewer resources as indicated in Table 4.4. Although the clock frequency of our design is not as high as the previous methods, the total execution time for our design is reduced by a factor 2.3x compared to the same FPGA target.

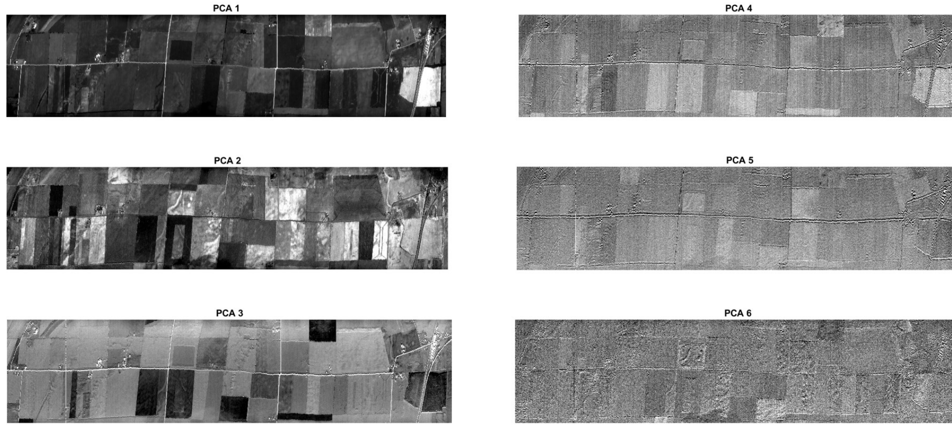


Fig. 4.15 The first 6 principal components of the HI data set. Our PCA accelerator in Zedboard produces the first 3 outputs (PCA1 to PCA3).

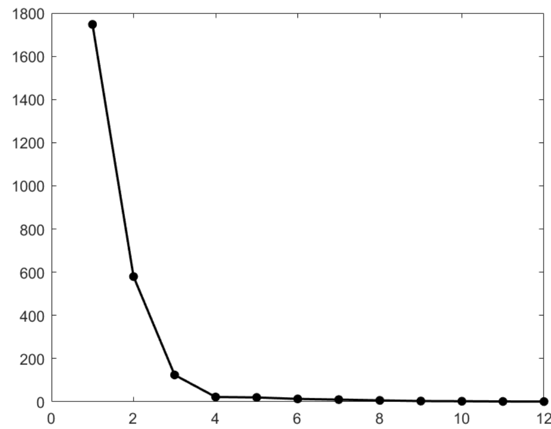


Fig. 4.16 Energy distribution of the eigenvalues for the Hyperspectral Imaging (HI) data set.

Table 4.4 Comparison of our PCA accelerator with other conventional methods. The input data dimensions are set to 30×16 for all designs.

Work	**Device**	**BRAM**	**DSP48**	**FF**	**LUT**	**freq (MHz)**	**Latency (Clock Cycles)**	**Execution Time (ms)**
[9]	Zynq ZC702	6 (2%)	95 (43%)	13,425 (12%)	18,884 (35%)	116	31,707,056	273
[83]	Virtex7 XC7VX485T-2	350 (16%)	2612 (78%)	304,596 (37%)	301,235 (76%)	183	289,352	1.6
This work (B_{max} = 8)	Virtex7 XC7VX485T-2	132 (6%)	385 (13%)	66,790 (10%)	145,220 (47%)	95	64,253	0.675

Table 4.5 shows the performance of our design compared to other PCA accelerators for spectral images that were also designed in HLS on a Zynq target. In [10] all the PCA functions were designed in HLS except for the EVD unit that was designed in software. A complete hardware design of the PCA algorithm using the standard method for covariance computation (all the bands are streamed at once without blocking) is presented in [86]. Our work uses instead the block-streaming method for covariance computation. The total number of bands is $B = 12$ and the block size in our method is $B_{max} = 3$. The data representation is floating-point in all of the methods compared in Table 4.5. As shown in the Table, in our design the DSP and BRAM usage is higher and the FF and LUT usage is lower. Despite the reduction in clock frequency, the total execution time of our design is the minimum (0.44 s) among the three accelerators.

Table 4.5 Comparison of the proposed PCA hardware design with other High Level Synthesis (HLS)-based accelerators. The dimensions of a spectral image data set ($640 \times 480 \times 12$) is selected for all of the designs.

Work	Execution Time (s)	BRAM (%)	DSP48 (%)	FF (%)	LUT (%)	freq (MHz)
[10] (PCA-SVD)	1.1	12	19	38	73	-
[86]	0.83	9	32	51	94	100
Ours ($B_{max} = 3$)	0.44	27	53	32	90	90

Finally, our FPGA design is compared with a GPU and MPPA implementation of the PCA algorithm [84] for different data dimensions as shown in Table 4.6. In our design, the target device is the Virtex7 FPGA and the block size is set to $B_{max} = 10$ as the numbers of bands are multiples of 10. For the smaller number of pixels (100×100), our FPGA design outperforms the other two implementations in GPU and MPPA in terms of execution time. For the larger number of pixels, the execution time for our design increases linearly and becomes more than the other designs. It has to be noted, however, that the typical power consumption in MPPAs and GPUs is significantly more than in FPGAs. In the radar chart in Figure 4.17, four important factors when selecting a hardware platform are considered and their mutual impact is analyzed. These factors are power consumption, latency per pixel, number of bands (input size) and energy. The axes are normalized to the range 0 to 1 and the scale is logarithmic for better visualization.

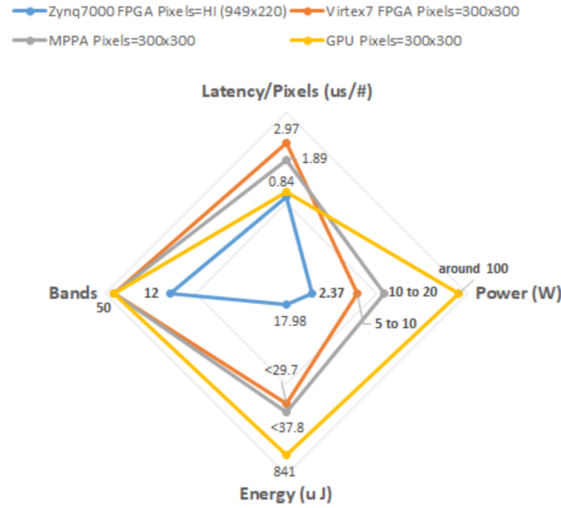


Fig. 4.17 Comparison of different hardware platforms between latency per pixel, power consumption, input size (bands) and energy.

Table 4.6 Execution time (ms) for the PCA implementation on GPU, Massively Parallel Processing Array (MPPA), and FPGA (our work). The first two designs on GPU and MPPA are from [84]

Dimensions	**MPPA**	**GPU**	**Ours (FPGA), B_{max} = 10**
$100 \times 100 \times 50$	140.4	69.28	40.47
$300 \times 300 \times 20$	47.2	70.87	62.37
$300 \times 300 \times 30$	80.1	70.22	121.9
$300 \times 300 \times 50$	170.7	75.74	268.11

As shown in Figure 4.17, for a small number of bands, a Zynq FPGA has a power consumption of only 2.37 W with a small latency. For larger bands, although GPUs and MPPAs have smaller latency than FPGAs, they consume much more power (especially GPUs). By taking into account the energy consumption that is smaller for FPGAs, one has to select the best hardware based on their needs and use case. Using an FPGA for the PCA accelerator results in a power efficient hardware that can be used for large input sizes without a significant increase in the total latency.

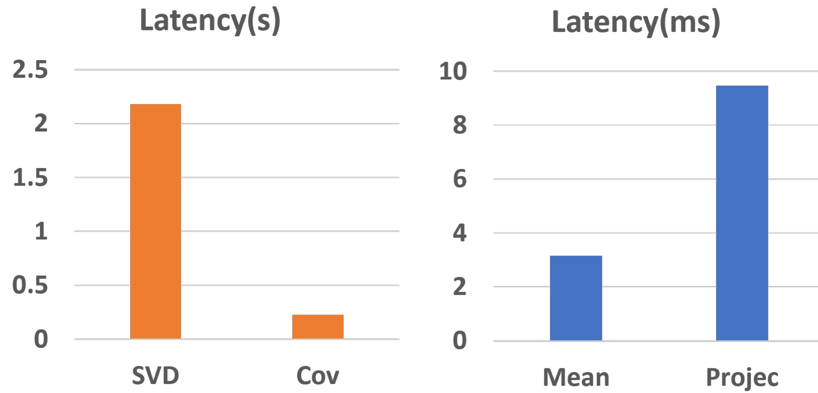


Fig. 4.18 Processing time for PCA compute units in FPGA with Microwave dataset

4.4.4 Evaluation on Microwave Data

PCA algorithm can be used in data dimensionality reduction for biomedical microwave applications. We compared the processing time and resource usage for each compute unit of PCA on a large Virtex FPGA (VC709U evaluation board) that is shown in Fig. 4.18. We found that the main dimension of PCA input (the number of independent elements of scattering matrix) can have a maximum value of 300 ($N_d = 300$) in our target FPGA, that corresponds to an MI system with 24 antennas. To have an evaluation of the latency, we set the number of samples to 100×100.

By using the block-streaming strategy described in section 4.3.1, the Covariance computation obtains a low latency, and the most critical part of the design is SVD as can be seen from Fig. 4.18, for which the optimized HLS built-in function is used.

4.5 Conclusions

In this chapter, we proposed a new hardware accelerator for the PCA algorithm on FPGA by introducing a new block-streaming method for computing the internal covariance matrix. By dividing the input data into several blocks of fixed size and reading each block in a specific order, there is no need to stream the entire data at once, which is one of the main problems of resource overuse in the design of PCA accelerators in FPGAs. The proposed PCA accelerator is developed in Vivado HLS tool and several hardware optimization techniques are applied to the same design in HLS to improve the design efficiency. A fixed-point version of our PCA design

is also presented, which reduces the PCA latency compared to the floating-point version. Different data dimensions and FPGA targets are considered for hardware evaluation, and a hyperspectral image data set is used to assess the proposed PCA accelerator implemented on Zedboard.

Compared to a similar RTL-based FPGA implementation of PCA using VHDL, our HLS design has a $2.3\times$ **speedup in execution time**, as well as a significant reduction of the resource consumption. **Considering other HLS-based approaches, our design has a maximum of** $2.5\times$ **speedup**. The performance of the proposed FPGA design is compared with similar GPU and MPPA implementations and, according to the results, the execution time changes with data dimensions. For a small number of pixels our FPGA design outperforms GPU and MPPA designs. For a large number of pixels the **FPGA implementation remains the most power-efficient one**.

Chapter 5

HLS-based Dataflow Hardware Architecture for Support Vector Machine

Support Vector Machine (SVM) is a supervised Machine Learning (ML) model widely used in different classification problems, such as image classification, medical diagnosis, object detection, and bioinformatics [90]. For example, in microwave imaging, SVM can detect a brain stroke from the electromagnetic scattering data [91, 92]. Since it is challenging to implement SVM in real-time embedded systems, several specialized hardware architectures have been recently proposed. Among them, those based on Field Programmable Gate Arrays (FPGAs) are preferable in embedded systems due to their flexibility and lower power. In [93], a review of recent FPGA accelerators for SVM is presented.

A parallel hardware architecture for SVM using a systolic array of vector processing units to process multiple support vectors (SVs) in parallel is proposed in [94] and extended to a cascade SVM classifier for real-time object detection in [95]. The main limitation of these works is that all the coefficients and SVs are stored in on-chip memory, which means that the number of SVs is limited by the memory resources in FPGA. In [96], another SVM architecture based on a Verilog RTL description is proposed. It uses on-chip FIFOs to store all the SVs and can process no more than 20 SVs in parallel, with no parallelism on the SVs dimension or the number of features N_f: as we will see, our design can support more features and a higher clock

frequency. The works in [97–100] also used RTL, either created manually or with Xilinx System Generator.

High Level Synthesis (HLS) enables a more efficient design space exploration than RTL manual design. In [11], an HLS design for SVM acceleration in the application of melanoma detection is proposed, which is extended for better performance in [12, 13]. Due to the assumption of local storage of SVM coefficients, this design is tested on small-scale problems with 27 features and a maximum of 346 SVs. In [101] a linear multi-class SVM is used for brain cancer detection in Hyper-Spectral Images (HSI). Due to the large dimensions of HSI datasets, the input vector could not be stored in local memory. However, due to the linear kernel used in this work, the main part of kernel computation is processed off-line and the weighted summation of all SVs is stored as a single vector, in a way that only one vector is used in on-line computations. In [102], a methodology to extract parallelism from software code is introduced for optimized annotation of C code with HLS directives, and is tested on SVM. Although different levels of hardware parallelism are explored in this work, the authors considered local storage for the inputs and did not report the time required to read and store all the inputs.

Previous SVM works usually ignored scalability, which allows the same design to be used for larger data dimensions. Usually, coefficients and SVs are stored in on-chip memory, which is a method useful for small scale problems but cannot be used for large data dimensions and/or low-cost FPGAs due to the lack of local memory. Indeed, it is possible to do the SVM computations while reading the required data from an external memory, hence increasing the overall throughput. This is the main idea on which our dataflow design is based.

