Hardware Accelerated Pose Tracking

Project Thesis AS2021

Master of Science in Engineering
Electrical Engineering

Author
Michael Schmid
Supervisor
Prof. Dr. Paul Zbinden, IMES
Project Partner
VRM Switzerland

OST - Ostschweizer Fachhochschule, Campus Rapperswil
December 3, 2025

This document is created with LaTeX and Ti*k*Z

The layout is based on Prof. Dr. Andreas Müller's *Partial Differential Equations* lecture notes

Abstract

Introduction With the advent of Artificial Intelligence (AI) and the ever improving neural networks, many new and unimaginable challenges can be overcome. In this paper, we focus on the task of Human Pose Estimation (HPE). The latest deep learning models combined with state-of-the-art hardware can perform HPE in real time. HPE refers to the estimation of a kinematic human body model from images.

Swiss startup *VRM Switzerland* has developed a pose tracking system to localize the 3D position of the human pose for their virtual-reality flight simulator. A multi-camera system with a top notch deep learning model and multiple Graphics Processing Units (GPUs) performs HPE with sufficient performance. To introduce HPE directly into other possible applications, the pose-tracking system should run on edge devices. The multi-GPU system is far from being an edge device. The Interdisciplinary Center for Artificial Intelligence (ICAI) at OST - Ostschweizer Fachhochschule (OST), together with *VRM Switzerland*, developed a new computationally efficient neural network for HPE called ICAIPose in a follow-up project.

Approach With the newly introduced Field Programmable Gate Arrays (FPGAs) for AI applications from Xilinx, the market leader in FPGAs, new opportunities arise. The main goal of this work is to implement ICAIPose on the high-performance Adaptive Compute Acceleration Platform (ACAP) FPGA VCK190 and the KV260 edge device with the AI Framework Vitis AI from Xilinx.

After a literature study on HPE on FPGAs, the first step was to acquire a suitable camera system. For the two FPGAs, the appropriate hardware platforms for the cameras were created. Then, a known and available computer vision application was presented in combination with the camera interface. After a time-consuming review of the correct versions for the camera interface and Vitis AI, the focus could be placed on ICAIPose. The ICAIPose network could be compiled for the Deep-Learning Processor Unit (DPU) on the FPGA with minor adjustments to the network. Thanks to the included Vitis AI Runtime Engine (VART) with its easy-to-use `Python` API, communication with the DPU is done via an embedded Linux on the FPGAs microprocessor.

Conclusion First of all, ICAIPose runs on both FPGA platforms with the proposed camera interfaces. ICAIPose runs with the used DPU configurations at 27 fps on the VCK190 and 8 fps on the KV260 in its original configuration. The performance achieved with ICAIPose, which requires about 100 GOps to process an image, is adequate. For the VCK190, better performance can be expected with a different DPU configuration. Xilinx' framework Vitis AI has been tested thoroughly and shows its strengths, but also some teething troubles.

Acknowledgement

I would like to express my great appreciation to my supervisor, Prof. Dr. Paul Zbinden, for his valuable and constructive support. Special thanks also go to Simon Walser, the author of the leading master's thesis on this work. They have helped me in many ways throughout the project with their extensive experience. I thank *VRM Switzerland* for openly providing their intellectual property.

Contents

Abbreviations

AI Artificial Intelligence

OST OST - Ostschweizer Fachhochschule

ICAI Interdisciplinary Center for Artificial Intelligence

VHDL Very High Speed Integrated Circuit Hardware Description Language

HDL Hardware Description Language

ANN Artificial Neural Network

CNN Convolutional Neural Network

ReLU Rectified Linear Unit

PReLU Parametric Rectified Linear Unit

DPU Deep-Learning Processor Unit

VART Vitis AI Runtime Engine

GPU Graphics Processing Unit

CPU Central Processing Unit

FPGA Field Programmable Gate Array

HPE Human Pose Estimation

ACAP Adaptive Compute Acceleration Platform

DSP Digital Signal Processor

PL Programmable Logic

PS Programmable System

SoC System on Chip

SoM System on Module

HPC High Performance Computing

PCB Printed Circuit Board

IAS Image Access System

TRD Target Reference Design

Chapter 1

Introduction

Motivation Physical therapy is becoming more and more popular and is often prescribed to patients. A relief of the specialists, a reduction of the health costs as well as an extraordinary patent care is desired. With the emergence of AI and the ever-improving neural networks, many new and unimaginable challenges can be overcome. The latest deep learning models in combination with cutting edge hardware can perform HPE in real-time. HPE refers to estimating a kinematic human body model from images. HPE, performed in AI, could provide a novel way to support patients during their exercise routines.

Swiss startup *VRM Switzerland* engineered a pose tracking system to localize 3D position of the human pose for their virtual reality flight simulator. A multi-camera system with top notch a deep learning model and multiple GPUs perform HPE with sufficient performance. The idea is to use this system for a new field of application, like physiotherapy. To introduce HPE directly to other possible applications, the pose tracking system should run on edge devices. The multi-GPU system is far from a edge device. The ICAI developed in a following project together with *VRM Switzerland* a new computationally efficient neural network for HPE called ICAIPose.

Approach With the newly introduced FPGAs for AI applications from Xilinx, the market leader in FPGAs, new opportunities arise. The main goal is to implement ICAIPose on the high-performance ACAP FPGA VCK190 and the KV260 edge device with the AI Framework Vitis AI from Xilinx. For this purpose, a camera system had to be evaluated and a corresponding hardware platform had to be created

The first step was to procure a suitable camera system. For the two FPGAs, the corresponding hardware platforms for the cameras had to be created. To test the developed camera interface, a well-known computer vision task is implemented. In a final step, the ICAIPose network is adapted and compiled for the two hardware platforms. The performance of HPE and the throughput in frames per second will be observed throughout the process.

Content HPE on FPGAs is a fairly novel approach, but there has been some work proposing such systems. In Chapter 2, a literature review on this topic is conducted. In addition, the basics of Deep Learning and HPE are discussed in detail. In Chapter 3, the main steps of this work are presented. The camera interface, a computer vision application, and the implementation of ICAIPose. In Chapter 4 the performance of the throughout and the general performance of the HPE task is shown.

Chapter 2

Fundamentals

2.1 Literature Study

In this part, the available literature on relevant HPE networks and their implementation on FPGAs is discussed. There are many published works on deep learning on FPGAs or on HPE with deep learning. In contrast, papers about HPE on FPGAs are rather rare.

Realtime Multi-Person 2D Pose Estimation using Part Affinity Fields (OpenPose) OpenPose is a deep learning network for multi-HPE. *"The approach uses a non-parametric representation, which we refer to as Part Affinity Fields (PAFs), to learn to associate body parts with individuals in the image."* The network won the COCO 2016 keypoint challenge and is very popular ever since [**?**].

An FPGA Realization of OpenPose based on a Sparse Weight Convolutional Neural Network In this paper from 2018, they introduced a sparse weight CNN to reduce the amount of memory size for weights. With that they proposed an indirect memory access architecture to realize the sparse CNN convolutional operation efficiently. They claimed to be the first FPGA implementation of OpenPose [**?**].

End-to-End Optimization of Deep Learning Applications A paper from 2020 about the software and hardware co-desing of FPGA integration into `TensorFlow`. The proposed architecture called FlexCNN can deliver high computation efficiency for different types of convolution layers using techniques including dynamic tiling and data layout optimization. For the case study they used OpenPose. [**?**]

2.2 Deep Learning

Many software systems of any kind are based on hard-coded parameters set by the engineer developing the system. AI can be used to generate knowledge from data. The generic term AI can be divided into three subcategories [**?**].

- **Machine learning** uses various techniques to analyze data. A typical example would be classifying data into two subsets. A machine learning algorithm has raw data as input, in order for such algorithms to work properly, the data must be represented appropriately.

- **Representation Learning** is used to learn the a good representation itself. Hand engineered representation can be very laborious and needs quite some guesswork sometimes. Machine learning algorithm can result in much better results and can be adapted quickly to a new set of data.
- **Deep Learning** provides solution which includes the machine learning and the representation learning part. The term "*deep*" comes from the multilayer structure of the presented networks in deep learning

2.2.1 Neuron

The perceptron or neuron presented by Rosenblatt [**?**], was the first neuron with learnable parameters. The neuron is inspired by the human brain, hence the name. Such a neuron is defined by

$$f(\mathbf{x}) = \hat{y}_1 = \sigma(\mathbf{w} \cdot \mathbf{x} + b), \tag{2.1}$$

where

$$\mathbf{w} \cdot \mathbf{x} = \sum_{i=1}^{n} w_i x_i, \tag{2.2}$$

is the definition of the scalar product, $\sigma(\cdot)$ is an activation function and n is the number of inputs. The visualization of the neuron can be seen in Fig. 2.1.

Such a neuron is a binary classifier for supervised learning, where the weights $\mathbf{w}$ and the bias term b are learned.

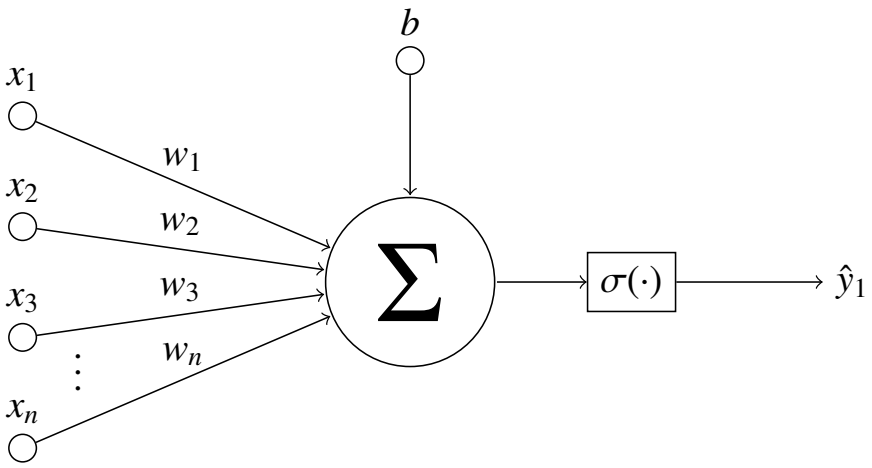


Figure 2.1: Graph of a neuron with n-inputs

2.2.2 Artificial Neural Network

This section about the Artificial Neural Network (ANN) and its training is heavily inspired by [**?**]. An ANN is composed of many neurons, which are connected layer-wise with one another. In Fig. 2.2 the structure of such a network can be seen.

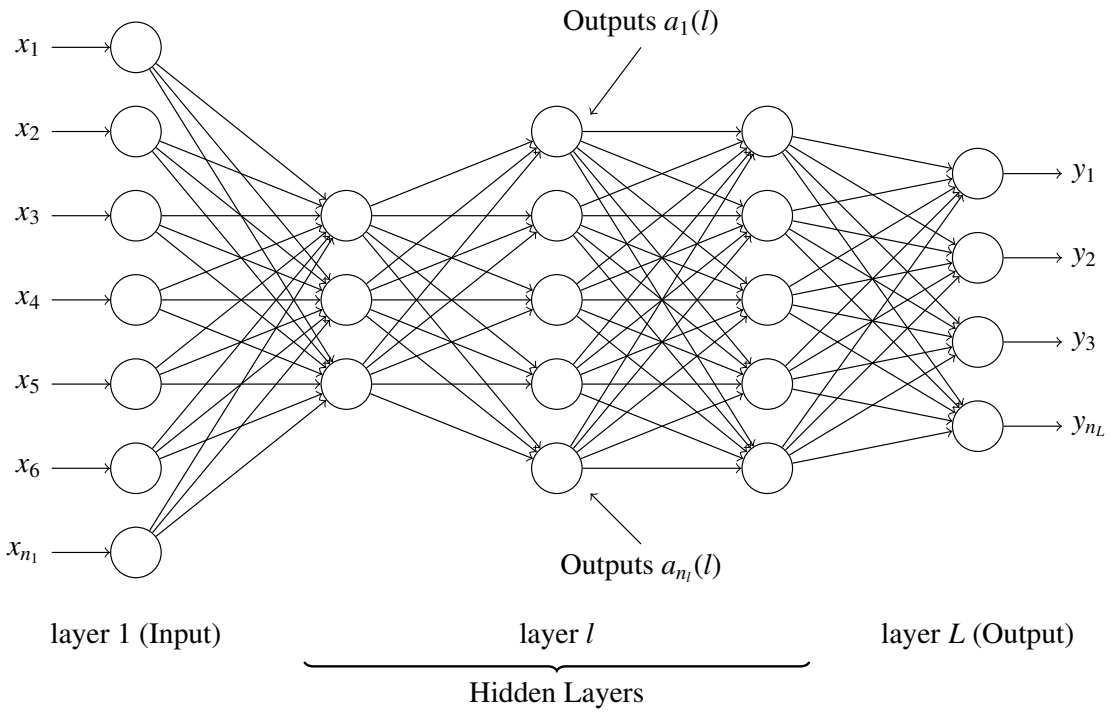


Figure 2.2: Graph of a ANN with L layers, which is also called *Fully Connected Neural Network* because each output of a neuron is connected with every neuron of the next layer. The first layer with n_1 nodes is connected to the hidden layers with n_l nodes. The last layer with n_L nodes provides the output.

