ELSYS

INSTITUT FÜR LEISTUNGSELEKTRONISCHE SYSTEME

Inference and Training of a Multilayer Perceptron in a Deep Reinforcement Learning Context on a FPGA

Thilo Wendt

TECHNISCHE HOCHSCHULE NÜRNBERG
INSTITUT ELSYS

www.th-nuernberg.de/elsys

Nuremberg Institute of Technology

Electrical Engineering, Precision Engineering and Information Technology

Master of Science
Applied Research in Engineering Sciences

Master's Thesis of
Thilo Wendt

Inference and Training of a Multilayer Perceptron in a Deep Reinforcement Learning Context on a FPGA

Summer Term 2022
Date of Submission: Mai 09, 2022

Supervisor
Prof. Dr. Armin Dietz

M. Sc. Tobias Schindler
Institute ELSYS

Keywords: FPGA, Artificial Neural Network, Batch Gradient Descent, High Level Synthesis

Notice: This declaration must be securely bound in all copies of the final thesis. (No spiral binding)

TECHNISCHE HOCHSCHULE NÜRNBERG
GEORG SIMON OHM

Declaration by the student in accord with examination rules and regulations

Personal information of the student:

Family name: Wendt Given name: Thilo Student ID number: 2921949

Faculty: Electrical Eng., Precision Eng., Information Tec Degree programme: Applied Research in Engineering Sciences

Semester: Sommersemester 2022

Title of the final thesis:

Inference and Training of a Multilayer Perceptron in a Deep Reinforcement Learning Context on a FPGA

I hereby declare that this work is my own and has not been submitted for examination purposes in any other context. I have indicated and acknowledged all sources, aids, and quotations used in its production.

Nuremberg, 05-09-2022

City, Date, Signature of the student

Declaration regarding the publication of the thesis named above

The decision to wholly or partially publish the thesis is, on principle, first and foremost the sole responsibility of the student author. According to the Copyright Act (UrhG), when a student produces a thesis, the author of the thesis acquires sole copyright and, in principle, also the resulting rights of use, such as first publication (§ 12 UrhG), distribution (§ 17 UrhG), reproduction (§ 16 UrhG), online use, etc., i.e., all rights pertaining to non-commercial or commercial exploitation.

The university and its employees will not publish theses or parts thereof without the agreement of the student author, in particular, it will not be placed in the publicly accessible section of the university library.

I hereby ☒ authorize, if and insofar as no conflicting agreements with third parties exist,

☐ do not authorize

that the above-named thesis may be made publicly accessible by the Technische Hochschule Nürnberg Georg Simon Ohm; if applicable, if indicated by an embargo annotation on the thesis, this will occur after the expiry of any publication embargo of

years (0 - 5 years after the date the thesis was submitted).

If authorization is granted, it is irrevocable; the thesis submission includes a pdf formatted version on a data storage medium for this purpose. Provisions in the respectively applicable Study and Examination Regulations about the type and scope of copies and materials that must be submitted as part of the thesis shall not be affected by the submission of the pdf format for publication.

Nuremberg, 05-09-2022

City, Date, Signature of the student

Print form

Data protection: The personal data you provide during your application is stored and processed by the Technische Hochschule Nürnberg Georg Simon Ohm. Further information on how the Technische Hochschule Nürnberg handles your personal data can be found at: https://www.th-nuernberg.de/datenschutz/

Abstract

This thesis addresses the design and verification of a multilayer perceptron (MLP) and the corresponding optimization algorithm, the batch gradient descent (BGD), on a FPGA using high level synthesis (HLS) for Xilinx devices. The solutions developed in this project are used in a reinforcement learning environment for the control of power electronic systems. The thesis briefly presents the principle of reinforcement learning, a mathematical description of the MLP and the BGD as well as programming techniques for HLS. The structure of the solutions and performance examinations are presented in the results part of the thesis. The project delivers functionally verified solutions for the execution on a FPGA. The solutions are able to process a three layer MLP with 16 inputs and outputs and 128 neurons in the hidden layer in 2,361 clock cycles at 100 MHz clock frequency which results in a runtime of 23.6 µs. The corresponding BGD for one training example features a minimum runtime of 13,141 clock cycles or 1,314 µs. However, performance is expected to further improve after resolving several issues described in the thesis.

Table of Contents

Nomenclature

Abbreviations

AI	Artificial Intelligence
ANN	Artificial Neural Network
AXI	Advanced eXtensible Interface
BGD	Batch Gradient Descent
BLAS	Basic Linear Algebra Subprograms
BRAM	Block RAM
CNN	Convolutional Neural Network
DDR	Double Data Rate RAM
DMA	Direct Memory Access
DPU	Deep Learning Processing Unit
DQN	Deep Q-Network
FIFO	First-In-First-Out (Buffer)
FPGA	Field Programmable Gate Array
GD	Gradient Descent
GPIO	General Purpose IO
GPU	Graphic Processing Unit
HDL	Hardware Description Language
HLS	High Level Synthesis
II	Initiation Interval
ILA	Integrated Logic Analyzer
IO	Input Output
IP	Intellectual Property
LUT	Look-Up Table
MLP	Multilayer Perceptron
PES	Power Electronic System
PIPO	Ping-Pong (Buffer)
RAM	Random Access Memory
ReLU	Linear Rectifier

RL	Reinforcement Learning
RTL	Register Transfer Level
SoC	System-on-a-Chip
TLF	Top Level Function
VHDL	Very High Speed Integrated Circuit HDL

Symbols for MLP and BGD

δ^l	Vector that holds the error of the layer l
δ^l_j	Error of the neuron j in the layer l
η	Learning rate
$\sigma^l(z)$	Activation function of the layer l
a^L	Output vector of the MLP
a^l	Vector that holds the output values of the layer l
a^l_j	Output value of the neuron j in the layer l
b	All bias vectors in the network
b^l	Vector that holds the biases of the layer l
b^l_j	Bias of the neuron j in the layer l
C	Cost function over the complete dataset
C_x	Contribution of one example to the cost function
L	Number of layers in the network
M	Number of examples in the training batch X
m	Index of the of the example in the batch X
N	Total number of examples in a dataset
n	Index of the of the example in the complete dataset
W	All weight matrices in the network
W^l	Weights between the the layers $l-1$ and l
$W^l_{j,i}$	Weight between the neuron a^{l-1}_i and a^l_j
X	Batch of training examples
x^n	Input vector that holds the values of the example n
$y(x)$	Vector that holds the desired output of the MLP for the input x
z^l_j	Input value of σ^l in the neuron a^l_j

Symbols in Reinforcement Learning

$\mathcal{A}$	Action space of the agent
$\mathcal{S}$	State space of the environment
π	Policy from which A is derived
π_*	Optimal policy to maximize the value
A	Action of the agent
$Q(S, A)$	Approximation of $q(S, A)$
$q(S, A)$	Value of the action A taken in the state S a.k.a. action-value function
R	Reward of the environment given to the agent
S	State of the environment
v	Accumulated reward a.k.a. value
$v(S)$	Value of a state

1 Introduction

Artificial intelligence (AI) algorithms are currently widely used to solve a broad range of problems in the field of power electronic systems (PES). AI can be applied in different phases of the product life-cycle including design, control and maintenance [1, p. 4633]. Within the scope of this project, the inference and the training of an artificial neural network (ANN) in a reinforcement learning (RL) approach for the control of a PES are investigated. In this context, the inference of the ANN is the computation with given operation parameters, while the training is referred to as the determination of optimal parameters for the application. The control algorithm is implemented on a field programmable gate array (FPGA) based rapid prototyping platform which provides the computational resources required to meet the hard real-time conditions, for which the target system demands [2].
The implementation on an embedded system with hard real-time requirements distinguishes the application from classic AI domains like computer vision, audio processing or big data analysis [3, p. 9]. Usually, these tasks run in unconstrained environments (e.g. in cloud environments backed by data centers) without fulfilling real-time requirements in the range of microseconds. In embedded system environments, a higher level of optimization concerning the design of the algorithm as well as the implementation on hardware is required. However, the application of AI in power electronics seems promising in situations where traditional control approaches fail in the consideration of non-linear effects [4, p. 27]. The consideration of non-linear effects in PES leads to optimized control algorithms which increase the efficiency of the system. Especially in space or weight constraint environments it is vital to operate the system close to the physical limits [5, p. 1].
The project is based on the outcomes presented in [2]. The paper describes the inference of multilayer perceptrons (MLP) on a FPGA. The application for the MLP is the approximation of the Q-Function in a RL environment which is further described in section 2.1. The author of the paper utilized the Matlab HDL Coder to generate a hardware description from a Simulink model which is suitable for the execution on a FPGA. While the implementation details of the different MLPs have been investigated extensively, the paper does not describe the training phase in an embedded environment. Furthermore, a verification of the generated hardware description has not been performed.
As a first step, the results from [2] are reproduced using the high level synthesis (HLS) tool from Xilinx. In contrast to the Simulink based approach, HLS allows the behavioral description in C++, which is then implemented in a hardware description language (HDL) like VHDL. The outcome can be manipulated for different design goals such as execution time, resource or power efficiency [6, p. 17]. Furthermore, the high level description of the algorithm is used to create a test bench to verify the behavior of the generated hardware description. The creation of the test bench is mainly covered by HLS. In contrast to a manual implementation in VHDL, this approach takes advantage of the implicit solution of HLS without the demand for extensive knowledge about hardware verification.

As a second step, the training of the MLP using HLS is investigated. The implementation focuses the maximum achievable execution speed on the considered FPGA. For this purpose, a high level description in C++ of the training algorithm is created which is suitable for the post processing by HLS. The training of the MLP is carried out with batch gradient descent (BGD).
The target platform is the UltraZohm which is a system specifically designed for the rapid prototyping of new control algorithms in the domain of PES. It is based on a Xilinx Zynq UltraScale+ SoC FPGA. For the application in PES, an ecosystem of interface boards for the acquisition of actual values and the output of switching signals to an inverter is available [7]. The outcome of the thesis is a contribution to the UltraZohm project, which is licensed under the open source Apache 2 license.

1.1 Related Work

The inference of neural networks on an embedded system is feasible nowadays due to the increased performance of the platforms. While implementations on graphic processing units (GPU) have been popular in the past, further optimizations in terms of power efficiency and throughput are currently examined. Xilinx offers deep learning processing units (DPU) specifically designed for the inference of convolutional neural networks (CNN) on various FPGA-based platforms. The solution operates with well-known frameworks like Caffe or TensorFlow [8]. While this solution seems promising for sophisticated problems that are subject to offline optimized CNNs, the combination with online training appears to be infeasible within the scope of this thesis. A lower level of complexity is offered by tools like HLS4ML that do not rely on a DPU. The tool infers a HLS description from common open source machine learning package models [9] and it has been successfully applied to an image processing problem in [10]. Adaptions of HLS4ML officially supported by Xilinx exist [11]. The potential of HLS4ML concerning real-time capabilities has been proven in [10] but a model of the ANN from one of the aforementioned AI frameworks is required. The vast majority of all solutions target the inference of CNNs on FPGAs which is suitable for complex tasks like image processing. In contrast, the ANNs described in [2] are much simpler and can be described in C++ effortlessly. While the usage of tools like HLS4ML is useful for complex ML applications with offline training, it just adds another layer of complexity to the problem investigated in this thesis.
In addition, very few research has been done on the online training of a MLP as it is required in a RL application. The training method used in this project is BGD. This algorithm has been directly implemented in HDL in [12], [13] and [14] but due to the high verification effort that comes with the realization in HDL, these solutions are not considered for the current project. In fact, a solution similar to [15] and [16], which take advantage of the implicit verification support of HLS, is desired. In both papers the algorithm is well described but no repository with the implementation is publicly available. Thus, the publications as well as the structures generated by HLS4ML are used as a starting point for the current project but they do not offer a solution.

1.2 Project Structure

The thesis is subdivided into two main work packages. As a first step, the inference of a MLP on a FPGA with HLS is examined. The theoretical basis for the implementation is given in section 2.2, while the implementation with HLS is explained in section 3.2. The outcome of the first work package forms the basis for the implementation of the BGD which is carried out in the second work package. The equations for the BGD implemented in section 3.3 are explained in section 2.3. [15] and [16] are considered for the second work package but these projects do not deliver a solution. The implementation of the MLP and the BGD follow the optimization guidelines presented in section 2.5.
The functional verification of the MLP and the BGD is presented in section 3.6. Consequently, performance benchmarking is presented in section 3.7. Final considerations are given in chapter 4.

Within the scope of this project, the following aspects have been investigated: [2] proposes the usage of the hard wired processor, which is available on the control platform. While the current project does not focus on the investigation of this approach, it is estimated if the solution designed with HLS is suitable for the execution on a general purpose processor without major code changes. The following research questions have been identified:

1. Which architecture for the implementation of a MLP in HLS leads to high performance while maintaining a reasonable resource footprint?
2. Is the implementation of the BGD algorithm feasible with HLS?
3. Which minimal execution time of a training run can be reached with the given hardware platform disregarding the usage of FPGA resources?
4. Are the HLS implementations suitable for execution on a processor without major code changes?

2 Theory

The following chapter presents the theoretical basis of the implementations described in chapter 3. This chapter introduces RL and embeds the role of the ANN in the context. A mathematical overview of MLPs and the corresponding BGD is given in section 2.2 and 2.3. Finally, section 2.5 presents the programming techniques for HLS applied in this project.

2.1 Reinforcement Learning

The ANN implemented in the current project is applied to a RL approach to control PES. In the following section, the context is explained, in which the ANN is used. The formal description follows the notation introduced in [17]. Fig. 2.1 shows the composition of the components in RL which are introduced in the following paragraphs.

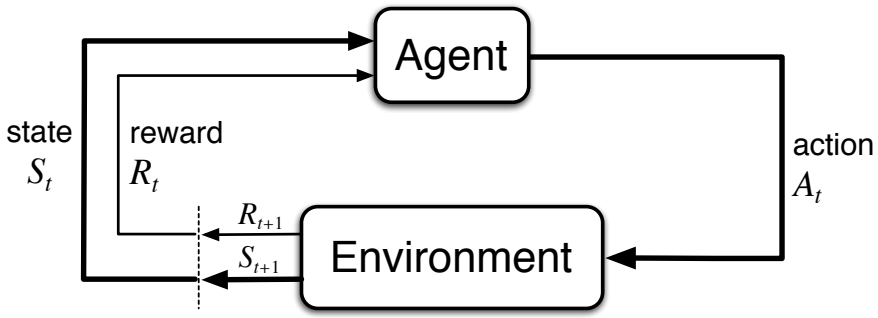

Figure 2.1: Overview of the components in RL from [17, p. 54]. The agent influences the environment by taking an action A. As a consequence, the environment transitions to the state S and feeds back a reward R to the agent.

Environment The environment in RL is the system which is scope to the control task [17, p. 53]. In the context of PES, the environment may be an electric power train composed of an inverter and a drive [18]. The designer of the RL selects a quantity available in the environment which defines the reward R to be fed back to the agent. The agent can be provided with a model of the environment in advance, which is considered as a model based method, or it infers the model during the learning process (model-free method) [17, p. 9].

State The state S describes the constitution of the environment, to which the agent has read access. In the concrete example of [18] the state of the environment is described by the compensation current and the rotor angle of the drive.

Agent The agent is the learner and the decision maker. It interacts with the environment by taking actions A and by sensing the feedback. In a PES context, the agent is the control unit that drives the inverter. The agent can take actions from a constraint action space. A is based on a

policy π which is a mapping of actions to states. Thus, π defines the probability of the agent to take an action in a specific state. Deterministic policies can be described as a function or a look-up table (LUT) but in general, polices are stochastic [17, p. 7]. The policy is optimized based on the reward the agent experiences after taking an action, which influences the environment [17, p. 53]. After the execution of A, the environment transitions to another state and the reward is fed back to the agent. The goal of the agent is the maximization of the reward [17, p. 7].