Most of the previous designs only considered binary classification or simple kernel functions. Multi-class classification is a more challenging problem requiring more computations, specially if complex kernels are used. In this work, we propose a scalable hardware accelerator for multi-class SVM classification in FPGA by using an efficient dataflow architecture designed entirely in HLS. These are our contributions:

- A specialized dataflow hardware accelerator for SVM algorithm in FPGAs that can scale to support different data dimensions while guaranteeing a high throughput.

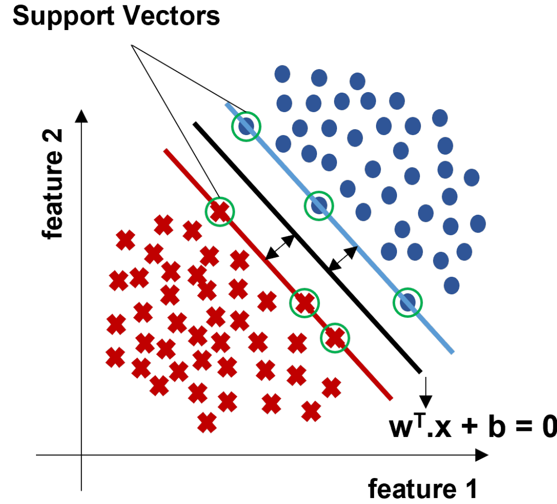


Fig. 5.1 SVM classification with linear kernel.

- Support for multi-class classification and various kernels.
- Adjustable parallelism by HLS-based configurations and efficient implementation of fixed-point design.

5.1 SVM Background

SVM algorithm for binary classification obtains a decision boundary as a hyper-plane that maximizes the margin between two classes as shown in Fig. 5.1. For linear classification, the hyper-plane can be expressed as $w^T x + b = 0$, where w and b are obtained during training, and x is a 1-D input vector with N_f elements. The inputs that lie on the margin are termed *Support Vectors* (SVs). A total number of N_{SV} vectors can be stored as a 2-D array of size $N_{SV} \times N_f$. To obtain w and b, a dual problem is constructed by using Lagrange multipliers (indicated as α_i parameter for each input). After training, α_i are obtained and all the inputs for which $\alpha_i \neq 0$ are regarded as support vectors (SV_i). Finally, w and b can be obtained from α_i and SV_i and the decision boundary can be written as

$$\text{Decision} = \sum_{i=1}^{N_{SV}} \alpha_i K(x, SV_i) + b, \tag{5.1}$$

in which SV_i is the ith support vector and $K(\cdot)$ is one of the *Kernel* functions in Tab. 5.1. For linear problems, the Kernel is a dot product; for non-linear problems, the Kernel will transform the input space into one where the classes are linearly separable. Note that in (5.1) α_i must be multiplied by the labels of the support vectors (y_i), which for clarity, we considered is done internally ($\alpha_i = \alpha_i y_i$).

Table 5.1 Kernel functions in SVM.

Kernels: $K(x, SVi)$	
Linear	$x \cdot SV_i$
Radial Basis Funtion (RBF)	$exp(-\frac{\lVert x-SV_i \rVert^2)}{2\sigma^2}$
Polynomial (Poly)	$(c_1(x \cdot SV_i)+c_2)^P$
Sigmoid	$\tanh(c_1(x \cdot SV_i)+c_2)$

For binary classification, the sign of *Decision* is simply used for the prediction. For multi-class, we used *one-vs-one* (*ovo*) method, in which all pairs of classes are compared using (5.1) and based on the majority vote, the final prediction is computed. With N_c classes, the total number of comparisons (i.e., the decision vector size) for *ovo* is $N_d = N_c(N_c - 1)/2$.

5.2 Proposed SVM Accelerator

To increase the overall throughput and reduce the on-chip storage, we propose the Dataflow hardware architecture illustrated in Fig. 5.2. Instead of storing all the SVM input data in on-chip memory, we read chunks of data from the external memory, transfer them through the FIFO channels, and do the subsequent computations while the next chunk is being read. We can design such a hardware architecture in HLS by defining a dataflow region between three functions in the main module related to the processing and reading the SVM data.

As shown in Fig. 5.2, three functions for SVM computations are *Read*, *Kernel*, and *Decision*. *Read* distributes the chunks of input read data through the FIFO channels. Since the major part of SVM computations is dedicated to the dot product between the input features, we match the size of a block of input features to a data chunk. *Kernel* reads the chunks from the FIFOs, calculates the kernel function and sends the result to the next FIFO channel. *Decision* receives the kernel output from the FIFO as well as three other input coefficients, and computes the final prediction,

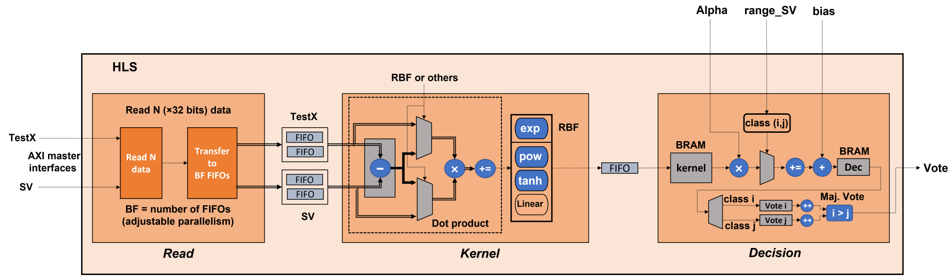


Fig. 5.2 Proposed SVM accelerator in HLS.

denoted as Vote in Fig. 5.2. We use Vivado HLS since we target Xilinx FPGAs and with this tool we can use a *dataflow* directive and *hls::stream* variables for the FIFO channels to obtain the implementation of Fig. 5.2.

5.2.1 Read SVM Inputs

The data are stored as 32-bit floating-point values in an external DDR memory. Depending on the maximum DDR data width (DW_{ddr}, in bits), DDR frequency (f_{ddr}), and working clock frequency (f_w), the maximum number of 32-bit data that can be read from the memory in one clock cycle is $N = (DW_{ddr}/32) \times ((2 \times f_{ddr})/f_w)$. For large-scale problems, storing SV in FPGA memory is the main limitation. In this work, we consider SVs and the test vector (x) stored in off-chip and on-chip memory, respectively. However, with enough memory bandwidth, both of them can be stored in external memory.

The *Read* function will transfer the input data to the *Kernel* function through BF FIFO channels, as indicated in the left part of Fig. 5.2, in such a way that the Kernel function can process BF data in parallel. Ideally, we want $N = BF$ because it is not possible to write more than N values to the FIFO channels in each clock cycle. However, by increasing BF we speed up the Kernel function, although we slow down the Read one. This is shown in Fig. 5.3, where by increasing BF we increase the Initiation Interval (II)[1] of the Read function, but this is in part masked by the overlapping of Read and Kernel functions due to the pipelining effect of the Dataflow implementation, and is compensated by the higher throughput of the Kernel function due to the greater parallelism. To keep the pipeline balanced, BF and N, which are defined as configurable variables in the HLS code, should be properly

[1] Minimum number of clock cycles before the next input data can be received.

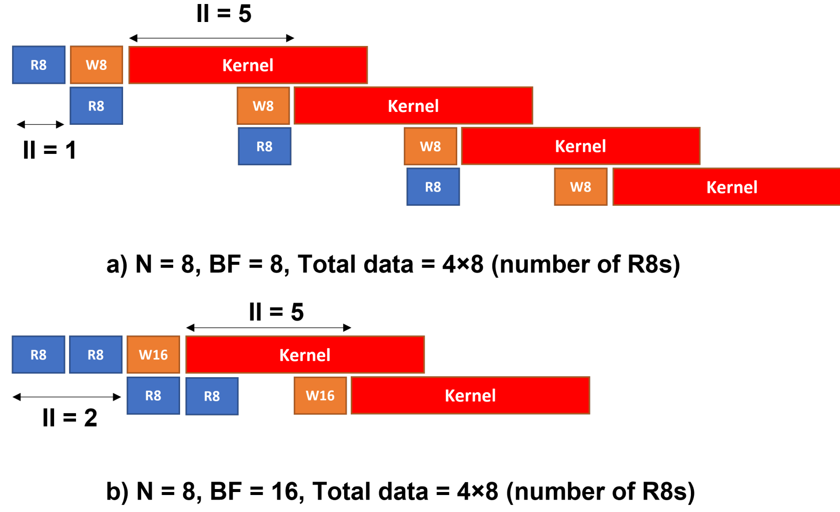


Fig. 5.3 Impact of the number of FIFO channels (BF) with a total of $4 \times 8 = 32$ data, (a) $BF = N$, (b) $BF = 2N$, overall latency is reduced.

related. For this, we use the SVM latency equation (T) from the maximum latency of *Read*, *Kernel*, and *Decision*

$$T \approx N_{SV} \cdot N_f \cdot \max\{\frac{II_R}{BF}, \frac{II_K}{BF}\}, \tag{5.2}$$

where the approximation comes from considering only the II of the various functions and from not considering the latency of the iterations. Indeed, II_R and II_K, are the II of *Read* and *Kernel* , respectively. Note that in general, the *Decision* latency must be incorporated in Eq. 5.2, as shown later for Microwave data set. However, in large-scale problems, usually the Decision latency is small and can be ignored in Eq. 5.2, so the total latency is determined by the maximum value between II_R/BF and II_K/BF. Therefore, a balanced pipeline would require $II_R = II_K$ and since $II_R = BF/N$, we should have $II_K = BF/N$. If we have $II_K < BF/N$ we end up with a memory-bound performance, otherwise the performance would be computation-bound. In the ideal case, if II for all the functions is 1, we must select $N = BF$ to match the throughput. Note that there is an iteration latency (L) to add to the latency of each function and that *Kernel* has a higher L than *Read*. Therefore, in a balanced dataflow, the latency between *Read* and *Kernel* is determined by the *Kernel* latency. As we will see, for this reason it is sometimes preferable not to have a perfectly balanced pipeline and select, for example, $BF = 2N$: *Read* will be the dominant part, but the overall latency will decrease.

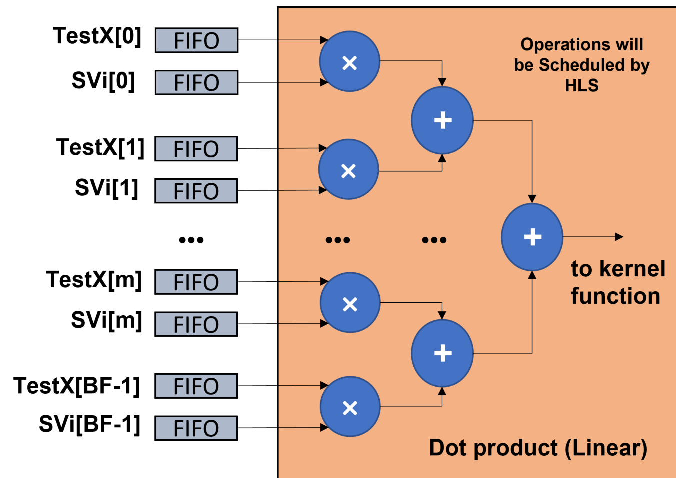


Fig. 5.4 Manual unrolling for kernel computation.

5.2.2 Kernel Computation

The hardware architecture for *Kernel* computation is shown in the middle part of Fig. 5.2. Note that to compute the RBF kernel, we need to subtract the inputs and compute its squared norm to be used in the exponential function. For other kernels, only the dot product between the inputs are computed. This is shown in Fig. 5.2 by a control signal (RBF or others) for clarity.

The parallelism of the computation is matched to the number of FIFO channels, as we instantiate BF parallel elements for the dot product, with BF multipliers (and subtractors for the RBF kernel). This can be obtained by *partial unrolling* directive in HLS with a factor of BF. Due to the accumulation in the dot product, if we use one scalar variable to store the accumulation (after $+=$ operation in Fig. 5.2), HLS tool cannot schedule the design with partial unrolling as we expect because of the data dependency on the scalar variable. Therefore, we use manual unrolling as shown in Fig. 5.4 (for clarity, only linear kernel is shown) and define an array of size BF to hold the result of each multiplication. By fully partitioning this array and computing the addition of all its elements (shown as adders), the dot product will be computed. After the dot product, a non-linear function selected based on the SVM kernel can be applied, if needed, otherwise for the linear case the dot product is passed directly to the output.

5.2.3 Decision Function

The Decision function, depicted in the right part of Fig. 5.2, stores the kernel FIFO stream from the previous function in a BRAM, computes the Decision based on (5.1), and stores the decision vector in another BRAM (Dec). For multi-class classification, the decision must be calculated for each pair of classes. The range and number of support vectors in each class are received from another input (range_SV). Based on the decision vector, the number of votes for each class is calculated and the majority vote determines the final prediction (Vote).

Decision computation consists of two main loops for one pair of classes. For the same reason described in Sec. 5.2.2 for the accumulation, we manually unrolled these loops with factor *DF* to increase the efficiency. *DF* is the third HLS parameter to control the level of hardware parallelism.

5.2.4 Fixed-Point Implementation

To design a fixed-point SVM accelerator, we converted the floating-point data received from the *Read* function to a fixed-point value in the *Kernel* function. We explored the accuracy loss in hardware by varying the bit widths of each variable. Once the optimum fixed-point precision for all variables are obtained, HLS synthesis tool estimates the performance. To obtain the optimum precision, we divided the variables into three main parts that are used in *Inputs*, *Kernel*, and *Decision* computations. For the inputs, we selected $< 10, 1 >$ ($< total, integer >$) for x, SV, $bias$ and $< 21, 5 >$ for α. For *Kernel* and *Decision* computations, we selected $< 21, 11 >$ and $< 28, 10 >$, respectively. For MNIST dataset, these values result in the minimum accuracy loss in hardware.