2.2.2.1 Forward Pass

The output $a_i(l)$ of the neuron i in layer l is illustrated in figure Fig. 2.3 and can numerically computed with

$$a_i(l) = \sigma\left(\sum_{j=1}^{n_{l-1}} w_{ij}(l)a_j(l-1)\right) + b_i(l), \tag{2.3}$$

where n_{l-1} are the number of nodes of the previous layer. This can be written in matrix notation, with

$$\mathbf{z}(l) = \mathbf{W}(l)\,\mathbf{a}(l-1) + \mathbf{b}(l) \qquad \text{where } l = 2, 3, \ldots, L, \tag{2.4}$$

where the weights are arranged as follows

$$\mathbf{W}(l) = \begin{bmatrix} W_{11}(l) & W_{12}(l) & \cdots & W_{1n_{l-1}}(l) \\ W_{21}(l) & W_{22}(l) & \cdots & W_{2n_{l-1}}(l) \\ \vdots & \vdots & \ddots & \vdots \\ W_{n_l1}(l) & W_{n_l2}(l) & \cdots & W_{n_ln_{l-1}}(l) \end{bmatrix} \tag{2.5}$$

The activation function is applied to each neuron separately.

$$\mathbf{a}(l) = \sigma[\mathbf{z}(l)] = \begin{bmatrix} \sigma\left(z_1(l)\right) \\ \sigma\left(z_2(l)\right) \\ \vdots \\ \sigma\left(z_{n_l}(l)\right) \end{bmatrix}. \tag{2.6}$$

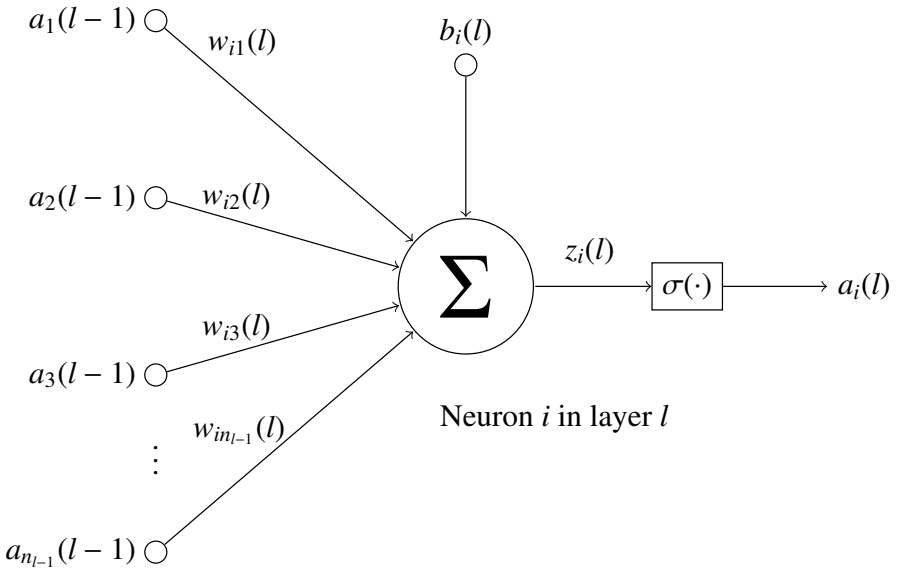


Figure 2.3: Graph of the i^{th} neuron in layer l of an ANN. The inputs $a_i(l-1)$ are from the output from the previous layer $l-1$, the weights w and the bias b are corresponding to the layer l, where $\sigma(\cdot)$ is an activation function

2.2.2.2 Training Artificial Neural Networks with Backpropagation

In order to use such an architecture of a neural network, the weights, possible millions of them, needs to be computed. In deep learning, one calls this process learning, because of its iterative algorithms.

Error Function Firstly, one needs to know how well the network performed for a given input when the desired output is know. The error of a neuron in the output layer can be defined as

$$E_j = \frac{1}{2}\left(r_j - a_j(L)\right)^2, \tag{2.7}$$

where j is the neuron, r_j the desired and a_j the actual response. For all Neurons of the output layer, the error

$$E = \sum_{j=1}^{n_L} E_j = \frac{1}{2}\sum_{j=1}^{n_L}\left(r_j - a_j(L)\right)^2 = \frac{1}{2}\left\|\mathbf{r} - \mathbf{a}(L)\right\|^2 \tag{2.8}$$

can be written as the euclidian vector norm.

Other Error Functions Different error functions are used to train neural networks, two common examples:

- The mean squared error

$$\text{MSE} = \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{n} \tag{2.9}$$

- Cross Entropy Loss/Negative Log Likelihood

$$\text{CE} = -\left(y_i log(\hat{y}_i) + (1-y_i) log(1-\hat{y}_i)\right) \tag{2.10}$$

Gradient Decent The main goal of the training is to find the weights and biases to minimize the error function. With gradient decent one calculates the gradient of the error function with respect to the weights and biases. Knowing the steepest direction of the error function, one can change the weights and biases a little so that the error function is slowly minimized.

To make the notation a bit more readable the simplification of the gradient of neuron j in the last layer L

$$\delta_j(L) = \frac{\partial E}{\partial z_j(L)} \tag{2.11}$$

is used.

The gradient of the last layer L in terms of the output $a_j(L)$ with the activation function σ is given by

$$\begin{aligned} \delta_j(L) &= \frac{\partial E}{\partial z_j(L)} \\ &= \frac{\partial E}{\partial a_j(L)} \frac{\partial a_j(L)}{\partial z_j(L)} \\ &= \frac{\partial E}{\partial a_j(L)} \frac{\partial \sigma\left(z_j(L)\right)}{\partial z_j(L)} \\ &= \frac{\partial E}{\partial a_j(L)} \sigma'\left(z_j(L)\right). \end{aligned} \tag{2.12}$$

Now one wants to know the gradient of every node in the network

$$\delta_j(l) = \frac{\partial E}{\partial z_j(l)}. \tag{2.13}$$

$\delta_j(l)$ is to be expressed with $\delta_j(l+1)$. Due to the backpropagation from the output, one starts with $\delta_j(L)$ and then computes $\delta_j(L-1)$ and with that result $\delta_j(L-2)$ and so on.

Equation (2.13) show how E changes with respect to a small change to any neuron in the network.

$$\begin{aligned} \delta_j(l) = \frac{\partial E}{\partial z_j(l)} &= \sum_i \frac{\partial E}{\partial z_i(l+1)} \frac{\partial z_i(l+1)}{\partial a_j(l)} \frac{\partial a_j(l)}{\partial z_j(l)} \\ &= \sum_i \delta_i(l+1) \frac{\partial z_i(l+1)}{\partial a_j(l)} \sigma'\left(z_j(l)\right) \\ &= \sigma'\left(z_j(l)\right) \sum_i w_{ij}(l+1)\delta_i(l+1) \end{aligned} \tag{2.14}$$

$\delta_j(l)$ is computed by all the $\delta_i(l+1)$ from the previous layer. $\frac{\partial a_j(l)}{\partial z_j(l)}$ is replaced with the derivative of the activation function. $\frac{\partial z_i(l+1)}{\partial a_j(l)}$ can be replaced with $w_{ij}(l+1)$ because $z_i(l+1) = w_{ij}(l+1)a_j(l)$

In the end we need to know how much the error changes with respect to a change to the weights.

$$\begin{aligned} \frac{\partial E}{\partial w_{ij}(l)} &= \frac{\partial E}{\partial z_j(l)} \frac{\partial z_j(l)}{\partial w_{ij}(l)} \\ &= \delta_i(l) \frac{\partial z_i(l)}{\partial w_{ij}(l)} \\ &= a_j(l-1)\delta_i(l) \end{aligned} \tag{2.15}$$

Similar to the weights the derivation of the error in respect to the bias term

$$\frac{\partial E}{\partial b_i(l)} = \delta_i(l) \tag{2.16}$$

The weights can now be updated using the Gradient Decent algorithm

$$\begin{aligned} w_{ij}(l) &= w_{ij}(l) - \alpha \frac{\partial E(l)}{\partial w_{ij}(l)} \\ &= w_{ij}(l) - \alpha \delta_i(l) a_j(l-1), \end{aligned} \tag{2.17}$$

similar for the bias terms

$$\begin{aligned} b_i(l) &= b_i(l) - \alpha \frac{\partial E(l)}{\partial b_i(l)} \\ &= b_i(l) - \alpha \delta_i(l) a_j(l), \end{aligned} \tag{2.18}$$

where α stands for the step size towards the minimum of the error function.

Matrix Notation Following the matrix notation of the previous shown equations.

Equaion (2.12) translates to

$$\boldsymbol{\delta}(L) = \begin{bmatrix} \delta_1(L) \\ \delta_2(L) \\ \vdots \\ \delta_{n_L}(L) \end{bmatrix} = \begin{bmatrix} \frac{\partial E}{\partial a_1(L)} \sigma'(z_1(L)) \\ \frac{\partial E}{\partial a_2(L)} \sigma'(z_2(L)) \\ \vdots \\ \frac{\partial E}{\partial a_{n_L}(L)} \sigma'(z_{n_L}(L)) \end{bmatrix} = \begin{bmatrix} \frac{\partial E}{\partial a_1(L)} \\ \frac{\partial E}{\partial a_2(L)} \\ \vdots \\ \frac{\partial E}{\partial a_{n_L}(L)} \end{bmatrix} \odot \begin{bmatrix} \sigma'(z_1(L)) \\ \sigma'(z_2(L)) \\ \vdots \\ \sigma'(z_{n_L}(L)) \end{bmatrix} \tag{2.19}$$

where $\odot$ represents the elementwise multiplication between two vectors or matrices.

$$\boldsymbol{\delta}(L) = \frac{\partial E}{\partial \mathbf{a}(L)} \odot \sigma'(\mathbf{z}(L)) \tag{2.20}$$

The vector $\boldsymbol{\delta}(L)$ accounts for one given input vector. The matrix $\mathbf{D}(L)$ accounts for all input vector one wants to use for the training algorithm.

$$\mathbf{D}(L) = \frac{\partial E}{\partial \mathbf{A}(L)} \odot \sigma'(\mathbf{Z}(L)) \tag{2.21}$$

Equation (2.14) translates to

$$\mathbf{D}(l) = \left(\mathbf{W}^T(l+1)\mathbf{D}(l+1)\right) \odot \sigma'(\mathbf{Z}(l)) \tag{2.22}$$

and the two gradient decent algorithms in matrix notation can be written as

$$\mathbf{W}(l) = \mathbf{W}(l) - \alpha \mathbf{D}(l)\mathbf{A}^T(l-1) \tag{2.23}$$

respectively,

$$\mathbf{b}(l) = \mathbf{b}(l) - \alpha \sum_{k=1}^{n_p} \boldsymbol{\delta}_k(l). \tag{2.24}$$

2.2.3 Convolutional Neural Network

Through an ANN, each input node is connected to every other input node. Let's imagine an image of a dog where the network must correctly classify the animal. The top right pixel and the bottom left pixel, in combination, contain little to no information about the animal in the image. Much more information is contained in the area around a pixel. Convolution with a learned filter extracts the information from a surrounding area. Also, convolution is a translation invariant operation, which is a very useful property since it does not matter where the dog is located in the image.

In Fig. 2.4 a typical Convolutional Neural Network (CNN) architecture with a 2D input image is shown.

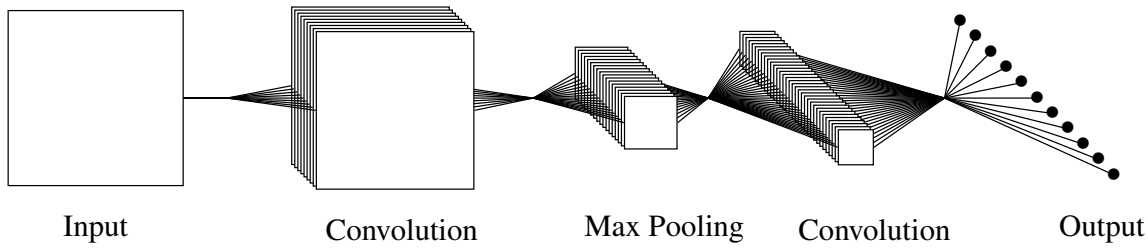


Figure 2.4: Graph of a CNN

The output of a CNN can be a convolutional layer or a ANN (mostly for classification).

2.2.3.1 Forward pass in a CNN

With the same notation as in section 2.2.2.1, the convolution with the kernel $w(l)$ from layer l with the output $a(l-1)$ of the previous layer

$$\begin{aligned} z_{x,y}(l) &= \sum_l \sum_k w_{l,k}(l) a_{x-l,y-k}(l-1) + b(l) \\ &= w(l) * a_{x,y}(l-1) + b(l), \end{aligned} \tag{2.25}$$

where l and k span the the dimension of the kernel.

The convolution can be written with the $*$ operator. The output of a convolutional layer with the activation function

$$a_{x,y}(l) = \sigma\left(z_{x,y}(l)\right). \tag{2.26}$$

With this convolution the output is smaller than the input. Sometimes it is desired to have the same output shape as the input shape, in which case the input is padded with zeros. In figure Fig. 2.5 the convolution of one input feature and three filters (kernel) is illustrated.

In figure Fig. 2.6 the convolution of four input features and three filters (kernel) is illustrated.

2.2.3.2 Layers

There are many different layers which are used in CNNs. Here are the relevant layer to this work shortly introduced.