Reward The reward R is the feedback formed by the environment and it is represented as a single numeric value. It is the direct consequence of the action taken by the agent in a specific state of the environment and it defines the goal of the optimization [17, p. 7]. For instance, in [18] the reward is defined as the root mean square of the speed error signal of the drive, which is scope to the optimization performed in the investigation.

Value While the reward represents the direct feedback from the environment, the value is the expected accumulated amount of reward starting from a specific state and thereafter following the policy π. Thus, it assesses the behavior of the agent in the long run. The value is expressed depending on the starting state S by the value function $v(S)$. In order to solve the optimization problem which is scope to the RL, a common method is the value estimation of the states and the consequent reward maximization by operating the environment in the states with the highest value. This is considerably more effortful than the reward optimization since the value of a state is determined over several iterations, whereas the reward for an action can be derived from the model directly [17, p. 8].
The value of a state is expressed by the value function $v_\pi(S)$. It outputs the value of a state under the condition that the agent follows the policy π. While $v_\pi(S)$ maps a value to a state, the function $q_\pi(S, A)$ represents the value of taking the action A in a given state S and thereafter following the policy π. It is referred to as the *action-value function for the policy* π [17, p. 70]. If it is not possible to describe these functions analytically, a suitable solution for finite problems with a small state space $\mathcal{S}$ and action space $\mathcal{A}$ is the definition with LUTs [17, p. 10-15]. Larger values for $\mathcal{S}$ and $\mathcal{A}$ result in a LUT size that is not manageable in a real-world application [2]. Instead, the action-value function is approximated. In deep reinforcement learning applications, the resulting function $Q(S, A)$ is described by a neural network which takes the state of the environment as an input and outputs the Q-value for all available actions in the current state. The agent can follow the policy π by choosing the action with the highest Q-value. The ANN that approximates $q(S, A)$ to $Q(S, A)$ is referred to as a Deep Q-Network (DQN). The DQN is altered depending on the reward the agent experiences from the environment during the training phase. With a sufficiently high number of training runs, the policy that is derived from $Q(S, A)$ converges to the optimal policy π_* [17, p. 13]. Within the scope of this project, the feasibility of online optimization of the DQN during operation is examined offering the possibility to react to a changing environment.

2.2 Multilayer Perceptrons

In a general case, the ANN explained in section 2.1 can have an arbitrary architecture, which is well suited to implement $Q(s, a)$. Since the implementation presented in the current report needs to fulfill real-time requirements a simple architecture with low latency on the control platform is desirable. The simplest architecture of an ANN is the MLP. The following section describes the structure of a MLP, the corresponding terminology and a matrix based notation for the computation of a MLP.
Fig. 2.2 shows a common representation of a MLP. It consists of an arbitrary number of layers L, an input layer a^1 and an output layer a^L. The layers between the input and the output are known as hidden layers. Each layer contains an arbitrary number of neurons which are stored in the vector a^l. The output of a neuron a^l_j is computed as follows [19]:

1. Summation of the output of the neurons of the previous layer a^{l-1}_i multiplied with the corresponding weight $W^l_{j,i}$.
2. Addition of a bias to the weighted sum from step 1. This intermediate quantity is identified by the symbol z.
3. Application of the activation function σ to z. The output of the activation function is the output of neuron a.

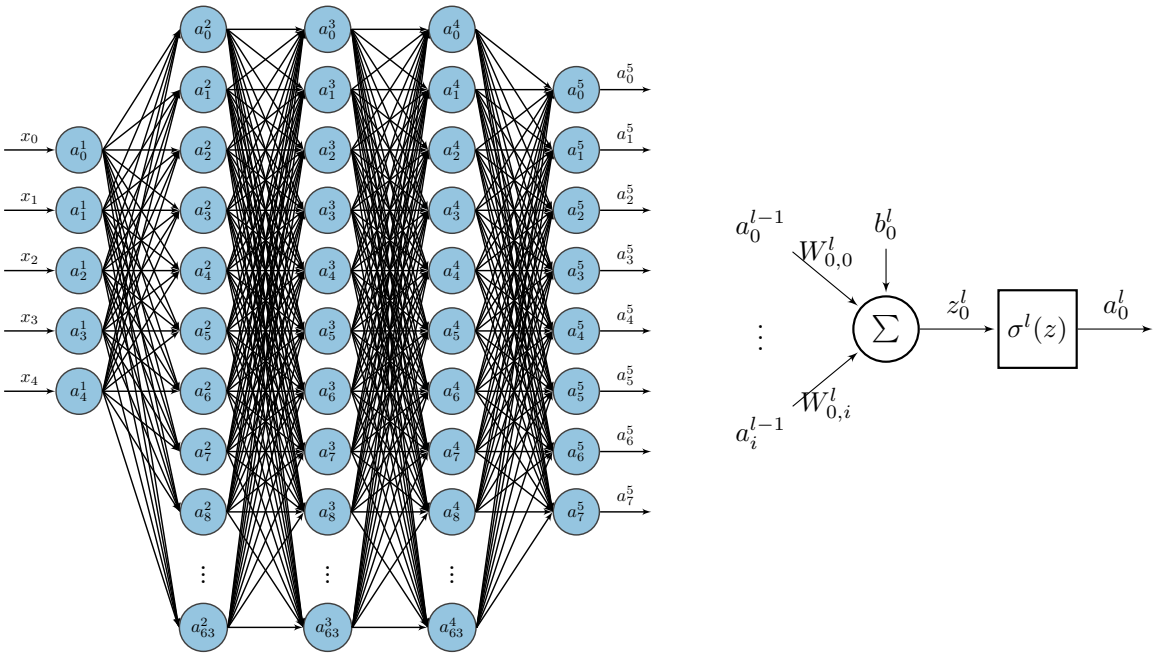

Figure 2.2: Graphic illustration of a MLP. All neurons of a layer are linked to the next layer with weighted connections (left), which results in a weighted sum followed by the activation function of the layer σ^l (right). The figures are based on illustrations from [2].

The parameters of a MLP can be summarized in a matrix representation allowing the description of the computation in a compact equation. Equation (2.1) describes the computation of z for a single neuron as the dot product between the row j of W^l and the addition of the corresponding bias value b_j^l.

$$z_j^l = b_j^l + \sum_i W_{j,i} \cdot a_i^{l-1} = W_j^l \cdot a^{l-1} + b_j^l \tag{2.1}$$

This approach is applied to the whole vector z^l by computing the product of the vector a^{l-1} and the matrix W^l and finally adding the vector b^l:

$$z^l = W^l a^{l-1} + b^l \tag{2.2}$$

The output vector of the layer a^l is obtained by applying the activation function σ^l to z^l:

$$a^l = \sigma^l(z^l) = \sigma^l(W^l a^{l-1} + b^l) \tag{2.3}$$

The representation from (2.3) is the basis for the implementation of the MLP. Within the scope of this report, the following activation functions are considered:

$$\sigma(z) = \text{sigmoid}(z) = \frac{1}{1+e^{-z}} = \frac{e^z}{1+e^z} \tag{2.4}$$

$$\sigma(z) = \text{ReLU}(z) = \begin{cases} z, & \text{if } z > 0 \\ 0, & \text{if } z \leq 0 \end{cases} \tag{2.5}$$

$$\sigma(z) = z \tag{2.6}$$

2.3 Batch Gradient Descent

In the current project, the gradient descent (GD) algorithm is the method to optimize the parameters of the MLP. The following section contains a brief description of the GD algorithm which is based on [19]. For a thorough introduction refer to this document.
In the literature, the algorithm is also known as *backpropagation*. Within the scope of this thesis, the term *backpropagation* is avoided due to its fuzzy meaning. The term only denotes that something is propagated backwards through an ANN while leaving the applied methods undefined. On the other hand, the term *batch gradient descent* distinctively defines the applied measures: First, the *gradient* of the cost function in regard to every parameter in the MLP is computed. Consequently, it is used to reduce the value of the cost function which is described by the term *descent*. In order to make this method applicable to large or infinite training datasets, it is applied to a *batch* of training examples and not to the complete dataset.
The flowchart in Fig. 2.3 illustrates the optimization process which is applied in this project. At the beginning of the optimization, the parameters are initialized randomly. During the optimization, the parameters are altered by the BGD algorithm.

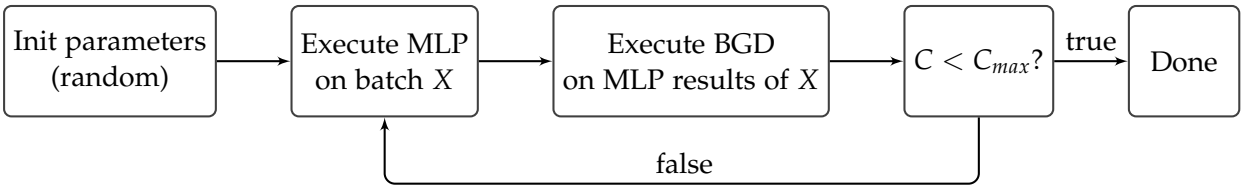

Figure 2.3: Flowchart of the optimization process with BGD. The goal of the optimization is the descent of the cost function C below a threshold C_{max}. This is achieved by iterating over the dataset with the MLP and altering the parameters with the BGD.

The goal of the optimization is the minimization of the cost function C. Within the scope of this project, the cost function is the mean squared error of the output of the MLP. Equation (2.7) shows the cost function for a complete dataset [19, eq. (26)].

$$C = \frac{1}{2N} \cdot \sum_{n=0}^{N-1} |y(x^n) - a^L(x^n)|^2 = \frac{1}{N} \cdot \sum_{n=0}^{N-1} C_x(x^n) \tag{2.7}$$

The cost function C depends on the desired output y and the actual output a^L of the MLP for a given training example x^n. In (2.7), N denotes the total number of training examples available in the dataset, while C_x is the contribution of a single training example to the value of the cost function C.
In order to alter the parameters of the MLP, a value that quantifies the contribution of every single weight and bias to the cost function in the network is required. This quantity is the partial derivative of the cost function in regard to every weight and bias in the network $\partial C_x / \partial w$ and $\partial C_x / \partial b$. The entity of all partial derivatives of the MLP is referred to as the gradient. Equation (2.8) shows the update of the parameters of the MLP [19, eq. (20)] [19, eq. (21)]. Metaphorically speaking, the gradient quantifies the positive contribution to the cost function. To minimize C, the gradient is subtracted from the current parameters. This process is scaled by the learning rate η.

$$
\begin{aligned}
W \to W' &= W - \eta \cdot \frac{1}{N} \cdot \sum_{n=0}^{N-1} \frac{\partial C_x(x^n)}{\partial W} \\
b \to b' &= b - \eta \cdot \frac{1}{N} \cdot \sum_{n=0}^{N-1} \frac{\partial C_x(x^n)}{\partial b}
\end{aligned}
\tag{2.8}
$$

Equation (2.8) depicts a major disadvantage of plain GD: Before updating the parameters, the partial derivatives for *all training samples in the dataset* must be computed. In case of large datasets or datasets with an unknown size, it is inconvenient to apply GD in this manner. Instead, GD is applied to a selection of training samples from the complete dataset. This method is known as batch gradient descent or stochastic batch gradient descent if the training samples are selected randomly. Assuming a large enough batch size M, the partial derivatives of a training batch X approximately equal the values for the complete dataset [19, eq. (18)]:

$$
\begin{aligned}
\frac{1}{M} \cdot \sum_{m=0}^{M-1} \frac{\partial C_x(x^m)}{\partial W} &\approx \frac{1}{N} \cdot \sum_{n=0}^{N-1} \frac{\partial C_x(x^n)}{\partial W} \\
\frac{1}{M} \cdot \sum_{m=0}^{M-1} \frac{\partial C_x(x^m)}{\partial b} &\approx \frac{1}{N} \cdot \sum_{n=0}^{N-1} \frac{\partial C_x(x^n)}{\partial b}
\end{aligned}
\tag{2.9}
$$

Consequently, $\partial C_x/\partial w$ and $\partial C_x/\partial b$ must be computed for each individual training sample in the training batch X. As a first step, the intermediate quantity δ^l is defined [19, eq. (29)]:

$$
\delta_j^l = \frac{\partial C_x}{\partial z_j^l} = \frac{\partial C_x}{\partial a_j^l} \cdot \sigma'(z_j^l) \tag{2.10}
$$

δ_j^l is known as the error of the neuron j in the layer l. It denotes, how much the output of a neuron a_j^l contributes to the cost function. Equation (2.10) is valid for all layers in the MLP but the quantity $\partial C_x/\partial a_j^l$ for the output layer is computed differently than for the hidden layers as shown in (2.13) and (2.15)
It is convenient to define δ^l in regard to the input of the activation z^l because the actual quantities of interest $\partial C_x/\partial w$ and $\partial C_x/\partial b$ are directly derived from it without the occurrence of the activation function σ. Metaphorically speaking, δ^l is computed by calculating the output error of a neuron $\partial C_x/\partial a_j^l$ and moving this value backwards through the activation function by multiplying it with its derivative. For hidden layers, δ^l is computed by propagating δ^{l+1} backwards through the network. Consequently, an expression for the error of the output layer must be defined first. Equation (2.10) demands for a partial derivative of C_x in regard to every element in the output vector a^l. Equation (2.11) shows the relation between a^l and C_x, which has been extracted from (2.7).

$$
C_x = \frac{1}{2} \cdot |y - a^L|^2 = \frac{1}{2} \cdot \sum_j (y_j - a_j^L)^2 \tag{2.11}
$$

The partial derivative of (2.11) with respect to every element of the output vector a_j^L is determined with the chain rule:

$$\frac{\partial C_x}{\partial a_j^L} = \frac{\partial}{\partial a_j^L}\left(\frac{1}{2}\cdot\sum_j (y_j - a_j^L)^2\right) = -1\cdot 2\cdot\frac{1}{2}\cdot(y_j - a_j^L) = a_j^L - y_j \tag{2.12}$$

Summarizing the results of (2.12) in the vector $\partial C_x/\partial a^L$ directly leads to the expression for the output error vector δ^L [19, eq. (30)]:

$$\begin{aligned}\frac{\partial C_x}{\partial a^L} &= a^L - y \\ \delta^L = \frac{\partial C_x}{\partial a^L}\odot\sigma'(z^L) &= (a^L - y)\odot\sigma'(z^L)\end{aligned} \tag{2.13}$$

The operator $\odot$ simply denotes the elementwise multiplication of two vectors which is known as the *Hadamard Product*. For the special case of a linear activation function $\sigma(z^L) = z^L$, the derivative of it is equal to one. This special case applies to the output layers of all MLPs considered in this project and δ^L is further simplified to (2.14):

$$\delta^L = a^L - y \tag{2.14}$$

Based on δ^L, the error is propagated backwards through the MLP by multiplying δ^L with the transposed weight matrix which results in the vector $\partial C_x/\partial a^l$. As introduced in (2.10), this vector is multiplied with the derivative of the activation function to obtain the error of the previous layer. Equation (2.15) delivers the general expression for the computation of the error in hidden layers [19, eq. (BP2)].

$$\begin{aligned}\frac{\partial C_x}{\partial a^l} &= (W^{l+1})^T\delta^{l+1} \\ \delta^l = \frac{\partial C_x}{\partial a^l}\odot\sigma'(z^l) &= ((W^{l+1})^T\delta^{l+1})\odot\sigma'(z^l)\end{aligned} \tag{2.15}$$

Equation (2.16) [19, eq. (BP4)] and (2.17) [19, eq. (BP3)] display the computation of the quantity of interest which is further used to update the parameters of the MLP as shown in (2.8). While the vector $\partial C_x/\partial b^l$ is simply equal to the error of the corresponding layer, a row of $\partial C_x/\partial W^l$ is computed by multiplying every value in the output vector of the previous layer a^{l-1} with one entry of δ^l. This is repeated for every entry in δ^l. After the iteration, the number of rows of the matrix $\partial C_x/\partial W^l$ equals the number of entries in the vector δ^l and the number of columns equals the number of entries in a^{l-1}. The matrix has the same dimensions as W^l.

$$\frac{\partial C_x}{\partial W_{j,k}^l} = a_k^{l-1}\cdot\delta_j^l \tag{2.16}$$

$$\frac{\partial C_x}{\partial b^l} = \delta^l \tag{2.17}$$

Regarding memory efficiency in the implementation, it is desirable to express $\sigma'(z^l)$ as a function of a^l. This approach avoids the buffering of z^l, while a^l is required anyways to compute (2.16). The simplification for the ReLU function is straightforward since it is partially a linear function, which outputs $\sigma(z) = z = a$ for positive input values:

$$\frac{d}{dz}\sigma(z) = \frac{d}{dz}\text{ReLU}(z) = \begin{cases} 1, & \text{if } a > 0 \\ 0, & \text{if } a \leq 0 \end{cases} \tag{2.18}$$

The derivative of the sigmoid function can be expressed as a function of sigmoid itself. This avoids the buffering of z^l because a^l is equal to $\sigma^l(z^l)$.

$$\begin{aligned} \frac{d}{dz}\sigma(z) &= \frac{d}{dz}\left(\frac{e^z}{1+e^z}\right) = \frac{e^z \cdot (1+e^z) - e^z \cdot e^z}{(1+e^z)^2} = \frac{e^z}{(1+e^z)^2} \\ &= \sigma(z) \cdot (1 - \sigma(z)) \\ &= a \cdot (1-a) \end{aligned} \tag{2.19}$$

In summary, (2.14), (2.15), (2.16) and (2.17) need to be implemented to compute the partial derivatives of the MLP. To update the parameters, (2.8) needs to be implemented for a training batch X. With the simplifications from (2.18) and (2.19), only the output vectors of the hidden layers a^l are required.