5.3 Results

Using Sklearn and an SVM model with RBF kernel, we first trained a classifier for the MNIST dataset. After training, we obtained 16036 SVs with size 784 ($N_{SV} = 16036$, $N_f = 784$). The test vector, support vectors, and other coefficients are sent to the SVM accelerator for the classification. We used Vivado HLS 2018.2 and the low-cost Zynq SoC of the ZedBoard for the evaluation of the performance. In addition to the

MNIST case, we measured the hardware performance for other data dimensions to compare with previous works.

To show the impact of HLS-controlled parameters we report processing time and resource usage in Tab. 5.2. Our accelerator can be used in various FPGA platforms by tuning *N* to adapt to the maximum memory bandwidth and by tuning *BF* to optimize the processing time. For the MNIST dataset, *DF* has no effect as the latency of Decision is negligible.

The first four experiments in Tab. 5.2 are with floating-point computation. In this case we have $II_K = 5$ and a computation-bound performance. Starting from the first experiment (*N*=*BF*=4), we can see that by increasing *BF* first to 8 and then to 16, the latency is reduced. Notice the increase in the resource usage due to *BF* and the negligible accuracy loss in fixed-point design compared to the floating-point one.

Table 5.2 MNIST dataset: performance and resource usage.

Experiment	**1**	**2**	**3**	**fix1**	**fix2**
N	4	4	4	8	8
BF	4	8	16	8	16
BRAM (%)	15	16	19	12	15
DSP48 (%)	20	23	28	7	11
FF (%)	8	10	13	11	16
LUT (%)	24	28	35	40	62
Latency (ms)	166.93	91.57	58.69	18.12	15.72
Accuracy (%)	98.56 (float)			98.55 (fixed)	

The last two experiments are with fixed-point computation. In this case II_K=1 and it is possible to have a memory-bound performance. Therefore, increasing *N* from 4 to 8, the maximum allowed by the Zedboard platform, can be helpful. When $N = BF$ (experiment fix1), although the pipeline is balanced, the Kernel iteration latency has an impact on the overall performance. Therefore, by increasing *BF* (experiment fix2), we obtain a further latency decrease.

Table 5.3 Comparison of different SVM kernels.

kernels	**RBF**	**Linear**	**Poly**	**Sigmoid**
Time (ms)	58.69	53.88	58.53	64.94
BRAM (%)	0	0	5	3
DSP48 (%)	23	9	24	31
FF (%)	6	3	6	9
LUT (%)	16	8	13	23

Tab. 5.3 compares the performance of different SVM kernels. Sigmoid kernel has the highest resource usage and time, which is related to its computational complexity. Tab. 5.4 shows a comparison of different HLS-based SVM accelerators in Zynq FPGA. Our dataflow design improves by about 10× the processing time with less BRAM usage and more LUTs. In Tab. 5.5 we compared the number of features, SVs, and processing time for our design with two other FPGA accelerators designed manually in RTL. Due to the high scalability, our design can support higher number of SVs and features, with higher frequency and 4.4× speed-up by using more resources.

Table 5.4 Comparison of the proposed accelerator with different SVs and same number of features ($N_f = 27$) in the same FPGA (model1 and model2 use different pre-processing methods on training data).

	[11]	**[12] (model1)**	**[12] (model2)**	**[13] (model1)**	**[13] (model2)**	**Proposed (dataflow)**
N_{SV}	248	61	248	248	346	**346**
freq (MHz)	100	250	250	250	100	100
Time (μs)	83.66	11.46	39.3	33.5	**136**	**13.15**
Interval (μs)	83.67	11.46	39.3	33.5	136	**8.15**
BRAM (%)	11	34	34	12	**11**	**2**
DSP48 (%)	61	2	2	61	61	88
FF (%)	13	10	28	13	13	30
LUT (%)	94	14	33	24	24	90

Table 5.5 Performance comparison with two manual RTL designs.

	[96]	**[100]**	**This work**
N_{SV}	100	60	100
N_f	500	1024	1024
FPGA	Virtex 5	Cyclone II	Zynq 7000
freq (MHz)	50	30	100
Time (ms)	0.25	2	0.45
#BRAM	-	-	39
#DSP48	52	20	83
#FF	9646	-	19687
#LUT	38179	14064	25758

5.3.1 Microwave Data Set

We evaluated our proposed SVM accelerator by using a medical microwave data set provided by the biomedical microwave research team in Politecnico di Torino [103].

The data set contains 462 features (the dimension of the scattering matrix) which are reduced to 110 feature after using PCA algorithm. There are 9 classes in this data set representing the type and locations of the brain stroke: stroke types (*ischemic, hemorrhagic, no stroke*) and locations (*top-right, top-left, bottom-right, bottom-left*). After the training phase, the total number of Support Vectors is 2009.

Table 5.6 shows the SVM accelerator performance and the impact of HLS parameters (N, BF, DF) on the latency and resource usage by using the *floating-point* data precision. Due to the latency of the *Kernel* function being the dominant part between the three main functions, the optimum HLS parameters are ($N = 2$, $BF = 8$, $DF = 2$), resulting in the total latency (equal to the kernel latency) of 3 ms. Note that increasing N and DF (from 2 to 8) will reduce the latency of *Read* and *Decision* functions, but there will be no improvement in the total latency. Also note that increasing BF (from 8 to 16) leads to a larger array to store the partial accumulations in kernel computation (refer to section 5.2.2); hence, a larger loop will be required to compute the *final* accumulation (addition of all elements of the array after partial accumulation). Therefore, using $BF = 16 > 8$ for the MI data set cannot reduce the total latency due to the increase in the iteration latency of the final accumulation loop.

Table 5.7 shows the similar results when we use fixed-point data precision. With fixed-point data type, the Decision latency is limited only by the number of memory ports used to store α_i parameter. Partitioning the α_i memory will not help due to the random access pattern. Therefore, increasing DF does not change the Decision latency. In contrast, with floating-point data type, due to a higher II_D (= 5), partial unrolling with DF can reduce the latency as well. The HLS estimation of the latency of the Decision function can be written as:

$$T_{Decision} = 2 \times (N_{class} - 1)^2 \times nSV_{max} \times II_D/DF \tag{5.3}$$

where nSV_{max} is the maximum number of Support Vectors in all the classes, and the minimum value for II_D/DF is 1.

From Table 5.7, it can be observed that increasing N and BF will reduce the Read and Kernel latency, respectively. When $BF = 16$, the Kernel latency is lower than Decision function, and due to the fixed-point precision, the minimum latency of Decision function is obtained from Eq. 5.3 with $(II_D/DF) = 1$, which results in the final latency of $0.55ms$.

Table 5.6 Performance analysis of SVM accelerator for medical microwave data set using floating-point data precision ($N_{SV} = 2009, N_{features} = 110, N_{samples} = 900$).

Experiment	float1	float2	float3	float4
N	2	2	8	8
BF	2	8	8	16
DF	2	2	8	8
BRAM (%)	4	3	3	3
DSP48 (%)	19	22	22	28
FF (%)	8	10	20	23
LUT (%)	22	28	48	55
Read Latency (ms)	1.1	1.1	0.3	0.3
Kernel Latency (ms)	6.5	3	3	3.1
Decision Latency (ms)	1.3	1.3	0.6	0.6
Total Latency (ms)	6.5	3	3	3.1

Table 5.7 Performance analysis of SVM accelerator for medical microwave data set using fixed-point data precision ($N_{SV} = 2009, N_{features} = 110, N_{samples} = 900$).

Experiment	fix1	fix2	fix3	fix4
N	2	4	8	8
BF	2	4	8	16
BRAM (%)	3	3	3	3
DSP48 (%)	5	5	7	10
FF (%)	7	9	11	17
LUT (%)	23	29	40	62
Read Latency (ms)	1.1	0.56	0.28	0.28
Kernel Latency (ms)	1.36	0.84	0.58	0.44
Decision Latency (ms)	0.55	0.55	0.55	0.55
Total Latency (ms)	1.36	0.84	0.58	0.55

5.4 Conclusions

We presented a scalable dataflow hardware architecture in FPGA by using HLS to accelerate SVM inference. The hardware parallelism can be controlled by three HLS-based configurations to adapt to small and large scale problems. In addition, a fixed-point design is introduced to speed up the computation. Experiments on different data dimensions and support vectors show **a minimum of $10\times$ latency improvement compared to similar HLS-based and $4.4\times$ improvement compared to RTL-based designs**.

Chapter 6

Hardware Design and Optimization of Neural Networks and ML Accelerators

Classic Machine Learning (ML) models, like Multi-Layer Perceptrons or Support Vector Machines, feature various hyper-parameters that must be tuned during training. In Deep Neural Networks (DNNs), the architectural parameters, like number of layers, neurons, kernel size, number of filters, etc., can be also considered as hyper-parameters to tune in order to maximize the accuracy achievable during training. When it comes to implementing a DNN in FPGA using a dedicated accelerator, however, the accuracy requirements and the corresponding network architecture that meets those requirements might be in contrast with hardware-related requirements and constraints, such as latency and FPGA resources utilization. Therefore, a trade-off must be found, and this can be done by solving a multi-objective optimization problem.

The usual design method consists of a Design Space Exploration (DSE) to fine-tune the hyper-parameters of an ML model, followed by another DSE that aims to optimize the hardware design for a given target.

This approach of separate DSE is shown in Fig. 6.1(a) and is useful for those applications where the ML accuracy has to be maximized and powerful hardware accelerators can be selected to meet the desired performance. However, being bounded to one specific small-size hardware architecture makes the design more

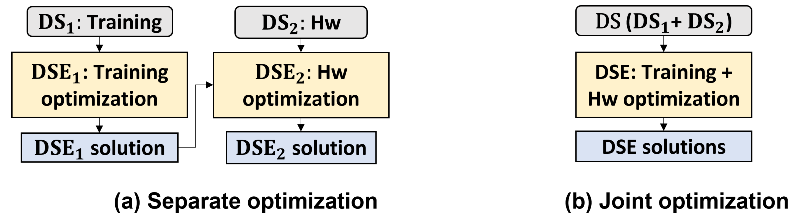


Fig. 6.1 Optimization of training hyper-parameters and hardware configurations: (a) traditional separate DSE, (b) more efficient joint DSE. DS_1 and DS_2 stand for Design Space of training and hardware design, respectively.

challenging, calling for the joint optimization strategy shown in Fig. 6.1(b). The joint method avoids lengthy iterations that occur when the selected ML model is incompatible with the hardware constraints (e.g., it exceeds the available resources or cannot meet the timing requirements) and can obtain a better trade-off between ML performance and hardware performance.

In this work, we propose a new framework based on Multi-Objective Bayesian Optimization with Constraints (MOBOC) on top of High Level Synthesis (HLS) to jointly optimize the network architecture and the HLS-based hardware configurations for FPGA devices. The search space supports multi-hardware configurations. In this multi-objective framework, we can assign multiple objectives and constraints related to the hardware and network performance, and use the truly multi-objective BO approach to find the optimal Pareto sets.

6.1 Related Work

To find the optimum hyper-parameters of an ML model during training, more powerful approaches than simple Grid Search and Random Search are in use nowadays, such as Auto-Sklearn, HyperOpt, Auto-Keras, and Keras-Tuner. In [104], a review of commonly-used methods in automated ML has been presented.

Regarding the DSE aimed at optimizing the hardware configurations, several works have focused on High Level Synthesis (HLS) as the hardware design tool and proposed different methodologies for the optimum selection of HLS *pragmas* ([105], [53]). In [106] Multi-Objective Bayesian Optimization is used to tune the HLS configurations. Although the generated Pareto Fronts are close to the actual

optimal points, it could not consider *constraints* in addition to multiple *objectives* in the optimization process which increases the exploration time.

Recently, there has been a growing interest in the joint optimization of hardware and training parameters, specially in the context of Deep Neural Networks (DNNs) for which Hardware-aware Neural Architecture Search (HW-NAS) has been introduced. In recent years, several approaches based on Hw-NAS have addressed the problem of co-optimizing the network and its hardware accelerator [107–113]. To solve the multi-objective optimization problem, different methods have been adopted. Some works use a two-stage optimization, which we call *separate* method [114], [115]. In the first stage the network parameters are tuned to maximize the accuracy, and in the second stage the hardware configurations are tuned to meet the hardware constraints; for example, data precision can be reduced. Other works try to reshape the problem into a single-objective one subject to some constraints on hardware performance [116], [117]. Another method combines multiple objectives in a single function and uses single-objective optimization approaches [118]. Although these co-optimization methods have been used in several recent works, merging multiple objectives can limit the performance of the optimization and degrade the final Pareto-optimal sets. The last methodology is to employ a *truly* multi-objective optimization approach to obtain the non-dominated Pareto solutions. This approach has attracted considerable attention in the evolutionary computation community [119], [120]. However, the computational complexity in evolutionary algorithms is the main limitation of these approaches.

Not all the Hw-NAS approaches aim to co-optimize the network and its hardware accelerator. Most of them actually fall in the category of *fixed hardware configuration* (fixed-Hw) in which the search space only consists of the model architecture [121]. If the hardware requirements are not met, the network architecture has to be changed. Another category of Hw-NAS is based on *multiple hardware configurations* (multi-Hw) which extends the search space to a combination of hardware configurations and network architectures, which is also what we consider in this work.