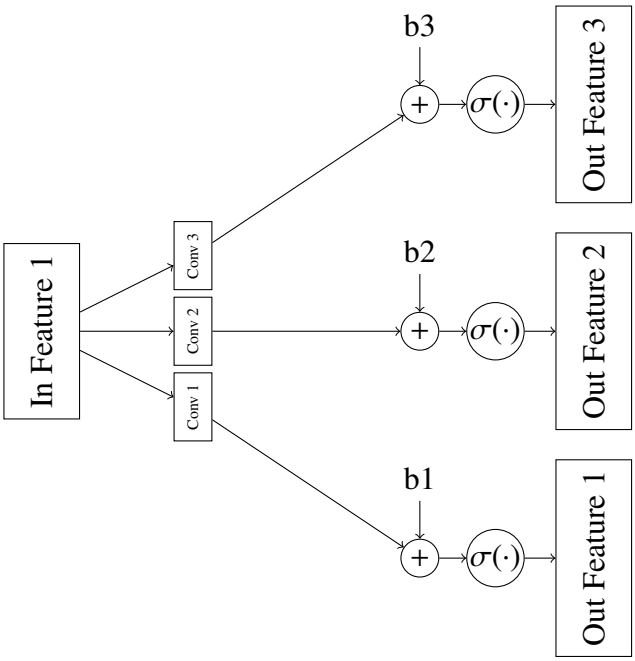


Figure 2.5: Graph of a convolutional layer with one input feature map and 3 filters. The input feature is convoled with three different kernels

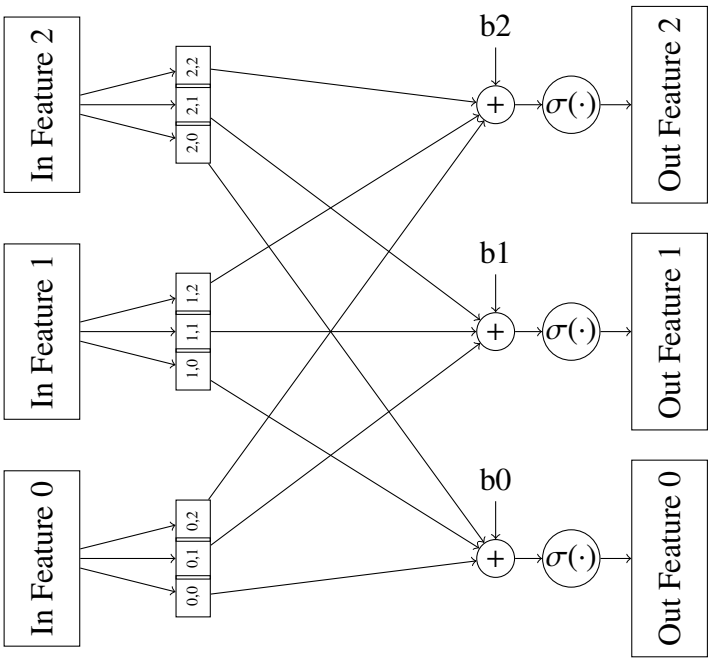


Figure 2.6: Graph of a convolutional layer with 3 input feature maps and 3 filters. Every input feature is convolved with four filters, in total there are $n_{convs} = n_{input} \cdot n_{filters}$ in this case $3 \cdot 3 = 9$ convolutions

Pooling for dimension reduction a pooling layer is used. As an example: Max pooling picks the max value of a certain neighborhood.

A 2×2 max-pooling layer would result in

$$\begin{bmatrix} 6 & -34 & 34 & -45 \\ 7 & -4 & 3 & 56 \\ 45 & 3 & 78 & 34 \\ -5 & 34 & -73 & 34 \end{bmatrix} \Rightarrow \begin{bmatrix} 7 & 56 \\ 45 & 78 \end{bmatrix}. \tag{2.27}$$

A 2×2 average-pooling layer would result in

$$\begin{bmatrix} 6 & -34 & 34 & -45 \\ 7 & -4 & 3 & 56 \\ 45 & 3 & 78 & 34 \\ -5 & 34 & -73 & 34 \end{bmatrix} \Rightarrow \begin{bmatrix} -6.25 & 12 \\ 19.25 & 18.25 \end{bmatrix}. \tag{2.28}$$

Up Sampling This layer repeats its value for n times in two dimensions. An example with $n = 2$

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 1 & 2 & 2 \\ 1 & 1 & 2 & 2 \\ 3 & 3 & 4 & 4 \\ 3 & 3 & 4 & 4 \end{bmatrix}. \tag{2.29}$$

Depthwise Convolution The principal of the depthwise convolution is shown in Fig. 2.7

Concatenate *"This layer takes as input a list of tensors, all of the same shape except for the concatenation axis, and returns a single tensor that is the concatenation of all inputs"* [**?**]

2.2.4 Activation Function

The activation function at the output of a neuron can be of various shapes. The main idea is to bring non-linearity into the network. A neuron by itself is a linear operation. Following the used common functions and some which are relevant to this work.

Softmax The Softmax function

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}} \quad \text{for } i = 1, \ldots, k \tag{2.30}$$

normalizes input to a probability distribution. The softmax activation is mostly used for multi-dimensional classification in the last layer of a network.

Sigmoid For binary classification the sigmoid function

$$\sigma(z) = \frac{1}{1 + e^{x}} \tag{2.31}$$

can be used. It also convert the input into a probability

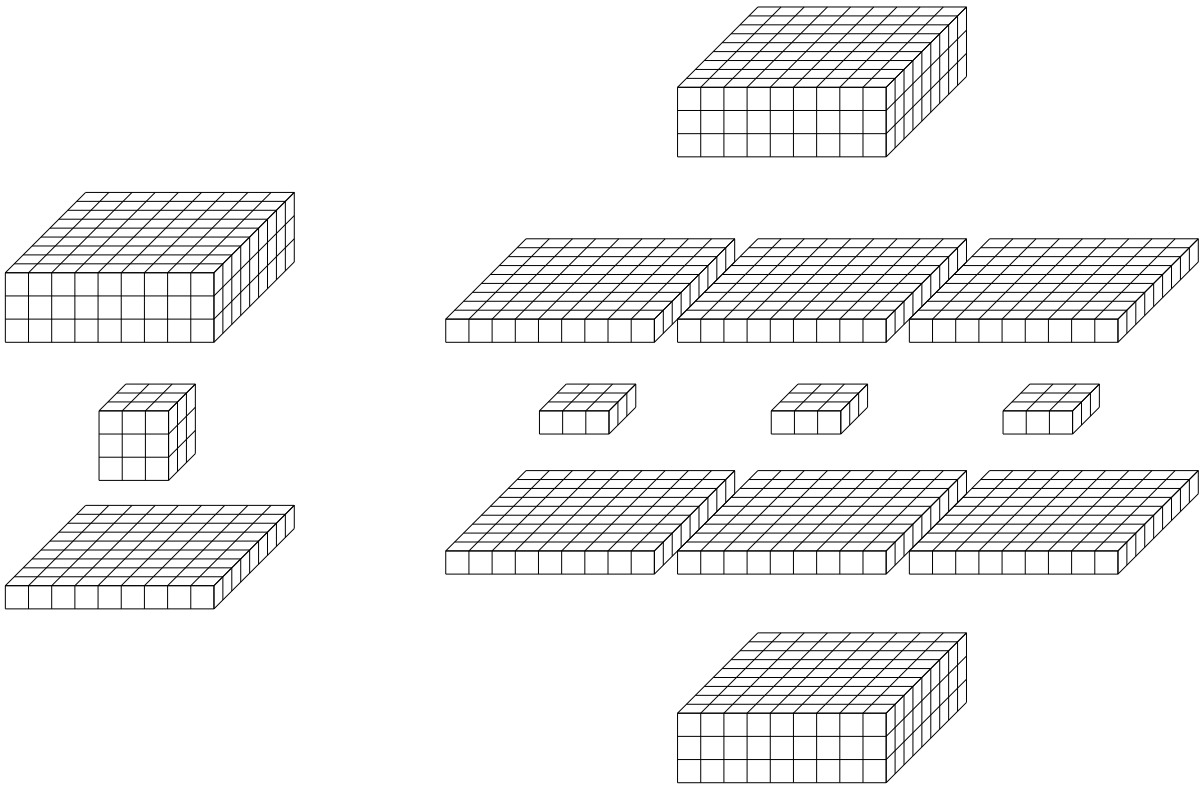


Figure 2.7: Depthwise convolution, where normal convolution is shown in the left part of the image. In the right part of the image the depthwise convolution is shown. The input is divided into three parts and folded separately, and the output is stacked. For the same output size as the depthwise convolution, the conventional convolution in this case would require three times the parameters.

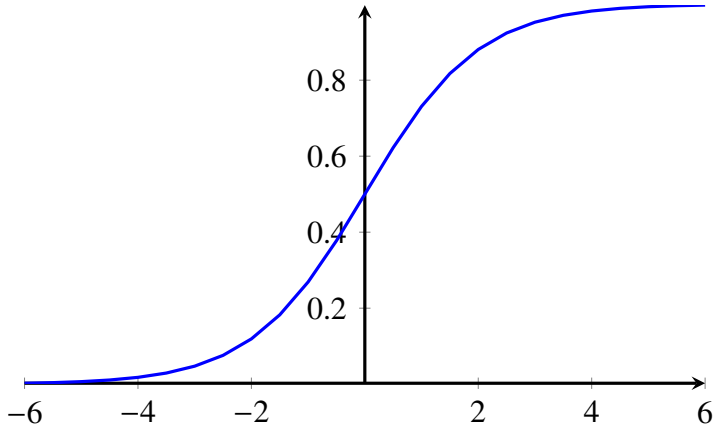


Figure 2.8: Sigmoid function

ReLU The Rectified Linear Unit (ReLU) function

$$\sigma(z) = \begin{cases} 0, & \text{if } z < 0 \\ z, & \text{otherwise} \end{cases} \tag{2.32}$$

is widely used and can be implemented on hardware with ease.

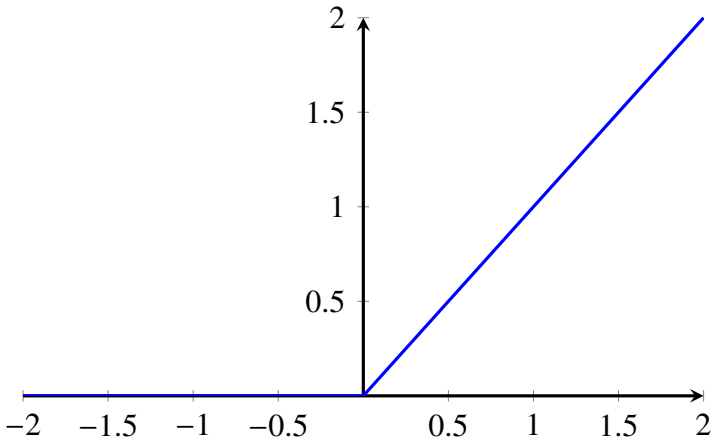


Figure 2.9: The ReLU function

LeakyReLU The LeakyReLU

$$\sigma(z) = \begin{cases} \alpha z, & \text{if } z < 0 \\ z, & \text{otherwise} \end{cases} \tag{2.33}$$

is very similar to the ReLU function with a predefined leak term α.

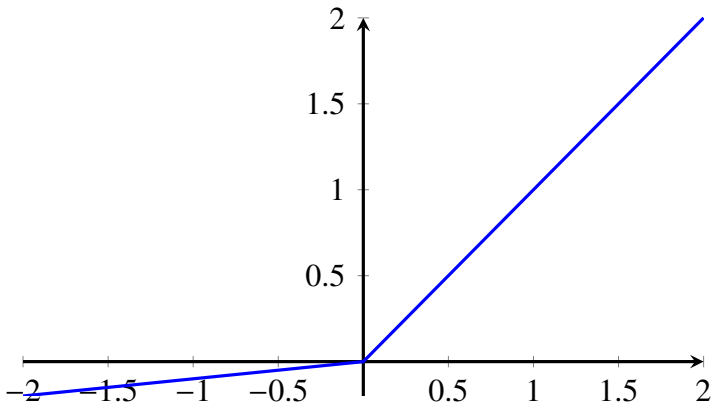


Figure 2.10: The LeakyReLU function

PReLU The Parametic Relu is very similar to the LeakyReLU but the leak term α is made into a learnable parameter.

Swish The Swish function

$$\sigma(z) = \frac{z}{1 + e^{-z}} \tag{2.34}$$

looks similar to the ReLU but is derivable for higher order derivatives [**?**].

Hard-Swish The Hard-Swish function

$$\sigma(z) = x\frac{\text{ReLU6}(x + 3)}{6} \tag{2.35}$$

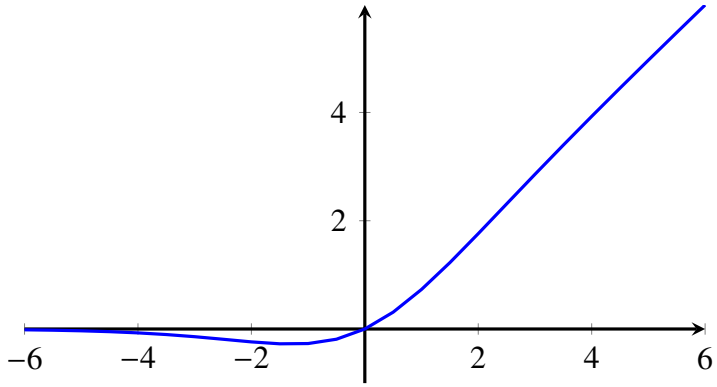


Figure 2.11: The Swish function

$$\text{ReLU6}(z) = \begin{cases} 0, & \text{if } z < 0 \\ z, & \text{if } z \leq z \leq 6 \\ 0, & \text{if } 6 \leq z \end{cases} \tag{2.36}$$

replaces the computationally expensive sigmoid with linear operations [**?**].