2.4 The XOR Problem

In the following section, the equations introduced in section 2.2 are illustrated with the XOR problem. It is a simple classification task, which is used to verify the basic functionality of the MLP and the corresponding training algorithm. In section 3.6, the XOR problem is solved with the solution developed in this project and the outcome is compared to the results from [16]. The goal of the optimization is a MLP that behaves like an XOR gate which is achieved by determining appropriate weights and biases using BGD. Fig. 2.4 displays a simple three layer MLP that is used to solve the XOR problem in the following example. The activation function for the hidden layer is sigmoid and linear for the output layer. The desired behavior is described in Tab. 2.1.

Table 2.1: Truth table of the desired function. n is a unique identifier for each sample, the vector x is the input of the MLP and the vector y is the *desired* output of the MLP. The right most column contains the boolean outputs of a XOR gate.

n	x_0	x_1	y_0	y_1	$XOR(x)$
0	0	0	0	1	false
1	0	1	1	0	true
2	1	0	1	0	true
3	1	1	0	1	false

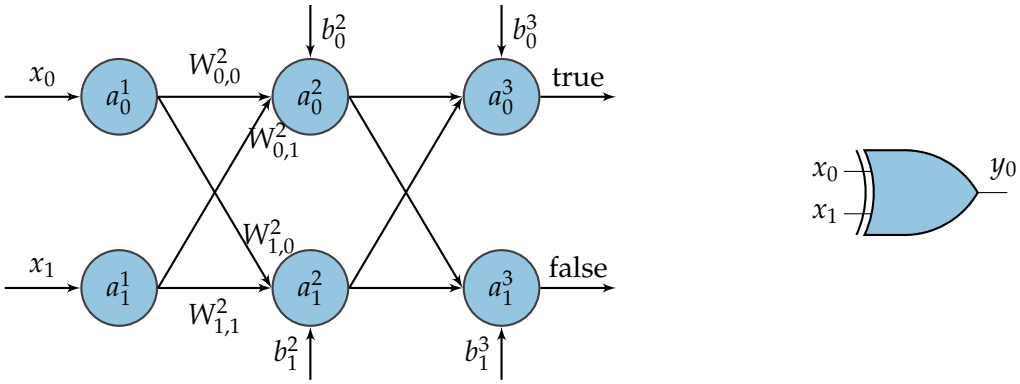

Figure 2.4: MLP to solve the XOR problem. The network consists of three layers with two neurons in each layer. The layers are connected by the weight matrices W^2 and W^3. The network shall behave like the XOR gate shown on the right side of the figure.

The following equations demonstrate that the MLP behaves as expected. The simplicity of the example allows the manual computation of the result. Nevertheless, the principle applies to more sophisticated problems but only the number and the size of the matrices are changed. The parameters of the MLP from Fig. 2.4 are set to the following values which have been determined with the BGD introduced in section 2.3:

$$W^2 = \begin{bmatrix} -2.02 & 2.42 \\ -1.66 & 1.65 \end{bmatrix} \qquad b^2 = \begin{bmatrix} 1.33 \\ -1.14 \end{bmatrix}$$

$$W^3 = \begin{bmatrix} -1.53 & 1.71 \\ 1.16 & -1.23 \end{bmatrix} \qquad b^3 = \begin{bmatrix} 1.12 \\ 0.0155 \end{bmatrix}$$

$$\sigma^2(z) = \frac{1}{1+e^{-z}} = \text{sigmoid}(z)$$

The MLP is expected to output *false* for the input values $x(n=0)$ since *XOR(0,0)* is equal to *false*. Regarding the output vector a^3 of the MLP, the maximum value is interpreted as the prediction of the MLP. In the current example, a_1^3 is greater than a_0^3. Since a_1^3 represents the value *false* the prediction of the MLP is *false* which matches the expected behavior. The execution of the MLP with the other input values produces similar results for the rest of the truth table.

$$a^1 = x(n=0) = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \qquad a^2 = \sigma(W^2 a^1 + b^2) \qquad a^3 = W^3 a^2 + b^3$$

$$y(n=0) = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \qquad = \begin{bmatrix} 0.791 \\ 0.242 \end{bmatrix} \qquad = \begin{bmatrix} -0.796 \\ 0.62 \end{bmatrix}$$

2.5 Programming for Vitis High Level Synthesis

The goal of this project is to implement the equations introduced in section 2.2 and 2.3 on a FPGA. Since the target platform is a Zynq UltraScale+ SoC FPGA from Xilinx the implementation relies on the Xilinx toolchain. Nevertheless, similar considerations can be applied to FPGAs from other vendors. Xilinx offers the Vitis HLS tool, which generates VHDL or Verilog code from a C++ description of the desired behavior. In comparison to traditional digital hardware design with direct implementation in VHDL or Verilog, HLS offers a higher level of abstraction and the design can be optimized for different constraints without major changes to the code base [6, p. 6].

2.5.1 Vitis HLS Workflow

Vitis HLS proposes a certain workflow for successful hardware development. Fig. 2.5 shows the proposed workflow. The first step is the description of the desired behavior with a subset of C++ that can be further processed by Vitis HLS. The tool introduces major limitations in comparison to the full feature set of C++ which are described in detail in [6, p. 95f.]. In order to verify the desired behavior, a proven software implementation is used to produce reference results. Within the scope of this report, this software implementation is referred to as the *reference implementation*. In the C-simulation step of the HLS workflow, the C++ description to be processed to the register transfer level (RTL) is tested against the reference implementation purely in software.
The second step is the generation of VHDL or Verilog code on RTL. Vitis HLS assesses the given C++ code for suitability for RTL implementation. If the code is eligible, a RTL description is generated. This process can be monitored by comprehensive log messages and a report about the estimated timing and resource characteristics of the solution. FPGA specific characteristics of the desired solution are given to the tool by `#pragma` directives or in global settings for the solution. Within the scope of this project, the synthesis settings are given to HLS by `#pragma` directives exclusively.
After the generation of the RTL description, Vitis HLS generates a test bench, where the RTL description is tested against the reference implementation. This step, which is known as *hardware software co-simulation*, allows a precise estimation of the runtime on hardware since different scenarios can be simulated and monitored by the reports created by Vitis HLS. The reports contain cycle accurate runtime information about the created solution.
If the co-simulation finishes successfully, the solution is exported to a packaged IP-core which can be inserted into a Vivado block design. Thereafter the deployment flow follows the normal procedure of synthesis and implementation of a Vivado block design [20, p. 185f.]. Besides the RTL implementation of the IP-core, Vitis HLS provides driver code in C which can be used to develop the software driver to control the IP-core. A comprehensive reference about the generated driver is available in [6, p. 408f.].

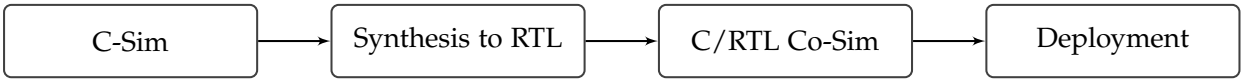

Figure 2.5: Flowchart of the Vitis HLS workflow based on [6, p. 6].

2.5.2 Definition of the IP-Core in HLS

Vitis HLS synthesizes a C++ function to RTL. Since this function incorporates the complete algorithm it is known as the top level function (TLF). Vitis HLS generates two types of interface protocols to control the IP-core. The block-level IO protocols monitor the state of the IP-core, whereas port-level IO protocols ship data to it. Tab. 2.2 shows the signals of the `ap_ctrl_hs` protocol. Concerning data transfer via port-level IO protocols, Vitis HLS offers several options [6, p. 177f.]:

- Data transfer via an AXI4 master, an AXI4-Lite slave or AXI4-Stream
- Data transfer via simple wire ports optionally with a handshake mechanism
- Memory interface protocol for direct interaction with block RAM (BRAM)

In this project, the control signals from Tab. 2.2 are given to the IP-core via an AXI GPIO, which is connected to the wire ports of the `ap_ctrl_hs` block-level protocol. A detailed description of the test setup is available in Appendix B. Configuration data is shipped directly via AXI4-Lite and ports for bulk data transfer are implemented as an AXI4 master. The AXI4 protocol is convenient to use in comparison to wire or memory interface ports because it is a memory mapped protocol. As a consequence, the AXI4-Lite of the IP-core can be mapped in the address space of a processor in the SoC and the AXI4 master can access the global memory space of the device.
The implementation from listing 2.1 serves as an example for the implementation of IP-cores with Vitis HLS. The hardware interface definition is explained in the current section whereas performance optimization is covered by section 2.5.3. Independent from the aforementioned symbols in MLPs and BGD, the example implements the following equation, in which all operands are vectors of the same size:

$$d = a \odot b + c \tag{2.20}$$

The `#pragmas` in line 5 to 10 define the hardware interface of the IP-core. Line 5 to 8 map the function arguments mentioned at the `port` option of the directive to AXI4 master ports which are named after the `bundle` option of the directive. The `offset = slave` option denotes that the address, on which the AXI4 master operates, is given via an AXI4-Lite slave. The `depth` option is important for proper behavior during co-simulation and it must be set to the number of transfers between the IP-core and the memory. It instructs the HLS tool to allocate an appropriately sized memory region for the operands in the RTL test bench. Setting this value too small or much too high usually causes a segmentation fault during co-simulation. Line 9 and 10 map the signals from Tab. 2.2 to the AXI4-Lite port which is also used to set the address offset of the AXI master ports. Furthermore, the directives instruct Vitis HLS to generate corresponding driver functions to access the signals. The signals from Tab. 2.2 are available because Vitis HLS is instructed to use the `ap_ctrl_hs` block-level protocol by the directive in line 10. If the block-level protocol is mapped to an AXI4-Lite port, it is not available as wire ports at the hardware interface of the IP-core. In summary, an IP-core with the AXI4 master ports `m_axi_read` and `m_axi_write` and the AXI4-Lite slave port `s_axi_control` is synthesized by Vitis HLS. The computation itself takes place in line 12 to 17. A major limitation of HLS is demonstrated by this example: the size of internal buffers `l_a`, `l_b`, `l_c` and `l_d` must be known before synthesis because dynamic memory

allocation is not possible on a FPGA. Apart from that, the implementation takes advantage of burst transfers by using the `memcpy()` function.

Table 2.2: Signals of the `ap_ctrl_hs` interface [6, p. 182].

Signal	Feature
ap_start	Set to high to start the IP-core
ap_idle	IP-core is idling if ap_idle is high
ap_ready	IP-core is ready to receive new data if ap_ready is high
ap_done	Becomes high for one clock cycle when the IP-core finishes the operation

```
1  #include <string.h> // required for memcpy()
2  #define ARRAY_SIZE 16
3
4  void multAdd(float *a, float *b, float *c, float *d){
5      #pragma HLS INTERFACE m_axi port = a offset = slave bundle = read depth =
       ↪  ARRAY_SIZE
6      #pragma HLS INTERFACE m_axi port = b offset = slave bundle = read depth =
       ↪  ARRAY_SIZE
7      #pragma HLS INTERFACE m_axi port = c offset = slave bundle = read depth =
       ↪  ARRAY_SIZE
8      #pragma HLS INTERFACE m_axi port = d offset = slave bundle = write depth =
       ↪  ARRAY_SIZE
9      #pragma HLS INTERFACE s_axilite port = return
10     #pragma HLS INTERFACE ap_ctrl_hs port = return
11
12     float l_a[ARRAY_SIZE], l_b[ARRAY_SIZE], l_c[ARRAY_SIZE], l_d[ARRAY_SIZE];
13     memcpy(l_a, a, ARRAY_SIZE * sizeof(float));
14     memcpy(l_b, b, ARRAY_SIZE * sizeof(float));
15     memcpy(l_c, c, ARRAY_SIZE * sizeof(float));
16     for(size_t i = 0; i < ARRAY_SIZE; i++){
17         l_d[i] = l_a[i] * l_b[i] + l_c[i];
18     }
19     memcpy(d, l_d, ARRAY_SIZE * sizeof(float));
20 }
```

Listing 2.1: Basic example of a HLS implementation with optimization potential. The directives in line 5 to 8 instruct the HLS tool to implement the function arguments as AXI with a read and a write port. The block-level protocol `ap_ctrl_hs` is mapped to a AXI4-Lite in line 9 to 10. The rest of the listing shows the communication with the main memory and the algorithm to be optimized.

2.5.3 Optimization for FPGA

The example from listing 2.1 shows a fully sequential implementation that does not leverage the parallel computing capabilities of the FPGA. [21] introduces several measures to increase resource utilization on FPGAs using HLS. The IP-cores developed in this project make use of loop pipelining and vectorization to enable parallelism on the operational level. A dataflow architecture is applied to enable task level parallelism.

Pipelining

Pipelining on an operational level allows the concurrent execution of operations featuring a latency of more than one clock cycle. In the following example, a latency L_{op} of four clock cycles for the computation of one element of the vector d is assumed. Fig. 2.6 shows a comparison between the non-pipelined design from listing 2.1 and a pipelined solution as in listing 2.2. The figure shows the desired behavior of a pipeline with an initiation interval (II) of one. The II denotes the number of clock cycles that must pass before the pipeline can accept new input values. The total latency of a pipelined design is given by equation 2.21, where N denotes the number of values to be processed [21]. In the given example, N is equal to `ARRAY_SIZE`.

$$L_{tot} = L_{op} + II \cdot (N - 1) \tag{2.21}$$

Equation (2.21) illustrates that the II must be as small as possible to achieve low latency. In a perfect pipeline, the II is equal to one but interface contention or loop-carried dependencies may prevent Vitis HLS from achieving this goal. While loop-carried dependencies do not cause significant problems in this project, interface contention due to insufficient memory bandwidth is the main performance bottleneck. Interface contention arises if a memory resource is unable to serve a request from the computational logic [21]. A typical scenario for this problem is the usage of vectorization with insufficient memory bandwidth.

```
1 for(size_t i = 0; i < ARRAY_SIZE; i++){
2 #pragma HLS PIPELINE II = 1
3     l_d[i] = l_a[i] * l_b[i] + l_c[i];
4 }
```

Listing 2.2: By inserting the directive in line 2, the loop is pipelined by Vitis HLS as presented in Fig. 2.6.