The optimization algorithms in Hw-NAS are usually divided into Reinforcement Learning (RL), Evolutionary algorithms (EA), Gradient-based methods, and Bayesian Optimization (BO) methods [121], [122]. RL and EA have been extensively used recently for both *fixed-Hw* and *multi-Hw* categories [123–127]. Despite their proven effectiveness in several works, they have some drawbacks. For instance,

EAs are computationally intensive due to their requirement for a large *population* size in each *generation*. In RL approaches, one of the main challenges is to customize the *policies* and *reward function* for each optimization problem.

Gradient-based methods fall in the *fixed-Hw* category. In these methods a super-network containing all the possible realizations in the search space is trained and a sub-network is sampled in each Hw-NAS iteration, guiding the search to the optimal network [128]. Although hardware metrics can be included in the loss function of the super-network, the search space includes only the network configurations, hence the fixed-Hw categorization.

BO approaches consider a Gaussian Process for each objective, which is a natural fit for the optimization problems in Hw-NAS, and no customization is required in contrast to RL methods [129, 130]. In addition, a truly multi-objective optimization can be achieved with BO methods. Despite these advantage, to the best of our knowledge, BO methods in FPGAs have been used only for the fixed-Hw category.

6.2 Multi-objective Framework for Training and Hardware Co-optimization

The proposed joint optimization methodology can be divided into four parts as illustrated in Fig. 6.2 (B-E). These parts are connected inside the multi-objective Bayesian Optimization framework (A). In our methodology, MOBOC will update the samples from the search space based on the expected improvement in the Pareto sets to find the optimum configurations for network and hardware accelerator. In the following, the details of MOBOC as well as each part of the framework will be discussed thoroughly.

6.2.1 Multi-Objective BO with Constraints (MOBOC)

Bayesian Optimization is a statistical method for the optimization of black-box functions that are expensive to be evaluated. In MOBOC, a Gaussian model is built for each objective and constraint in the optimization problem (a few initial points from real evaluations are required in the beginning). The next step is to build an acquisition function based on the expected improvement of the estimated Pareto sets

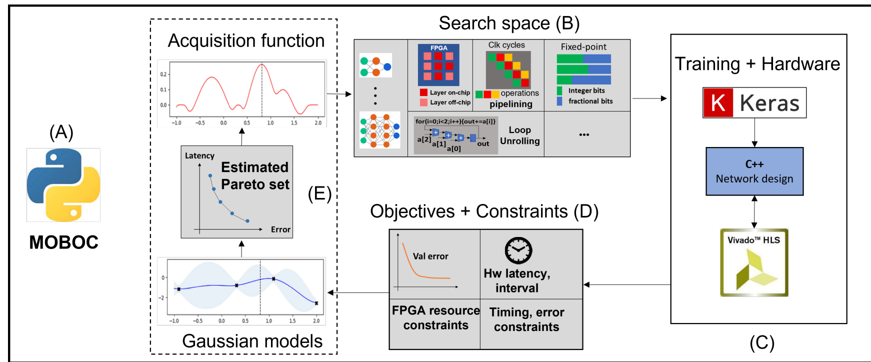


Fig. 6.2 Proposed methodology for training and hardware co-optimization in FPGA devices.

obtained from the Gaussian models. The maximum value of the acquisition function will suggest a new point in the search space. The objectives and constraints are evaluated for the new point and guide the search towards the optimum configurations by updating the Bayesian model at each iteration [131].

6.2.2 Search Space

The search space consists of the combination of all the possible values of the network training hyper-parameters and of the HLS-based hardware configurations. Note that any ML model can be used in this framework provided that we have its HLS code ready for implementation in FPGA. In this work, however, we focus on a DNN classifier for the MNIST dataset that is inspired from Lenet-5 [132] and consists of two Convolutional, two max-pooling, and three dense layers. For each layer, the set of hyper-parameters includes the number of neurons, filters, and kernel size.

The hardware accelerator design in FPGA is obtained from an HLS description of the network. Several hardware knobs can configure the accelerator in many ways, such as the choice between on-chip or off-chip memory for storing the weights of a layer, specific HLS pragmas for loop pipelining and for loop unrolling with a configurable unroll factor, enabling or disabling the Dataflow pragma for task-level concurrency, the configuration of the fixed-point precision, and the value of the clock frequency. The optimal choice of these HLS *directives* and hardware parameters is achieved by MOBOC in conjunction with the optimal training parameters.

6.2.3 Function Evaluations

At each MOBOC iteration, a new point is suggested from the search space for which the objectives and constraints must be evaluated. For this evaluation, we used Keras for training and HLS C-simulation and Synthesis for hardware verification.

6.2.4 Objectives and Constraints Extraction

In principle, we can assign any number of objectives and constraints in this framework, although the larger the number, the longer the time to update the Bayesian model in each iteration. In this work, we selected as objectives to optimize the prediction error on the validation set, the hardware latency, and the throughput; we selected as constraints the FPGA resource limits (BRAM, DSP48, FF, LUT), the clock period, and the maximum admissible error ($< 10\%$). Note that instead of using the prediction error during training, we use the prediction error after execution in hardware: in this way the error includes also the effect of the fixed-point precision.

6.2.5 Update Bayesian model

The new evaluation is used to update the Gaussian models. For the acquisition function, Predictive Entropy Search for Multi-objective Optimization with Constraints (PESMOC) [133] is used to update the expected improvement of the estimated Pareto set. After the update, the maximum value of the acquisition function corresponds to the parameters of the new suggestion. We can set a convergence criteria or a maximum number of iterations to stop the procedure.

6.3 Evaluation on Neural Networks

6.3.1 Multi-Layer Perceptron (MLP)

For the evaluation, we used the MNIST dataset to train a Multi-Layer Perceptron (MLP) to be implemented in a Zynq7000 FPGA. We used hls4ml [14] to convert the MLP model to a synthesizable C++ code. The model hyper-parameters and the ranges of the HLS and hardware knobs are in Tab.6.1.

Table 6.1 Ranges of parameters for the joint training/hardware optimization method.

Inputs	Clk (ns)	Hidden Layers	Neurons	Precision #total	Precision #Integer	Reuse factor	Array Partition	Learning rate	Regul-arization rate
Ranges	4 - 7	1 - 3	32 - 256 step = 32	12 - 16	4 - 6	1 - 4	2^x $x = [1-8]$	$1 \times 10^{(-x)}$ $x = [2-7]$	$1 \times 10^{(-x)}$ $x = [2-7]$

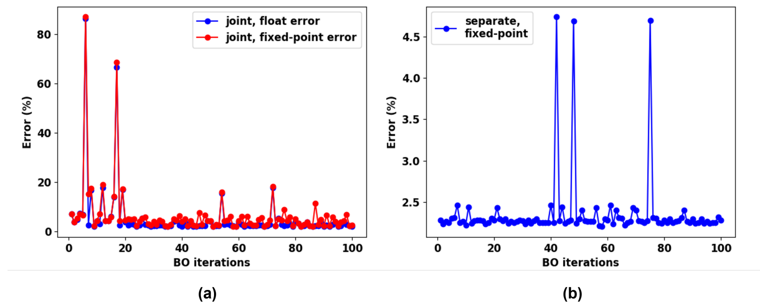


Fig. 6.3 Percentage of training error (float error) and hardware error (fixed-point error) in each BO iteration, (a) proposed joint optimization, (b) separate optimization.

Fig. 6.3 compares the evolution of the training error using a joint (Fig. 6.3(a)) and a separate (Fig. 6.3(b)) optimization approach. Fig. 6.3(a) shows that both floating-point error during training and fixed-point error in hardware converge as the BO iterations progress. Since the separate method returns only the best training result, Fig. 6.3(b) shows only the fixed-point inference error and shows an immediately low error in all BO iterations (less than 5%). This is because the separate hardware design starts with a neural network already optimized in terms of training error, which only needs to be tailored to the hardware target.

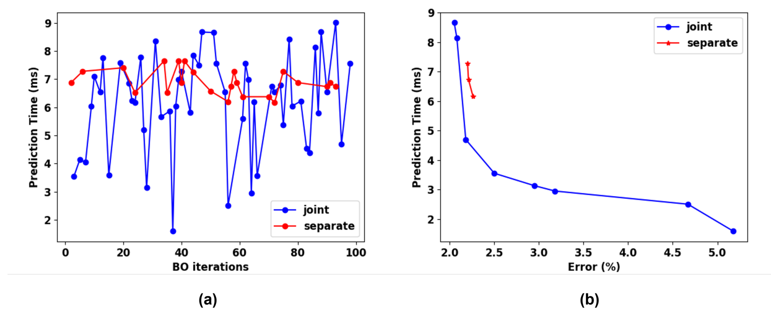


Fig. 6.4 Comparison of (a) prediction time and (b) Pareto fronts.

This last optimization of the separate method, however, is constrained by the initial training. As a result, the BO cannot reach the same latency performance of the

joint optimization. This is visible in Fig. 6.5(a), with the relatively high prediction time for the separate optimization method (red points).

The efficiency of the proposed methodology is apparent in Fig. 6.5(b), which compares the Pareto curves obtained by separate (red) and joint (blue) optimization methods. In the red curve, the training optimization is done by Keras-Tuner. Note that in this case Keras-Tuner suggests an initial MLP with three layers and a number of neurons that could not fit in the FPGA due to excessive BRAM usage, leading to a failure in the subsequent hardware BO. This required a second iteration to limit the neurons range from $[32-256]$ to $[32-128]$ in the training DSE, which returned a feasible three-layer MLP with 128 neurons in each layer, low error but relatively high prediction time. The subsequent hardware DSE returned only three (red) Pareto points. On the contrary, the joint method returns many more valid Pareto points (blue) because of its ample maneuverability in the combined trainig and hardware design spaces. Most importantly, the blue points dominate the red ones, as clearly shown in Fig. 6.5(b).

6.3.2 Convolutional Neural Networks (CNN)

The FPGA target for the implementation of the CNN accelerator inspired from Lenet-5 is a Zynq-7000 SoC (XC7Z020-CSG484). For the implementation of MOBOC, the latest *Spearmint* package is used [133]. The search space is described in Tab. 6.2, in which *On Chip* refers to the option of selecting on-chip BRAMs to store the weights and bias values for each layer (as opposed to an off-chip external DRAM), hence 5 options are available for 5 layers. Total and Integer bits in Tab. 6.2 determine the fixed-point data precision used that can be individually configured for the weights, the activations, and the accumulations for the intermediate calculations (i.e., sum of products). Note that the search space is quite large (with total configurations on the order of 10^{16}) and it would take years to do an exhaustive search to determine the best configurations. It should be mentioned that the search space can be extended to include other parameters such as learning rate. However, the BO convergence would be slower: in the current setting 100 BO iterations are sufficient to obtain a Pareto set, with more parameters more iterations would be needed.

Fig. 6.5 and Tab. 6.3 show the Pareto-optimal points in the space defined by prediction error evaluated in hardware (hardware error, *x* axis) and execution latency

(Time, *y* axis) achieved by three different methods: Random search, Separate, and the proposed Joint method. The search space for all the methods is the same and the total number of iterations for each method is set to 100. In the Random case, we randomly choose in each iteration a point in the search space. In the Separate method, we first optimize the network hyper-parameters to obtain the best accuracy, and then BO optimizes the hardware configurations to maximize the hardware performance. In the Joint method, BO jointly optimizes the choice of the hyper-parameters and the hardware configuration to maximize both prediction accuracy and performance. Note that in Random search, as opposed to other two methods, we cannot set a constraint for hardware error, so there is one point with error larger than the admissible threshold ($>$ 10%). With the exception of one single point randomly detected by the Random search, all the other points in both the Random and Separate method are Pareto-dominated by the points found by the Joint method.

To help speed-up the BO convergence toward results with low prediction error, we modified the objective function with a non-linear exponential of the error. As shown in Fig. 6.6 where the error is reported as a function of the BO iterations, the exponential definition of the error in the objective function helps reach a quick convergence toward solutions with low hardware error, compared to a linear definition. The final results in this work are based on the exponential error.

Finally, Fig. 6.7 shows all the points suggested by the proposed joint MOBOC and the other two methods in 100 iterations. For better clarity, a log scale is selected for the hardware error. We can see that due to the error threshold in MOBOC, points are more concentrated in the error range below 10% for our joint method, which helps in better convergence of the optimization process by searching in the area of interest.

Table 6.2 Search space featuring network architecture and hardware configurations (UF:Unroll Factor)

Configs	Range	Configs	Range
#Filters Conv1	[2,20]	Dataflow	active/inactive
		On Chip	active/inactive
#Filters Conv2	[2,20]	Clk (ns)	[8,20]
		Total bits	[10,20]
Kernel size	2*([1,4]) + 1	Integer bits	[1,9]
#Neurons Dense1	[50,150]	Conv1 UF	2^([0,4])
#Neurons Dense2	[50,150]	Conv2 UF	2^([0,4])

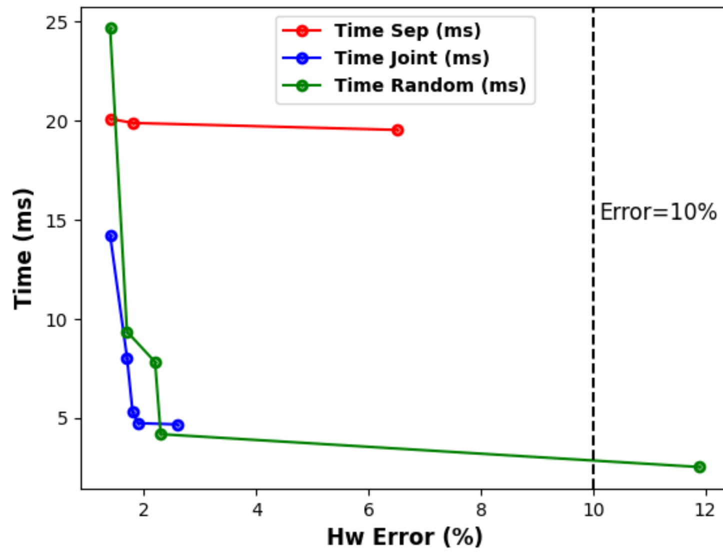


Fig. 6.5 Pareto-points found by the joint approach, random search, and conventional separate method in the space of prediction error (Hw error) and execution latency (Time). Total number of iterations is 100 for all methods.