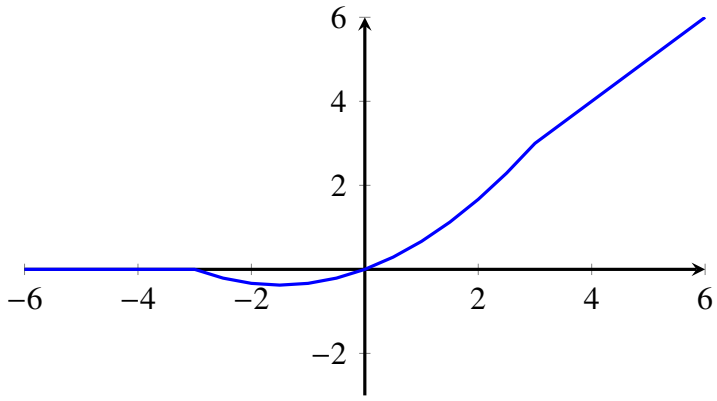


Figure 2.12: The Hard-Swish function

2.3 Hardware for Deep Learning

Deep learning algorithms can be very computationally intensive. For modern networks, the number of operations for a given input can be as high as 500 billion. Different hardware architectures such as Central Processing Units (CPUs), GPUs and FPGAs are used to execute the Deep Learning algorithms. Training of such networks is mostly performed on servers with GPUs. This is due to the fact that all popular Deep Learning frameworks have been developed for GPUs.

However, for inference, the hardware to choose depends heavily on the application. In general, a GPUs is a good and easy solution, but if the application requires a cost- or energy-efficient implementation, other solutions such as CPUs (μ-controllers) or FPGAs should be considered. FPGAs have a wide range of products, from low-cost and low-power to high-performance solutions. Therefore, FPGAs are a valuable option for low-cost implementation as well as high-performance deep learning inference.

CPU A CPU is primarily designed to process a wide range of task. A CPU with its arithmetic logic unit can process deep learning algorithms with ease. However, if one wants to parallelize the algorithms, a CPU very quickly reaches its limits.

GPU A GPU is built to perform repetitive calculations in parallel. Deep learning with its very simple algorithms, mostly a form of matrix multiplication, is ideal for GPUs.

FPGA Compared to CPUs and GPUs, FPGAs are well suited for embedded applications and have lower power consumption. FPGA designs can be built for custom data types and are not limited by architecture like GPUs. Also, the programmability of FPGAs make it easier to adapt them for safety and security concerns [**?**].

2.4 Deep Learning Frameworks

2.4.1 Tensorflow

`TensorFlow` is an end-to-end open source platform for deep learning. Neural network of any form can easily be implemented and trained. It runs both on CPU and GPU.

Following a simple ANN:

```
import tensorflow as tf

inputs = tf.keras.layers.Input()
net = tf.keras.layers.Dense(32, activation='relu')(Input)
net = tf.keras.layers.Dense(64, activation='relu')(Input)
net = tf.keras.layers.Dense(10, activation='softmax')(Input)

model = tf.keras.models.Model(inputs=inputs, outputs=net)
```

A CNN with a ANN for the classification can be written with the following code.

```
import tensorflow as tf

inputs = tf.keras.layers.Input()
net = tf.keras.layers.Conv2D(filters=4, kernel_size=(3,3), activation='relu')(inputs)
net = tf.keras.layers.MaxPooling2D(pool_size=(2, 2))(net)
net = tf.keras.layers.Conv2D(filters=16, kernel_size=(3,3), activation='relu')(net)
net = tf.keras.layers.Flatten()(net)
net = tf.keras.layers.Dense(32, activation='relu')(net)
net = tf.keras.layers.Dense(10, activation='softmax')(net)

model = tf.keras.models.Model(inputs=inputs, outputs=net)
```

2.4.2 Vitis AI

There are several possible methods to run inference algorithms on Xilinx FPGAs.

- Use the provided framework Vitis AI
- Write high-level code for the inference algorithms and synthesize it for the FPGA with the High-Level-Synthesis
- Describe the hardware with low level Hardware Description Language (HDL) such as `Very High Speed Integrated Circuit Hardware Description Language (VHDL)` or `Verilog`

For this work Vitis AI is used.

Vitis AI framework accelerates AI inference on Xilinx FPGAs. The training is preferably done on GPUs. Vitis AI is composed of optimized IP cores, tools, libraries and example designs. With is high abstraction level Vitis AI can be used without any FPGA hardware knowledge.

Vitis AI:

- *Supports mainstream frameworks and the latest models capable of diverse deep learning tasks.*
- *Provides a comprehensive set of pre-optimized models that are ready to deploy on Xilinx devices.*
- *Provides a powerful quantizer that supports model quantization, calibration, and fine tuning. For advanced users, Xilinx also offers an optional AI optimizer that can prune a model by up to 90 % with a tolerable accuracy loss.*
- *Provides layer-by-layer analysis to help with bottlenecks.*
- *Offers unified high-level C++ and Python APIs for maximum portability from Edge to Cloud.*
- *Customizes efficient and scalable IP cores to meet your needs for many different applications from a throughput, latency, and power perspective.* [**?**]

2.4.2.1 Usage of Vitis AI

In Fig. 2.13 is the general work-flow illustrated.

- Generate the network in your deep learning framework of choice (`Tensorflow, Caffe, PyTorch`)
- Optimize the model (network pruning) with the *AI Quantizer*. This requires a commercial license.
- Quantize the trained model to `INT8`
- Compile the model for a specific DPU
- Generating the DPU and a FPGA hardware platform
  - Use a pre-defined Linux image for given platforms
  - Build a custom platform either with Vitis or Vivado
- Programming the DPU with VART or the Vitis AI Library
  - With `C++` or `Python` API

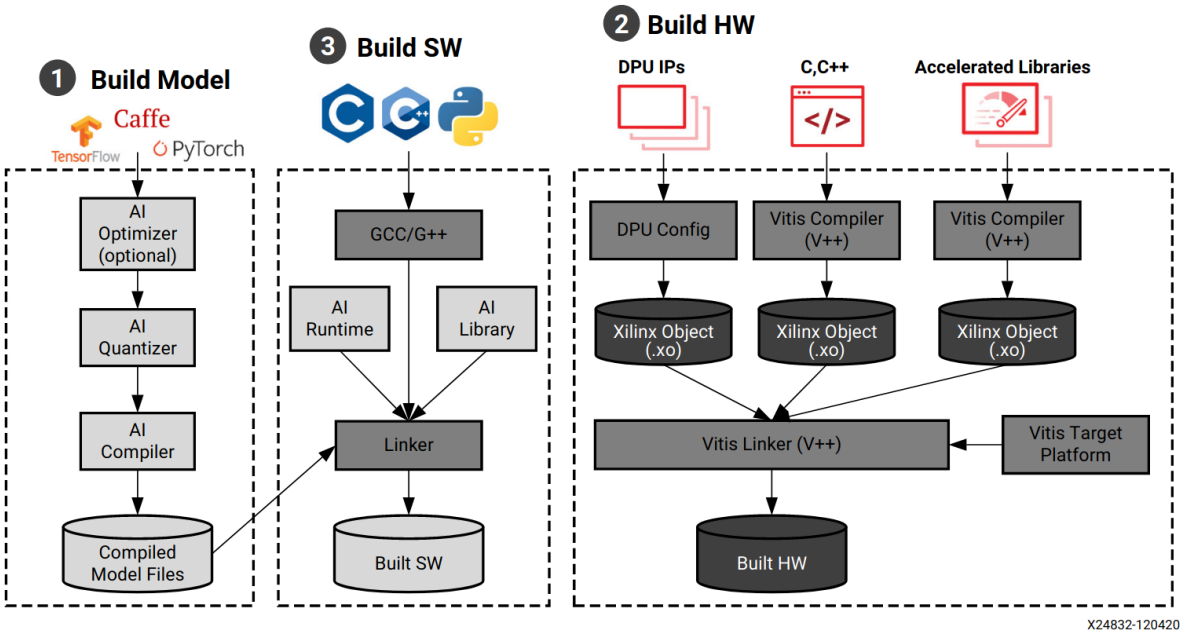


Figure 2.13: The deep learning model is built using the framework of choice. The AI quantizer quantizes the `float32` weights and biases to `INT8`. With the information about the FPGA design, the AI compiler can compile the deep learning network for the FPGA. `C/C++` or `Python` is then used to build the software to run the network on the DPU via the μ-controller on the FPGA. The hardware can be built using either Vitis or Vivado, pre-built Linux images can be downloaded for quick prototyping.

2.4.2.2 Deep-Learning Processor Unit

"DPU is a programmable engine optimized for deep neural networks. It is a group of parameterizable IP cores pre-implemented on the hardware with no place and route required. It is designed to accelerate the computing workloads of deep learning inference algorithms widely adopted in various computer vision applications, such as image/video classification, semantic segmentation, and object detection/tracking. The DPU is released with the Vitis AI specialized instruction set, thus facilitating the efficient implementation of deep learning networks." [**?**]

A a complete system with the DPU on a FPGA could look similar to the one shown in Fig. 2.14. One can see, that the usage of a processor is necessary for a application with the DPU.

2.5 Human Pose Estimation

In this work we focus on single person HPE. For detailed information about the deep learning HPE networks conduct [**?**]. The output of such a network are 15 confidence heatmaps one for each keypoint (Fig. 2.16). In figure Fig. 2.15 on the right all 15 confidence maps are summed up. To find the relevant keypoint on a single confidence map, the coordinates of the max value has to be found. With that information the pose estimation can be drawn onto the original image.

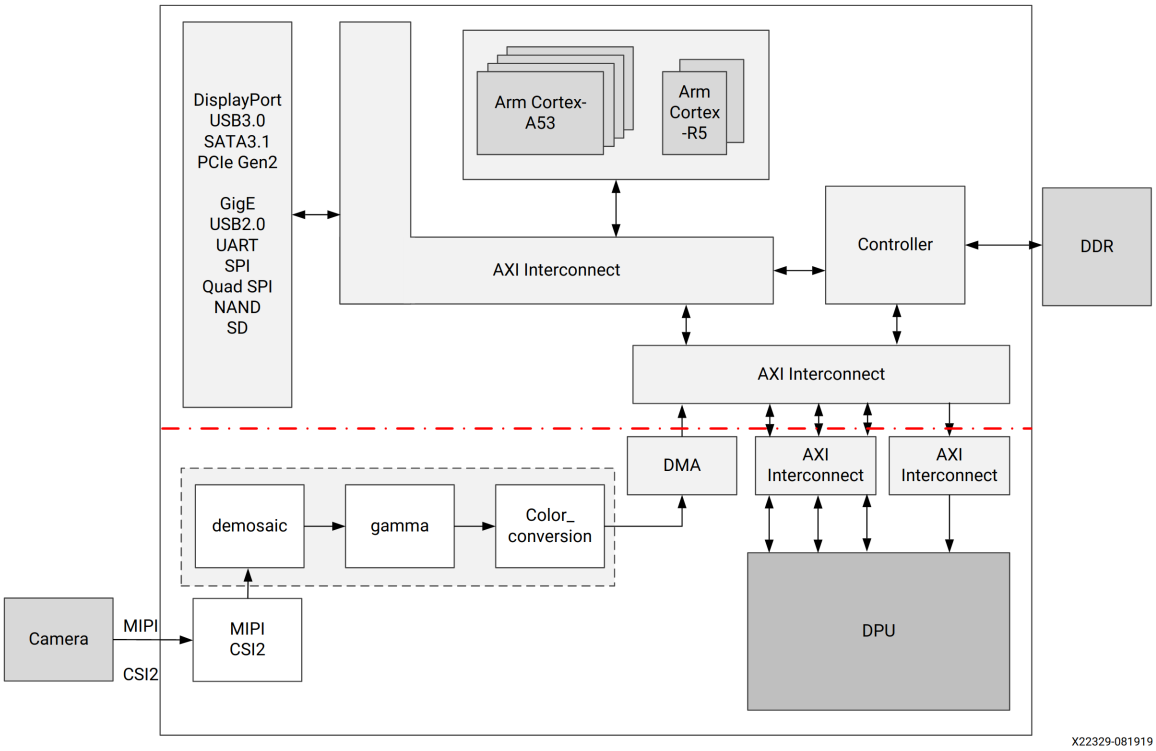


Figure 2.14: A complete deep learning system on a FPGA with Vitis AI and a DPU. A camera interface is shown. The data flow is all controlled by a processor with the *AXI-Interface* in combination with the DDR