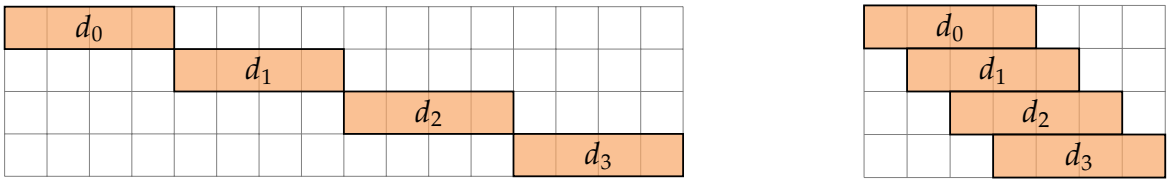

Figure 2.6: Visualization of loop pipelining. The diagram on the left side shows a non-pipelined implementation like listing 2.1. Inserting a pipeline directive like in listing 2.2 into the computation leads to an implementation with a timing shown in the diagram on the right side [6, p. 398].

Vectorization

While pipelining increases the utilization of a single compute unit, vectorization is applied to scale an algorithm over multiple compute units. The *#pragma HLS UNROLL* directive instructs HLS to execute a loop fully in parallel. As shown in listing 2.3, the iteration space of the outer loop is folded by the the degree of parallelism (line 3). The directive in line 7 instructs HLS to compute `parEntries` of the vector `l_d` concurrently. From a hardware perspective, `parEntries` compute units that implement $d_n = a_n \cdot b_n + c_n$ are instantiated, which results in a timing illustrated by Fig. 2.7. However, unrolling is only applicable if the memory resource holding the vectors `l_a`, `l_b`, `l_c` and `l_d` provide sufficient memory bandwidth to concurrently read and write `parEntries` operants. Insufficient memory bandwidth leads to interface contention and a pipeline with an II greater than one. HLS provides the directive *#pragma HLS ARRAY_PARTITION* and *#pragma HLS ARRAY_RESHAPE* to increase memory bandwidth of a given array [6, p. 366], but it is still a limited resource. Therefore, vectorization cannot scale infinitely and other measures to increase performance must be considered [21].

```
1  constexpr unsigned int parEntries = 4;
2  ...
3  for(size_t i = 0; i < ARRAY_SIZE / parEntries; i++){
4  #pragma HLS PIPELINE II = 1
5      size_t offset = i * parEntries;
6      for(size_t j = 0; j < parEntries; j++){
7  #pragma HLS UNROLL
8          l_d[i] = l_a[offset + j] * l_b[offset + j] + l_c[offset + j];
9      }
10 }
11 ...
```

Listing 2.3: Instructing Vitis HLS to apply vectorization using loop unrolling. The iteration space is folded by the degree of parallelism in line 3 and the inner loop is executed fully in parallel by inserting the directive in line 7. If sufficient memory bandwidth is provided, the implementation results in a timing shown in Fig. 2.7.

Figure 2.7: Visualization of vectorization. In contrast to pure pipelining, several values are computed in parallel. Successful vectorization requires sufficient memory bandwidth of the buffer that holds the operands.

Dataflow

While vectorization and pipelining provide optimizations on an operational level, dataflow enables task level pipelining [6, p. 374]. Listing 2.4 shows a dataflow implementation of the example from listing 2.1. For dataflow optimization, the algorithm is divided into functions, which are connected by streams. In a dataflow region, the producer-consumer paradigm must be respected, which denotes that a stream of data must be produced and consumed at the same rate. In the given example, the multiplication unit must produce a new value every clock cycle, while the adder unit must consume the values at the same rate. If this balance is disrupted, the pipeline stalls and the design does not operate at maximum performance. To relax the situation, the depth of the stream can be adjusted with the directive `#pragma HLS STREAM depth=<int>` but this only applies to applications where the producer works in a non-continuous batch mode and the consumer can drain the pipeline. However, this measure does not decrease the total latency because the compute unit with the highest II dictates the performance of the whole design [21].
While streams are generally realized as ping-pong buffers (PIPO) in RAM, the usage of the `hls::stream` class from Vitis HLS enforces the implementation as a first-in-first-out buffer (FIFO). In contrast to a PIPO, a FIFO is accessed sequentially only which results in a more efficient implementation. A value written to a FIFO by the producer traverses the complete FIFO before the consumer reads it once [6, p. 486]. In the current example, the algorithm is split up in a multiplication and an adder unit which are optimized with pipelining and vectorization on the operational level.
Xilinx provides a collection of HLS functions and constructs that are optimized for Xilinx devices which are referred to as the Vitis libraries [6, p. 32]. Within the scope of this report, the basic linear algebra (BLAS) functions of the libraries are used. Besides the computational building blocks, which are designed for a dataflow implementation, the Vitis libraries offer functions to move data from memory to streams and vice versa. Vectorization is provided by the `WideType` template class, which is configured with the datatype of the values and the number of entries in the vector as shown in line 5 of listing 2.4. The current example makes use of `WideType` and the corresponding functions to interact with memory. The computational building blocks provided by the Vitis libraries are connected as demonstrated in line 32 to 39 of listing 2.4.
The dataflow implementation may seem cumbersome in the current example but it enables a modular software design in HLS with building blocks that are efficiently connected. The algorithm is split up into small pieces which are optimized using a divide and conquer approach. Dataflow architectures enable design reuse and enforce an efficient implementation [21].

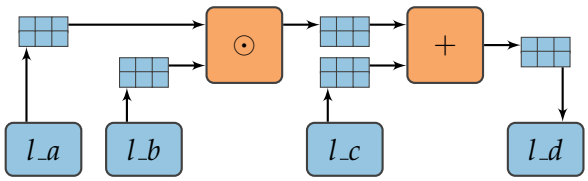

Figure 2.8: Structure of the dataflow implementation.

Synchronization of dataflows

In the case of the algorithms to be implemented in this project, it is necessary to synchronize a dataflow because the computation depends on previously calculated results. This dependency is visible in (2.3) and (2.15), where the computation relies on the results of the previously processed layer. The results are buffered in a local BRAM array but Vitis HLS does not allow such a buffer to be the destination and source of a dataflow. Therefore, the content from the destination buffer is copied to a new source buffer which forces the dataflow function to finish before the next execution starts. Indeed, this leads to a sequential execution of the algorithm but due to the dependencies explained above, the algorithm cannot be further parallelized.

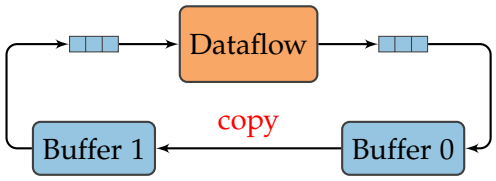

Figure 2.9: Synchronization mechanism between dataflow functions. The dataflow function is fully drained into the destination buffer 0, whereafter its content is copied to the source buffer 1. Thereafter, the dataflow function can be executed again with buffer 1 as source.

Further optimizations

The usage of vectorization by simply widening the data paths and processing more elements per clock cycle draws on memory bandwidth. In comparison to computational resources and memory size, the bandwidth to move data is a very limited resource in FPGAs. As mentioned above, pure vectorization, which is also known as *horizontal unrolling*, can not scale infinitely. Vitis HLS only offers the directives `#pragma HLS UNROLL` to apply horizontal unrolling together with `#pragma HLS ARRAY_PARTITION` and `#pragma HLS ARRAY_RESHAPE` to increase memory bandwidth. In order to further increase parallelism, these techniques must be combined with *vertical unrolling*. By vertically unrolling an algorithm, memory bandwidth is used economically by only fetching data once and thereafter reusing it throughout the whole algorithm. While horizontal unrolling draws on rather limited memory bandwidth between processing elements, vertical unrolling stores data in local caches at the expense of widely available buffer space [21]. Within the scope of this thesis, these optimizations have been used indirectly through the Vitis libraries [22].

```
1  #include "hls_stream.h"
2  #include "Vitis_Libraries/blas/L1/include/hw/xf_blas.hpp"
3  #define ARRAY_SIZE 16
4  constexpr size_t parEntries = 4;
5  using Vector_t = xf::blas::WideType<float, parEntries>;
6  using VectorStream_t = hls::stream<Vector_t::t_TypeInt>;
7
8  void multiply(VectorStream_t& a, VectorStream_t& b,
9                VectorStream_t& result, size_t n){
10     for(size_t i = 0; i < n / parEntries; i++){
11 #pragma HLS PIPELINE II = 1
12         Vector_t l_a, l_b, l_result;
13         l_a = a.read(); l_b = b.read();
14         for(size_t j = 0; j < parEntries; j++){
15 #pragma HLS UNROLL
16             l_result[j] = l_a[j] * l_b[j];}
17         result.write(l_result);}
18 }
19 void add(VectorStream_t& a, VectorStream_t& b,
20          VectorStream_t& result, size_t n){
21     for(size_t i = 0; i < n / parEntries; i++){
22 #pragma HLS PIPELINE II = 1
23         Vector_t l_a, l_b, l_result;
24         l_a = a.read(); l_b = b.read();
25         for(size_t j = 0; j < parEntries; j++){
26 #pragma HLS UNROLL
27             l_result[j] = l_a[j] + l_b[j];}
28         result.write(l_result);}
29 }
30 void multAdd(float *a, float *b, float *c, float *d){
31     ...
32 #pragma HLS DATAFLOW
33     VectorStream_t l_strA, l_strB, l_strC, l_strMult, l_strD;
34     xf::blas::readVec2Stream<float, parEntries>(l_a, ARRAY_SIZE, l_strA);
35     xf::blas::readVec2Stream<float, parEntries>(l_b, ARRAY_SIZE, l_strB);
36     multiply(l_strA, l_strB, l_strMult, ARRAY_SIZE);
37     xf::blas::readVec2Stream<float, parEntries>(l_c, ARRAY_SIZE, l_strC);
38     add(l_strMult, l_strC, l_strD, ARRAY_SIZE);
39     xf::blas::writeStream2Vec<float, parEntries>(l_strD, ARRAY_SIZE, l_d);
40     ...
41 }
```

Listing 2.4: Fully optimized Vitis HLS implementation. The functions `add(...)` and `multiply(...)` are optimized with unrolling and pipelining on the operational level. Using the directive in line 32, the functions are pipelined on a task level.

3 Results

The following chapter presents the implementation and verification of the IP-cores developed in this project. Section 3.1 describes constraints that apply to the MLP as well as the BGD. Consequently, the implementation of both IP-cores and shortcomings of the solutions are described in section 3.2 to 3.5. Finally, the outcomes of the functional verification and the performance examinations are presented in section 3.6 and 3.7. Fig. 3.1 summarizes the solution that has been developed within this project. A detailed communication example between the components is presented in the sequence diagram of Fig. 3.5.

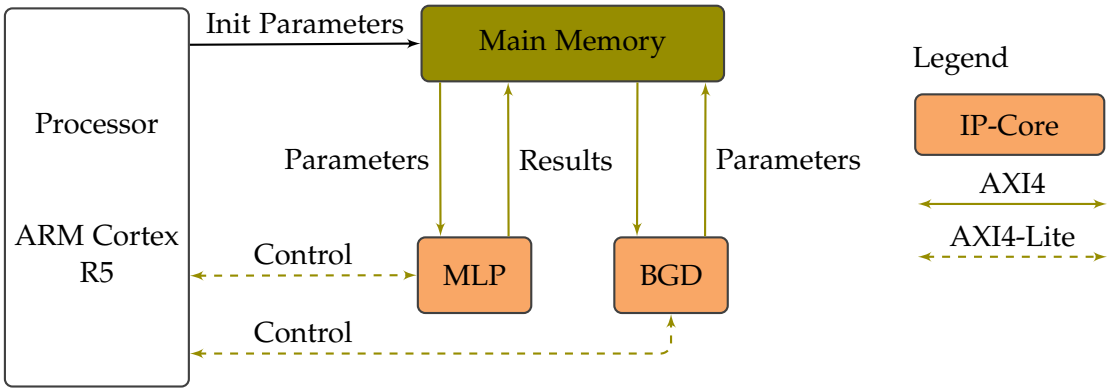

Figure 3.1: Composition of the MLP and the BGD with the main memory and the processor. The term *parameters* summarizes the entity of all weights and biases, whereas the term *results* describes the results of all layers of the MLP.

3.1 Implementation Constraints

The MLP and the BGD algorithm are implemented as a dataflow architecture with vectorization as described in section 2.5.3. The degree of parallelism is set by the variable `ParEntries` in the header file `Settings.hpp`. In comparison to a pure software version like [23], the current implementation introduces a lower degree of flexibility:

- The maximum size of the MLP is constrained by the amount of BRAM allocated at synthesis time.
- Every hidden layer has the same number of neurons, which is adjustable during runtime.
- The degree of parallelism `ParEntries` is adjusted before synthesis.
- $\log_2(\texttt{ParEntries})$ must be an integer greater or equal to zero.
- The number of neurons in every layer must be an integer multiple of the degree of parallelism. This also applies to the input and output layer.
- Every hidden layer has the same activation function which, is adjustable before synthesis.
- The output layer has a linear activation.

The possibility of decoupling the number of inputs and outputs from the degree of parallelism has been investigated. Due to dependencies between the layers in the BGD and unsuccessful co-simulation of the MLP, the independency of these design parameters is not possible.
Despite the above mentioned constraints, the implementation offers a greater degree of flexibility than [2]. Respecting the limitations introduced by the vectorization and assuming sufficient BRAM resources, the following parameters are adjustable during runtime:

- The number of input neurons.
- The number of output neurons.
- The number of hidden layers, which must be always greater or equal to one.
- The batch size of the BGD.
- The learning rate of the BGD.
- The initial transfer of the parameters controlled by `loadParameters`.
- The export behavior of the intermediate results in the MLP controlled by `exportLayers`.

3.2 Implementation of the MLP

Fig. 3.2 shows the implementation of the MLP as a dataflow architecture. First, the weight matrices W and the bias vectors b are transferred to the internal BRAM of the IP-core via AXI. Similar to the example from section 2.5 the internal BRAM is declared as an array of a fixed size. By declaring this array as `static`, it is persistent over multiple executions of the IP-core [6, p. 116]. Therefore, the transfer of the MLP parameters only needs to be done initially or if the parameters have been altered by the BGD kernel.
After transferring the MLP parameters as well as the input values to the BGD, the function `processLayer()` is executed for every hidden layer. Referring to (2.3), the computation of the MLP is split up in the matrix vector processing and the application σ^l to z^l. The Vitis BLAS library provides a template function for matrix vector multiplication in a dataflow environment:

```
template <typename t_DataType, unsigned int t_LogParEntries>
void gemv(
    const unsigned int m,
    const unsigned int n,
    const t_DataType alpha,
    hls::stream<typename WideType<t_DataType, (1 <<
     ↪  t_LogParEntries)>::t_TypeInt>& M,
    hls::stream<typename WideType<t_DataType, (1 <<
     ↪  t_LogParEntries)>::t_TypeInt>& x,
    const t_DataType beta,
    hls::stream<typename WideType<t_DataType, 1>::t_TypeInt>& y,
    hls::stream<typename WideType<t_DataType, 1>::t_TypeInt>& yr)
```

The function computes the following equation:

$$y_r = \alpha(M \cdot x) + \beta \cdot y; \tag{3.1}$$

With $y_r = z^l$, $M = W^l$, $x = a^{l-1}$, $y = b^l$ and $\alpha = \beta = 1$, the function is used to compute the matrix vector multiplication and the addition of the bias. Consequently, σ^l is applied to z^l to compute a^l. Transfer functions between BRAM and streams are provided by the Vitis BLAS library. Since the output layer has a linear activation the function `outputLayer()` omits σ^l and outputs directly the result of `gemv()`. The functions `processLayer()` and `outputLayer()` implement a dataflow but since the computation of the following layer depends on the results of the previous layer a^{l-1} the executions are synchronized as illustrated by Fig. 2.9. After processing all layers of the network, the results are transferred back to the DDR via AXI. The IP-core offers the parameters `loadParameters` and `exportLayers` that are adjustable during runtime. The option `loadParameters` determines if the weights and biases are transferred to the internal BRAM before the computation of the MLP whereas `exportLayers` controls whether all intermediate results from all layers or only the output layer are transferred to the DDR. As explained in section 3.3, all intermediate results are required to execute the BGD algorithm.