Table 6.3 Pareto points obtained by three methods.

Random		Separate		Joint	
Error (%)	Time (ms)	Error (%)	Time (ms)	Error (%)	Time (ms)
1.4	24.67	1.4	20.1	1.4	14.22
1.7	9.36	1.8	19.89	1.7	8.06
2.2	7.83	6.5	19.54	1.8	5.31
2.3	4.19	-	-	1.9	4.74
11.9	2.53	-	-	2.6	4.68

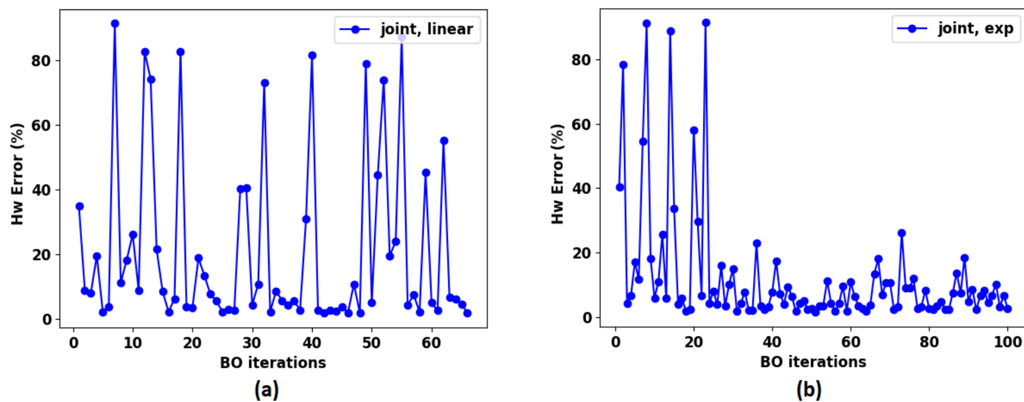


Fig. 6.6 Hardware error in our joint method in each BO iteration with (a) linear and (b) exponential error function.

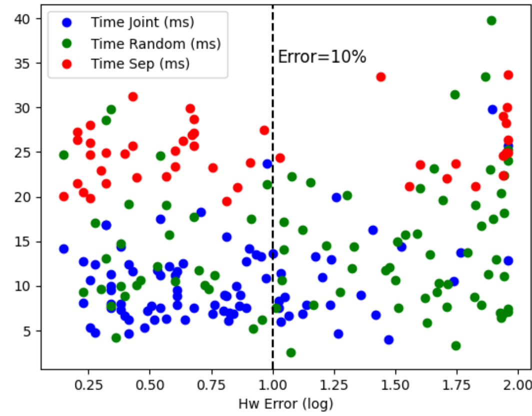


Fig. 6.7 Total points suggested by the joint, separate, and random search methods; note the concentration of the joint method on low errors ($< 10\%$).

6.3.3 MLP in Microwave Data Set

Detection and classification of anomalies from the scattering matrix can be done by using Neural Networks (NN). Multi-Layer Perceptron (MLP) is a fully connected network that can be used in MI diagnosis. In [134], an MLP is used for breast cancer detection with microwave sensing. When designing a hardware accelerator for the MLP, there are several parameters that must be tuned to maximize the performance as shown in previous sections. We proposed the methodology described in section 6.2 to achieve the optimum set of parameters and we evaluated the efficiency of the Parero fronts by using the MNIST data set. In this part, we focus on the hardware performance of the MLP accelerator by considering an optimum set of hyper-parameters for our design. To design and evaluate the MLP accelerator, we used hls4ml [14] that is a tool to convert recurrent ML models from Python to a synthesizable code that can be used in Vivado HLS. We used the same dataset of microwave measurements that we used for the evaluation of SVM accelerator introduced in Chapter 5, which contain 4500 samples of scattering matrix with 462 elements. It consists of 9 classes representing the presence, type, and location of the brain stroke. For feature extraction, PCA is used that results in the reduction of features to 110. We selected an MLP with 3 hidden layers. The numbers of neurons per layer are $220, 64, 64$, respectively. The target device is a Zynq SoC (ZedBoard), and we used fixed-point precision for the hardware implementation by leveraging hls4ml. The accuracy before and after hardware implementation, resource usage, and

processing time are depicted in Table 6.4 and Figs. 6.8 and 6.9. Note the negligible accuracy loss in the hardware accelerator due to the reduced precision.

Table 6.4 Evaluation of MLP performance in microwave anomaly detection. Fixed-point precision is selected in hardware with total and integer widths of 16 and 10, respectively.

training accuracy (%)	test accuracy (%)	hardware accuracy (%)	Latency (ms)
99.6	98	97.5	1.12

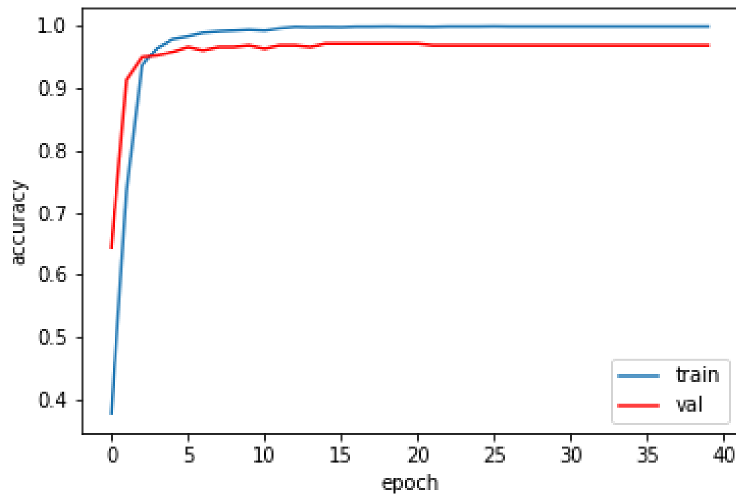


Fig. 6.8 Accuracy of MLP during training by MI dataset

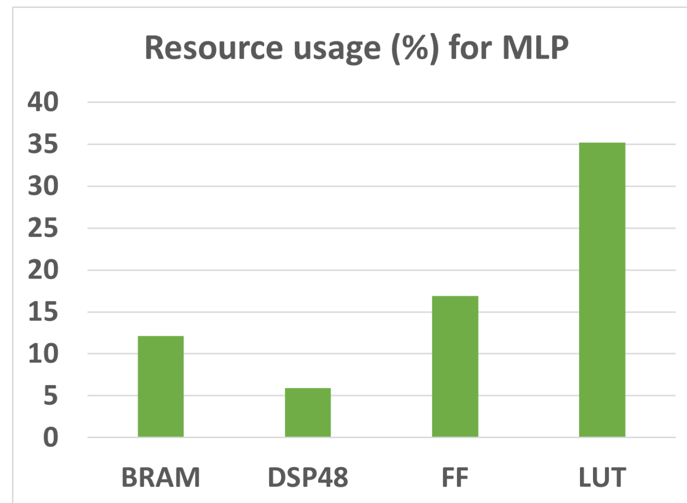


Fig. 6.9 Resource usage for MLP in Zynq FPGA

6.4 Conclusions

In this chapter, we proposed a new methodology for joint optimization of DNN training and hardware configurations in FPGAs using Keras and Vivado HLS for training and hardware evaluation, respectively, within a multi-objective Bayesian Optimization (BO) framework. In contrast to more common BO approaches, the search space includes multiple hardware configurations in addition to the hyper-parameters used for DNN training. Moreover, conflicting objectives related to training and hardware performance can be considered separately to achieve a Pareto-optimal set of configurations without merging them into a single objective function. We compared our joint methodology with conventional approaches, random search and separate optimization method. The **Pareto set achieved with our method outperforms those obtained with the other two methods, with** $1.7\times$ **and** $1.4\times$ **improvement in execution time for the minimum error compared to random and separate methods, respectively**. In the future, other co-optimization algorithms based on RL or EA can be compared with this work to further analyse the efficiency of the proposed approach.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

In this thesis we presented efficient hardware design methodologies to accelerate the execution of domain-specific kernels that are recurrent in a broad domain of applications. We focused on two main application areas that are biomedical microwave techniques and Machine Learning (ML). Specifically, we proposed different hardware accelerators for 3D FDTD, PCA, SVM, as well as design optimization techniques for ML models, including Multi-Layer Perceptrons (MLPs) and Convolutional Neural Networks (CNNs). We used High Level Synthesis (HLS) as the trending approach to design these hardware accelerators and optimize their performance. The flexibility, completeness (considering substantial details in each algorithm), efficiency, and high level design techniques and hardware optimization methodologies are the distinguishing features of the proposed accelerators compared to previous related works. Specifically, these are the main contributions and achievements in our thesis:

- The accelerator for 3D FDTD considers the CPML boundary conditions for all directions, models the dispersive materials, and is designed as an internal step for a non-linear iterative image reconstruction algorithm in medical Microwave Imaging. Two architectures were proposed based on the number of interfaces and the amount of consumed hardware resources. The first one is the *Large* design which has 18 interfaces connecting the accelerator to the external memory (which is still compatible with the memory specifications), requires

a higher memory bandwidth, consumes more resources, and results in the highest speed *per antenna*. The second architecture is the *Small* design, in which the number of interfaces is reduced to 15 and there is lower resource usage. In this way, although the speed performance will degrade *per antenna*, it is possible to fit more antennas in a single FPGA (meaning that we can execute FDTD for more parallel antennas in one FPGA) due to the lower resource consumption per antenna. Therefore, the execution time for multiple antennas in the Small design will improve compared to the Large design. For the reference data set with the dimensions of $70 \times 70 \times 70$, each FPGA in the Small and Large design can be used for 3 and 2 antennas, respectively. The execution time per antenna for our FPGA accelerator is $3.38s$ for the Small design which is slightly better than the conventional GPU design with Acceleware library ($4.88s$). In addition, we presented a multi-FPGA design in which multiple antennas can be assigned to multiple FPGAs. The system level evaluation of our multi-FPGA design with 24 antennas and 8 UltraScale+ FPGAs shows a $13\times$ speed-up compared to the single GPU design on Tesla P40 (Acceleware).

- The proposed PCA accelerator in FPGA consists of several compute units, including the computations of *Mean*, *Covariance*, *SVD or EVD*, and *Projection of inputs*. Although separate acceleration techniques for each compute unit have been already proposed, their integration in an efficient hardware accelerator has not been considered before. We proposed a flexible FPGA accelerator for PCA that can be used for different data dimensions, thanks to the HLS design methodology. In addition to the floating-point design, a fixed-point implementation was proposed to efficiently use the hardware resources in FPGA. For the computation of Covariance matrix, an efficient block streaming methodology was introduced to read blocks of input data from the external memory, store them in on-chip memory and process them inside the hardware accelerator, while reading the next block of data from external memory. In addition, to accelerate the computation of singular values or eigenvalues in hardware, we used either an optimized SVD accelerator in floating-point precision, or an efficient EVD accelerator in fixed-point precision. The SVD accelerator was based on a built-in floating-point hardware block in one of the HLS tool libraries, which was optimized to be used in PCA algorithm. The fixed-point EVD accelerator was designed entirely in HLS starting from its

software code in C++. We evaluated the performance of our PCA accelerator on a hyperspectral imaging data set with both fixed-point and floating-point precision and with different block sizes in the blocking method. Different data dimensions were also analyzed, and compared to the GPU or multi-core CPU designs we could achieve either power efficiency or performance speed-up. In addition, we compared our HLS design an RTL implementation of PCA using VHDL and achieved a 2.3× speed-up as well as significant reduction of resource usage.

- FPGA accelerator for SVM uses a novel dataflow architecture in which the required input data are transferred to the accelerator while the SVM computations are performed. The proposed accelerator is scalable, meaning that it is not limited by the number of dimensions of Support Vectors and can be used for large data dimensions when there is limited on-chip memory. In addition, a fixed-point implementation was presented for the SVM accelerator which could improve the speed. The entire accelerator was designed using the HLS tool and we could apply different hardware optimization techniques. One of the main characteristics of our design is having *adjustable parallelism* which enables the designer to specify the amount of parallelism in hardware depending on the available resources. For this purpose, three parameters were introduced to control the level of parallelism in hardware, which determine the number of data to read from memory (N), the number of FIFO channels in the data flow region (BF), and the loop unroll factor in the computation of Decision function (DF). As opposed to most of the conventional SVM accelerators in FPGA, multi-class classification is also supported in our design. In addition, we explored the impact of different SVM kernels on the hardware performance. For the evaluation, we compared our HLS-based hardware accelerator with recent works which were designed using either HLS or the manual RTL approach. A 4.4× latency improvement could be achieved compared to the RTL design, and a minimum of 10× improvement in the latency was obtained compared to a similar HLS-based design.