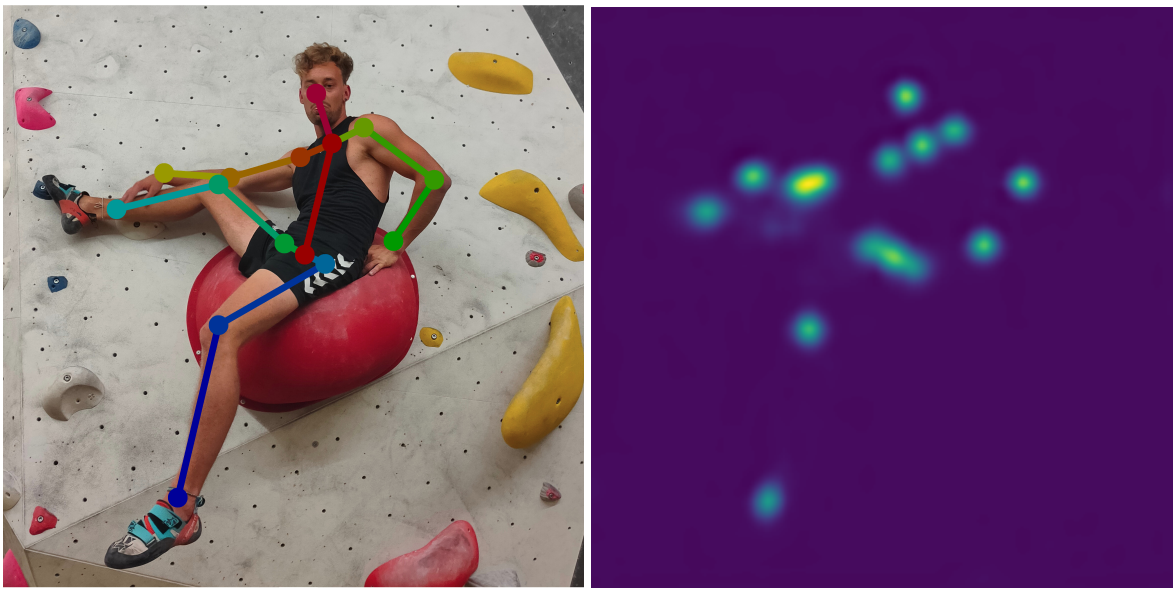


Figure 2.15: Left: input image with the drawn pose. On the right all 15 confidence maps summed up

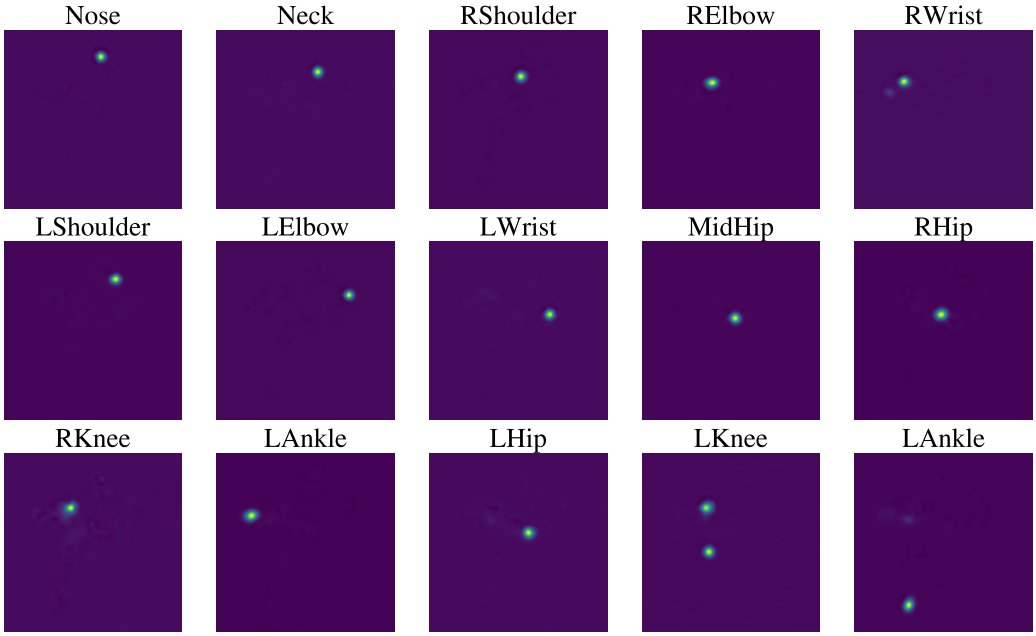


Figure 2.16: The 15 confidence output maps from a HPE network. Each map is related to a certain body part, where the coordinates of the max value represent the highest probability for that body part. One can clearly see, that the network was not quite sure if it has detected the left or the right knee. But if one compares to the input image in Fig. 2.15 the network detected everything correctly

Chapter 3

Methods

The main part of this project is divided into three portions. The first one is the implementation of the camera interface for the two FPGAs hardware platforms. The second part is to run a typical computer vision algorithm using this camera interface. Finally, a deep learning network developed in-house by OST shall be ported from a typical GPU implementation to both FPGA platforms.

3.1 Camera Interface

This section explains how to use the cameras on the FPGAs evaluation boards. The most important thing is to use the camera interface alongside with Vitis AI.

3.1.1 GStreamer

"`GStreamer` is a library for constructing graphs of media-handling components. The applications it supports range from simple playback, audio and video streaming, to complex audio (mixing) and video (non-linear editing) processing [?, p. 40]."

On Xilinx FPGAs `GStreamer` can be installed on PetaLinux . `Gstreamer` can be used with terminal commands or in various programming languages such as `Python` or `C/C++`.

Following `bash` command reads a video stream from source `/dev/video0` and prepares it to be processed by an application.

Listing 3.1: Example GStreamer string

```
gst-launch-1.0 v4l2src device=/dev/video0 io-mode=4 ! \
video/x-raw, width=3840, height=2160, format=NV12, framerate=60/1 ! \
queue ! appsink
```

3.1.2 VCK190

The Versal VCK190 is an evaluation kit that uses the new ACAP with AI cores. The need for the AI engine arose from computationally intensive applications that use Digital Signal Processors (DSPs) and AI. DSPs are typically used with a HDL, which can be very time consuming to develop. AI engines are expected to have a higher level of productivity with a higher level of

abstraction. In Fig. 3.1 the main structure of a High Performance Computing (HPC) system and in Fig. 3.2 the implementation in Xilinx' ACAP.

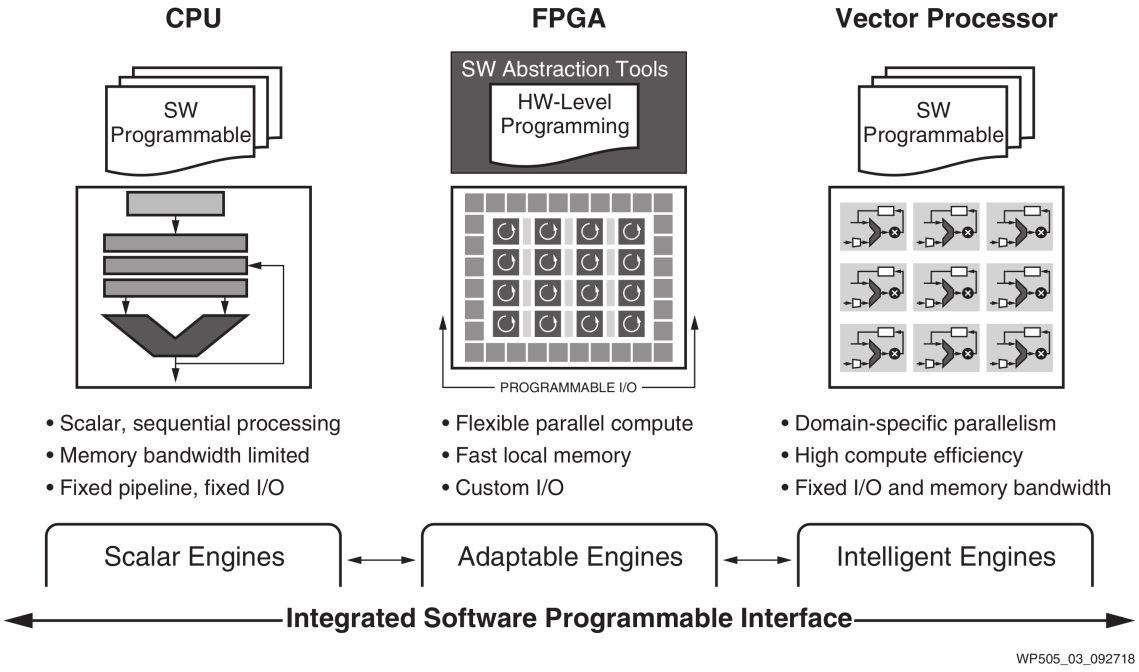


Figure 3.1: The three main parts of a modern FPGA system [**?**]

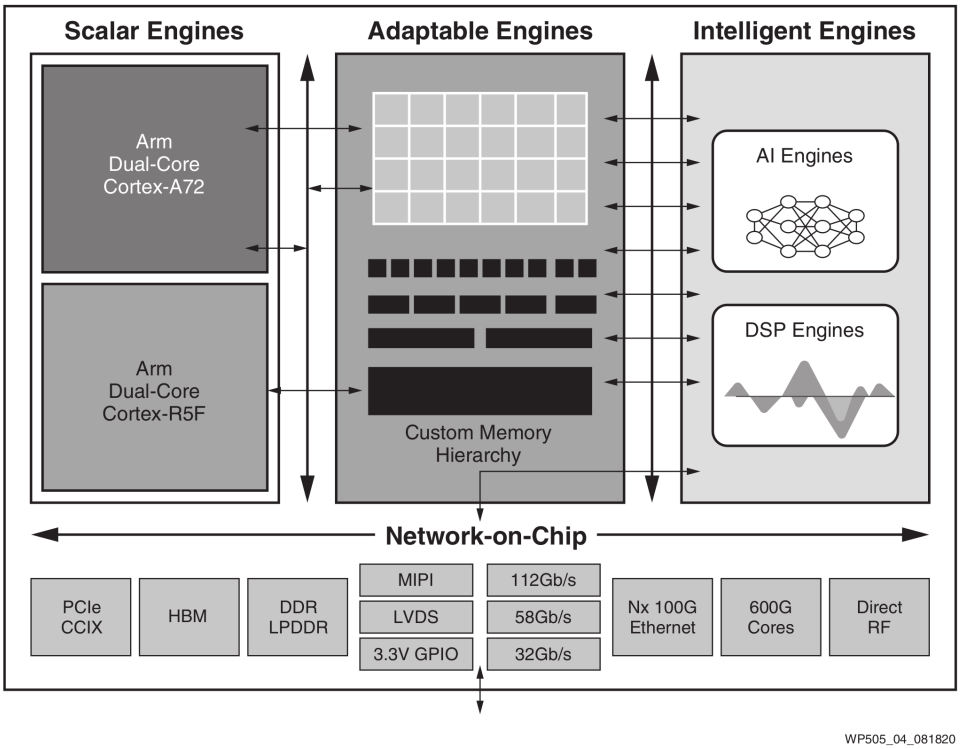


Figure 3.2: Block diagram of an ACAP [**?**]

Xilinx promises four primary benefits with AI Engines.

- *Deliver three to eight times more compute capacity per silicon area versus PL implementation of compute-intensive applications*
- *Reduce compute-intensive power consumption by 50 % versus the same functions implemented in PL*
- *Provide deterministic, high-performance, real-time DSP capabilities*
- *Dramatically improve the development environment and deliver greater designer productivity* [**?**]

All three elements of an ACAP, Programmable Logic (PL), Programmable System (PS) and AI engines can be programmed with `C/C++`.

camera For the VCK190 platform, the LI-IMX274MIPI-FMC camera was purchased from Leoparding Imaging inc. It uses MIPI as the video interface, an FMC connector and can deliver 4K at up to 60 fps.

hardware platform System on Chip (SoC) designs become quite complex when using a camera interface. Developing each project from scratch would take an fair amount of time, especially in the prototyping phase. Xilinx provides a large number of pre-built designs for various applications as part of its open source policy. For the VCK190, a Target Reference Design (TRD) with various platforms is made available via GitHub [**?**].

The Versal Base TRD consists of a series of platforms, accelerators, Jupyter notebooks and applications targeting the VCK190 evaluation board. A platform is a Vivado design with a pre-instantiated set of I/O interfaces and a corresponding PetaLinux BSP and image that includes the required kernel drivers and user-space libraries to exercise those interfaces. Accelerators are mapped to FPGA logic resources and/or AI Engine cores and stitched into the platform using the Vitis unified software platform toolchain.

The platform we use for the chosen camera:

Mipi Rx Single Sensor and HDMI Tx Platform: This Vitis platform captures video from either a file source, USB webcam, or an image sensor using the IMX274 MIPI FMC Module and displays it on a HDMI monitor. Along with video, audio from a file can be replayed using the HDMI transmitter. Accelerator functions can be added into this platform using the Vitis acceleration flow.

Fortunately, Vitis AI for the VCK190 uses this TRD and thus the camera interface for the LI-IMX274MIPI-FMC camera.

There are now two possible ways to use this platform: [**?**]

- Either download the prebuilt Linux image for the vck190
- Or built the platform from scratch with given scripts under *Vitis-AI/dsa/XVDPU-TRD*
  - The configurations of the DPU can be adapted
  - With this option own hardware designs can easily be added

With this platform Vitis AI as well as the camera can be used in its full extent. In table Table 3.1 is the hardware utilization for the used hardware platform shown.