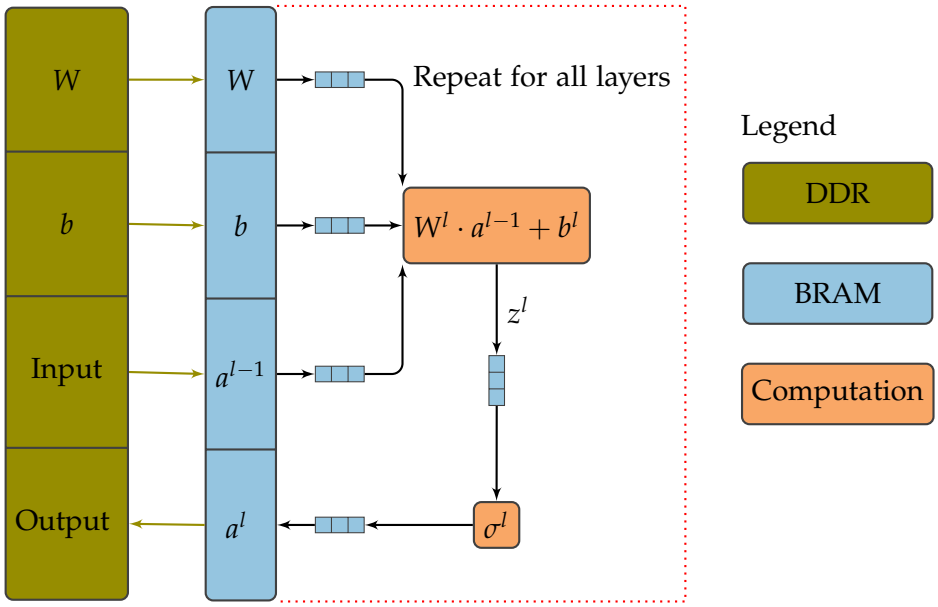

Figure 3.2: Block diagram of the HLS implementation of the MLP. The algorithm is split up in the matrix vector processing and the application of σ^l to z^l. The compute units are connected with streams. The red boundary describes the function `processLayer()` which computes the output of one layer.

3.3 Implementation of the BGD

Fig. 3.3 illustrates the implementation of the BGD algorithm in HLS. The algorithm works on a training batch X which is processed by the MLP before. Prior to the actual algorithm, the desired behavior y_X and the results from the MLP a_X of the whole training batch are transferred to the IP-core via AXI. Similar to the MLP, the BGD IP-core offers a runtime adjustable parameter `loadParameters` which controls whether W and b are transferred prior to the execution of the algorithm. The algorithm is split up in two phases:

1. Computation of the averaged partial derivatives $\sum \frac{\partial C}{\partial W}$ and $\sum \frac{\partial C}{\partial b}$
2. Update of the MLP parameters

To determine the partial derivatives of a layer l, the error δ^l is computed first. δ is described by (2.14) and (2.15). Both equations are relatively simple to implement in HLS. For the multiplication $(W^{l+1})^T \delta^{l+1}$ the function `gemv()` from the Vitis libraries is used. The transposition of W^{l+1} is performed by the custom transfer function `gem2StreamTranspose()` that reads a matrix from BRAM to a stream in a transposed fashion. This avoids the buffering of a transposed version of the weight matrices.
While the computation of δ fulfills the producer-consumer paradigm, a bottleneck situation is present at the computation of the matrix $\partial C / \partial W^l$ which is expressed by (2.16). Assuming that the number of iterations of the loop in line 4 of listing 3.1 `p_n / t_ParEntries` is greater than one, the stream `p_delta` is not read every clock cycle i.e. the II is greater than one. Supposing a reliable producer, the stream between the computation of δ and $\partial C / \partial W^l$ is written more frequent than it is read which results in a stalling pipeline. The situation is illustrated by Fig. 3.4: In this case, the reading rate of the compute unit calculating $\partial C / \partial W^l$ is only half of the writing rate of the producer of δ^l. As a result, the input stream fils up faster than it is drained by the consumer and the design stalls. Stalling of the producer can be prevented by increasing the stream depth but the overall latency is not reduced because the II of the consumer is not decreased. The situation worsens with increasing size of the vector a^{l-1}. In order to optimize the situation, a consumer that operates at the same rate as the producer is required. The issue has been identified in this project but remains to be solved in a further revision. The development of a better solution was not possible within the timeline of the thesis.

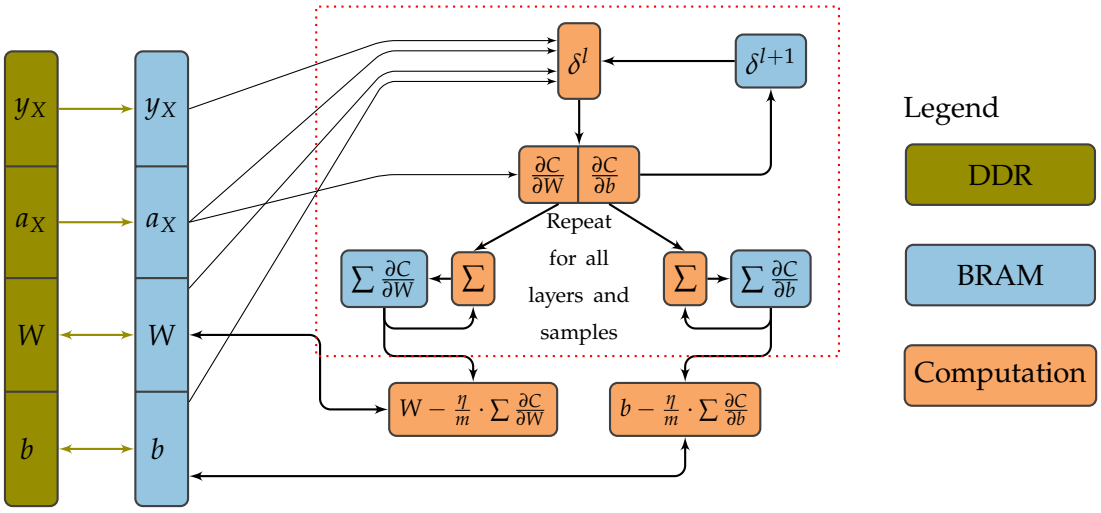

Figure 3.3: Block diagram of the HLS implementation of the BGD. The algorithm works on a batch X of multiple samples which are given by the MLP. The sequence in the red boundary is executed for every sample in the batch. The BGD implements a dataflow like in Fig. 3.2 but in favor of simplicity, the streams between the computing elements are omitted.

```
for (size_t m = 0; m < p_m; m++){
    xf::blas::WideType<t_DataType, 1> l_delta = p_delta.read();
    p_biasGradient.write(l_delta);
    for (size_t n = 0; n < p_n / t_ParEntries; n++){
#pragma HLS PIPELINE
        xf::blas::WideType<t_DataType, t_ParEntries> l_outputPrev =
         ↪ p_outputPrev.read();
        xf::blas::WideType<t_DataType, t_ParEntries> l_weightGradient;
        for (size_t j = 0; j < t_ParEntries; j++){
#pragma HLS UNROLL
            l_weightGradient[j] = l_outputPrev[j] * l_delta[0];
        }
        p_weightGradient.write(l_weightGradient);
    }
}
```

Listing 3.1: HLS implementation of the computation of the gradients. `p_m` and `p_n` denote the number of rows and columns in the matrix $\partial C/\partial W^l$ whereas `t_ParEntries` denotes the degree of parallelism. `p_delta`, `p_outputPrev`, `p_biasGradient` and `p_weightGradient` are `hls::stream` function arguments.

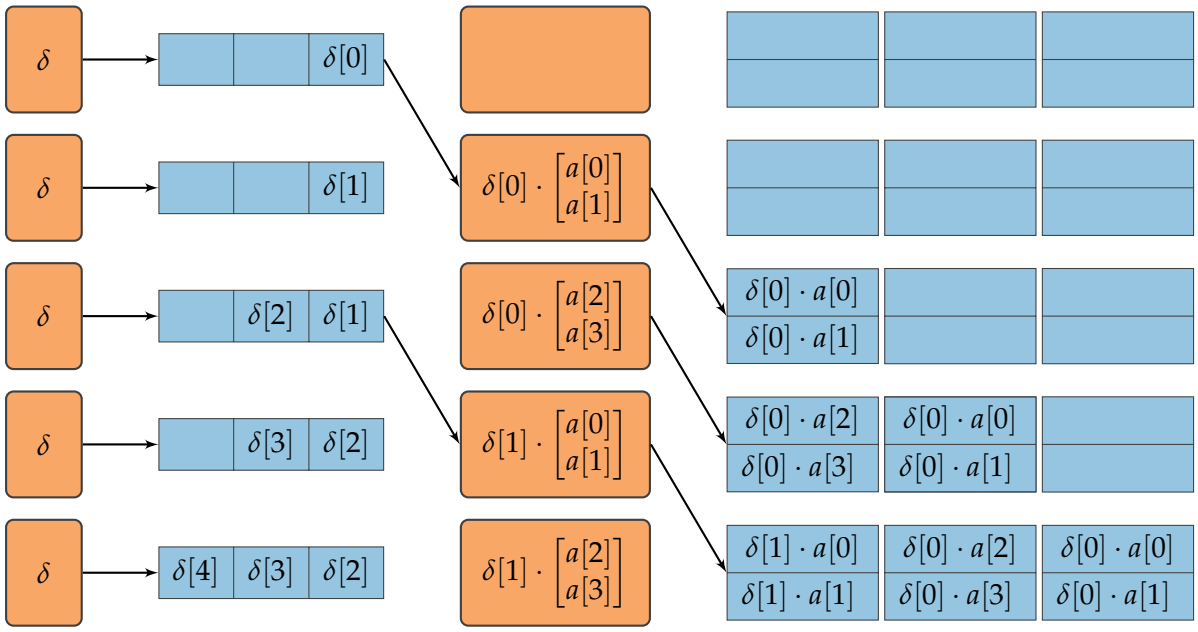

Figure 3.4: Illustration of the bottleneck situation at the computation of the gradients. In this scenario, `ParEntries` = 2 and the vector a^{l-1} has four entries. Assuming a reliable consumer on the right of the process, a depth of one for the output stream is sufficient but in this case, it has been set to three for illustration purposes. The input stream of a^{l-1} is omitted for simplification.

3.4 Combination of the BGD and the MLP

The BGD depends on the results produced by the MLP. Fig. 3.5 shows the sequence diagram of the processing of one batch on the target platform, where a processor orchestrates the execution order. The MLP and the BGD access a shared memory in the main system memory, which is assumed to be DDR, to exchange results of the MLP and new parameters computed by the BGD. After the initialization of the parameters by the processor, the MLP loads them once and processes all training samples in the batch. The results of all layers are transferred to the DDR after the computation has finished. Consequently, the BGD loads the parameters and the results of the MLP. After the update of the parameters as described in section 3.3, the parameters are stored back to the DDR and the MLP updates its parameters at the next execution. Further executions of the BGD do not load the parameters from the main memory because the copy in the local BRAM still matches the version in the DDR.

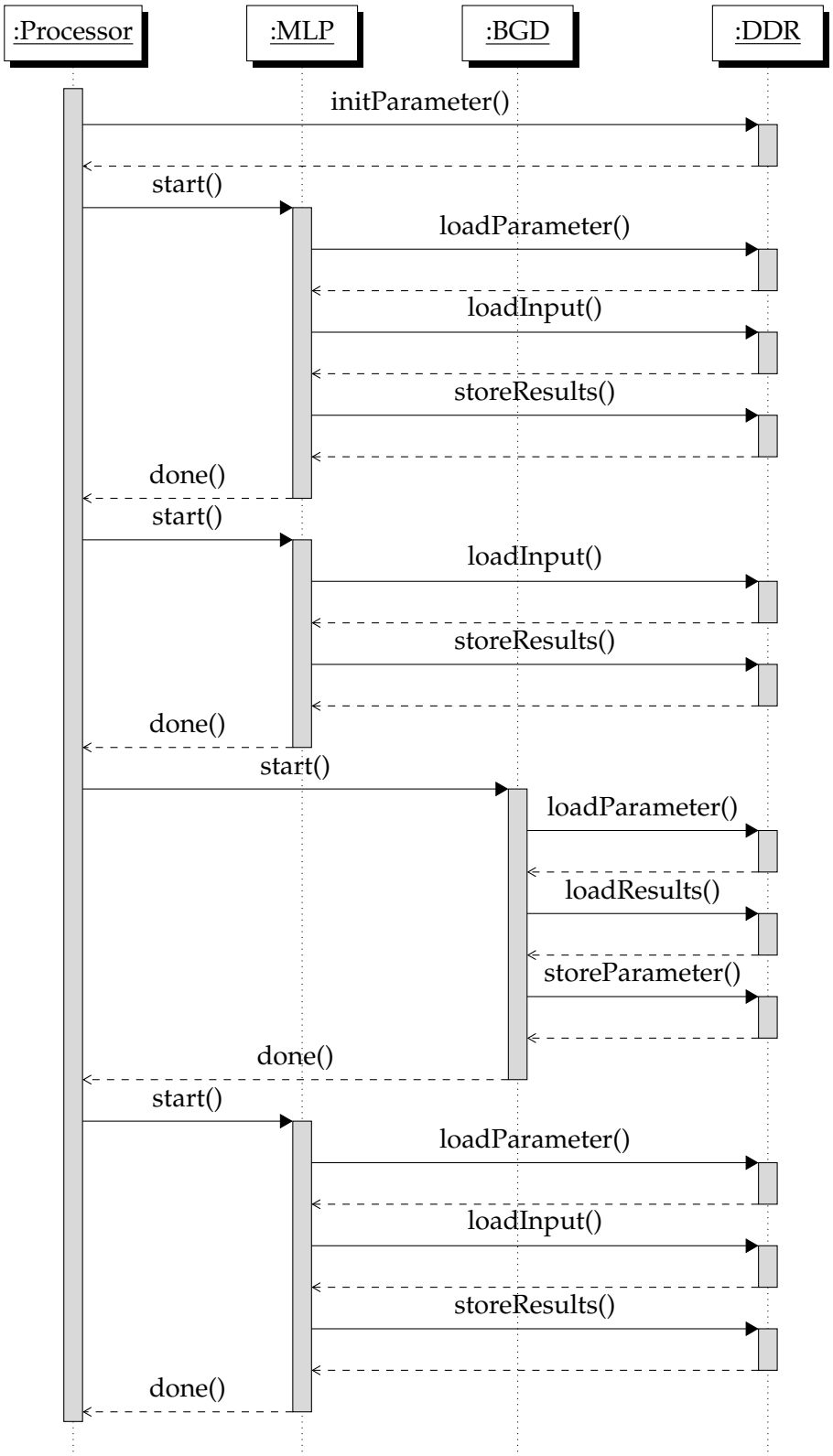

Figure 3.5: Sequence diagram of the execution order of the MLP and the BGD. The diagram shows the processing of one batch with a size of two including the consequent update of the MLP.