- Finally, for the optimum design of Machine Learning models and specifically Neural Networks in FPGAs, a new methodology was proposed that could jointly optimize the training hyper-parameters and HLS-based hardware configurations based on multi-Objective Bayesian optimization. Although the

proposed approach can be used for any ML model, we did a primary evaluation on two network architectures, which are Multi-Layer Perceptron (MLP) and Convolutional Neural Networks (CNNs). Compared to other Hw-NAS solutions based on Bayesian Optimization, the search space in our method supports *multi-Hw* configurations and consists of a combination of network architectures and hardware configurations. In addition, the available hardware resources in the FPGA can be set as *Constraints* in the optimization problem together with different contrasting *Objectives* such as hardware latency and accuracy loss. We considered a large search space and compared our joint optimization methodology with the separate optimization and the Random search approach. From the comparison, we could notice the improvement in the Pareto sets obtained by the proposed joint approach, with $1.7\times$ and $1.4\times$ improvement in the execution time for the minimum error compared to the random and separate methods, respectively, in a Lenet5-inspired CNN architecture, and $1.43\times$ improvement in the prediction time without an increase in the prediction error compared to the separate design, in the MLP network architecture.

The above-mentioned hardware accelerators are highly beneficial for high-performance embedded computing systems. As stated previously, 3D FDTD is used in non-linear iterative medical microwave image reconstruction, and its acceleration helps in reducing the amount of time needed to reconstruct the final image. FPGA acceleration of SVD/EVD (included in PCA) is useful in linear microwave image reconstruction algorithms. Hardware acceleration of PCA, SVM, and Neural Networks are advantageous in not only microwave imaging, but also in other Machine Learning applications which use these algorithms.

7.1.1 List of Published Papers

In the following, the list of the published papers is presented:

1. M. A. Mansoori, P. Lu and M. R. Casu, "FPGA Acceleration of 3D FDTD for Multi- Antennas Microwave Imaging Using HLS," in *IEEE Access*, vol. 9, pp. 122696-122711, 2021.

2. M. A. Mansoori.; M. R. Casu, “High Level Design of a Flexible PCA Hardware Accelerator Using a New Block-Streaming Method”, *Electronics* 2020, 9, 449.

3. M. A. Mansoori and Mario R. Casu, “HLS-Based Flexible Hardware Accelerator for PCA Algorithm on a Low-Cost ZYNQ SoC,” *2019 IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC)*, 2019, pp. 1-7.

4. M. A. Mansoori and Mario R. Casu, “Efficient FPGA Implementation of PCA Algorithm for Large Data using High Level Synthesis,” *2019 15th Conference on Ph.D Research in Microelectronics and Electronics (PRIME)*, 2019, pp. 65-68.

5. D. O. Rodriguez Duarte, M. A. Mansoori, J. A. Tobon Vasquez, G. Turvani, M. R. Casu and F. Vipiana, “evelopment of an EM Device for Cerebrovascular Diseases Imaging and Hardware Acceleration for Imaging Algorithms within the EMERALD Network,” *2019 13th European Conference on Antennas and Propagation (EuCAP)*, 2019, pp. 1-3.

6. M. A. Mansoori, Mario R. Casu, “Efficient Training and Hardware Co-Design of Machine Learning Models”, In: *Applications in Electronics Pervading Industry, Environment and Society. ApplePies 2021*. Lecture Notes in Electrical Engineering, vol. 866, 2022

7. M. A. Mansoori, Mario R. Casu, “Hardware Acceleration of Biomedical Microwave Techniques using High Level Synthesis”, In *16th European Conference on Antennas and Propagation (EuCAP)*, 2022 (accepted).

8. M. A. Mansoori, Mario R. Casu, “HLS-based dataflow hardware architecture for Support Vector Machine in FPGA”, In: *International Symposium on Circuits and Systems (ISCAS)* 2022 (accepted).

7.2 Future Work

As the future work and in order to improve the performance of the proposed hardware accelerators, we provide some suggestions in the following:

- To implement PCA algorithm in FPGA, we used a built-in function for SVD in HLS and optimized its performance. However, the HLS implementation of SVD uses floating-point data precision. As a future work, it is highly beneficial to have a fixed-point implementation of SVD. For this purpose, we need to develop a high level implementation of SVD algorithm by using fixed-point data precision with efficient data widths, which can result in the minimum accuracy loss while increasing the throughput and reducing the power consumption.

- We designed the EVD accelerator that can use either fixed-point or floating point precision. However, its performance can be improved by using a block-streaming method (similar to the PCA accelerator). It can be extended to be used for the fixed-point SVD accelerator.

- In 3D FDTD, it is possible to reduce the memory usage by reducing the block size in the spatial blocking method and using overlapped tiling approach instead. One of the drawbacks of this approach is that it results in some redundant computations in the overlapped regions which are termed "halo" regions.

  In addition to "spatial blocking" and overlapped tilling approach, it is possible to improve the throughput by exploiting "temporal blocking". Temporal blocking allows for pipelining multiple iterations of FDTD without storing the intermediate results in the external memory. However, combination of spatial and temporal blocking for 3D FDTD with CPML boundary conditions creates additional challenges. For example, local memory usage will increase due to the storage of multiple spatial blocks in different iterations. In addition, by using overlapped tilling approach in parallel with temporal blocking, the size of the halo regions will increase in subsequent iterations making it more difficult to design the FDTD accelerator with these optimization techniques in HLS.

- In ML applications, a classifier is usually used after the feature extraction step. Therefore, the combination of PCA algorithm (as the feature extractor) and SVM algorithm (as the classifier) is useful in a variety of applications. A hardware accelerator containing the combination of PCA and SVM can be considered as a future work.

- In addition to other ML classifiers such as KNN, Decision Trees, and Random forest, the ensemble of different classifiers can be implemented in hardware to optimize their performance.
- We can obtain the optimum values of block size in PCA or FDTD, and other parameters in HLS computing kernels by using the proposed co-optimization framework in Chapter 6.
- We can extend the new co-optimization framework in Chapter 6 for the larger networks (GANs, CNNs, ...).

References

[1] William J. Dally, Yatish Turakhia, and Song Han. Domain-specific hardware accelerators. *Commun. ACM*, 63(7):48–57, jun 2020.

[2] Michael Dossis. High-level synthesis for embedded systems. In Kiyofumi Tanaka, editor, *Embedded Systems*, chapter 16. IntechOpen, Rijeka, 2012.

[3] Ivan Nunes Da Silva, Danilo Hernane Spatti, Rogerio Andrade Flauzino, Luisa Helena Bartocci Liboni, and Silas Franco dos Reis Alves. Artificial neural networks. *Cham: Springer International Publishing*, 39, 2017.

[4] Meha Desai and Manan Shah. An anatomization on breast cancer detection and diagnosis employing multi-layer perceptron neural network (mlp) and convolutional neural network (cnn). *Clinical eHealth*, 4:1–11, 2021.

[5] Frank Vahid and Tony Givargis. *Embedded System Design: A Unified Hardware/Software Introduction*. John Wiley & Sons, Inc., USA, 1st edition, 2001.

[6] Razvan Nane, Vlad-Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Brown, Fabrizio Ferrandi, Jason Anderson, and Koen Bertels. A survey and evaluation of fpga high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(10):1591–1604, 2016.

[7] Heiner Giefers, Christian Plessl, and Jens Förstner. Accelerating finite difference time domain simulations with reconfigurable dataflow computers. *ACM SIGARCH Computer Architecture News*, 41(5):65–70, 2014.

[8] Koji Okina, Rie Soejima, Kota Fukumoto, Yuichiro Shibata, and Kiyoshi Oguri. Power performance profiling of 3-d stencil computation on an fpga accelerator for efficient pipeline optimization. *ACM SIGARCH Computer Architecture News*, 43(4):9–14, 2016.

[9] Amine Ait Si Ali, Abbes Amira, Faycal Bensaali, and Mohieddine Benammar. Hardware pca for gas identification systems using high level synthesis on the zynq soc. In *2013 IEEE 20th International Conference on Electronics, Circuits, and Systems (ICECS)*, pages 707–710. IEEE, 2013.

[10] Mathias Schellhorn and Gunther Notni. Optimization of a principal component analysis implementation on field-programmable gate arrays (fpga) for analysis of spectral images. In *2018 Digital Image Computing: Techniques and Applications (DICTA)*, pages 1–6. IEEE, 2018.

[11] Shereen Afifi, Hamid GholamHosseini, and Roopak Sinha. A low-cost fpga-based svm classifier for melanoma detection. In *2016 IEEE EMBS Conference on Biomedical Engineering and Sciences (IECBES)*, pages 631–636. IEEE, 2016.

[12] Shereen Afifi, Hamid GholamHosseini, and Roopak Sinha. Svm classifier on chip for melanoma detection. In *2017 39th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, pages 270–274. IEEE, 2017.

[13] Shereen Afifi, Hamid GholamHosseini, and Roopak Sinha. A system on chip for melanoma detection using fpga-based svm classifier. *Microprocessors and Microsystems*, 65:57–68, 2019.

[14] Farah Fahim, Benjamin Hawks, Christian Herwig, James Hirschauer, Sergo Jindariani, Nhan Tran, Luca P Carloni, Giuseppe Di Guglielmo, Philip Harris, Jeffrey Krupa, et al. hls4ml: An open-source codesign workflow to empower scientific low-power machine learning devices. *arXiv preprint arXiv:2103.05579*, 2021.

[15] Wenyi Shao and Todd McCollough. Advances in microwave near-field imaging: Prototypes, systems, and applications. *IEEE microwave magazine*, 21(5):94–119, 2020.

[16] Rohit Chandra, Ilangko Balasingham, Huiyuan Zhou, and Ram M Narayanan. Medical microwave imaging and analysis. In *Medical Image Analysis and Informatics: Computer-Aided Diagnosis and Therapy*, pages 451–466. CRC Press, 2017.

[17] Hyuna Sung, Jacques Ferlay, Rebecca L Siegel, Mathieu Laversanne, Isabelle Soerjomataram, Ahmedin Jemal, and Freddie Bray. Global cancer statistics 2020: Globocan estimates of incidence and mortality worldwide for 36 cancers in 185 countries. *CA: a cancer journal for clinicians*, 71(3):209–249, 2021.

[18] Amran Hossain, Mohammad Tariqul Islam, Md. Tarikul Islam, Muhammad E. H. Chowdhury, Hatem Rmili, and Md. Samsuzzaman. A planar ultrawide-band patch antenna array for microwave breast tumor detection. *Materials*, 13(21), 2020.

[19] Daniel Oloumi, Robert S. C. Winter, Atefeh Kordzadeh, Pierre Boulanger, and Karumudi Rambabu. Microwave imaging of breast tumor using time-domain uwb circular-sar technique. *IEEE Transactions on Medical Imaging*, 39(4):934–943, 2020.

[20] Mario R. Casu, Marco Vacca, Jorge A. Tobon, Azzurra Pulimeno, Imran Sarwar, Raffaele Solimene, and Francesca Vipiana. A cots-based microwave imaging system for breast-cancer detection. *IEEE Transactions on Biomedical Circuits and Systems*, 11(4):804–814, 2017.

[21] Daniele Jahier Pagliari, Mario R. Casu, and Luca P. Cartoni. Acceleration of microwave imaging algorithms for breast cancer detection via high-level synthesis. In *2015 33rd IEEE International Conference on Computer Design (ICCD)*, pages 475–478, 2015.

[22] Saptarshi Mukherjee, Lalita Udpa, Satish Udpa, Edward J. Rothwell, and Yiming Deng. A time reversal-based microwave imaging system for detection of breast tumors. *IEEE Transactions on Microwave Theory and Techniques*, 67(5):2062–2075, 2019.

[23] Natalia K. Nikolova. Microwave imaging for breast cancer. *IEEE Microwave Magazine*, 12(7):78–94, 2011.

[24] Imran Sarwar, Giovanna Turvani, Mario R. Casu, Jorge A. Tobon, Francesca Vipiana, Rosa Scapaticci, and Lorenzo Crocco. Low-cost low-power acceleration of a microwave imaging algorithm for brain stroke monitoring. *Journal of Low Power Electronics and Applications*, 8(4), 2018.

[25] David O. Rodriguez-Duarte, Jorge A. Tobón Vasquez, Rosa Scapaticci, Lorenzo Crocco, and Francesca Vipiana. Assessing a microwave imaging system for brain stroke monitoring via high fidelity numerical modelling. *IEEE Journal of Electromagnetics, RF and Microwaves in Medicine and Biology*, 5(3):238–245, 2021.

[26] Igor Bisio, Claudio Estatico, Alessandro Fedeli, Fabio Lavagetto, Matteo Pastorino, Andrea Randazzo, and Andrea Sciarrone. Brain stroke microwave imaging by means of a newton-conjugate-gradient method in <inline-formula> <tex-math notation="latex">l^p </tex-math></inline-formula> banach spaces. *IEEE Transactions on Microwave Theory and Techniques*, 66(8):3668–3682, 2018.