Resource	Utilization	Available	Utilization %
LUT	382665	899840	38
FF	503538	1799680	28
BRAM	238	967	25
URAM	334	463	72
DSP	1228	1968	62
AI Cores	96	400	24 %

Table 3.1: Hardware utilization camera interface for the VCK190

GStreamer The camera can be accessed by the `GStreamer` string in Listing 3.2. More on that in the section 3.1.3

Listing 3.2: GStreamer string for the VCK190

```
gst-launch-1.0 mediasrcbin media-device=/dev/media1 name=channel  channel. ! video/x-raw,
    width=3840, height=2160, format=BGR, framerate=60/1
```

3.1.3 Kria KV260 Vision AI Starter Kit

The Kria KV260 is an evaluation kit with the K26 System on Module (SoM).

The SoM integrates a Zynq UltraScale+, runtime memory and a power supply solution, it basically provides everything that is really needed to run the SoC. If you want to implement a SoM in a custom design, you just need to design the required peripherals and a power source on a Printed Circuit Board (PCB). Designing the PCB for the SoC and memory would be by far the most time consuming part, so using a SoM can save time.

In Fig. 3.3 the block structure of KV260 is shown, where all hardware parts of the SoM can be seen, as well as all implemented peripherals on the carrier board.

IAS Interface The Image Access System (IAS) interface implements OnSemi IAS camera module interface supporting four MIPI lanes. MIPI is a simple protocol for point to point image and video transmission between cameras and host devices [**?**]. The IAS interface are connected directly to the Zynq UltraScale+ MPSoC.

The AR1335 camera was included with the KV260, it delivers a video stream up to 4k, 30 fps.

Hardware Platform Similar to the VCK190, a predefined hardware platform is used. The pre-built Vitis AI Linux image for the KV260 does not include the camera interface, unlike the

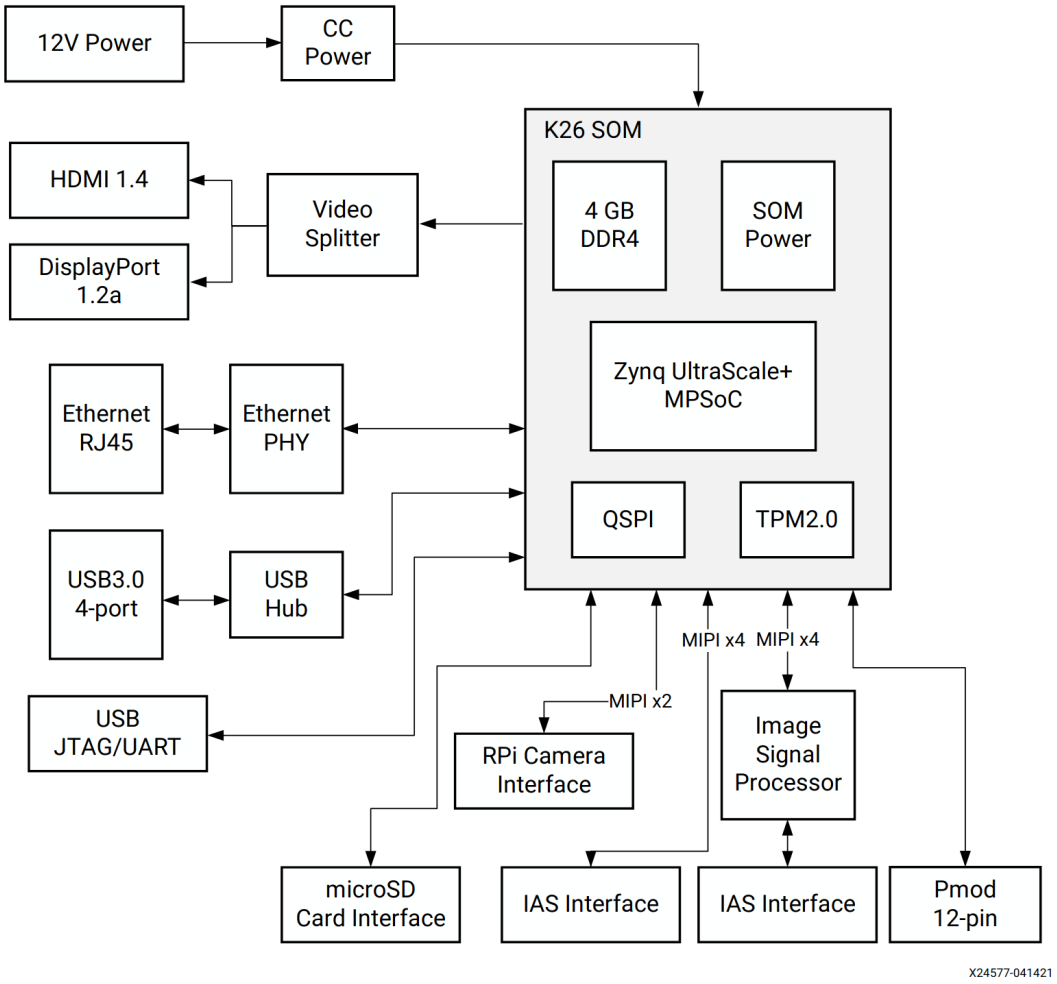


Figure 3.3: Block design of the Kria KV260 Vision AI Starter Kit [**?**]

one for the VCK190. However, an AI example design platform for a Vitis AI demonstration on the KV260 contains everything.

The Vitis AI libraries needed to run VART can then be installed within the Linux system on the KV260. In this procedure, one must make sure that all versions match each other.

- The DPU for the KV260 board is based on the Petalinux 2020.2 with Vitis AI 1.4.0 [**?**]
- The smart camera example platform is based on Petalinux 2021.1 [**?**]
- Therefore, the two Petalinux versions are not identical, but the previous version of the smart camera example uses Petalinux 2020.2
- The 2020.2 smart camera version uses Vitis AI 1.3.0, one has to keep this in mind.

In Table 3.2 the hardware utilization for the used hardware platform is shown.

GStreamer While running the smart camera example on the KV260 the corresponding C++ code generates a large GStreamer string, similar to the Listing 3.1. This string launches the whole application with video capture, preprocessing, deep learning inference and displaying the output over HDMI. The exact string can be seen in Listing 3.3.

Resource	Utilization	Available	Utilization %
LUT	81198	117120	69
FF	135276	234240	58
BRAM	102.5	144	71
URAM	52	64	81
DSP	583	1248	47

Table 3.2: Hardware utilization camera interface for the KV260

Listing 3.3: GStreamer string from the smart camera app on the KV260

```
gst-launch-1.0 mediasrcbin media-device=/dev/media0 v4l2src0::io-mode=dmabuf v4l2src0::stride
    -align=256  ! video/x-raw, width=1920, height=1080, format=NV12, framerate=30/1  ! tee
    name=t  ! queue ! ivas\_xmultisrc kconfig="/opt/xilinx/share/ivas/smartcam/facedetect/
    preprocess.json"  ! queue ! ivas\_xfilter kernels-config="/opt/xilinx/share/ivas/smartcam
    /facedetect/aiinference.json"  ! ima.sink\_master ivas\_xmetaaffixer name=ima ima.src\
    _master ! fakesink  t. ! queue max-size-buffers=1 leaky=2 ! ima.sink\_slave\_0 ima.src\
    _slave\_0  ! queue ! ivas\_xfilter kernels-config="/opt/xilinx/share/ivas/smartcam/
    facedetect/drawresult.json"  ! queue ! kmssink driver-name=xlnx plane-id=39 sync=false
    fullscreen-overlay=true  ! queue ! kmssink driver-name=xlnx plane-id=39 sync=false
    fullscreen-overlay=true  ! queue ! ivas\_xfilter kernels-config="/opt/xilinx/share/ivas/
    smartcam/facedetect/drawresult.json
```

The first part of listing, shown in Listing 3.4 is responsible for the video capture on the KV260.

Listing 3.4: GStreamer string for the video capture on the KV260

```
gst-launch-1.0 mediasrcbin media-device=/dev/media0 v4l2src0::io-mode=dmabuf v4l2src0::stride
    -align=256  ! video/x-raw, width=1920, height=1080, format=NV12, framerate=30/1
```

With the `Python` program, shown in Listing 3.5 the video stream from the IAS camera can be displayed.

Listing 3.5: `Python` program for displaying a video stream from the camera

```
import cv2

cap = cv2.VideoCapture("mediasrcbin media-device=/dev/media0 v4l2src0::io-mode=dmabuf
    v4l2src0::stride-align=256  ! video/x-raw, width=100, height=100, format=NV12,
    framerate=30/1 ! videoconvert! appsink")

while(True):
    ret, frame = cap.read()

    cv2.imshow('frame', frame)

    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

cap.release()
cv2.destroyAllWindows()
```

3.2 Computer Vision Application

The Vitis AI Model Zoo, is a repository of pretrained models for deep learning inference on FPGAs with Vitis AI [**?**]. Various models with different are available.

Naming Rules *Model name:* `F_M_(D)_H_W_(P)_C_V`

- `F` *specifies training framework:* `cf` *is Caffe,* `tf` *is Tensorflow,* `tf2` *is Tensorflow 2,* `dk` *is Darknet,* `pt` *is PyTorch*
- `M` *specifies the model*
- `D` *specifies the dataset. It is optional depending on whether the dataset is public or private*
- `H` *specifies the height of input data*
- `W` *specifies the width of input data*
- `P` *specifies the pruning ratio, it means how much computation is reduced. It is optional depending on whether the model is pruned or not*
- `C` *specifies the computation of the model: how many Gops per image*
- `V` *specifies the version of Vitis AI*

For example, `cf_refinedet_coco_360_480_0.8_25G_1.4` *is a RefineDet model trained with* `Caffe` *using COCO dataset, input data size is 360*480, 80 % pruned, the computation per image is 25 Gops and Vitis AI version is 1.4.*

For each network, one can download either the float/quantize or the compiled model for a specific hardware platform. With the float/quantize models, one can quantize and compile the model for a custom DPU implementation.

Demo application with Vitis AI Library The Vitis AI Library provides demo applications for several models from the Model Zoo [**?**].

After the complete setup of the Vitis AI Library on the board and host computer, there is a specific way to run the demos on the FPGA. The C++ applications are built with a *host cross compiler* for the device on the host computer. Depending on the Petalinux version a different *host cross compiler* has to be used. The compiled executable are then copied to the device.

There are different applications per model (OpenPose as example).

- `test_jpeg_openpose`
  - One input image is processed with this program and the result is shown
- `test_performance_openpose`
  - With a list of images, the performance of the model is computed
- `test_video_openpose`

  - Either a video file or are video stream can be processed and displayed with this program
  - A GStreamer string can be used for the video stream

- `test_accuaracy_openpose`

  - To test the model accuracy, prepare your own image dataset, image list file and the ground truth of the images

3.2.1 Examples

Following two HPE models have been tested on both FPGA platforms: Because the prebuilt hardware platform for the VCK190 is used, the model could be directly be downloaded from the model zoo and did not needed to be re-compiled. Whereas on the KV260 a different DPU as in the prebuilt image is used, the models had to be compiled for the according DPU

- OpenPose: `cf_openpose_aichallenger_368_368_0.3_189.7G_1.4`
  - VCK190
    - 13 fps with camera interface
    - 21 fps without camera interface
  - KV260
    - 4 fps with camera interface
    - 4 fps without camera interface

- Hourglass: `cf_hourglass_mpii_256_256_10.2G_1.4`
  - VCK190
    - 36 fps with camera interface
    - 83 fps without camera interface
  - KV260
    - 30 fps (camera limit) with camera interface
    - 39 fps without camera interface

3.3 Implementation of ICAIPose

ICAIPose developed by an fellow engineer at the ICAI at OST, is a neural network for HPE [**?**]. The main goal of this part is to use a deep learning model developed using standard methods for a GPU and export it for execution on an FPGA.

3.3.1 Supported Layers for the DPU

A neural network consists of different layers, such as convolutional layers, fully connected layers or activation layers. Vitis AI implements with the DPU a hardware platform that can compute these layers. Due to the limitation of the architecture of an FPGA, not all layers are supported by the DPU. Also, Vitis AI works slightly differently for each deep learning framework, i.e., a layer may be supported by `PyTorch` but not by `Tensorflow`. The user has to check the supported layers for his framework, sometimes it is necessary to change the framework. In the Vitis AI user guide all supported layers for each framework are listed [**?**] .

3.3.2 ICAIPose

ICAIPose has about 11 million learnable parameters in the configuration used and uses 103 GOps per input image. The input is an RGB image with dimensions 256x256. Although it is a fairly large network, it consists of only six layers:

- Conv2D
  - Zero padding to the input is applied, the output of the convolution has the same dimension as the input
  - Kernel size is 3x3
- PReLU
- Concatenate
- UpSampling2D
  - 2x2 upsampling
- DepthwiseConv2D
- MaxPooling2D
  - 2x2 MaxPooling

ICAIPose is written in the `Tensorflow 2` framework.

Problems with PReLU The product guide for the DPU of the Versal ACAP states that Parametric Rectified Linear Unit (PReLU) is a supported activation function [**?**]. Unfortunately, it was not possible to get Vitis AI to work with the PReLU layer on different frameworks. This issue was reported on their GitHub page.

LeakyReLU Instead of the activation function PReLU, the very similar LeakyReLU, which is supported by all DPUs (Versal and Zynq Ultrascale+) is used.

While testing the LeakyReLU activation function, a bug was found. The Vitis AI compiler for `Tensorflow 2` cannot compile the LeakyReLU for the DPU, so it wants to compute it on the CPU. With `Tensorflow 1` the compilation works fine. This bug has been reported on the Vitis AI GitHub repository.