3.5 Shortcomings

While the macro structure of the implementation is mature, there are still some shortcomings to be fixed in further optimizations. The biggest issue is insufficient memory bandwidth. As mentioned in section 2.5.3, insufficient memory bandwidth leads to interface contention which prevents pipelining. When synthesizing the HLS designs, Vitis HLS is unable to schedule the load operations from the internal BRAM with the desired II of one due to insufficient memory bandwidth. Currently, internal arrays are declared as follows:

```
float bramArray[arraySize];
```

Increasing memory bandwidth by partially partitioning this one-dimensional array with the `ARRAY_PARTITION` directive either with block or cyclic partitioning leads to a freeze in the co-simulation. Similar behavior has been observed when using *`#pragma HLS ARRAY_RESHAPE`*. A promising approach is vectorization by using a two-dimensional array and fully partitioning the second dimension:

```
float bramArray[arraySize / ParEntries][ParEntries];
#pragma HLS ARRAY_PARTITION variable=bramArray complete dim=2
```

By fully partitioning the second dimension of the array, every element of the vector is stored in an individual BRAM instance, which increases overall memory bandwidth and enables the reading of `ParEntries` in one clock cycle [6, p. 366f.]. This solution is a compromise between the economic utilization of BRAM resources and the exploitation of memory bandwidth. By fully partitioning only the second dimension of the array, `ParEntries` BRAM instances are allocated. This enables the scheduling of a parallel access of `ParEntries` values in one clock cycle while maintaining a reasonable resource footprint. In contrast, a fully partitioned and equally sized one-dimensional array would consume `arraySize` memory entities. Since `arraySize` is usually much larger than `ParEntries` the complete partitioning of large one-dimensional arrays is inefficient or even infeasible.
This declaration is equivalent to using the `WideType` class of the Vitis libraries, where the entries of the vector are stored in a fully partitioned array [22]. An equally sized array using the `WideType` class is declared as follows:

```
xf::blas::WideType<float, ParEntries> bramArray[arraySize / ParEntries];
```

A replacement of the buffers with a vectorized version has not been examined within the scope of this project. It is strongly recommended to investigate this possibility in future redesigns because the current shortage of memory bandwidth prevents the solution to exploit the full potential of parallel computing on the FPGA. The severity of this issue is illustrated by the performance examinations presented in section 3.7.

When replacing the narrow buffers with a vectorized solution, the following aspects need to be considered: Currently, the implementation heavily relies on the data mover functions provided by the Vitis libraries to move data between streams and memory. These functions only provide interfaces to non-vectorized arrays. As a consequence, custom data mover functions need to be implemented, which provide an interface to vectorized arrays. Furthermore, the on-the-fly transposition of matrices as implemented by the function `gem2StreamTranspose()` may not perform well due to the columnwise access of the matrix. Since the BGD only works on transposed versions of the matrices it could be stored in a transposed fashion to overcome this issue. Nevertheless, this workaround requires a transposition of the matrix when interacting with the main memory. Another shortcoming is the narrow AXI to communicate with the main system memory. Currently, only one value per clock cycle is transferred to the IP-core which equals a width of four byte of the data bus. By replacing the non-vectorized arguments of the TLF with vectorized `WideType` arrays, the data bus between the system memory and the IP-core can be widened up to 64 byte. By widening the data path, up to 4096 byte per request to the main memory can be transferred [6, p. 384].
Finally, the synchronization mechanism introduced in section 2.5.3 is not elegant. A perfect dataflow implementation without pipeline draining between the layers would be preferable. [16] claims to have implemented such a dataflow architecture but there is no implementation available. Since the function `vec2GemStream()`, which interfaces between BRAM and `gemv()` from the Vitis libraries, feeds the input vector to the output stream multiple times a dataflow equivalent needs to cache the incoming stream while feeding it directly to the output stream. After the input stream is exhausted, the dataflow equivalent needs to read from the internal cache. If producer and consumer operate at different write and read rates as introduced in section 3.3, the situation gets even more sophisticated.

3.6 Functional Verification

Fig. 3.6 shows the general structure of the test bench that is used to verify the HLS implementations of the MLP and the BGD in all development stages. The open source project Cranium serves as a reference implementation for the functional verification. It is a header-only plain C implementation of the desired algorithms, which makes it usable in the HLS test bench and on the target platform [23]. In the Vitis HLS test bench, the designs are encapsulated in container classes, which realize the structure from Fig. 3.6. On the target platform, a shared memory is allocated by the processor and initialized by Cranium. Thereafter, the kernels are executed as illustrated by the sequence diagram from Fig. 3.5 and the results are compared to the reference implementation executed on the processor.
All MLP configurations tested in section 3.7 produce the same results as the reference implementation on the target platform with an error in the order of magnitude of 10^{-8}. The BGD algorithm is verified by solving the XOR problem described in section 2.4. With the configuration given in Tab. 3.1, the algorithm solves the XOR problem in 1065 epochs which equals the performance of the reference implementation. However, each individual epoch is executed faster than the software reference.
In order to estimate the performance of the implementation on solving the XOR problem, [16] serves as a benchmark. The architecture of the implementation from [16] is not clearly described but it is assumed that it uses a similar configuration to Tab. 3.1. The major difference to the current project is the integration of the MLP and the BGD in a single IP-core. This offers performance benefits over a modular approach because the communication via the main system memory is omitted, which is the main source of latency. Therefore, [16] performs better on solving the XOR problem but the flexibility is worse. Fig. 3.7 illustrates the outcome of the comparison. As shown in Fig. 3.5, the training involves multiple executions of the MLP IP-core. After each execution, the IP-core needs to be reconfigured for the next training example, which is done by the processor via AXI4-Lite. This reconfiguration is the main source of latency and is only displayed in the gross value. The net value is the sum of the latencies of the MLP and the BGD disregarding the reconfiguration of the IP-cores.

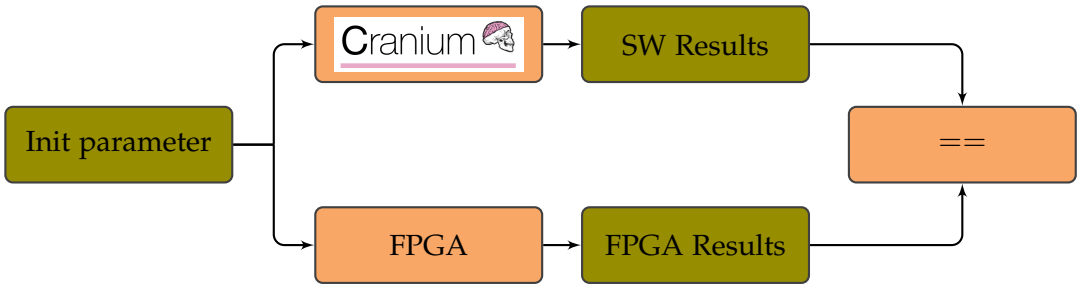

Figure 3.6: Structure of the test bench with Cranium [23] serving as a reference implementation. The initialization of the parameters is also performed by Cranium.

Table 3.1: Configuration of the network solving the XOR problem.

Parameter	Value
`numberOutputs`	2
`numberInputs`	2
`numberNeurons`	2
`numberHiddenLayers`	1
Activation function of the hidden layers	Sigmoid
Activation function of the output layer	Linear
Batch size	4
Number of training examples	4
Learning rate (η)	0.5

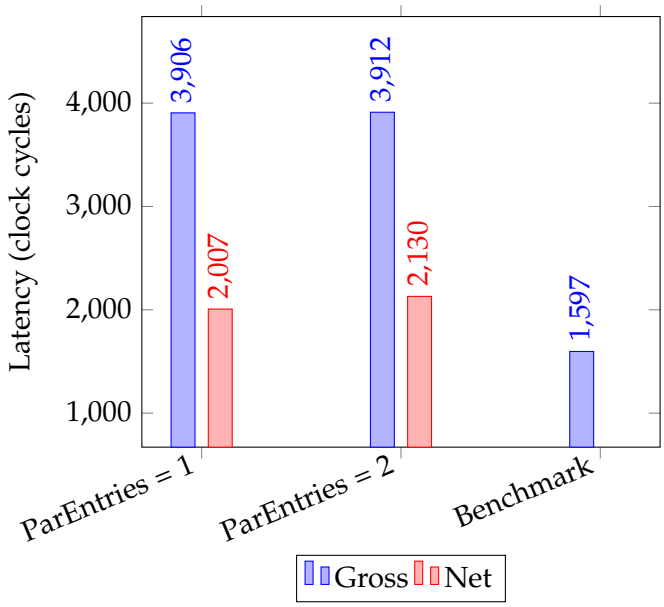

Figure 3.7: Performance of the implementation solving the XOR problem. The diagram shows the latency of one epoch of the BGD. The gross value includes reconfiguration of the IP-cores by the processor via AXI4-Lite, whereas the net value only considers the execution latency of the IP-cores.

3.7 Performance Examination

The following section presents the examination of the performance achieved by the networks MLP-1 and MLP-2 and the corresponding BGD algorithm. Section 3.7.1 to 3.7.3 only present the examinations, while section 3.8 contains an explanation of the results. The measurements have been carried out with the setup described in Appendix B. Tab. 3.2 shows the configurations of MLP-1 and MLP-2. Apart from the number of inputs and outputs, the architectures match the networks in [2] but to carry out the investigation up to a degree of parallelism of 16, an equal minimum vector size is required (see section 3.1).

Table 3.2: Configuration of the networks used for performance investigations.

Parameter	MLP-1	MLP-2
`numberOutputs`	16	16
`numberInputs`	16	16
`numberNeurons`	128	64
`numberHiddenLayers`	1	3

Tab. 3.3 shows the global settings for all simulation runs and executions on the target platform. While the parameters prefixed with `hwNumber` determine the size of the internal BRAM, which limits the size of the MLPs to be processed by the IP-cores, the actual size of the network is set during runtime via AXI4-Lite and equals the settings given in Tab. 3.2. The oversizing of the BRAM buffers enables design reuse of the same IP-core for MLP-1 and MLP-2.

Table 3.3: IP-core settings applied to all measurements. Parameters prefixed with `hw` determine the size of the allocated BRAM which limits the size of the MLPs to be processed.

Parameter	Value
`hwNumberOutputs`	32
`hwNumberInputs`	32
`hwNumberNeurons`	128
`hwNumberHiddenLayers`	3
Clock period	10 ns
Output activation	Linear

The software reference implementation from [23] is not used as a performance benchmark because it is not optimized for low latency on the target platform. As a result, the reference implementation performs three orders of magnitude worse than the hardware implementation which is not suitable to estimate the performance. Instead, the investigations in the following section use the results from [2] for the latency of MLP-1 and MLP-2. For the BGD on such networks, the literature does not supply suitable results for comparison. [2] only provides values for MLPs with ReLU activation function but as shown in section 3.7.2 the networks with a sigmoid activation function perform similarly.

During the performance investigation, the degree of parallelism `ParEntries` and the activation function of the hidden layers have been altered. The diagrams in the following sections display the latency for all possible values of `ParEntries` from one to eight executed on the target platform and the corresponding latency of the co-simulation from Vitis HLS. Additionally, `ParEntries` = 16 has been simulated but not executed on the target platform because a higher degree of parallelism does not further decrease the latency. The diagrams always show the minimum achievable latency which implies that the MLP parameters are not transferred from the main memory before the algorithm is executed and that the results of the hidden layers are not transferred to the main system memory after execution. Tab. 3.5 summarizes additional latencies for these operations. The batch size of the BGD is set to one. Latency induced by bigger batches is presented in section 3.7.3.

3.7.1 Examination of Networks with ReLU Activation

This section presents the performance examination of MLP-1 and MLP-2 with ReLU activation of the hidden layers and the corresponding BGD algorithm. Tab. 3.4 contains the latencies from [2] which serve as a benchmark. Fig. 3.8 and 3.9 present the results of the examination. The implementations of the current project perform similarly to the benchmark for `ParEntries` = 1 and `ParEntries` = 2 but higher values do not lead to a further performance increase.

Table 3.4: Benchmark values for the MLPs from [2]. The values have been measured on MLPs with a ReLU activation function in the hidden layers.

	MLP-1			MLP-2		
ParEntries	**1**	**2**	**32**	**1**	**4**	**32**
Latency (clock cycles)	4,910	2,454	348	10,220	2,552	368

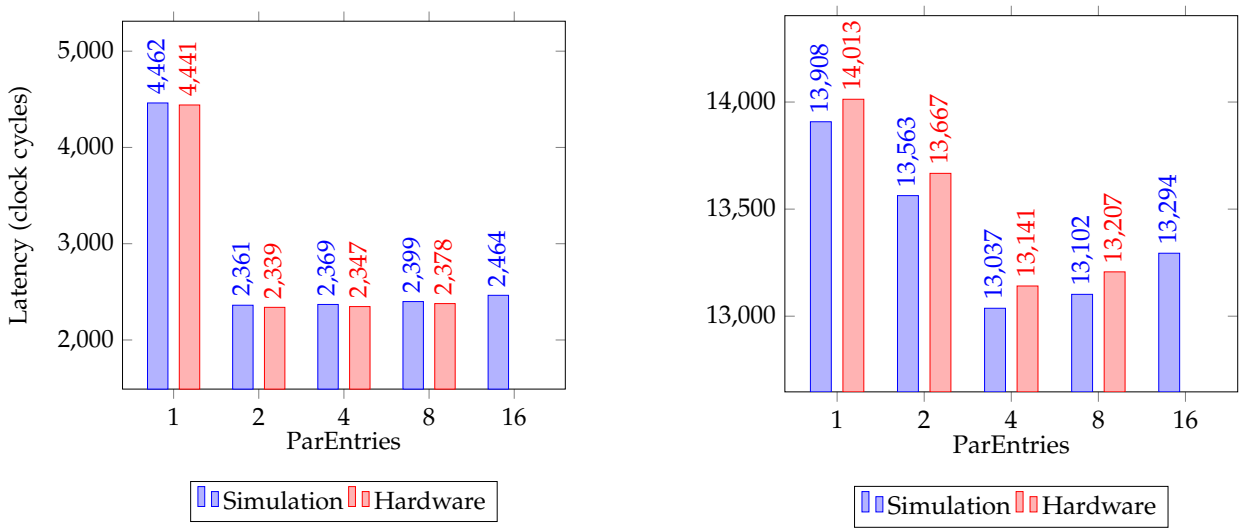

Figure 3.8: Performance of a network with one hidden layer, 128 neurons and ReLU activation of the hidden layers a.k.a. MLP-1. Left: Latency of the MLP. Right: Latency of the corresponding BGD.