[27] Igor Bisio, Claudio Estatico, Alessandro Fedeli, Fabio Lavagetto, Matteo Pastorino, Andrea Randazzo, and Andrea Sciarrone. Variable-exponent lebesgue-space inversion for brain stroke microwave imaging. *IEEE Transactions on Microwave Theory and Techniques*, 68(5):1882–1895, 2020.

[28] Syed Ahsan, Maria Koutsoupidou, Eleonora Razzicchia, Ioannis Sotiriou, and Panagiotis Kosmas. Advances towards the development of a brain microwave imaging scanner. In *2019 13th European Conference on Antennas and Propagation (EuCAP)*, pages 1–4, 2019.

[29] Marco Salucci, Alessandro Polo, and Jan Vrba. Multi-step learning-by-examples strategy for real-time brain stroke microwave scattering data inversion. *Electronics*, 10(1), 2021.

[30] Thomas Haugland Johansen, Kajsa Møllersen, Samuel Ortega, Himar Fabelo, Aday Garcia, Gustavo M Callico, and Fred Godtliebsen. Recent advances in hyperspectral imaging for melanoma detection. *Wiley Interdisciplinary Reviews: Computational Statistics*, 12(1):e1465, 2020.

[31] Iqbal H Sarker. Machine learning: Algorithms, real-world applications and research directions. *SN Computer Science*, 2(3):1–21, 2021.

[32] Shaveta Dargan, Munish Kumar, Maruthi Rohit Ayyagari, and Gulshan Kumar. A survey of deep learning and its applications: a new paradigm to machine learning. *Archives of Computational Methods in Engineering*, 27(4):1071–1092, 2020.

[33] Fioralba Cakoni, David Colton, and Peter Monk. *The linear sampling method in inverse electromagnetic scattering*. SIAM, 2011.

[34] Massimo Brignone, Giovanni Bozza, Andrea Randazzo, Michele Piana, and Matteo Pastorino. A hybrid approach to 3d microwave imaging by using linear sampling and aco. *IEEE Transactions on Antennas and Propagation*, 56(10):3224–3232, 2008.

[35] Mehmet Nuri Akıncı, Tuğhan Çağlayan, Selçuk Özgür, Uğur Alkaşı, Habibullah Ahmadzay, Mehmet Abbak, Mehmet Çayören, and Ibrahim Akduman. Qualitative microwave imaging with scattering parameters measurements. *IEEE Transactions on Microwave Theory and Techniques*, 63(9):2730–2740, 2015.

[36] Jacob D Shea, Barry D Van Veen, and Susan C Hagness. A tsvd analysis of microwave inverse scattering for breast imaging. *IEEE Transactions on Biomedical Engineering*, 59(4):936–945, 2011.

[37] Md Delwar Hossain and Ananda Sanagavarapu Mohan. Cancer detection in highly dense breasts using coherently focused time-reversal microwave imaging. *IEEE Transactions on Computational Imaging*, 3(4):928–939, 2017.

[38] Maged A Aldhaeebi, Khawla Alzoubi, Thamer S Almoneef, Saeed M Bamatraf, Hussein Attia, and Omar M Ramahi. Review of microwaves techniques for breast cancer detection. *Sensors*, 20(8):2390, 2020.

[39] Raquel Cruz Conceição, Johan Jacob Mohr, Martin O'Halloran, et al. *An introduction to microwave imaging for breast cancer detection*. Springer, 2016.

[40] Colin Gilmore, Aria Abubakar, Wenyi Hu, Tarek M Habashy, and Peter M van den Berg. Microwave biomedical data inversion using the finite-difference contrast source inversion method. *IEEE Transactions on Antennas and Propagation*, 57(5):1528–1538, 2009.

[41] Igor Bisio, Claudio Estatico, Alessandro Fedeli, Fabio Lavagetto, Matteo Pastorino, Andrea Randazzo, and Andrea Sciarrone. Brain stroke microwave imaging by means of a newton-conjugate-gradient method in <inline-formula> <tex-math notation="latex">l^p </tex-math></inline-formula> banach spaces. *IEEE Transactions on Microwave Theory and Techniques*, 66(8):3668–3682, 2018.

[42] Zhenzhuang Miao and Panagiotis Kosmas. Multiple-frequency dbim-twist algorithm for microwave breast imaging. *IEEE Transactions on Antennas and Propagation*, 65(5):2507–2516, 2017.

[43] Matteo Pastorino. *Microwave imaging*. John Wiley & Sons, 2010.

[44] John B Schneider. Understanding the finite-difference time-domain method; 2010. *URL www. eecs. wsu. edu/schneidj/ufdtd*, 2015.

[45] Declan O'Loughlin, Martin O'Halloran, Brian M. Moloney, Martin Glavin, Edward Jones, and M. Adnan Elahi. Microwave breast imaging: Clinical advances and remaining challenges. *IEEE Transactions on Biomedical Engineering*, 65(11):2580–2590, 2018.

[46] Jorge A. Tobon Vasquez, Rosa Scapaticci, Giovanna Turvani, Gennaro Bellizzi, David O. Rodriguez-Duarte, Nadine Joachimowicz, Bernard Duchêne, Enrico Tedeschi, Mario R. Casu, Lorenzo Crocco, and Francesca Vipiana. A prototype microwave system for 3d brain stroke imaging. *Sensors*, 20(9), 2020.

[47] Acceleware Ltd 2016 Acceleware FDTD solvers http://www.acceleware.com/fdtd-solvers.

[48] Johannes de Fine Licht, Maciej Besta, Simon Meierhans, and Torsten Hoefler. Transformations of high-level synthesis codes for high-performance computing. *IEEE Transactions on Parallel and Distributed Systems*, 32(5):1014–1029, 2020.

[49] Sakari Lahti, Panu Sjövall, Jarno Vanne, and Timo D Hämäläinen. Are we there yet? a study on the state of high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(5):898–911, 2018.

[50] Benjamin Carrion Schafer and Zi Wang. High-level synthesis design space exploration: past, present, and future. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(10):2628–2639, 2019.

[51] Young-Kyu Choi, Yuze Chi, Jie Wang, and Jason Cong. Flash: Fast, parallel, and accurate simulator for hls. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(12):4828–4841, 2020.

[52] Lorenzo Ferretti, Jihye Kwon, Giovanni Ansaloni, Giuseppe Di Guglielmo, Luca P Carloni, and Laura Pozzi. Leveraging prior knowledge for effective design-space exploration in high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):3736–3747, 2020.

[53] Jieru Zhao, Liang Feng, Sharad Sinha, Wei Zhang, Yun Liang, and Bingsheng He. Performance modeling and directives optimization for high-level synthesis on fpga. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(7):1428–1441, 2019.

[54] Ximin Wang, Song Liu, Xuan Li, and Shuangying Zhong. Gpu-accelerated finite-difference time-domain method for dielectric media based on cuda. *International Journal of RF and Microwave Computer-Aided Engineering*, 26(6):512–518, 2016.

[55] Zhang Bo, Xue Zheng-hui, Ren Wu, Li Wei-ming, and Sheng Xin-qing. Accelerating fdtd algorithm using gpu computing. In *2011 IEEE International Conference on Microwave Technology & Computational Electromagnetics*, pages 410–413. IEEE, 2011.

[56] Shuo Liu, Bin Zou, Lamei Zhang, and Shulei Ren. Heterogeneous cpu+gpu-accelerated fdtd for scattering problems with dynamic load balancing. *IEEE Transactions on Antennas and Propagation*, 68(9):6734–6742, 2020.

[57] Hanyong Zhang, Yuan Lei, Haizeng Ye, and Yuhan Gong. Bistatic radar cross section prediction of 3-d target based on gpu-fdtd method. In *2018 12th International Symposium on Antennas, Propagation and EM Theory (ISAPE)*, pages 1–4. IEEE, 2018.

[58] Tobias Kenter, Jens Förstner, and Christian Plessl. Flexible fpga design for fdtd using opencl. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–7. IEEE, 2017.

[59] Yasuhiro Takei, Hasitha Muthumala Waidyasooriya, Masanori Hariyama, and Michitaka Kameyama. Fpga-oriented design of an fdtd accelerator based on overlapped tiling. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, page 72. The Steering Committee of The World Congress in Computer Science, Computer ..., 2015.

[60] Hasitha Muthumala Waidyasooriya, Tsukasa Endo, Masanori Hariyama, and Yasuo Ohtera. Opencl-based fpga accelerator for 3d fdtd with periodic and absorbing boundary conditions. *International Journal of Reconfigurable Computing*, 2017, 2017.

[61] Chang Kong and Tao Su. Parallel hardware architecture of the 3d fdtd algorithm with convolutional perfectly matched layer boundary condition. *Progress In Electromagnetics Research C*, 105:161–174, 2020.

[62] Hasitha Muthumala Waidyasooriya, Yasuhiro Takei, Shunsuke Tatsumi, and Masanori Hariyama. Opencl-based fpga-platform for stencil computation and its optimization methodology. *IEEE Transactions on Parallel and Distributed Systems*, 28(5):1390–1402, 2016.

[63] Hamid Reza Zohouri, Artur Podobas, and Satoshi Matsuoka. Combined spatial and temporal blocking for high-performance stencil computation on fpgas using opencl. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 153–162, 2018.

[64] Rie Soejima, Koji Okina, Keisuke Dohi, Yuichiro Shibata, and Kiyoshi Oguri. A memory profiling framework for stencil computation on an fpga accelerator with high level synthesis. *ACM SIGARCH Computer Architecture News*, 42(4):69–74, 2014.

[65] Shuo Wang and Yun Liang. A comprehensive framework for synthesizing stencil algorithms on fpgas using opencl model. In *Proceedings of the 54th Annual Design Automation Conference 2017*, pages 1–6, 2017.

[66] Kentaro Sano, Yoshiaki Hatsuda, and Satoru Yamamoto. Multi-fpga accelerator for scalable stencil computation with constant memory bandwidth. *IEEE Transactions on Parallel and Distributed Systems*, 25(3):695–705, 2013.

[67] Kane Yee. Numerical solution of initial boundary value problems involving maxwell's equations in isotropic media. *IEEE Transactions on antennas and propagation*, 14(3):302–307, 1966.

[68] Junnan Shan, Mihai T Lazarescu, Jordi Cortadella, Luciano Lavagno, and Mario R Casu. Cnn-on-aws: Efficient allocation of multikernel applications on multi-fpga platforms. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 40(2):301–314, 2020.

[69] Shakti K Davis, Barry D Van Veen, Susan C Hagness, and Frederick Kelcz. Breast tumor characterization based on ultrawideband microwave backscatter. *IEEE transactions on biomedical engineering*, 55(1):237–246, 2007.

[70] Elisa Ricci, Simone Di Domenico, Ernestina Cianca, Tommaso Rossi, and Marina Diomedi. Pca-based artifact removal algorithm for stroke detection using uwb radar imaging. *Medical & biological engineering & computing*, 55(6):909–921, 2017.

[71] Bárbara Oliveira, Martin Glavin, Edward Jones, Martin O'Halloran, and Raquel Conceição. Avoiding unnecessary breast biopsies: Clinically-informed 3d breast tumour models for microwave imaging applications. In *2014 IEEE Antennas and Propagation Society International Symposium (APSURSI)*, pages 1143–1144. IEEE, 2014.

[72] Branislav Gerazov and Raquel C Conceicao. Deep learning for tumour classification in homogeneous breast tissue in medical microwave imaging. In

IEEE EUROCON 2017-17th International Conference on Smart Technologies, pages 564–569. IEEE, 2017.

[73] Mustafa U Torun, Onur Yilmaz, and Ali N Akansu. Fpga, gpu, and cpu implementations of jacobi algorithm for eigenanalysis. *Journal of Parallel and Distributed Computing*, 96:172–180, 2016.

[74] Server Kasap and Soydan Redif. Novel field-programmable gate array architecture for computing the eigenvalue decomposition of para-hermitian polynomial matrices. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(3):522–536, 2013.

[75] Xinying Wang and Joseph Zambreno. An fpga implementation of the hestenes-jacobi algorithm for singular value decomposition. In *2014 IEEE International Parallel & Distributed Processing Symposium Workshops*, pages 220–227. IEEE, 2014.

[76] Shuiping Zhang, Xin Tian, Chengyi Xiong, Jinwen Tian, and Delie Ming. Fast implementation for the singular value and eigenvalue decomposition based on fpga. *Chinese Journal of Electronics*, 26(1):132–136, 2017.

[77] Yafeng Ma and Dong Wang. Accelerating svd computation on fpgas for dsp systems. In *2016 IEEE 13th International Conference on Signal Processing (ICSP)*, pages 487–490. IEEE, 2016.

[78] Yen-Liang Chen, Cheng-Zhou Zhan, Ting-Jyun Jheng, and An-Yeu Wu. Reconfigurable adaptive singular value decomposition engine design for high-throughput mimo-ofdm systems. *IEEE transactions on very large scale integration (VLSI) systems*, 21(4):747–760, 2012.

[79] Mrudula V Athi, Seyed Reza Zekavat, and Allan A Struthers. Real-time signal processing of massive sensor arrays via a parallel fast converging svd algorithm: Latency, throughput, and resource analysis. *IEEE Sensors Journal*, 16(8):2519–2526, 2016.

[80] Darshika G Perera and Kin Fun Li. Embedded hardware solution for principal component analysis. In *Proceedings of 2011 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, pages 730–735. IEEE, 2011.

[81] Daniel Fernandez, Carlos Gonzalez, Daniel Mozos, and Sebastian Lopez. Fpga implementation of the principal component analysis algorithm for dimensionality reduction of hyperspectral images. *Journal of Real-Time Image Processing*, 16(5):1395–1406, 2019.