Vitis AI provides not only typical deep learning functions, but also some general mathematical functions. To avoid the switch from TF2 to TF1, an attempt was made to compile the leakyReLU with other working functions. In Fig. 3.4 the calculation graph of the workaround is shown.

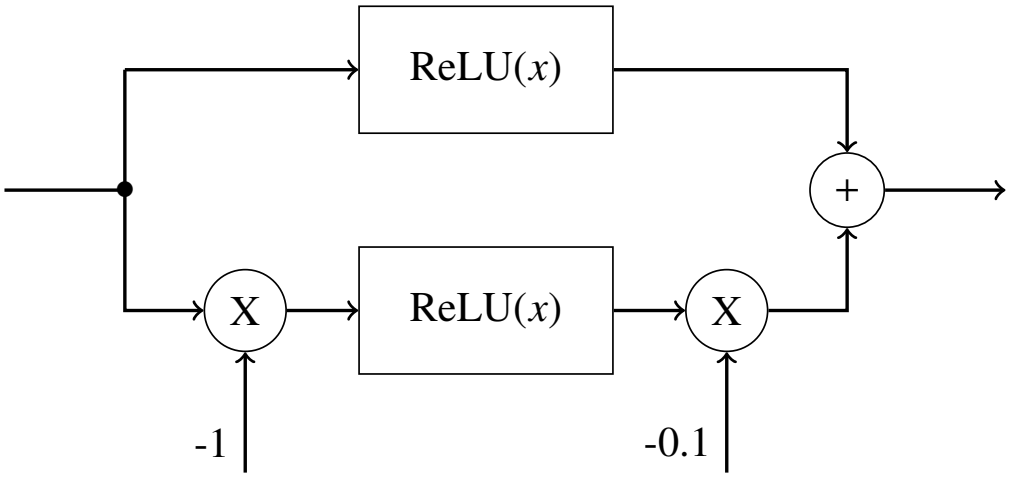


Figure 3.4: LeakyReLU composed from the normal ReLU function, multiplications with a constant and the addition of two Tensors

The multiplication by a constant has shown to be a problem for Vitis AI. The appropriate function for the multiplication, the Lambda function, Listing 3.6 is not supported by Vitis AI.

Listing 3.6: TF2 Lambda Function

```
output = tf.keras.layers.Lambda(lambda x: x * scale)(input)
```

The supported multiplication layer (Listing 3.7) multiplies two tensors. If one wants to multiply a tensor with a scalar, the scalar needs to be a tensor with the same shape as the other one.

Listing 3.7: TF2 Lambda Function

```
output = tf.keras.layers.Multiply()([input1, constTensor])
```

These constant tensors generate a huge amount of redundant data, so the workaround is not applicable. The change to TF1 was unavoidable.

3.3.2.1 Quantization and Compiling

The ICAI delivers a trained version of ICAIPose with its weights and biases. The PReLU activation function of the original network was replaced by the LeakyReLU function and the network was re-trained in `Tensorflow 2`.

In the following bullet points are the steps shown, how to get the pre-trained network to a state in which it can be used with Vitis AI.

- Export the weights from the TF2 network.

  - `model.save_weights('weights.h5')`

- Import the weights in the same network but written in `TF1` (`Keras`)

  - `model.load_weights('weights.h5')`

- Convert the `h5`-file into a `Tensorflow1` checkpoint and protobuf file
- The normal Vitis AI flow for `Tensorflow1` can now be used

For the quantization a set of 100-1000 images without the ground truth have to be passed to the Vitis AI quantizer. The network is saved with the newly `int8` weights and biases and can be used by the compiler. For every DPU configuration a hash-fingerprint is generated, the compiler knows with that hash for which DPU it has to compile the network.

Quantization has been shown to be a fairly robust process. Quantization can be problematic for some specific network architectures. For the ICAIPose network, quantization was initially performed with some random images. One might think that quantization with the same test data set as the benchmarking test set should increase performance. It turned out that this did not significantly change the performance in this case.

3.3.3 Running on a FPGA

Vitis AI provides with the VART a simple way to control the DPU from the CPU with `Python`. The video processing is done with the `OpenCV`-library. The program is built around two *First in First Out*-memory structures, the `queueIn` FIFO and the `queueOut` FIFO. A multithreaded program controls the three main tasks of the system:

- TaskCapture: Capturing the video stream an saves the frames for the DPU into `queueIn`
- TaskWorker: Getting the next frame from the FIFO fetches it after the pre-processing to the DPU. With the result of the DPU the pose can be drawn over the input image. The result is stored into `queueOut` FIFO
- TaskDisplay takes the frames from `queueOut` and displays them

Conficence Threshold The confidence map shows where the network assumes a keypoint to be, the higher the probability, the higher the amplitute. One can set a certain threshold so that a keypoint with a low probability is not assigned to a keypoint. This helps in detecting the pose when, for example only the upper body is presented to the network. For the `INT8` network this threshold is a different one than for the `float` model, because the confidence map has a different distribution.

Chapter 4

Results

Based on the successful implementation of the ICAIPose neural network on both hardware platforms, the throughput performance and the overall quality of the HPE can be discussed. Due to the quantization of the neural network some kind of loss is expected in contrast to a `float32` network.

Following DPU-configuration has been used for this tests:

- VCK190
  - DPUCVDX8G_ISA1_C32B3
    - This results in a peak performance of 30.72 TOps
  - DPUCVDX8G_ISA1_C64B3
    - This results in a peak performance of 61.44 TOps
    - This implementation did not improve performance and is therefore not included in the following Table 4.1.
- KV260
  - DPUCZDX8G_ISA0_B3136_MAX_BG2
  - B3136 means that the DPU has a peak operation of 3136 per clock cycle
  - With a typical clock frequency of 350 MHz this would result in a peak performance of 1.097 TOps

4.1 Throughput Performance

In Table 4.1 the throughput results can be seen. The results from the Jetson Xavier NX has been provided by the ICAI

As an additional information the DPU without the camera interface (on the VCK190 with the 256×256 network), process 46 fps.

Hardware Platform	Network	GOps per image	Throughput
VCK190	ICAIPose 256 × 256	103 GOps	27 fps
KV260	ICAIPose 256 × 256	103 GOps	8 fps
Jetson Xavier NX	ICAIPose 256 × 256	103 GOps	8 fps
VCK190	ICAIPose 200 × 200	63 GOps	40 fps
KV260	ICAIPose 200 × 200	63 GOps	12 fps

Table 4.1: Throughput ICAIPose with 256 × 256 and 200 × 200 input size

4.2 Human Pose Estimation Performance

To compare the performance of the network in the sense of HPE a dataset with 2142 images has been used. Some example of the dataset with its ground truth (confidence map) are shown in Fig. 4.1. Three networks are compared to the ground truth:

- ICAIPose with the PReLU activation function
  - This was the original ICAIPose network, only due to the limitations of the DPU has the LeakyReLU activation function been introduced. To see if some performance was lost with the new activation function, this network is also compared to the ground truth.
- ICAIPose with the LeakyReLU activation function
- ICAIPose with the LeakyReLU activation function and `Int8` quantization

In Fig. 4.2 an example image with the applied HPE of all networks is shown.

4.2.1 $PCK_h@\tau$ Metric

A common metric is the $PCK_h@\tau$, the description about this metric from [**?**]:

$PCK_h@\tau$ is defined as the fraction of predictions residing within a distance τl from the ground truth location (see Fig. 4.3). l is 60 % of the diagonal d of the head bounding box, and τ the accepted percentage of misjudgment relative to l. [**?**]

4.2.2 Key Points

Another logical metric is to compare the key points of the detected human pose to the one in the ground truth. This is relative similar to section 4.2.1 but needs less information.

With

$$\overline{d} = \frac{1}{n}\sum_{i=0}^{n}\sqrt{\left(x_i^{(\text{truth})} - x_i^{(\text{network})}\right)^2 + \left(y_i^{(\text{truth})} - y_i^{(\text{network})}\right)^2} \tag{4.1}$$

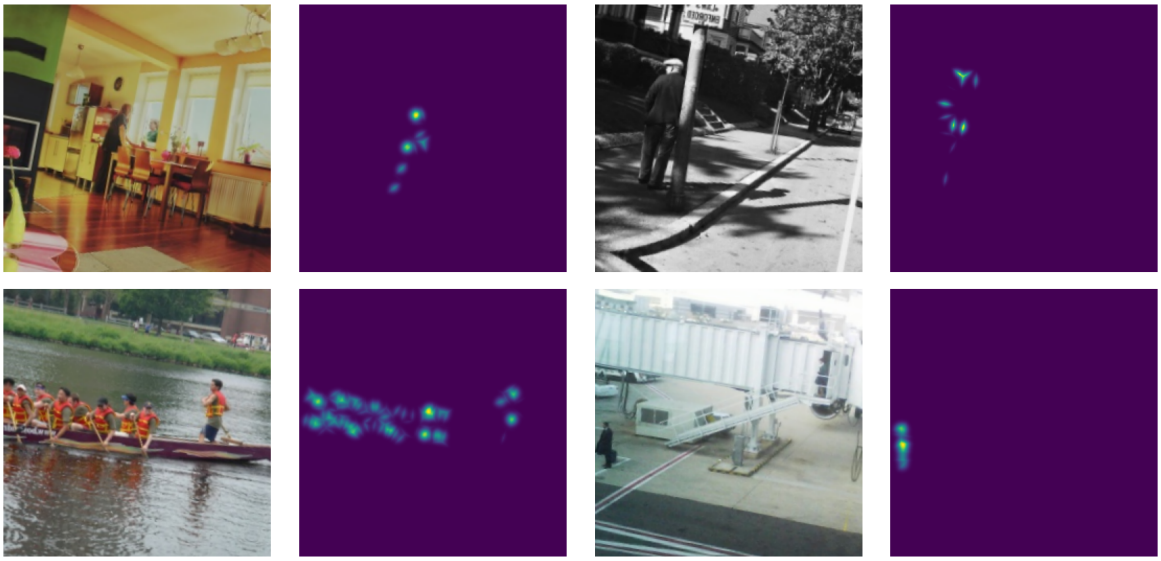


Figure 4.1: Left: images of the dataset, Right: the corresponding confidence map

Figure 4.2: An Example of the dataset with the pose estimated by all networks and the ground truth

the mean distance between the keypoints is computed, where x and y are the coordinates.

The problem with this metric is in the data set. The algorithm used to determine the pose from the confidence map only works for a single person, but the dataset contains many images with multiple people. In Fig. 4.2 this metric could be applied, but in Fig. 4.4 it would not give

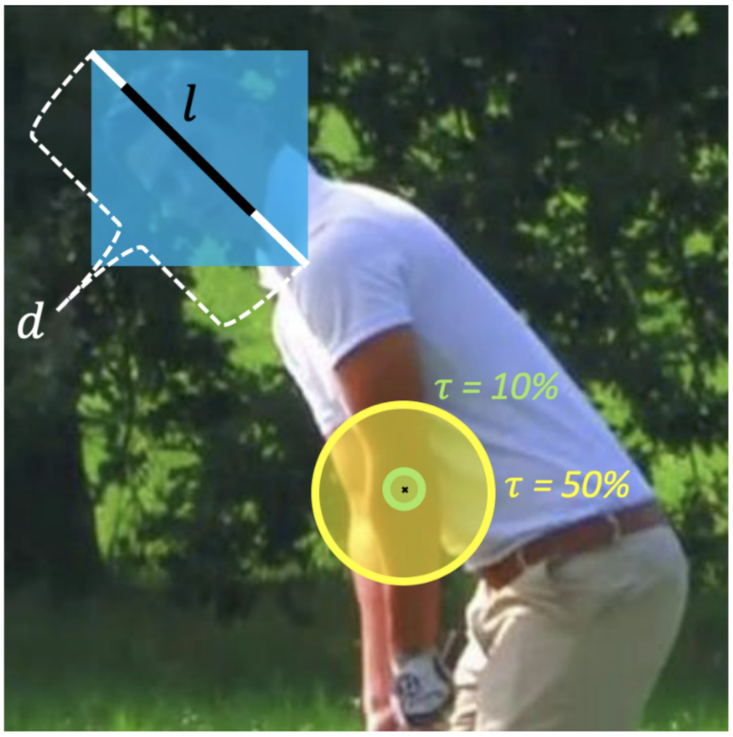


Figure 4.3: The evaluation metric $PCK_h@\tau$, with $PCK_h@50$ (yellow) and $PCK_h@10$ (green) [**?**]

a meaningful result.

Figure 4.4: An Example of a image from the dataset with two people. Because the algorithm just detects the maximum in the confidence map not all keypoints are necessarily assigned to the same person

4.2.3 Mean Squared Error

A possible way to analyze the performance is to directly to compare the confidence map from the ground truth to the confidence map of the network under test.

The mean squared error can be computed pixel-wise

$$\text{MSE} = \frac{1}{n^2} \sum_{i=0}^{n} \sum_{j=0}^{n} \left(X_{i,j}^{(\text{truth})} - X_{i,j}^{(\text{network})} \right) \tag{4.2}$$

where $X_{i,j}$ is the pixel of the ground truth confidence map at position i, j. The confidence map $X^{(j)}$ has to be standardized with

$$\hat{X}^{(j)} = \frac{X^{(j)} - \mu^{(j)}}{\sigma^{(j)}}, \tag{4.3}$$

where

$$\mu^{(j)} = \frac{1}{n^2} \sum_{i=0}^{n} \sum_{j=0}^{n} X_{i,j}^{(j)} \tag{4.4}$$

is the mean and

$$\sigma^{(j)} = \sqrt{\frac{1}{n^2} \sum_{i=0}^{n} \sum_{j=0}^{n} \left(X_{i,j}^{(j)} - \mu^{(j)} \right)^2} \tag{4.5}$$

is the standard deviation. In Fig. 4.5 a example is shown of the mean squared error (pixelwise) on the `INT8` LeakyReLU network.