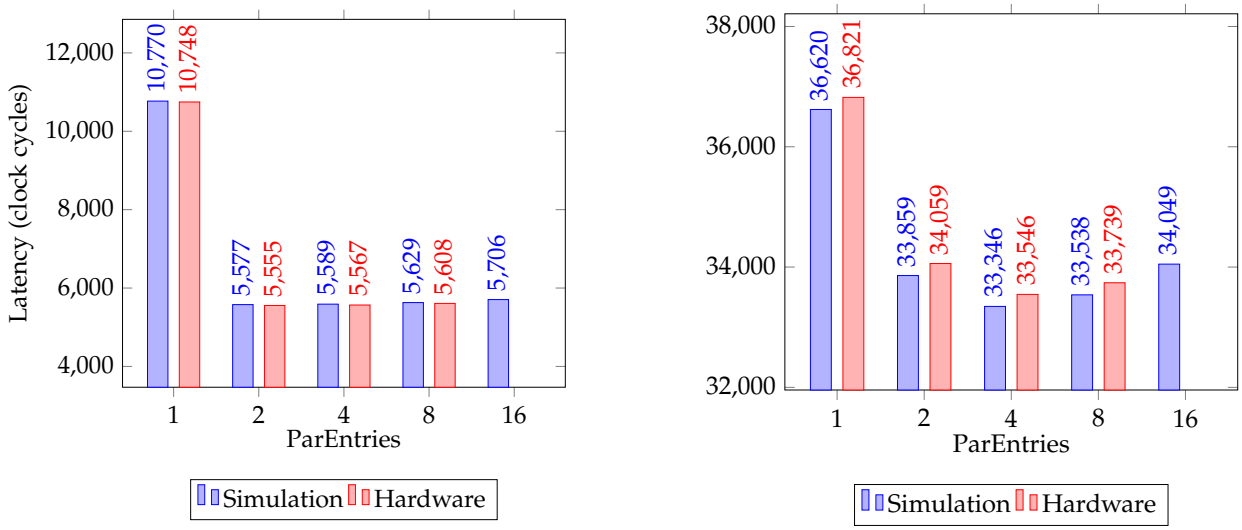

Figure 3.9: Performance of a network with three hidden layers, 64 neurons and ReLU activation of the hidden layers a.k.a. MLP-2. Left: Latency of the MLP. Right: Latency of the corresponding BGD.

3.7.2 Examination of Networks with Sigmoid Activation

The examinations from section 3.7.1 are repeated with the sigmoid activation function for the hidden layers. The MLP and the BGD perform similarly to the configuration of section 3.7.1.

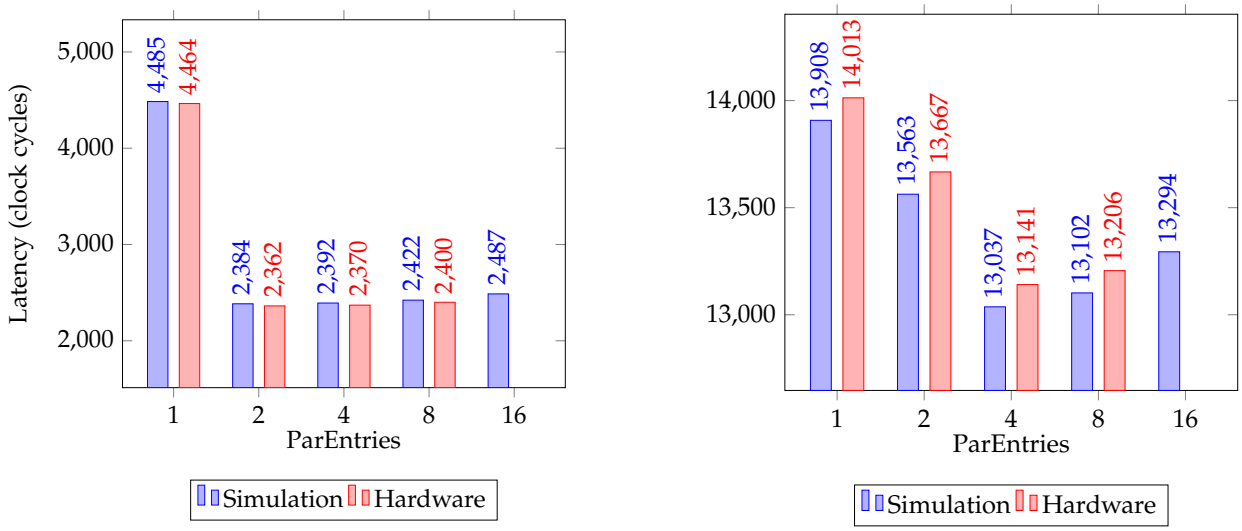

Figure 3.10: Performance of a network with one hidden layer, 128 neurons and sigmoid activation of the hidden layers a.k.a. MLP-1. Left: Latency of the MLP. Right: Latency of the corresponding BGD.

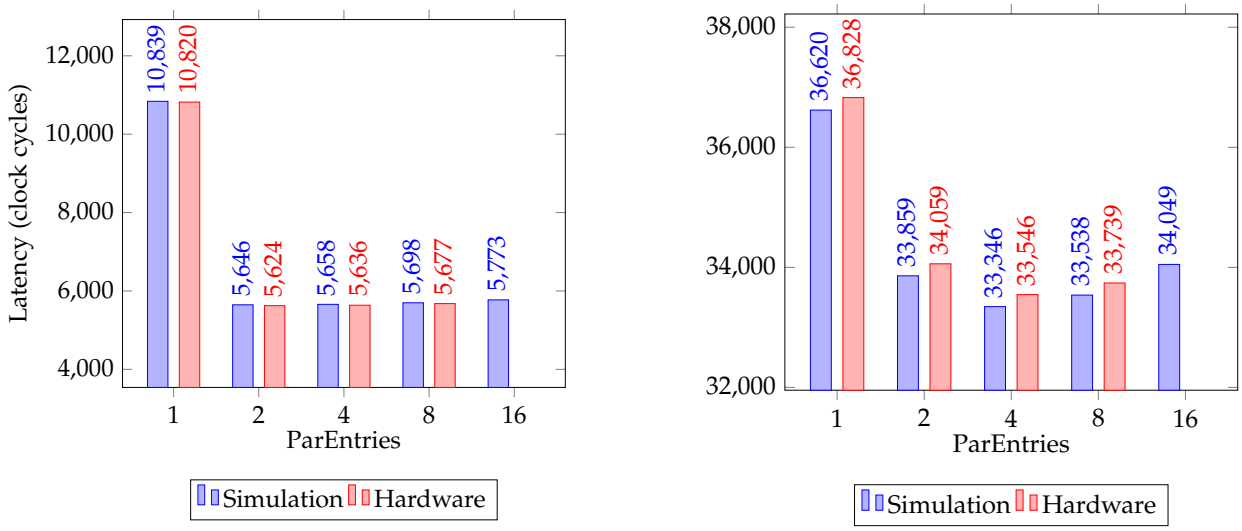

Figure 3.11: Performance of a network with three hidden layers, 64 neurons and sigmoid activation of the hidden layers a.k.a. MLP-2. Left: Latency of the MLP. Right: Latency of the corresponding BGD.

3.7.3 Additional Latencies

The examinations presented in section 3.7.1 and 3.7.2 display the minimum achievable latency. However, the IP-cores offer options to transfer the weights and biases from the main system memory to the internal BRAM. Additionally, the MLP exports the results of the hidden layers to the main system memory when setting the parameter `exportLayers`. The latencies from Tab. 3.5 are added to the aforementioned values when carrying out the corresponding operation. The values apply to all configurations disregarding the degree of parallelism and the activation function. The duration of the AXI transfer approximately equals the number of values to be transferred plus some overhead for the bus management.

Table 3.5: Additional latencies in clock cycles induced by the export of the results of the hidden layers and the update of the weights and biases of the MLP.

	MLP-1		MLP-2	
	Simulation	**Hardware**	**Simulation**	**Hardware**
Load Parameters	4,269	4,336	10,477	10,550
Export Layer	306	352	425	475

Increasing the batch size of the BGD induces additional latency on execution. Fig. 3.12 presents the latency for each additional training example in a batch of the BGD. The additional latency only depends on the size of the MLP but not on the activation function.

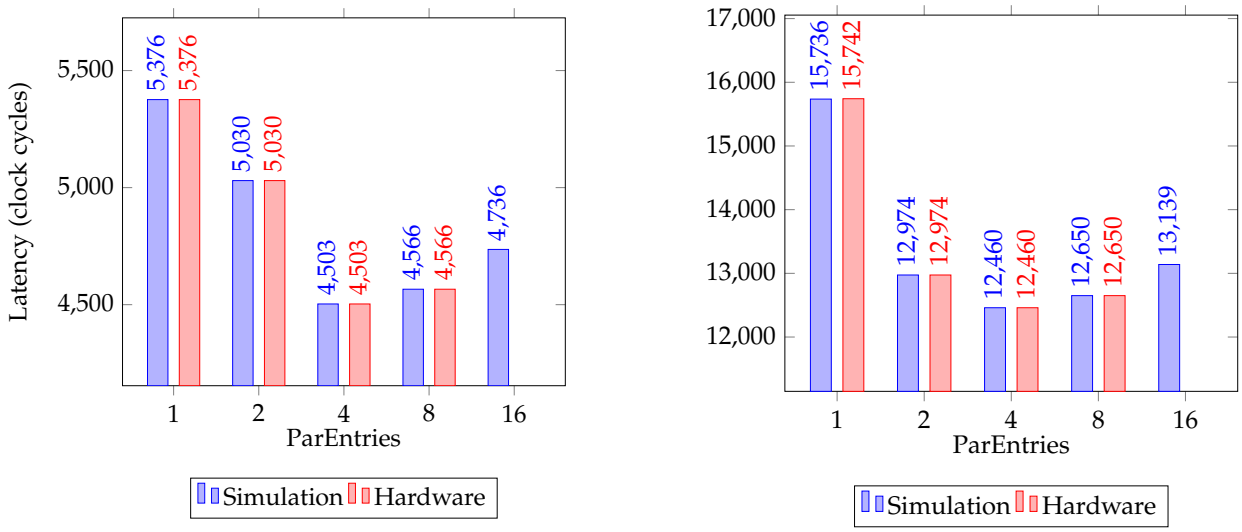

Figure 3.12: Latency of the BGD for each additional training example. The diagram on the left displays the values for MLP-1 and the diagram on the right represents MLP-2.

3.8 Summary and Interpretation

The performance examinations show that the implementation of the MLP is able to compete with the benchmark for a low degree of parallelism. The latencies for `ParEntries` = 1 and `ParEntries` = 2 are similar to the benchmark. While increasing `ParEntries` to 2 cuts the latency in half, higher degrees of parallelism do not lead to further improvement. In fact, the MLP performs even worse when further increasing `ParEntries`. The interface contention of the internal BRAM buffers discussed in section 3.5 is expected to be the reason for this behavior. While increasing the degree of parallelism may reduce the latency of the actual computation, the increased delay caused by insufficient memory bandwidth prevents overall performance improvement.
A similar behavior is observed at the BGD. The current implementation of the BGD features the lowest latency for `ParEntries` = 4, while higher degrees of parallelism lead to higher latency. In addition to the interface contention issue, the implementation suffers from the inaccurate producer-consumer situation described in section 3.3. In contrast to the MLP, a higher degree of parallelism only reduces the overall latency by about 9 %.
In contrast to the benchmark from [2] which implements the MLPs with a fixed point data type, the examinations have been carried out with a single-precision floating-point data type. Performance with a fixed-point data type has not been investigated but it stands out that the execution with floating-point performs similarly for low degrees of parallelism to a fixed-point implementation. However, it remains to be investigated if this performance is caused by the dataflow architecture or by the computational resources of the target platform.
The performance examinations show that the result of the co-simulation is eligible to estimate the latency on the target platform. Lengthy synthesis and implementation runs are only required in the final testing phase but all optimizations can be applied by exclusively using the simulation tools offered by Vitis HLS. In fact, the comprehensive log output of Vitis HLS even facilitates the optimization in simulation in comparison to testing on hardware. Minor differences between the simulation and the hardware are caused by the differences in scheduling the AXI transfers. Fig. 3.12 illustrates this behavior: In contrast to the other measurements, Fig. 3.12 only displays the additional runtime caused by computations while disregarding the AXI transfer. In this measurement, the latencies on the hardware match the simulation result.

4 Summary and Outlook

The success of the project is estimated by referring to the research questions from section 1.2: For a low degree of parallelism, the dataflow architecture of the MLP performs similarly to the benchmark, which proves the implementation methodology to be suitable. Performance for a high degree of parallelism is expected to rise after solving the interface contention issue described in section 3.5. Despite the performance issue, the implementation offers great flexibility and runtime configuration options compared to other solutions. In summary, the project delivers a solid basis with potential for optimization.
The implementation of the BGD on a FPGA using Vitis HLS is feasible. The functional verification delivered a positive result and the hardware solution runs at a lower latency than a software implementation. However, the third research question, which asks for minimum latency without resource constraints, cannot be answered within the scope of project because an increased resource usage does not lead to performance improvement. The solution suffers from interface contention and a non-perfect task level pipeline as described in section 3.3 and 3.5. When overcoming these issues, the solution is expected to perform much better than the current version. Nevertheless, the implementation also offers great runtime configuration and forms a solid basis for further improvements.
Answering the last research question, the HLS C++ implementations of this project are not suitable for performance optimized execution on a general purpose processor. The constructs used in the C++ description are specifically tailored for the post processing with Vitis HLS. Furthermore, Vitis HLS does not support important software features like dynamic memory allocation, which leads to rather inflexible solutions compared to a software implementation. In summary, a FPGA as a target platform differs too much from a processor and therefore optimizations are applied differently.

Outlook

It is strongly recommended to solve the issues mentioned in section 3.5 but further optimizations can be applied. The concept of a dataflow architecture can be ported to the interfaces of the IP-cores by using AXI4-Stream, which enables the outsourcing of the main memory accesses to a specific direct memory access (DMA) unit. Since a DMA unit feeds a specified amount of data from the main memory to the IP-core without processor intervention the overall system performance can be increased by avoiding high latency reconfigurations via AXI4-Lite.
Furthermore, the BRAM buffers, which are currently defined in the HLS project, can be externalized. External BRAM blocks in the Vivado block design enable the user to modify the buffer sizes without changing the HLS project. While this modification does not lead to lower latency, the usability of the IP-cores is improved.

List of Figures

List of Tables

List of Listings

Bibliography

[1] S. Zhao, F. Blaabjerg, and H. Wang, "An Overview of Artificial Intelligence Applications for Power Electronics," *IEEE Transactions on Power Electronics*, vol. 36, no. 4, pp. 4633–4658, Apr. 2021.

[2] T. Schindler and A. Dietz, "Real-Time Inference of Neural Networks on FPGAs for Motor Control Applications," in *2020 10th International Electric Drives Production Conference (EDPC)*, Dec. 2020, pp. 1–6.

[3] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*., The MIT Press, 2016., [Online]. Available: `https://www.deeplearningbook.org/` (visited on 04/27/2022).

[4] B. K. Bose, "Artificial Intelligence Techniques: How Can it Solve Problems in Power Electronics?: An Advancing Frontier," *IEEE Power Electronics Magazine*, vol. 7, no. 4, pp. 19–27, Dec. 2020.

[5] A. Mai, "Modellierung von Drehfeldmaschinen mit künstlichen neuronalen Netzen," Nuremberg Institute of Technology, Nuremberg, Aug. 2021.

[6] Xilinx, "Vitis High-Level Synthesis User Guide (2020.1)," 2020., [Online]. Available: `https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_1/ug1399-vitis-hls.pdf` (visited on 03/21/2022).

[7] S. Wendel, A. Geiger, E. Liegmann, *et al.*, "UltraZohm — A Powerful Real-Time Computation Platform for MPC and Multi-Level Inverters," in *2019 IEEE International Symposium on Predictive Control of Electrical Drives and Power Electronics (PRECEDE)*, May 2019, pp. 1–6.

[8] Xilinx, "Vitis AI User Guide," Jul. 2021., [Online]. Available: `https://www.xilinx.com/content/dam/xilinx/support/documentation/sw_manuals/vitis_ai/1_4/ug1414-vitis-ai.pdf`.

[9] Fast Machine Learning Lab. "HLS4ML Documentation," HLS4ML. (2021), [Online]. Available: `https://fastmachinelearning.org/hls4ml/index.html` (visited on 09/22/2021).