[82] Abhishek Das, David Nguyen, Joseph Zambreno, Gokhan Memik, and Alok Choudhary. An fpga-based network intrusion detection architecture. *IEEE Transactions on Information Forensics and Security*, 3(1):118–132, 2008.

[83] Uday A Korat and Amirhossein Alimohammad. A reconfigurable hardware architecture for principal component analysis. *Circuits, Systems, and Signal Processing*, 38(5):2097–2113, 2019.

[84] Ernestina Martel, Raquel Lazcano, José López, Daniel Madroñal, Rubén Salvador, Sebastián López, Eduardo Juarez, Raúl Guerra, César Sanz, and Roberto Sarmiento. Implementation of the principal component analysis onto high-performance computer facilities for hyperspectral dimensionality reduction: Results and comparisons. *Remote Sensing*, 10(6):864, 2018.

[85] Mohammad Amir Mansoori and Mario R Casu. Efficient fpga implementation of pca algorithm for large data using high level synthesis. In *2019 15th Conference on Ph. D Research in Microelectronics and Electronics (PRIME)*, pages 65–68. IEEE, 2019.

[86] Mohammad Amir Mansoori and Mario R Casu. Hls-based flexible hardware accelerator for pca algorithm on a low-cost zynq soc. In *2019 IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC)*, pages 1–7. IEEE, 2019.

[87] D. Manolakis and G. Shaw. Detection algorithms for hyperspectral imaging applications. *IEEE Signal Processing Magazine*, 19(1):29–43, 2002.

[88] James Demmel and Krešimir Veselić. Jacobi's method is more accurate than qr. *SIAM Journal on Matrix Analysis and Applications*, 13(4):1204–1245, 1992.

[89] Larisa Beilina, Evgenii Karchevskii, and Mikhail Karchevskii. *Numerical linear algebra: Theory and applications*. Springer, 2017.

[90] Jair Cervantes, Farid Garcia-Lamont, Lisbeth Rodríguez-Mazahua, and Asdrubal Lopez. A comprehensive survey on support vector machine classification: Applications, challenges and trends. *Neurocomputing*, 408:189–215, 2020.

[91] Marco Salucci, Alessandro Polo, and Jan Vrba. Multi-step learning-by-examples strategy for real-time brain stroke microwave scattering data inversion. *Electronics*, 10(1):95, 2021.

[92] Lei Guo and Amin Abbosh. Stroke localization and classification using microwave tomography with k-means clustering and support vector machine. *Bioelectromagnetics*, 39(4):312–324, 2018.

[93] Shereen Afifi, Hamid GholamHosseini, and Roopak Sinha. Fpga implementations of svm classifiers: a review. *SN Computer Science*, 1(3):1–17, 2020.

[94] Christos Kyrkou and Theocharis Theocharides. A parallel hardware architecture for real-time object detection with support vector machines. *IEEE Transactions on Computers*, 61(6):831–842, 2011.

[95] Christos Kyrkou, Christos-Savvas Bouganis, Theocharis Theocharides, and Marios M Polycarpou. Embedded hardware-efficient real-time classification with cascade support vector machines. *IEEE transactions on neural networks and learning systems*, 27(1):99–112, 2015.

[96] Murad Qasaimeh, Assim Sagahyroon, and Tamer Shanableh. Fpga-based parallel hardware architecture for real-time image classification. *IEEE Transactions on Computational Imaging*, 1(1):56–70, 2015.

[97] Sumeet Saurav, Ravi Saini, and Sanjay Singh. Fpga based implementation of linear svm for facial expression classification. In *2018 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pages 766–773. IEEE, 2018.

[98] Yiyue Jiang, Kushal Virupakshappa, and Erdal Oruklu. Fpga implementation of a support vector machine classifier for ultrasonic flaw detection. In *2017 IEEE 60th international midwest symposium on circuits and systems (MWSCAS)*, pages 180–183. IEEE, 2017.

[99] Liu Han, Zhao Yue, and Xie Guo. Image segmentation implementation based on fpga and svm. In *2017 3rd International Conference on Control, Automation and Robotics (ICCAR)*, pages 405–409. IEEE, 2017.

[100] Marta Ruiz-Llata, Guillermo Guarnizo, and Mar Yébenes-Calvino. Fpga implementation of a support vector machine for classification and regression. In *The 2010 International Joint Conference on Neural Networks (IJCNN)*, pages 1–5, 2010.

[101] Abelardo Baez, Himar Fabelo, Samuel Ortega, Giordana Florimbi, Emanuele Torti, Abian Hernandez, Francesco Leporati, Giovanni Danese, Gustavo M Callico, and Roberto Sarmiento. High-level synthesis of multiclass svm using code refactoring to classify brain cancer from hyperspectral images. *Electronics*, 8(12):1494, 2019.

[102] Renato Campos and Joao MP Cardoso. On data parallelism code restructuring for hls targeting fpgas. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 144–151. IEEE, 2021.

[103] Valeria Mariano, Jorge A. Tobon Vasquez, Mario R. Casu, and Francesca Vipiana. Model-based data generation for support vector machine stroke classification. In *2021 IEEE International Symposium on Antennas and Propagation and USNC-URSI Radio Science Meeting (APS/URSI)*, pages 1685–1686, 2021.

[104] Yi-Wei Chen, Qingquan Song, and Xia Hu. Techniques for automated machine learning. *ACM SIGKDD Explorations Newsletter*, 22(2):35–50, 2021.

[105] Atefeh Sohrabizadeh, Cody Hao Yu, Min Gao, and Jason Cong. Autodse: Enabling software programmers to design efficient fpga accelerators. *ACM*

Transactions on Design Automation of Electronic Systems (TODAES), 27(4):1–27, 2022.

[106] Atefeh Mehrabi, Aninda Manocha, Benjamin C Lee, and Daniel J Sorin. Bayesian optimization for efficient accelerator synthesis. *ACM Transactions on Architecture and Code Optimization (TACO)*, 18(1):1–25, 2020.

[107] Yunyang Xiong, Ronak Mehta, and Vikas Singh. Resource constrained neural network architecture search: Will a submodularity assumption help? In *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 1901–1910, 2019.

[108] Sheng-Chun Kao, Geonhwa Jeong, and Tushar Krishna. Confuciux: Autonomous hardware resource assignment for dnn accelerators using reinforcement learning. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 622–636, 2020.

[109] J. Ney, D. Loroch, V. Rybalkin, N. Weber, J. Kruger, and N. Wehn. Half: Holistic auto machine learning for fpgas. In *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*, pages 363–368, Los Alamitos, CA, USA, sep 2021. IEEE Computer Society.

[110] Hayeon Lee, Sewoong Lee, Song Chong, and Sung Ju Hwang. Hardware-adaptive efficient latency prediction for NAS via meta-learning. In A. Beygelzimer, Y. Dauphin, P. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, 2021.

[111] Md Shahriar Iqbal, Jianhai Su, Lars Kotthoff, and Pooyan Jamshidi. Flexibo: Cost-aware multi-objective optimization of deep neural networks. *CoRR*, abs/2001.06588, 2020.

[112] Lile Cai, Anne-Maelle Barneche, Arthur Herbout, Chuan Sheng Foo, Jie Lin, Vijay Ramaseshan Chandrasekhar, and Mohamed M. Sabry Aly. Tea-dnn: the quest for time-energy-accuracy co-optimized deep neural networks. In *2019 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pages 1–6, 2019.

[113] Yao Yang, Andrew Nam, Mohamad M. Nasr-Azadani, and Teresa Tung. Resource-aware pareto-optimal automated machine learning platform. *CoRR*, abs/2011.00073, 2020.

[114] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. Once-for-all: Train one network and specialize it for efficient deployment. In *International Conference on Learning Representations*, 2020.

[115] Song Han, Han Cai, Ligeng Zhu, Ji Lin, Kuan Wang, Zhijian Liu, and Yujun Lin. Design automation for efficient deep learning computing. *CoRR*, abs/1904.10616, 2019.

[116] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, and Quoc V. Le. Mnasnet: Platform-aware neural architecture search for mobile. *CoRR*, abs/1807.11626, 2018.

[117] Bichen Wu, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Peter Vajda, Yangqing Jia, and Kurt Keutzer. Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[118] Chi-Hung Hsu, Shu-Huan Chang, Da-Cheng Juan, Jia-Yu Pan, Yu-Ting Chen, Wei Wei, and Shih-Chieh Chang. MONAS: multi-objective neural architecture search using reinforcement learning. *CoRR*, abs/1806.10332, 2018.

[119] Xiangxiang Chu, Bo Zhang, Ruijun Xu, and Hailong Ma. Multi-objective reinforced evolution in mobile neural architecture search. *CoRR*, abs/1901.01074, 2019.

[120] Zhichao Lu, Ian Whalen, Vishnu Boddeti, Yashesh Dhebar, Kalyanmoy Deb, Erik Goodman, and Wolfgang Banzhaf. Nsga-net: Neural architecture search using multi-objective genetic algorithm. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '19, page 419–427, New York, NY, USA, 2019. Association for Computing Machinery.

[121] Hadjer Benmeziane, Kaoutar El Maghraoui, Hamza Ouarnoughi, Smaïl Niar, Martin Wistuba, and Naigang Wang. A comprehensive survey on hardware-aware neural architecture search. *CoRR*, abs/2101.09336, 2021.

[122] Lukas Sekanina. Neural architecture search and hardware accelerator co-search: A survey. *IEEE Access*, 9:151337–151362, 2021.

[123] Weiwen Jiang, Xinyi Zhang, Edwin Hsing-Mean Sha, Lei Yang, Qingfeng Zhuge, Yiyu Shi, and Jingtong Hu. Accuracy vs. efficiency: Achieving both through fpga-implementation aware neural architecture search. *CoRR*, abs/1901.11211, 2019.

[124] Weiwei Chen, Ying Wang, Shuang Yang, Chen Liu, and Lei Zhang. You only search once: A fast automation framework for single-stage dnn/accelerator co-design. In *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1283–1286, 2020.

[125] Mohamed S. Abdelfattah, Lukasz Dudziak, Thomas Chau, Royson Lee, Hyeji Kim, and Nicholas D. Lane. Best of both worlds: Automl codesign of a cnn and its hardware accelerator. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2020.

[126] Ye Yu, Yingmin Li, Shuai Che, Niraj K. Jha, and Weifeng Zhang. Software-defined design space exploration for an efficient dnn accelerator architecture. *IEEE Transactions on Computers*, 70(1):45–56, 2021.

[127] Philip Colangelo, Oren Segal, Alexander Speicher, and Martin Margala. Automl for multilayer perceptron and FPGA co-design. *CoRR*, abs/2009.06156, 2020.

[128] Han Cai, Ligeng Zhu, and Song Han. Proxylessnas: Direct neural architecture search on target task and hardware. *CoRR*, abs/1812.00332, 2018.

[129] Dimitrios Stamoulis, Ermao Cai, Da-Cheng Juan, and Diana Marculescu. Hyperpower: Power- and memory-constrained hyper-parameter optimization for neural networks. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 19–24, 2018.

[130] Edgar Liberis, Lukasz Dudziak, and Nicholas D. Lane. *μNAS: Constrained Neural Architecture Search for Microcontrollers*, page 70–79. Association for Computing Machinery, New York, NY, USA, 2021.

[131] Stewart Greenhill, Santu Rana, Sunil Gupta, Pratibha Vellanki, and Svetha Venkatesh. Bayesian optimization for adaptive experimental design: A review. *IEEE Access*, 8:13937–13948, 2020.

[132] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[133] Eduardo C. Garrido-Merchán and Daniel Hernández-Lobato. Predictive entropy search for multi-objective bayesian optimization with constraints. *Neurocomputing*, 361:50–68, 2019.

[134] Tyson Reimer, Jorge Sacristan, and Stephen Pistorius. Improving the diagnostic capability of microwave radar imaging systems using machine learning. In *2019 13th European Conference on Antennas and Propagation (EuCAP)*, pages 1–5. IEEE, 2019.

Appendix A

List of Acronyms

HLS High Level Synthesis

HDL Hardware Description Language

FPGA Field Programmable Gate Array

ML Machine Learning

MI Microwave Imaging

FDTD Finite Difference Time Domain

II Initiation Interval

PCA Principal Component Analysis

EVD Eigenvalue Decomposition

SVD Singular Value Decomposition

SVM Support Vector Machine

ANN Artificial Neural Network

DNN Deep Neural Network

CNN Convolutional Neural Network

MLP Multi-Layer Perceptron

CPML Convolutional Perfectly Mached Layer

BO Bayesian Optimization

NRE Non Recurrent Engineering

MRI Magnetic Resonance Imaging

VNA Vector Network Analyzer

HI Hyperspectral Imaging

TSVD Truncated Singular Value Decomposition

SLR Super Logic Region

PC Principal Component

MPPA Massively Parallel Processing Array

Cov Covariance

PU Projection Unit

PS Processing System

PL Parallel Logic

SoC System on Chip

SV Support Vector

RBF Radial Basis Function

MOBOC Multi Objective Bayesian Optimization with Constraints

DSA Domain Specific Accelerator

DSE Design Space Exploration

HW-NAS Hardware aware Neural Architecture Search

RL Reinforcement Learning

EA Evolutionary Algorithm

PESMOC Predictive Entropy Search for Multi-objective Optimization with Constraints