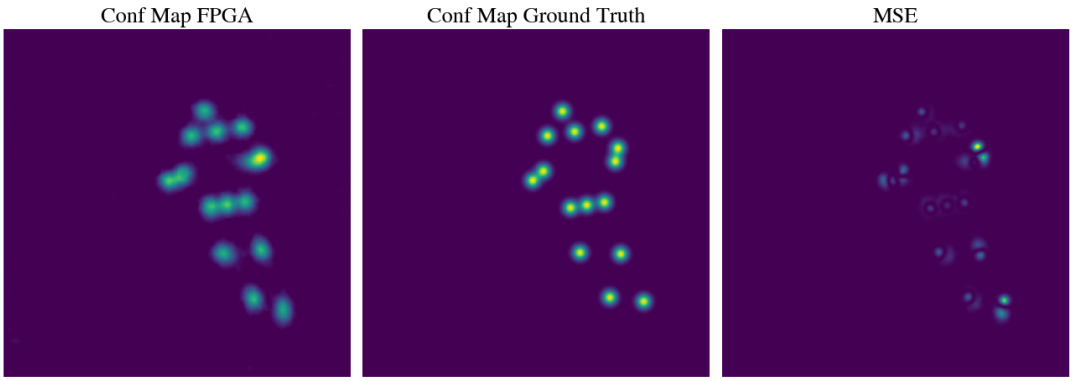


Figure 4.5: Left: The confidence map computed by the `INT8` network of the example shown in Fig. 4.2. Middle: confidence map of ht ground truth. Right: Mean squared error shown pixel-wise

Problem with Translation and Scale Variance The means squared error is highly translation and scale variant. A pose can be detected quite well, but if there is a small translation to the ground truth, the mean squared error will penalize this translation. Also for an input image where the the detected pose is small, the confidence maps are more likely to be similar.

4.2.3.1 results

In Table 4.2 the MSE of the three networks are listed.

Network	MSE
PReLU	0.9348
LeakyReLU	0.8109
LeakyReLU `INT8`	0.9332

Table 4.2: MSE of the trhee networks

Due to the problem with the translation and the scale this results must be treated with caution. But it shows a general tendency, the LeakyReLU network works just as well or even a bit better than the original PReLU network. With the quantization of the LeakyReLU network from `float32` to `INT8` some accuracy is lost, but it performed similar to the PReLU network.

Chapter 5

Conclusion

Literature Study The literature study has shown that, HPE has been implemented on FPGAs rather for the purpose of implementing a large deep learning model on an FPGA, than for of running complete HPE application. This work presents a HPE implementation which runs solely on an FPGA.

Performance With 8 fps for the KV260 platform and 27 fps for the VCK190 platform, The reached performance is adequate. If we recall the peak performance of the VCK190 platform (30 TOps) and KV260 (1.1 TOps) and the required operations by the ICAIPose network to compute a frame (100 GOPS), one can comprehend the numbers. For the VCK190 the camera interface or more precisely the used method to communicate has shown to be the bottleneck in the system, the DPU could process 46 fps. And for the KV260 with a theoretical throughput of 10 fps are the reached 8 fps reasonable. The Nvidia Jetson Xavier NX with its peak performance of 21 TOps could also only deliver 8 fps for the ICAIPose.

The loss due to quantization of weights and biases resulted in a small but acceptable loss of accuracy.

Vitis AI Many basic building blocks have been laid in this project. For the first time, a camera system and a self-developed neural network on an FPGA with the deep learning frame work Vitis AI Vitis AI from Xilinx is a great tool, but it is still at an early stage. Especially with the new hardware like the VCK190 and KV260, there are still some bugs and the documentation is lacking in some places. In the course of this work, new versions and additional documentation were created that could have saved weeks if they had been available from the beginning. The opensource policy and the ever growing community around Vitis AI help a lot to improve the tool. As an example, in the last week of this project Vitis AI 2.0 was released, with which many new possibilities are available. Even with these difficulties with Vitis AI, it is by far the best developed framework for deep learning on FPGAs. Of course there is always the possibility to build your own system on HDL level or help with a HLS. However, this would require an enormous amount of time and an excellent hardware knowledge.

In conclusion, Vitis AI is the way to go and one needs to keep and eye on it with its extremely fast development.

A Glimpse to Future Projects With the foundations laid, the project can be built upon as desired and the focus could be more on the implementation rather than the framework. There are many possible approaches for improvement. The interface between the camera and the PS

as could be redesigned, as well as possible acceleration on computer vision algorithms on the PL.

Chapter 6

Declaration of Authorship

I hereby declare that I am the sole author of this student research paper and that I have not used any sources other than those listed in the bibliography and identified as reference. I further declare that I have not submitted this thesis at any other institution in order to obtain a degree.

Michael Schmid

Rapperswil, December 3, 2025

Appendices

Appendix A

Task

OST Ostschweizer Fachhochschule

Master project thesis | **HS**
Michael Schmid | **2021**

MASTER PROJECT THESIS

for

Michael Schmid

Hardware Accelerated Pose Tracking

1. INTRODUCTION

Background of application

Physical therapy is a growing problem in an aging society. Moreover, physical therapy as it is provided today tends to be expensive, unpleasant, and limited by the availability of physical therapists. This results in unsatisfied patients, incorrectly prescribed physical therapy, and eventually skyrocketing healthcare costs.

Along with this, the proliferation of neural networks has shattered the boundaries of what is possible in the field of human motion analysis. New generations of hardware accelerators facilitate real-time operation of extremely complex computer vision tasks such as human pose estimation (HPE). HPE refers to the task of estimating a kinematic human body model from images. It provides quantitative human motion information that may be helpful for personalized physical therapy.

Special information about this work

Swiss startup VRMotion engineered a pose tracking system to localize 3D position of human skeletal keypoints in realtime. Their multi-camera system utilizes state-of-the-art neural networks and multiple high-performance graphics cards to estimate a kinematic body model of the person in the detection area.

To bring HPE to the physio patient, the pose tracking system will need to run on edge devices with little memory and low processing power, which is infeasible with VRMotion's current system. In a previous Master's thesis, a new neural network architecture, EfficientPose by Groos et al., has been implemented. EfficientPose is more computationally efficient than VRMotion's current system and is able to run on edge devices, such as the Jetson Xavier NX. In this work, the implementation of EfficientPose on FPGA devices such as the XILINX Versal VCK 190 Eval Kit or the XILINX Kria KV260 should be studied and a comparison between FPGA- and GPU-based devices should be performed.

OST Ostschweizer Fachhochschule

Master project thesis HS
Michael Schmid 2021

2. SCOPE OF WORK

Project thesis 1

1. Study literature on hardware-accelerated pose tracking implementations and camera interfaces for FPGA applications. Summarize your findings in a short literature study
2. Get yourself familiar with existing tools and evaluation boards for the purpose of hardware accelerated AI implementations and camera interfaces.
3. Select an evaluation board suited for low-cost implementation of one channel. Note that the second evaluation board, the Versal VCK 190 Eval Kit from XILINX, is given for this thesis.
4. Realize first the camera interfaces on both evaluation boards. The system shall only consist of a camera, the camera-interface, a RAM block for stream-based camera image storage, a stream-based read process from the RAM block and an interface of the stream to further processing units.
5. Add a classical computer vision algorithm example to both systems by using known algorithms and code. Alternatively or in addition to the above, you may also start your hardware implementation with a simple classical Deep Neural Network implementation.
6. Implement the algorithm based on "EfficientPose": https://arxiv.org/pdf/2011.04307.pdf. The implementation shall be based on the Master Project Thesis by Simon Walser.
7. Test your implementation with a previously designed test bench.
8. Compare the results of the implementations on boths boards and draw conclusions concerning performance, efficiency, cost, power, etc.
9. Compare your implementation(s) with the implementation on Nvidia Jetson Xavier NX by Simon Walser.

Documentation

1. A documentation must be written.
2. At the beginning of the work, a project plan must be created and agreed upon with the supervisor and industrial partner.
3. During the work, various review meetings must be scheduled to review important work steps. A minute shall be written for each of these meetings.

3. GOALS

- Understand theory, design and test of hardware accelerated AI implementations on FPGA.
- Realize a design that outperforms the existing implementatoin on Nvidia hardware in terms of speed (30 – 50 fps) while maintaining the resolution, hardware cost, power consumption….?.

4. REPORT

A report about the project has to be written in which all made considerations are described in detail. Investigations, calculations and tests inclusive illustrations have to be documented. The report shall be written legibly and clearly structured. The following items may not be missing. Table of content, abstract (half to a full page), original text of the assignment, introduction, main part, results summary of 2 - 4 pages and the appendix (with all relevant data, detailed information and bibliography). The

entire report not including appendix shall not exceed 60 pages. One printed copy of the report and all relevant data together with the report in electronic form must be handed over to the advisor.

5. APPOINTMENT SCHEDULE

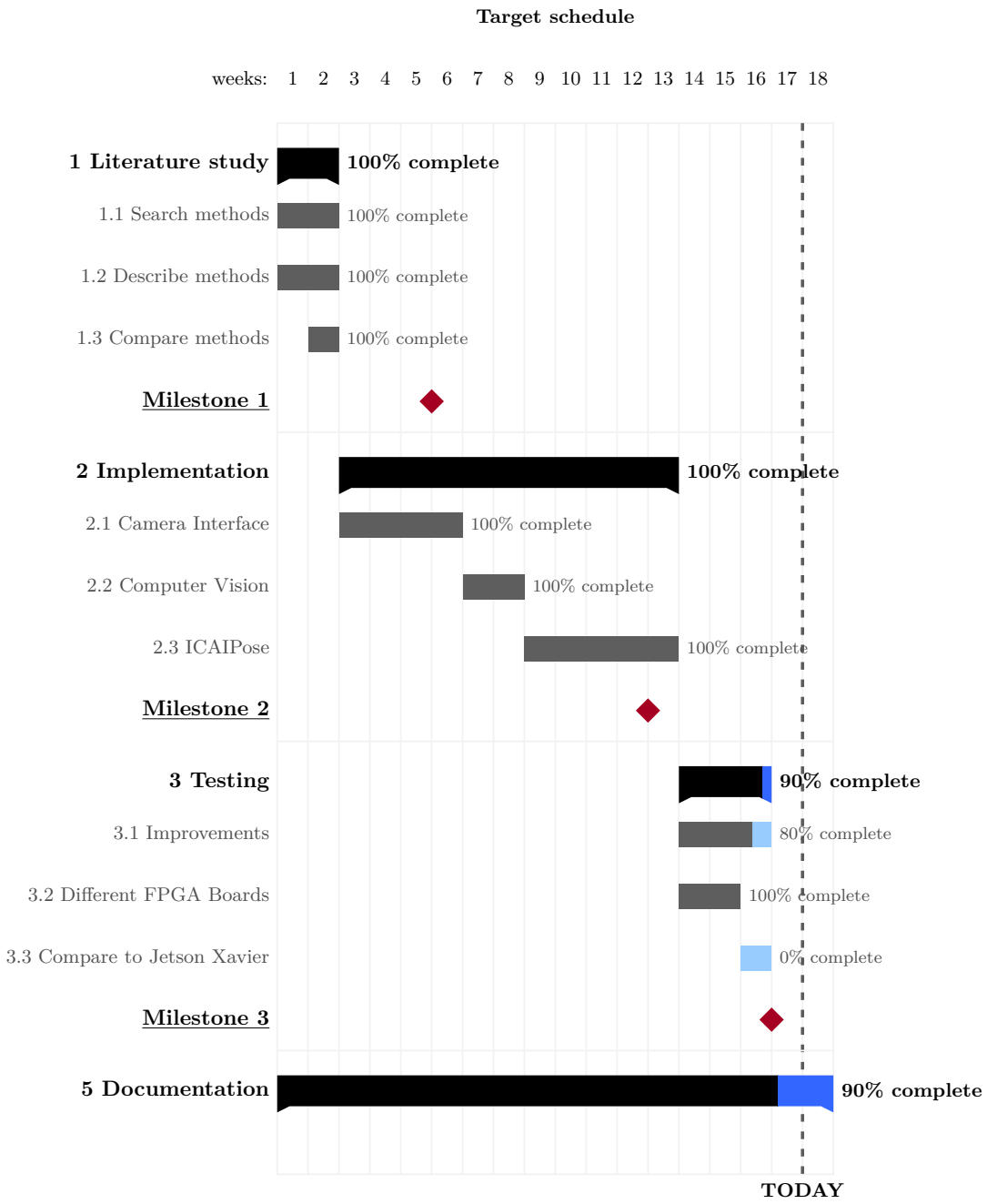
- Project start — Thursday, 20. September 2021
- Delivery of report — tbd, 2022
- Presentation — tbd.

6. ORGANIZATION

- Advisor — Prof. Dr. Paul Zbinden (+41 58 257 45 84, paul.zbinden@ost.ch)
- Industrial partner
- Meetings — weekly meetings
- Work place — @HSR, room 8.221

Appendix B

Schedule



List of Figures

List of Tables