[10] K. Lin, "Convolutional Layer Implementations in High-Level Synthesis for FPGAs," University of Washington, Washington, Jun. 2021., [Online]. Available: `https://cds.cern.ch/record/2776765` (visited on 04/19/2022).

[11] Xilinx, *RFNoC HLS NeuralNet*, Jul. 2019., [Online]. Available: `https://github.com/Xilinx/RFNoC-HLS-NeuralNet` (visited on 04/19/2022).

[12] H. M. Vo, "Implementing the on-chip backpropagation learning algorithm on FPGA architecture," in *2017 International Conference on System Science and Engineering (ICSSE)*, Jul. 2017, pp. 538–541.

[13] S. Murugan, K. P. Lakshmi, J. Sundar, and K. Mathi Vathani, "Design and Implementation of Multilayer Perceptron with On-chip Learning in Virtex-E," *AASRI Procedia*, 2nd AASRI Conference on Computational Intelligence and Bioinformatics, vol. 6, pp. 82–88, Jan. 2014.

[14] R. G. Gironés, R. C. Palero, J. C. Boluda, and A. S. Cortés, "FPGA Implementation of a Pipelined On-Line Backpropagation," *Journal of VLSI signal processing systems for signal, image and video technology*, vol. 40, no. 2, pp. 189–213, Jun. 2005.

[15] N. Afianah, A. E. Putra, and A. Dharmawan, "High-Level Synthesize of Backpropagation Artificial Neural Network Algorithm on the FPGA," in *2019 5th International Conference on Science and Technology (ICST)*, vol. 1, Jul. 2019, pp. 1–5.

[16] A. A. Bataineh, D. Kaur, and A. Jarrah, "Enhancing the Parallelization of Backpropagation Neural Network Algorithm for Implementation on FPGA Platform," in *NAECON 2018 - IEEE National Aerospace and Electronics Conference*, Jul. 2018, pp. 192–196.

[17] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, Second Edition., Cambridge, Massachusetts: The MIT Press, 2018., [Online]. Available: `https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf` (visited on 04/27/2022).

[18] T. Schindler, L. Foss, and A. Dietz, "Comparison of Reinforcement Learning Algorithms for Speed Ripple Reduction of Permanent Magnet Synchronous Motor," in *IKMT 2019 - Innovative Small Drives and Micro-Motor Systems; 12. ETG/GMM-Symposium*, Sep. 2019, pp. 1–6.

[19] M. Nielsen, *Neural Networks and Deep Learning.*, Determination Press, 2015., [Online]. Available: `http://neuralnetworksanddeeplearning.com` (visited on 01/14/2022).

[20] Xilinx, "UltraFast Design Methodology Guide for the Vivado Design Suite," 2020., [Online]. Available: `https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_1/ug949-vivado-design-methodology.pdf` (visited on 03/22/2022).

[21] J. d. F. Licht, M. Besta, S. Meierhans, and T. Hoefler, "Transformations of High-Level Synthesis Codes for High-Performance Computing," *CoRR*, vol. abs/1805.08288, Nov. 2020.

[22] Xilinx, *Vitis Libraries*, Sep. 16, 2020., [Online]. Available: `https://github.com/Xilinx/Vitis_Libraries` (visited on 11/03/2021).

[23] D. Soni, *100/Cranium*, Feb. 5, 2017., [Online]. Available: `https://github.com/100/Cranium` (visited on 03/05/2022).

A HLS Project Structure

The repository contains the following directories:

- `include`: Header files included in the kernels and the test bench.
  - `KernelHelper.hpp`: Custom dataflow functions to implement the kernels in the directory `kernel`.
  - `MlpContainer.hpp`: Forward declaration of the TLF `MLP` and the simulation wrapper class `MlpContainer`. This class offers utilities to simulate the MLP kernel.
  - `BgdContainer.hpp`: Forward declaration of the TLF `BGD` and the simulation wrapper class `BgdContainer`. This class offers utilities to simulate the BGD kernel.
  - `Settings.hpp`: Synthesis time adjustments to the kernel and simulation settings.
  - `Simulation.hpp`: Simply includes the files `MlpContainer.hpp`, `BgdContainer.hpp` and `Simulation.hpp` as an include wrapper for the main test bench.
- `kernel`: HLS implementation of the kernels.
  - `Mlp.cpp`: HLS implementation of the MLP kernel.
  - `Training.cpp`: HLS implementation of the BGD kernel.
- `test`
  - `Test.cpp`: Test bench for MLP and BGD kernel.

Furthermore, the Vitis libraries and the reference implementation Cranium [23] are included as git submodules. Fig. A.1 shows the include structure of the HLS project. The kernel domain and the test bench are strictly separated. The container files contain a forward declaration to the TLFs but the implementations in the `KernelHelper.hpp` header file are not included in the test bench in `Test.cpp`.

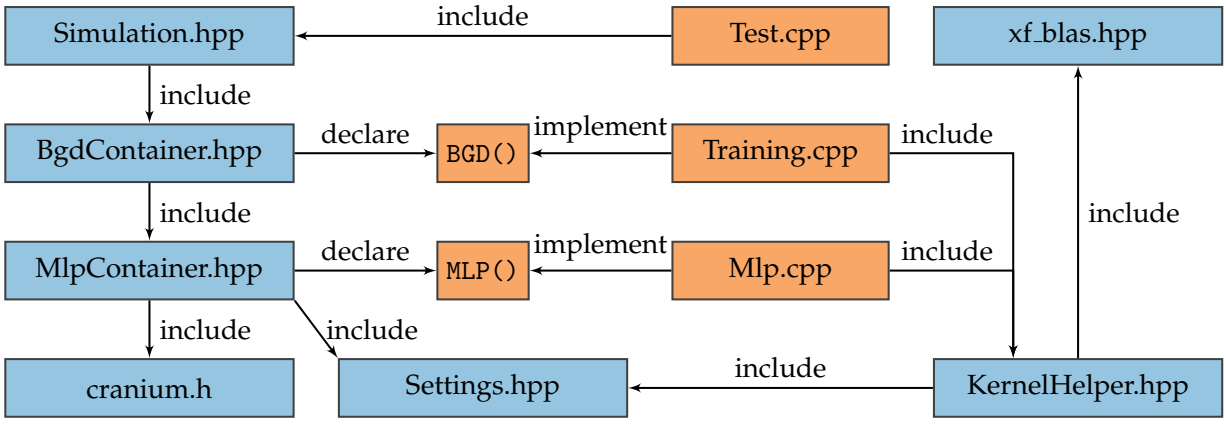

Figure A.1: Include structure of the HLS project.

B Integration in the Vivado Block Design

For the examinations on the IP-cores, the `ap_ctrl_hs` has not been mapped to a AXI4-Lite. Instead, the interface is available as wire ports in the Vivado block design which enables an ILA to display the signals of the interface. The `ap_ctrl_hs` interface has been defined with the following directive.

```
#pragma HLS INTERFACE ap_ctrl_hs port = return
```

HLS implements a control interface shown in Fig. B.1. Fig. B.2 shows the sub hierarchy which contains the AXI GPIOs and the ILA for control and debugging of the IP-cores in hardware. The ILA uses the `ap_start` signal as a trigger and `ap_done` signals a successful execution. The timing measurements from section 3.7 have been carried out with the ILA. `ap_start` is asserted by software from the integrated processor via AXI4-Lite. Fig. B.3 shows an example measurement with the ILA.

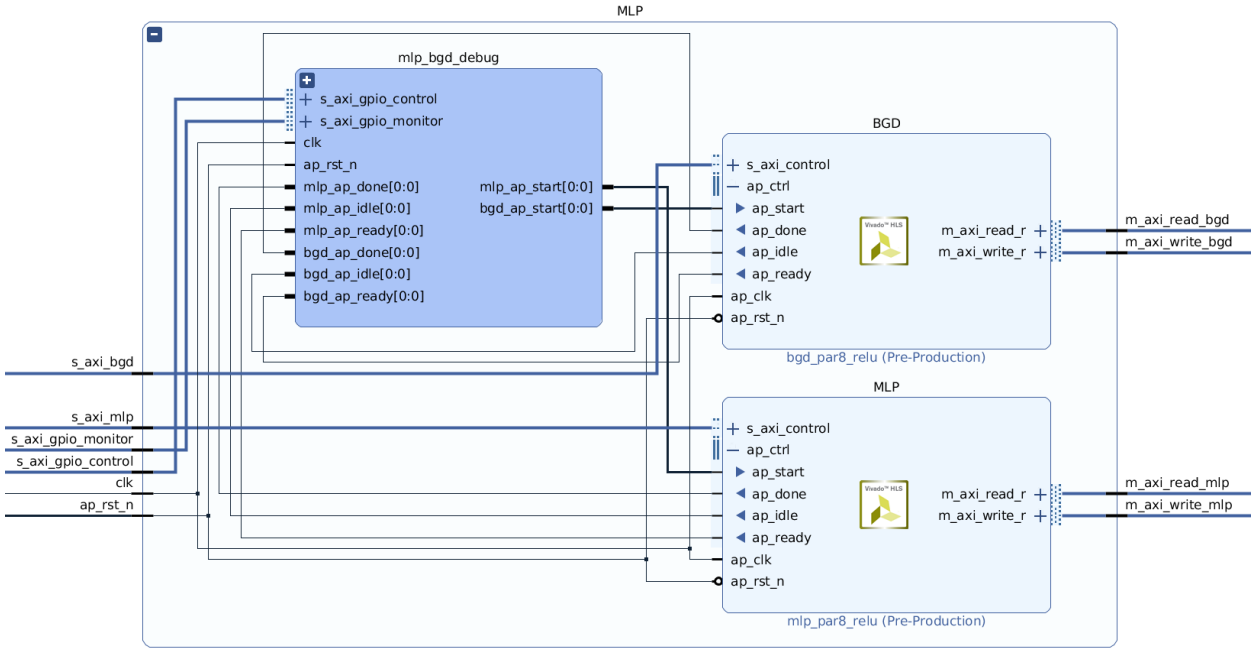

Figure B.1: MLP and BGD IP-cores in the Vivado block design.

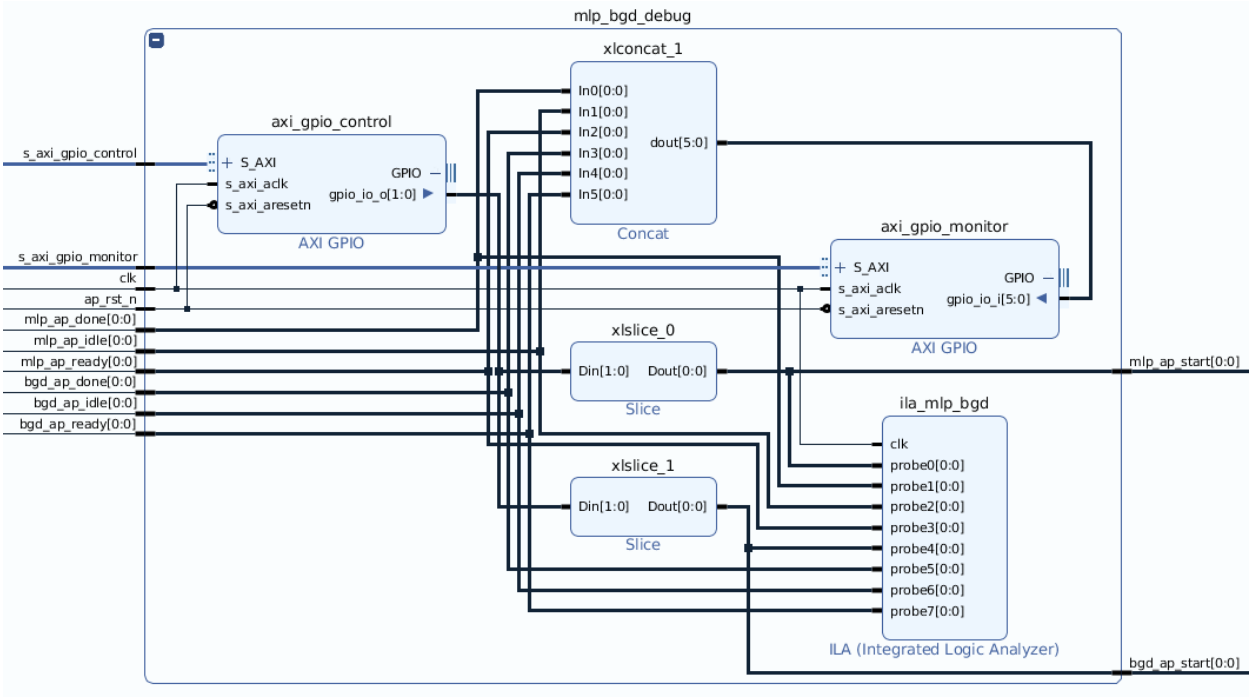

Figure B.2: Control and debug sub hierarchy for the MLP and the BGD.

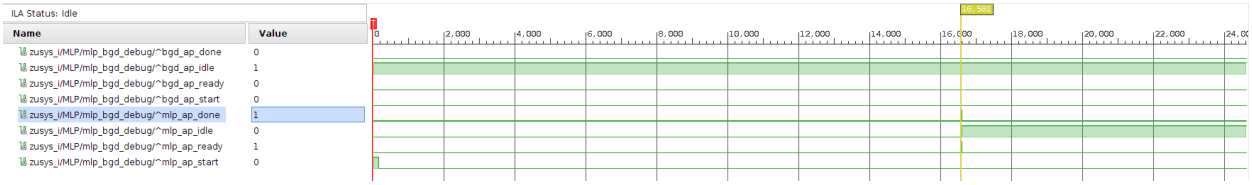

Figure B.3: Example measurement with the ILA to determine the latency on hardware. The measurement is triggered by the rising edge of the `ap_start` signal at the sample zero and the latency is measured with the cursor at the rising edge of the `ap_done` signal. In this measurement, the latency is 16,581 clock cycles.

C Development Environment

Within the scope of this project, the following tools and libraries have been used:

- Vits HLS Version 2020.1
- Vivado Version 2020.1
- Vitis IDE Version 2020.1
- Vitis libraries [22]:
  - Tag: v2020.1_update1
  - Commit hash: 3b8c61d377c9be4695e6b73deeebcb81bd9e90a8
- Cranium [23]:
  - Commit hash: 87184ef6acdb7751e9c5c5ea1be3d46bc0d69ee6
- Target Platform: UltraZohm second generation with a Xilinx Zynq UltraScale+ SoC FPGA no. xczu9eg-ffvc900-1-e

D Directory Structure of the Archive

This thesis includes an archive with the implementations of this project and relevant literature. The archive is available on the enclosed CD. The data carrier has the following directory structure:

- `arm_cortex_r5_software`: Software version of the UltraZohm project which has been used to control the IP-cores. In order to work correctly, the corresponding Vivado Platform with the Xilinx driver files is required. Relevant directories and files for this project:
  - `./IP_cores/uz_mlpHls/`: Software driver for the MLP IP-core.
  - `./IP_cores/uz_bgdHls/`: Software driver for the BGD IP-core.
  - `./sw/uz_mlpBgdHls_testbench.c`: Test bench that has been used to carry out the verification and performance examination of the IP-cores.
  - `./include/uz_mlpBgdHls_testbench.h`: Header file to be included in the main C-program that declares the test bench function.
- `exported_ip_cores`: IP-cores that have been used during the verification and performance examinations as ZIP files. These files can be imported into the Vivado IP catalog.
- `hls_code`: HLS project as described in Appendix A.
- `literature`: Relevant literature used in this project.
- `measurements`: Measurement protocol that contains the results of the performance examinations presented in section 3.7.
- `thesis_latex`: LaTeX project of this thesis including a compiled PDF version.